A Framework for Supporting Shared Interaction in Distributed Product Development Projects

Babak Amin Farshchian

Information Systems Group
Department of Computer and Information Science
Faculty of Physics, Informatics and Mathematics
Norwegian University of Science and Technology

May 22, 2001

Abstract

Geographically distributed software development projects are becoming commonplace due to the wide-spread use of network technologies. Software development is an activity that is based on intensive cooperation among groups of developers. Therefore, network technologies used to support such projects have to provide proper support for this cooperation.

It is shown in this thesis that the *(software) product* being developed by a group of developers plays an essential role as a *resource for cooperation*. The product is used for externalizing knowledge, for resolving misunderstandings, and for coordinating the daily activities of the developers. Continuous, flexible and customized access to the product is shown to be a necessary prerequisite for using the product as a resource for cooperation. It is also shown, through a case study of a geographically distributed project, that geographical distribution hampers continuous, flexible, and customized access to the product. This often leads to break-downs in coordination and cooperative learning in product development projects.

A *product-based shared interaction model* is proposed as a suitable approach for supporting distributed product development projects in using the product as a resource for cooperation. The model emphasizes the uncertain nature of product development processes, and therefore does not impose any predefined patterns of cooperation. Instead, the model simulates some of the properties of the physical space. Awareness mechanisms are used to provide the developers with continuous information about the modifications to the shared product. A flexible interface to the product allows the developers to modify and annotate the product according to their emerging needs. Local customization mechanisms are used to allow the product to be used as a boundary object.

The product-based shared interaction model is formalized in form of a generic framework. The framework consists of three service layers, each with a well-defined set of services. The lower layers of the framework are concerned with cooperation in large groups, and use a *shared product space* for supporting shared interaction involving the product. The higher layers of the framework emphasize close and dynamic cooperation among the developers in smaller groups, and use *centers of interaction* for supporting cooperation in shared workspaces. The framework is implemented in form of a generic shared interaction platform that can be used to construct a range of cooperation support environments for product development projects. One such environment is described in details and demonstrates how the framework can solve some of the problems of geographically distributed product development projects.

Preface

This thesis is submitted to the Norwegian University of Science and Technologies for the doctoral degree "doktor ingeniør." The work reported has been carried out during the time period 1995–2000 at the Information Systems Group, Department of Computer and Information Science, under the supervision of Professor Arne Sølvberg. Parts of the work have been done in the context of the EU-sponsored project Aquarius, where I was involved during 1996-1998.

During the last five years I have enjoyed a dynamic and encouraging environment at IDI. The work presented in this thesis is a result of my interaction with this environment and the wonderful people who work there. My supervisor Prof. Arne Sølvberg has provided me with a perfect combination of freedom and guidance. I can say without any hesitation that it has been an honor for me to work in his group. My colleagues in IS group throughout the last five years have contributed continuously to this work through discussions and ideas. I thank in particular Terje Brasethvik, Hallvard Trætteberg, Arne Dag Fidjestøl, Tom Reidar Henriksen, Xiaomeng Su, Harald Haibo Xiao, and Håvard Jørgensen for their contributions. I also thank Lisbeth Vaagan and Anne Berit Dahl for their valuable help with preparing the thesis.

Many diploma and project students have been actively involved in defining and implementing the ideas in this thesis. I thank all of them. Their contributions are referred to throughout this thesis. In particular I thank the following: Yin Bin, Tieyan Li, Jinghai Rao and Xiaomeng Su for their efforts in implementing MultiCASE. Nils–Helge Garli and Anders Lund for implementing the last version of Gossip. Stig Peter Olsrød and Trond Isaksen for implementing ICE.

During the first half of year 2000 I was given the opportunity to stay at the Cooperation Technologies Laboratory of University of Milano Bicocca. I thank Prof. Giorgio De Michelis, the head of the laboratory, and Alessandra Agostini for making my stay possible and very instructive. I thank Andersen Consulting Forskningsfond for their economical contributions to my stay in Milano, and to the work presented here. I thank my colleagues at Telelogic Norge AS, in particular Ståle Deraas, for providing me with an enjoyable and instructive work environment, and for giving me the freedom to finish the last steps in my study.

My wife, Monica Divitini, deserves a special acknowledgement. She has been helping me both with forming the ideas presented in this thesis, and with thoughtful reviewing of almost everything I wrote. Her presence has also been invaluable in my difficult moments, when I thought I would never be able to finish. I thank you for all this Monica, and above all for giving Veronica to me.

Babak Amin Farshchian
Trondheim May 22, 2001

Contents

Chapter 1

Introduction

The aim of this thesis is to develop collaboration technologies that can support geographically distributed software development projects. Software development is considered as the process of creating a new *product* based on the stated needs of a customer. This process involves refinement of requirements and ideas, externalization of developers' knowledge, creation of an agreed-upon understanding of the needs and solutions, as well as development of software that will support the customers in their business. The process of product development is highly cooperative. Cooperation among large groups of people is needed because one person's knowledge, authority, and available time are not enough for developing a large product.

Cooperation is considered in this thesis to be the social interaction that happens among the developers on a day-to-day basis, and that contributes to the refinement of the product. Cooperation is distinguished from *control*. The assumption is that control and cooperation are two aspects of product development that have to co-exist in order to create high-quality products.

Most product development environments, such as CASE (Computer Aided Systems Engineering) and SE (Software Engineering) tools, have focused on supporting the formal aspects of product development, including project management aspects. This focus on formal aspects has played an important role in the acceptance and advancement of conceptual modeling and development methods. However, we see two important reasons why future product development environments cannot afford to underestimate the cooperative aspects of product development. First, more and more often development projects become geographically distributed. Social interaction that was previously supported by physical proximity breaks down when this proximity does not exist anymore. Development environments have to advance their support for cooperation in a way that is coherent with how developers cooperate face-to-face. Second, since cooperation takes a large part of day-to-day activities of developers, by supporting cooperation (even in case of co-located groups) a development environment can create a medium for capturing the knowledge that is necessary for the development of the product *as this knowledge is created*.

The perspective on cooperation as an emergent social process is strongly based on our first-hand experience from a distributed product development project called ALPHA (1997-1998). AL-

PHA was a research and development project involving approximately 40 developers from three European countries. The aim of the project was to develop an Internet-based information system for knowledge sharing and cooperation in the European aquaculture sector (more details on this project are provided in Chapter 2 of this thesis). The characteristics of the project were:

- *Local control*: The involved parts were universities and research centers with a high degree of local autonomy. This made it difficult to impose any strong control, at the same time stressing the cooperative aspects of product development.
- *Complex product*: The project was set out to develop an innovative and quite large product based on a set of abstract needs. This necessitated dynamism, creativity, and intensive cooperation.
- *Distribution*: The project demonstrated a high level of both geographical and intellectual distribution. The participants had diverse background knowledge, ranging from aquaculture scientists to project managers and computer scientists. In addition, all the participants were facing geographical distribution and had to use some form of collaboration technology in order to cooperate with others.

The ALPHA experience revealed to us the important impact of cooperation on facilitating knowledge creation and coordination in product development projects, and the negative impact of geographical separation of project participants on the quality of the resulting product. ALPHA has also demonstrated the need for computer-based systems to support seamless interaction between the formal and informal.

The research reported in this thesis is conducted in the Information Systems (IS) group at the Norwegian University of Science and Technology. IS group has a strong tradition in developing formalisms and tools for information systems modeling. Within this tradition, cooperation has always been considered as an important part of the overall process of developing information systems (Sølvberg 2000, Andersen 1994, Lindland, Sindre and Sølvberg 1994, Andersen and Sølvberg 1993). The need to support geographically distributed groups of developers has emerged with the wide-spread use of Internet in the recent years, and has further increase the need for development environments that can support these distributed groups in their work. This thesis finds itself in the research tradition of IS group, and contributes to the research in information systems development by suggesting and developing novel systems for supporting cooperation in distributed project teams.

1.1 Research Questions

The overall research question this thesis tries to answer is:

> *How can we support, through computer-based tools and environments, cooperation among developers in geographically distributed product development projects?*

As a necessary step in answering this overall question, the thesis also tries to give an answer to the following questions:

- **RQ1:** What is the nature of cooperation in product development groups, i.e. what is the meaning of “cooperative product development”?

- **RQ2:** What is the effect of geographical distribution on cooperative product development?
- **RQ3:** What kind of computer-based tools and environments are needed for supporting cooperative product development?

1.2 Approach

We have approached the problem through defining a framework for enhancing product development environments with cooperation support. This framework, called IGLOO, is developed as a result of our experience from ALPHA, but also as a result of developing and testing a number of prototypes. Our participation in ALPHA in particular gave rise to several important observations:

- Descriptions and specifications of the product being developed are used actively by developers as *resources for cooperation*. These descriptions and specifications are used for exchanging knowledge and for cooperative learning. They are also powerful coordination mechanisms, used by developers on a day-to-day basis for aligning their actions to that of others.
- A precondition for using the descriptions of the product as a resource for cooperation is that developers can participate in *shared interaction* with each other and with these product descriptions. Shared interaction, as opposed to individual interaction, is interaction that is visible in a shared context. Shared interaction is easily achieved in co-located groups. For geographically distributed groups explicit efforts are needed in order to enable shared interaction.
- The emergent nature of product development, the high degree of local control, and the difficulty of enforcing global procedures and tools require *flexibility* and *interoperability* in the computer-based support.

Different prototypes were developed in the course of this research in order to find out how computer-based systems can best support the above observations. The focus of all these prototypes has been on *the product being developed* by a group of developers. This focus is not only based on our observation from ALPHA. It is also based on the fact that most conventional product development tools and environments support a strong notion of a product and its specifications stored in a central repository. Constructing cooperation support technology that can co-exist with already-existing and widely used tools and environments necessitates an appreciation of what has been already used as a successful support mechanism.

The developed prototypes have resulted in the generic *IGLOO framework* with the following properties:

- IGLOO introduces a new perspective on the product being developed by a project team. It regards the product and its descriptions primarily as a resource for cooperation. IGLOO makes active use of these descriptions, not only as a container of information, but also as a place for exchanging knowledge, resolving misconceptions, and coordinating the efforts of the developers.
- IGLOO emphasizes the flexibility of the cooperative work that is needed for creating a new product. Instead of specifying policies for how this cooperation would or should happen,

IGLOO provides a medium for supporting learning, and for capturing knowledge that is created as a result of this learning.

- IGLOO supports the interplay between the formal and the social. It is recognized that product development consists of highly informal interactions among developers. It is these informal interactions that eventually result in formal products.
- IGLOO supports geographically distributed groups by providing a *virtual space* for enabling shared interaction. This virtual space makes extensive use of *awareness* mechanisms developed in the CSCW (Computer Supported Cooperative Work) research field in order to allow developers to participate in a shared environment, regardless of their geographical location.
- IGLOO is defined as an "operative system" for cooperation. IGLOO can be used for integrating already existing tools into a coherent cooperative environment.

In addition to the prototypes that were developed during this research, the core IGLOO framework has been designed and partly implemented in form of a *generic implementation*. This generic implementation can be developed into a full-fledged cooperative product development environment. In addition, IGLOO framework and its generic implementation can be used to enhance existing conventional CASE and SE tools with support for awareness and shared interaction. Our approach has been to focus on the cooperative aspects of product development *first*, and then extend to the more formal aspects. This means for instance that although IGLOO places the product and its representations in the center of attention, it does not support traditional repository operations such as transformations, version control, and configuration management. We believe the approach of "shared interaction first" has given us the freedom to focus on what is important for cooperation, without compromising cooperation support for other purposes.

1.3 Background

The research reported in this thesis is by and large constructive. Our goal has been to design and develop innovative systems for supporting cooperation in geographically distributed product development teams. This is in line with the research tradition of the IS group, where the results of the reported research are to be applied in a larger framework of support tools for information systems engineering (Sølvberg 2000). Grounding the design of cooperation technologies in real world observations of groups is a critical success factor for research within the tradition of CSCW (Schmidt and Bannon 1992, Olson and Olson 1991), where this thesis is partly located.

The method undertaken in this thesis is a mixture of empirical investigations, literature study, and prototyping activities. The resulting IGLOO framework and its generic implementation are based on the following sources of data:

- The empirical study of ALPHA, and the analysis of everyday problems that distributed product development project similar to ALPHA experience. This case study, in combination with published empirical investigations of cooperative product development done by other researchers, has largely shaped the constructive part of the thesis.
- The study of existing support technology, their shortcomings and their strong points, and the feasibility of future technological solutions.

- The development of various prototypes. Some of these prototypes have been used by end users, and in this way have provided valuable feedback for the development of new prototypes. Some have served as demonstrators of ideas.

The author's involvement in ALPHA resulted in a prototype of a Web-based knowledge-sharing system called ICE (Internet Collaboration Environment). ICE is a *radically tailorable*[1] system, allowing its users to create Web-based knowledge-intensive applications through an easy-to-use tailoring interface. These applications serve the emerging knowledge-sharing needs of the users. All user-tailored ICE applications are based on a predefined set of building blocks, i.e. a predefined set of *knowledge object types* and a specific type of *shared workspace*. ICE was defined cooperatively by ALPHA team, and was designed and developed by the author and a group of project and diploma students who were supervised by the author. The development took approximately 30 work-months over one year. ICE is described in Appendix A in this thesis. More details on the different parts of the prototype can be found in (Farshchian and Divitini 1997) and in the various technical reports and diploma theses (Knudsen and Solheim 1999, Asbjørnsen and Ellingsen 1997, Damskog 1997, Sivesind and Grimstad 1997, Olsrød and Isaksen 1996).

During the project, several versions of ICE were released. Each version was used by the participants for testing and collecting feedback for the next version. Feedback was collected both for adding new functionality and for improving existing functionality. These usage and feedback cycles allowed us to evaluate ICE with respect to supporting a distributed group of developers. The analysis of the usage of ICE together with other WWW applications and mailing lists used by ALPHA members threw light on some strong and weak points of the prototype. The strong points were the *tailorability* and *accessibility* of ICE. End users could freely tailor their own shared workspaces, and structure multiple workspaces into hierarchies. They could freely insert knowledge objects such as documents, drawings, source code, etc. in these spaces. In this way the users could build knowledge-intensive applications tailored to their real needs. In addition, ICE had an intuitive WWW-based user interface and a limited email interface. In this way the functionality of the prototype was available to anybody using a standard Web browser or email client.

However, ICE had limitations in three areas that were critical for product development. First, ICE provided *limited support for coordination and cooperative learning*. ICE lacked support for visibility of overall work. The shared workspaces tailored by the users were disconnected from each other. This meant that user activities and contents in single workspaces were completely invisible to remote users who were not using those workspaces. Moreover, ICE provided little active support for delivery of information to its users. Because of these limitations, often users were not aware of what other remote users were doing, with consequences for coordination and long-term learning (see Chapter 2 for more on these issues). Second, ICE had a *weak notion of product*. ICE had little support for representing and sharing of conceptual objects such as models and descriptions of the product being developed. In addition, ICE had no support for relations among different objects. Third, ICE was *weak in supporting explicit communication* among its users. Asynchronous communication was supported by external mailing lists, and support for synchronous communication was mainly lacking.

[1] The term *radically tailorable* was introduced by Malone, Lai and Fry (1995) to denote computer systems that allowed their end-users to tailor applications that were fundamentally different from each other. This tailoring was called radical because most tailoring mechanisms only allow end-users to change the "surface behavior" of an application. In these cases, the functionality of the underlying application is the same before and after the tailoring activity.

After ALPHA was finished, we did an analysis of the cooperation that had taken place among the project participants. We tried to understand how a geographically distributed product development project such as ALPHA, with participants having different background knowledge and skills, used collaboration technologies to conduct their work. This analysis revealed the importance of being able to continuously share representations of the product being developed in form of *interconnected knowledge artifacts*. Our analysis also revealed the importance of *flexible computer support* for shared interaction in geographically distributed teams. The results of this analysis, together with a comparison to related literature, are presented in Chapter 2 of this thesis.

This empirical analysis guided our second round of prototyping. This time we focused on the shortcomings of ICE. In particular we tried to increase the support for visibility of work using a *shared product space* and associated *awareness support mechanisms* for active delivery of information. We put less emphasis on the tailorability and accessibility aspects (which were the strong points of ICE) because of limitation in development time and resources. We designed and implemented a prototype of a product development tool, called MultiCASE, with focus on shared interaction centered around the product being developed by a project group. This prototype was designed and implemented by the author and four diploma students supervised by the author. The development consumed approximately 30 work-months and lasted six months. Details about the prototype's functionality can be found in (Farshchian 2000b, Farshchian 1999) and in Chapter 5 of this thesis. Details about the implementation of MultiCASE can be found in (Bin, Farshchian, Li, Rao and Su 1999). In short, MultiCASE is a synchronous product development tool built on top of a shared product space consisting of objects and relations among these objects. MultiCASE allows a group of distributed developers to share the model of a product (a software architecture) through the shared product space, and to interact with this model through synchronous shared workspaces. MultiCASE has a much stronger notion of a product model. In addition, shared workspaces in MultiCASE contrary to those in ICE are not completely isolated from each other. MultiCASE helps its users to keep themselves aware of the activities involving other parts of the product than those currently in focus. The shared product space is in charge of producing enough *awareness information* to let this keeping aware happen.

Our experience from developing ICE and MultiCASE, and from analyzing ALPHA, resulted in the development of *IGLOO framework*, which is described in details in this thesis.[2].

IGLOO builds on a *product-based shared interaction model* as outlined in Chapter 4 of this thesis. This model consists of a *shared product space* and *centers of interaction*. The shared product space provides a virtual space where product developers can be involved in product development activities. The product development activities of the developers may happen in centers of interaction. A center of interaction provides customized access to the underlying product and creates a context for the activities happening inside it. IGLOO framework is a detailed formalization of this model in form of three *service layers*. The service layers are partly implemented in form of an operative system for supporting cooperation. Generic implementations of each layer are developed partly and used for testing the functionality of the framework (Farshchian 2000a, Garli and Lund 2000, Lie-Nielsen 2000, Rømma and Skjønhaug 2000). IGLOO is developed in a way to support the creation of *IGLOO networks*. IGLOO networks are collections of product development tools that communicate through IGLOO framework and create a flexible environment

[2]Igloos are built on a landscape of ice. They are an integral part of the landscape since they are built using the same underlying ice. At the same time, each igloo gives protection to the eskimos living in it, separating them from the harsh and messy environment outside. In our framework, the ice landscape can be thought of as being a large and messy product, while each igloo gives shelter to a small group of developers.

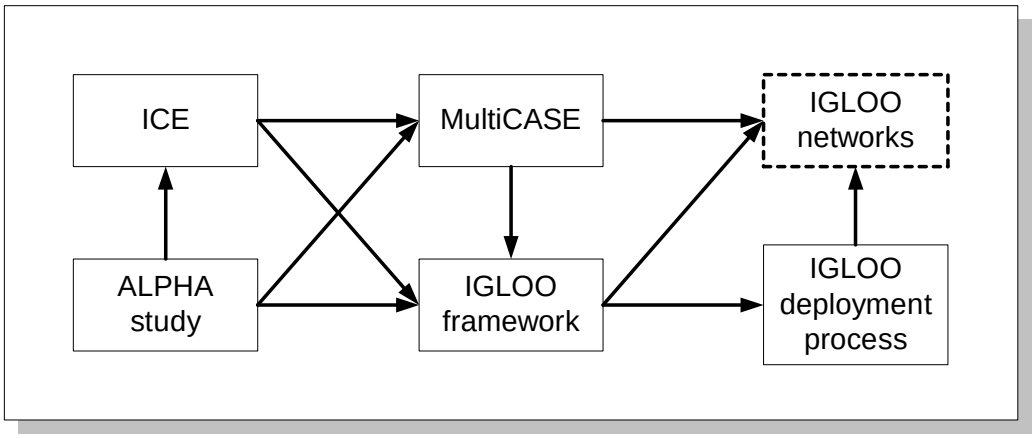


Figure 1.1: The research contributions.

for cooperative product development. The deployment of IGLOO framework happens through *IGLOO deployment process*.

We developed IGLOO as a framework rather than a single application. By framework we mean that IGLOO is a definition of a set of generic services instead of being a specific application. Practical issues were one reason for making this decision. Implementing a specific application required us to implement a large amount of functionality within the application (functionality that was not the focus of this research, such as advanced repository services), and promoting the application as the best one among all the existing advanced development tools. Having chosen to develop IGLOO as a framework has provided us with a number of advantages, such as allowing third-party tools and environments to be integrated into a cooperative environment through well-defined service interfaces. In addition, the services of the framework do not enforce any strict "cooperation policies" on the applications that can be developed or integrated into the framework. This means that widely different cooperation scenarios can be supported by the framework.

Figure 1.1 shows roughly some of the main contributions of this thesis and the dependencies among them. IGLOO networks will be developed through deploying IGLOO framework. Although an example IGLOO network is described in this thesis, IGLOO networks are future work and are not treated in details in this thesis. The box with IGLOO networks is drawn with dashed lines in Figure 1.1 in order to indicate this. Table 1.1 shows an overview of the developed systems and frameworks, and their strong and weak points.

Table 1.1: Systems developed during the reported research, and their strong and weak points with respect to support for cooperative product development

System/ framework	**Strong points**	**Weak points**
ICE	Ease of use – Accessible and open user interface – High level of support for tailorability.	Weak notion of product model and interdependencies among product parts – No support for active delivery of (awareness) information – Little support for explicit communication.
MultiCASE	Strong notion of a large product model – Support for dependencies among product parts and among developers – Active delivery of (awareness) information – Support for synchronous cooperation.	Little support for tailorability – Lack of an open and extendable architecture.
IGLOO framework	Supports strong points of both ICE and MultiCASE – Supports a high degree of tailorability and flexibility in defining different types of product models – Support highly tailorable awareness services for active delivery of information.	See Chapter 10.

IGLOO framework is developed as a result of several cycles of development and empirical evaluation. The possibility of being involved in a real world project has provided essential insight into problems that may occur as a result of geographical distribution in product development teams. Together with an study of the literature on software engineering as a cooperative activity we believe we have been able to base our technical contributions on real requirements. The development of prototypes has played a central role in understanding the complexity of distributed systems and their existing limitations.

1.4 Contributions

The main contributions of this thesis are the following:

- An empirical analysis of cooperative product development: This analysis shows the importance of computer support for *shared interaction*. It also demonstrates the importance of integrating support for formal and informal levels of product development in the same support environment. In particular, the interaction between the formal product and the informal processes of *cooperative learning* and *coordination* need to be explicitly supported for geographically distributed groups.

- A product-based model of shared interaction: This model is an improvement of existing models of interaction with respect to support for product development. The central role of product is emphasized by extending the notion of an artifact to support coordination, knowledge creation and cooperative learning. Centers of interaction allow flexible interaction with the product. The model acknowledges the differences between cooperation in small and large groups, and is an attempt to provide an integrated support for both.
- A framework for cooperative product development: IGLOO framework formalizes the product-based interaction model into detailed definitions of three service layers that provide different levels of cooperation support.
- A generic implementation of the framework: The framework is partly implemented in form of three generic servers, one for each service layer. These servers provide a scalable distributed architecture for the framework and facilitate the deployment of the framework.
- A method for deploying the framework: Due to a high degree of flexibility, IGLOO framework has to be customized for each individual project. A companion method is developed to facilitate the deployment process of the framework in different types of project.

1.5 The Structure of the Thesis

Chapter 2 is an analysis of cooperative product development. The analysis shows the importance of the product as a resource for cooperation. This chapter also present a case study of ALPHA as an example of geographically distributed product developed project. The chapter concludes with a set of requirements for product development environments.

Chapter 3 present a state-of-the-art survey of computer-based tools and environments for supporting cooperative product development. We focus on configuration management and CASE tools, and shared workspace applications. The survey gives an overview of the feasibility of computer-based systems, and also reveals a number of shortcomings in the existing systems.

Chapter 4 outlines a product-based shared interaction model. This model enhances the existing shared interaction models with concepts that are specific for cooperative product development, such as shared product space and centers of interaction.

Chapters 5– 9 define IGLOO framework. Chapter 5 gives an overview of the framework and describes an example, i.e. MultiCASE. Chapters 6– 8 describe the three service layers in IGLOO framework. Chapter 9 outlines the IGLOO deployment process and its different activities.

We evaluate IGLOO framework through applying the IGLOO deployment process to ALPHA in Chapter 10. Chapter 11 concludes the thesis and proposes some directions for future research.

Chapter 2

Cooperative Product Development

2.1 Introduction

This chapter provides an analysis of the key role that the product being developed plays as a *resource for cooperation* in product development projects. The analysis is based on data from two sources. First, a review of the literature on the social aspects of product development is presented in Section 2.2. We focus on co-located teams, and see how the product is used for supporting cooperation in co-located teams. Second, our own experience from a real world distributed project is presented as a case study in Section 2.3. This case study further investigates the effect of geographical distribution in the practice of product development. Section 2.4 presents a set of requirements for supporting collaboration in product development environments. Before proceeding to these sections, some basic concepts used in this chapter and the rest of the thesis are defined in the following.

The (software) product and its developers are in focus in this chapter. We make an analytical distinction between the product being developed, and the end-product of a development project. In this thesis we use the term *product* when we refer to *the product being developed* by the development project, while the term *end-product* is used to denote the resulting *executable software* that is used by its intended users in its intended environment. The product contains all requirements and design documents, drawings, minutes, notes, source code, intermediate prototypes[1], etc. that are generated during the lifetime of a development project. The end-product is the executable code and its user documentation that are delivered to the customers of the project as the end result of the project. The product itself plays a "dual role" in the development process (Seltveit 1994). The product is used for supporting communication and understanding among the developers in the development project. The same product and its different configurations are later used for (automatically or semi-automatically) manufacturing the end-product in form of an executable software (see

[1]Note that an earlier version of an end-product may play a central role in supporting the communication among the developers when they develop a future version. In this case the version is regarded as an intermediate prototype for the next cycle of development.

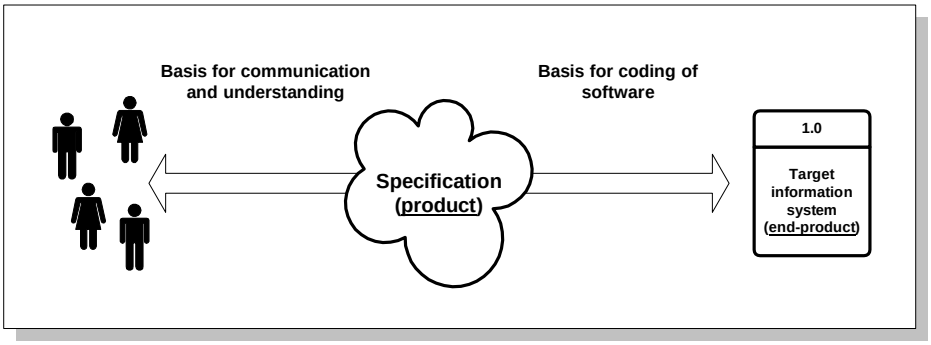


Figure 2.1: Duality of systems specifications. Adapted from Seltveit (1994).

Figure 2.1). Our concern is about using the product for communication and understanding among developers.

We use the term *developer* to denote all the people who play a visible role in the product development activities. Developers can be analysts, different domain experts, project managers, programmers, etc. A *product development project*, or project for short, is the organizational unit that is actually performing the product development activities. A *product development group*, or group for short, is a subset of the developers in a project who are involved in solving a specific issue or a task related to a product. *Product development activities*, or activities, are those activities of the developers that influence the product, i.e. result in refinements to the product. These activities do not need to be those that modify the product directly. For instance, chance encounters and opportunistic discussions in hallways and coffee rooms always play an important role in shaping a product. A *product development environment*, or development environment for short, is a technical infrastructure, tool, or technology that is used to support the developers in a project during their product development activities.

The product is seen as constituted by a set of interconnected artifacts called *product objects*. Examples of product objects are requirements documents used for collecting user requirements, user interface mock-ups for demonstrating future usage scenarios, formal and semi-formal models describing the problem domain, design documents describing the technical design of the computer system, source code files written in a programming language, etc. Product objects do not exist in isolation, but their meaning is normally defined in relation to other product objects. There exist different *relations* among product objects. Examples of relations are dependency relations, import relations, "part of" relations, etc. An end-product is created through transforming a *configuration* of product objects into an executable software. A configuration will normally include a subset of all the available product objects. *Cooperative product development* is the process of creating these product objects and relations. This process consists of intensive cooperation among the developers, which in turn results in externalized knowledge that constitutes and shapes the product. As we will see in this chapter, this process is a highly emergent one. Because of this emergent nature of the process, it is important to consider the product, together with its developers, existing within a continuous *social environment* (see Figure 2.2). In this social environment, the product is regarded as an "anchor" for supporting cooperation among the developers.

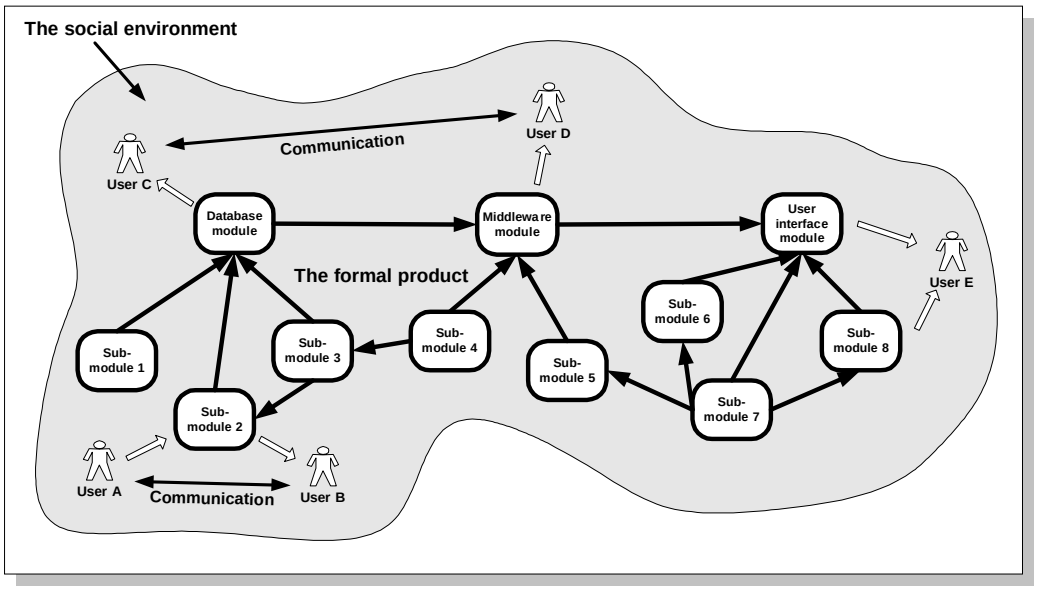


Figure 2.2: A product is always developed within a social environment.

2.2 Cooperative Product Development

This section presents an analysis of cooperative product development. This analysis is based on empirical and theoretical studies of cooperative product development as published in the literature. We focus in particular on the role of the product in the development process, and the way it supports cooperation among developers. The product being developed is a central part of any product development project because it is used as the basis for manufacturing the end-product. The success or failure of the whole project will normally be related to the quality of the product. Most product development environments, such as CASE and SE tools, are built around a product (which is often stored in a central repository). Interacting with the product takes a considerable part of the activities of the developers. Our analysis, which is summarized in this section, shows clearly that the product being developed plays an essential role in supporting the cooperation among the developers in the course of a project. This perspective on products has implications for current product development environments, where the assumption is often that the product is a (semi-) formal specification used as the basis for various transformations.

The rest of this section is organized as follows. First, three properties of products are examined. These are the properties of being *externalized knowledge*, *boundary object*, and *coordination mechanism*. From our analysis of the literature we believe these three properties are the most central ones with respect to cooperation. Second, we look at how co-located teams utilize these properties of the product. It is shown that physical proximity plays an important role in using the product as a resource for cooperation. Physical proximity (e.g. being in the same room, or in the same building within reasonable physical distance from other developers) offers continuous access to the product, supports flexible interaction with the product, and allows groups of devel-

opers to easily cooperate on different tasks related to the product. A summary of the discussion is shown in Table 2.1. In this table, the three properties of the product are show in the left column, their positive effects on cooperation are shown in the middle column, and the effects that physical proximity has on utilizing the properties are shown in the right column.

Table 2.1: Three properties of product that are important for supporting cooperation, with the necessary pre-conditions for utilizing them

Property	**Support for cooperation**	**The role of physical proximity**
Product is externalized knowledge	Supports cooperative learning, criticism, creativity.	Shared physical space: embodies the developers and the product – supports continuous exchange of information related to the product – supports flexible and customized interaction between the developers and the product.
Product is boundary object	Supports understanding across different communities of practice – Facilitates negotiation of local understandings – Supports information sharing.	Shared physical space: allows the developers to customize the product to their local needs – offers low-cost and dynamic communication channels for resolving misunderstandings.
Product is coordination mechanism	Supports coordination of day-to-day activities of developers.	Shared physical space: allows continuous access to information about modifications to the product – supports access to information about the process through which these modifications are made.

2.2.1 The properties of the product

Our analysis of the literature on social aspects of product development shows that at least three properties of products are crucial for cooperative product development. First, a product is externalized knowledge. Developers cooperate in order to externalize their knowledge and confront it with that of other developers. The product is created as the result of this knowledge creation and learning process. Second, a product is a boundary object. Every development project consists of developers with varying domain knowledge and background. The same product is often used by all these developers in order to support the different local understandings of accumulated knowledge. Third, a product is a coordination mechanism. Information about the status of the product is used effectively by the developers in order to coordinate their day-to-day work with that of other developers.

Product is externalized knowledge

Developing large products is a knowledge-intensive process (Waterson, Clegg and Axtell 1997, Walz, Elam and Curtis 1993, Curtis, Krasner and Iscoe 1988). This is true in two senses. First,

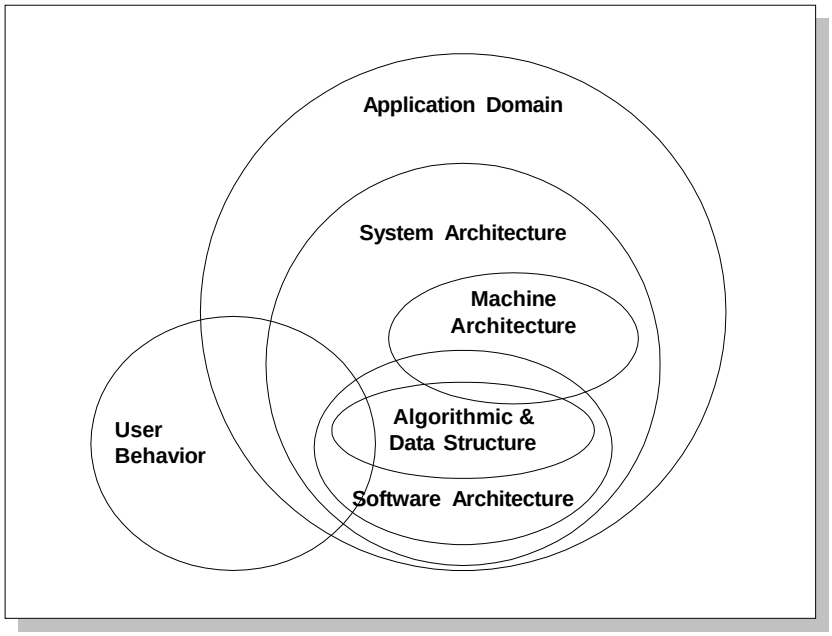


Figure 2.3: Different types of domain knowledge involved in software development. Adopted from Curtis et al. (1988).

due to the product's complex operational environment there is a need for different kinds of specialized and overlapping *domain knowledge* in order to be able to develop the product (see Figure 2.3). Second, products themselves are often complex, and knowing them in detail requires a large amount of knowledge about the product, i.e. *product knowledge*. Mutual knowledge is a prerequisite for effective cooperation (Krauss and Fussell 1990). Mutual knowledge facilitates communication and makes cooperation more effective. Product knowledge constitutes a large part of this mutual knowledge in a project. Developers need to know enough about the product's conceptual model in order to be able to communicate about it, and to cooperate with each other for its development. Product and domain knowledge are often interwoven, and the boundaries are changing dynamically during the course of a project. In fact, product knowledge is gained through acquiring and integrating the domain knowledge of the developers. On the other hand, in most cases product knowledge does not only mean knowledge about an isolated technical construct. Product knowledge also may include knowledge about how the end-product will function in its future environment, and how it will interact with organizational, social, and technical aspects of this environment[2].

Product knowledge is thus knowledge that is created, shared and modified by the developers.

[2]There are of course other types of knowledge that are necessary in any project. These include knowledge about the project's organization, knowledge about the market, knowledge about competing products and organizations, etc. We will here focus only on domain and product knowledge because they are more directly related to product development environments.

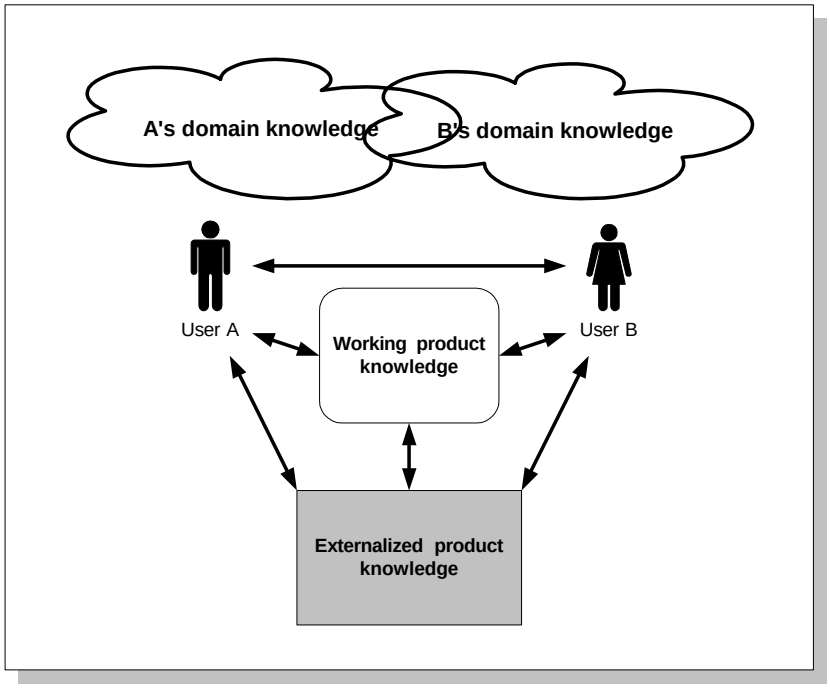


Figure 2.4: Knowledge creation in product development. Arrows denote flow of information.

Product knowledge is mainly non-existing in the beginning of a project. It is created through a "socialization" process where the tacit domain knowledge of each developer is confronted with that of other developers (Nonaka and Takeuchi 1995). It is important to note that, at any time during the course of the project, product knowledge is only partly created. It is also important to note that product knowledge resides only partly in the existing product objects and relations. This means that the product development project as a whole will normally know much more about the product than what is recorded in the externalized product. We will use the term *externalized product knowledge* to denote product knowledge that is recorded in form of product objects and relations. The term *working product knowledge* is used to denote product knowledge that is created (i.e. understood and agreed upon by a group of developers) but is not yet recorded as part of the product. Figure 2.4 shows an overview of these concepts.

From this perspective, the product (i.e. product objects and relations) is the externalized product knowledge. Product objects and relations among them are used for externalizing product knowledge because they are tangible and accessible in a shared context. Product objects often start and accelerate cycles of knowledge creation as they play the role of "hooks" for attaching new knowledge (Curtis et al. 1988). Externalization of product knowledge happens interactively, and is the basis for a continuous *learning process*:

> "*Since we are normally used to talking about knowledge only when it is explicitly given, a learning cycle can be characterized as the updating or revision of the re-*

> *spective artifacts. A learning cycle in software development may thus be identified with the production of a new version; in software engineering it may be the development of a new generation of tools and methods.*" (Keil-Slawik 1992, p.174)

Each new version of a product object is thus a refinement of externalized product knowledge. Product development can from this perspective be seen as a process of convergence from vague ideas into increasingly clearer conceptual models (Potts and Catledge 1996). Product as knowledge that is gradually externalized plays an important role in supporting a long term *cooperative learning* process in the development project. This learning happens mainly through observation of changes and gradual confrontation of ideas, i.e. it is a process of *situated learning*. What is learned is normally unique for each project, and depends on the type of the product that is being developed.

Product is boundary object

Different *communities of practice* (Brown and Duguid 1991) will understand and/or interpret situations in different ways. This is in particular true in product development projects. These projects are often constituted by developers with highly varying backgrounds and domain knowledge (i.e. belonging to different communities of practice). Groups of developers work on different parts of a large product, and they often employ their specialized perspectives when solving issues related to the product. In addition, the situation or problem domain that the product tries to support is often complex. Local interpretations are often present and necessary for coping with the complexity. The role of the product here is that of "unifying" the different communities of practice and their views. For instance, a module in a software product might be seen as an artifact fulfilling some organizational or business requirements, as an operational unit capable of performing some concrete operations, or as a technical artifact constituted by pieces of executable code and hardware, all depending on the background and local context of its developers. The product has to provide a common view, but also support the local interpretations of the involved developers. In other words, the product is a *boundary object* as defined by Star and Griesemer (1989):

> "*. . . an analytical concept of . . . objects which both inhabit several intersecting social worlds . . .* and *satisfy the informational world requirements of each of them. Boundary objects are objects which are both plastic enough to adapt to local needs and the constraints of the several parties employing them, yet robust enough to maintain a common identity across sites.*" (Star and Griesemer 1989, p.393).

Product as boundary object is located in the boundary of several communities of practice. In this way, it plays an important role in making possible cooperation among developers divided by organizational, cultural, temporal, physical, or other types of distances. According to Star and Griesemer there are four types of boundary objects, all of which can also be identified in product development activities:

- *Repositories*: These are collections ("piles") of objects that are indexed in a standardized fashion. Repositories are modular in that different people can use different objects from the "pile." An example is the set of all the product objects constituting a product, or the set of technical manuals available to the development team. Each developer or group of developers will use only a subset of the repository.

- *Ideal type*: This is the object that can be adapted to different contexts. It is abstracted from all contexts and is fairly vague. It serves as a means for communicating and cooperating symbolically. "A 'good enough' road map for all parties". For instance, an ER diagram can be used for communication between programmers and domain experts.

- *Coincident boundaries*: Common objects which have the same boundaries but different internal contents. For instance, the architecture of the product has the same boundaries (the same boxes and the same arrows among the boxes) but can be visualized differently for a performance analyst (interested in bottlenecks in the architecture) and a project manager (interested in the progress of work in each box).

- *Standardized forms*: Boundary objects devised as methods of common communication across dispersed work groups, e.g. a standardized bug report.

The product as boundary object supports cooperation among the members of a community of practice by allowing them to create local understandings. Creating local annotations, or using a notation that is commonly used by the members of the community of practice, allows these members to cooperate through the product more efficiently. The "standard" representation of the product, which is used by intersecting communities of practice, allows these communities to cooperate across their discipline and to confront their domain knowledge with that of other communities. Being understandable both within and across communities of practice, the product plays a central role in sharing information and opinions in a product development project.

Product is coordination mechanism

The two previous properties of product, i.e. product as externalized knowledge and boundary object, are based on the notion of product as container of information. Regardless of their contents, products themselves also have a coordinating effect and are often used as resources for coordinating the day-to-day conduct of the developers. This coordinative effect of products is in particular studied by Grinter (1995) in case of configuration management tools. Grinter shows that information about the status of the different files in a configuration management tool is an important resource for developers when coordinating their actions. For instance, in the case that she studied it was normal that a developer would delay his changes to a file if the file was already checked out by someone else. Similar scenarios have been shown by Tellioğlu and Wagner (1997) and Rogers (1993).

Using products as coordination mechanism depends on their *predictable behavior* and their ability to support *peripheral awareness* (Robinson 1993). Although a product does not put many restrictions on how it can be changed or accessed otherwise, its behavior is often predictable. Many product objects have a predictable form in that they are based on some standard notation/formalism, and can be subject to only a predefined set of operations/processes. Conventions among developers about how to use different product objects can also play an important role in increasing predictability. In the case of configuration management tools, as studied by Grinter (1995), the programmers had established local conventions that would encourage them to talk to each other before checking out files that were already checked-out by others. These conventions would only work if all the programmers adhered to them, and if the tool provided enough information about the status of each file.

Because of their existence in a shared context, products support peripheral awareness by providing continuous information about their status and the operations performed on them by others. For instance, a configuration management tool normally contains a view of all the files belonging to a product. This view continuously shows which files are checked out by whom, which files change status from unlocked to locked etc. This information is extensively used by developers as peripheral awareness, and for finding out "at a glance" who is doing what without explicitly talking to them (Grinter 1995).

This coordinative effect also applies to large and composite software products. In many cases these products will consist of thousands of product objects and relations. The relations among the product objects may result in dependencies among the developers working on these objects. In these cases, not only single product objects but also the overall *structure* of the product becomes a coordination mechanism. This large-scale coordinative effect is often a deciding factor for organizing the overall product development process. The mutual relation between work organization and the structure of the system under development is well-documented. Conway's law (Conway 1968) stated that the structure of a large system being develop by a large group will often mirror the organization of the work needed for developing it. Grinter, Herbsleb and Perry (1999) show the effect of Conway's law in different cases of product development projects, where divisions are often based on the architecture of the product. Acknowledging the crucial impact that dependencies among different parts of a software product have on the organization of the overall work, Parnas (1972) introduced the idea of "information hiding" as an attempt to effectively divide a system into sub-modules. According to Parnas, a system should be divided into sub-systems not based on external factors such as the organizational demands of the project team developing it, but based on minimizing the dependencies among the work activities needed to develop each sub-system.

The type of "implicit" coordination that is supported by products (i.e. the artifacts of the work) is different than "explicit" coordination through for instance language and speech acts (Winograd and Flores 1986). One important difference is that implicit coordination does not impose any sequencing of actions, as is normal in explicit coordination. Products support *multiplicity* in that they allow themselves to be used in diversified and often unanticipated ways (Robinson 1993). This is an important property of products because product development projects are often highly uncertain and emergent, and any prediction of action sequences should only be done at a high level of abstraction.

2.2.2 Utilizing the properties in co-located settings

In order to use the product for knowledge creation, cooperative learning and coordination, developers have to be able to continuously exchange various information about the product and its evolution. In addition, the product should be seamlessly integrated with the social interaction among the developers so that developers can freely modify and refine the product. In case of large projects, it is also necessary that the developers are able customize their interaction with the product in order to prevent information overload. E.g. a small group of developers working on a small part of a large product may need to be able to focus on that part, at the same time keeping an overview of other activities in the project. For co-located groups, the shared *physical space* is a strong facilitator of utilizing the three properties of the product. The following sections discuss this facilitator role of physical space that happens through providing *continuous*, *flexible*, and *customizable* access to the product.

Continuous access to product

Continuous access to the product is a precondition for cooperation because of the uncertain and emerging nature of product development. The process of developing new products is that of a "conversation with the situation"(Schön 1983). New and innovative solutions are tried out constantly by the developers, and their consequences are discussed and evaluated. Product development is a wicked problem, where the problem itself is never completely defined before its solution is developed (Sølvberg and Kung 1993). Information about the current status of the product, and the activities of other developers involving the product, play an important role in choosing the next step in the development, keeping the developers up-to-date, and enabling them to cooperate with each other more effectively. Continuous access to the product has *short-term* and *long-term* benefits. These benefits allow developers to use the product as a resource for cooperation.

The short-term benefit is connected to using the product as a coordination mechanism. From this perspective, any access to the shared product is potentially of importance to the developers. Information about these accesses is used by the developers as a resource for coordinating their conduct with that of others. This is the underlying reason for the importance of monitoring the product, as pointed out for instance by Vessey and Sravanapudi (1995) and Grinter (1995). Of importance for effective coordination is not only information about what changes are made to the content of the product, but also knowing through which processes these changes are made. E.g. in a project it is often of crucial importance to know who is changing a product object. Knowing that the changer is an expert in the field might give a completely new meaning to the change that is made to the product (Schmidt and Bannon 1992).

From a long-term perspective, continuous flow of information about accesses to the product is a precondition for enabling the developers to learn about the product, and to more easily externalize their knowledge. This is closely related to using the product as a boundary object, and the product being externalized knowledge. Being exposed to a continuous flow of information about the product's evolution facilitates the cooperative learning processes that are necessary for efficient cooperation (Walz et al. 1993).

Continuous access to the product requires that the product and the developers reside in a shared space. A shared space does not necessarily mean a shared physical space, but a space that allows the developers to *continuously* exchange information. It is important to emphasize the necessity of being *embodied* in a space rather than merely using the space as a communication tool. This distinction is closely related to the distinction between *intentional* and *consequential* communication (Gutwin and Greenberg 1999). Intentional communication is initiated when a person (the sender) feels the need to communicate with another person (the receiver). Verbal communication is normally intentional, but also gestures are used for intentionally conveying ideas (Tang and Leifer 1988). Consequential communication happens as a *consequence* of being in a shared space. By being embodied in a space, people and artifacts "diffuse" information. The sender of this information might not see the need for initiating communication; it is merely the receiver who "discovers" the communicational value of information that is available in the shared space. Consequential communication is important for product development because of the emergent nature of the development process. It is often difficult for a developer who modifies the product to know who will need to be informed about a particular modification.

Table 2.2: Elements of awareness information normally found in a shared workspace (from Gutwin et al. 1996)

Element	**Relevant Questions**
Identity	Who is participating in the activity?
Location	Where are they?
Activity Level	Are they active in the workspace?
	How fast are they working?
Actions	What are they doing?
	What are their current activities and tasks?
Intentions	What are they going to do?
	Where are they going to be?
Changes	What changes are they making?
	Where are changes being made?
Objects	What objects are being used?
Extents	What can they see?
Abilities	What can they do?
Sphere of Influence	Where can they have effects?
Expectations	What do they need me to do next?

Being embodied in a shared physical space guarantees that changes to the product are continuously observable by all developers. In physical space, consequential communication happens continuously and often unconsciously due to the embodiment of the participants in the space (Robertson 1996). Table 2.2 for instance shows a few types of information that are normally used as resources for consequential communication in a shared workspace (e.g. a meeting room). Such information is often called *awareness information* (Gutwin and Greenberg 1999, Dourish and Bellotti 1992). Awareness information is not only important for supporting consequential communication, but also creates *opportunities* for intentional communication.

Flexible access to product

The distinction between the formal and the informal is treated under different banners by researchers investigating the intersection between formal (often technical) constructs, e.g. computer-based information systems, and social aspects of work practices (Grudin 1994b, Orlikowski 1992, Schmidt and Bannon 1992). In particular, Robinson (1991) has argued for a double-level language property of cooperative work:

> "*In general, it can be said that any non-trivial collective activity requires effective communication that allows both ambiguity and clarity. These ideas of ambiguity and clarity can be developed as the 'cultural' and the 'formal' aspects of language as used by participants in projects and organisations. 'Computer support' is valuable insofar as it facilitates the separation and interaction between the 'formal' and the 'cultural'*". (Robinson 1991, p.43)

This double-level property of cooperative work is highly present in product development. Products are often formal constructs, while cooperation among developers can be highly infor-

mal (Kraut and Streeter 1995). In order for developers to use the product as a resource for cooperation, the product should be accessible through an interface that is flexible enough to be integrated with the social interaction among the developers. Physical space and physical proximity play a crucial role in supporting such a flexible interface. Being in a shared physical space, developers can easily externalize their product knowledge, create a common understanding of the product, and coordination their actions using the product.

The process of externalizing product knowledge happens through a dialectic process between the social interaction among the developers and the already externalized product knowledge. For instance, a large part of verbal communication in a product development project results in product knowledge, e.g. natural language documents. In fact Rittel (1972) regards the whole communicative activity as that of generating knowledge; utterances of the types *issue*, *argument*, etc. are connected in a conversation web (an IBIS, Issue-Based Information System) in order to create new knowledge. The process of creating an IBIS can be highly flexible. A set of empirical studies of small group meetings performed by Olsons and their colleagues (Olson et al. 1992, Olson et al. 1996) actually show a clear unconscious tendency by the meeting participants to follow a conversation structure similar to that of Rittel's IBIS (Rittel 1972). At the same time it is shown that conversations among the participants seamlessly switch from knowledge creation to clarification, explanation, and other types of conversation. In addition, all this conversing is mixed with non-verbal communication such as gestures and body language (Tang 1991).

Regarding the property of product as a boundary object, there exist again a large amount of social interaction with the sole aim of clarifying misunderstandings and creating a shared understanding of the product. It is difficult to predict the different ways in which a product object is perceived or used. Curtis et al. (1988) argue that the assumption that product objects produced by one group convey all the information needed by the next group using them creates strong barriers to communication and knowledge sharing within organizations:

> "*The communication needs of teams were poorly served by the written documentation since it could not provide the dialectic necessary to resolve misunderstandings about requirements or design decisions among project members. Rather, forging a common understanding of these issues required interaction.*" (Curtis et al. 1988, pp.1280–81)

In fact, there are no "perfectly understandable" products (Robinson and Bannon 1991). Too much emphasis on the formal product may result in an underscoring of informal communication, and the crucial role that this informal communication plays in improving the quality of the product (Potts and Catledge 1996, Kraut and Streeter 1995). Inter-personal communication is used not only when product objects are incomplete or ambiguous, but also as a vehicle for creativity, criticism, and negotiation.

Using the product as a coordination mechanism emphasizes the importance of flexible coordination. Instead of imposing strict regulations on the activities of developers, the product provides clues for predicting future actions (Robinson 1993). Coordination through the product emerges as a result of observing past and present interactions and predicting their consequence. This gives developers the freedom to *negotiate* the terms of coordination (Tellioğlu and Wagner 1997), i.e. it promotes "negotiated order" instead of "predefined order." In this way, each developer can align his activities to those of others. This negotiation is often done in a "low" level of interaction in order to allow developers to get along with their day-to-day activities. It therefore requires a high degree of flexibility in the social interactions among the developers (Button and Sharrock 1995). In the words of Button and Sharrock:

> "*Although there were 'formal devices' that were used to keep themselves informed of each others' work and others of their's such as regular scheduled meetings at which they would review their progress, they also, as a day-to-day feature of their work developed* ad hoc *methods for keeping others' progress in view and for making their own progress visible to others. These ways included: (i) involving themselves in the resolution of each others' problems by talking these problems over; (ii) knowing, through their engineering experience, just where what they might now do would impact upon someone else and informing that person about what they were doing; and (iii) working out between themselves 'standardised' ways of doing common tasks that would figure in each module and which otherwise could have been done in many different ways*" (1996, p.379).

Using the product as a resource for cooperation requires support for flexible and dynamic interaction with the product. Physical space allows for this dynamicity and flexibility by providing low-cost means of communication and interaction (Clark and Brennan 1991). Verbal and non-verbal utterances in physical space are easy to produce, and can be used extensively for creating knowledge, resolving misunderstandings, and negotiating order in presence of uncertainties.

Customized access to product

The last two points emphasize ease of access to product-related information. This "being aware" of the state of the product will become impractical when the size of the product grows. In many situations the developers will not have the cognitive ability to keep themselves updated about all the changes to the product. In addition, this will be useless and time-consuming. Using the product as a resource for cooperation will therefore require that each developer or group of developers can *customize* their interaction with the product in order to meet their local needs. This customization process is often supported by the resources provided by the physical space.

Development of large products consists of a number of parallel or sequential activities, or *tasks*. Fixing a bug in the database module, changing the layout of the graphical user interface, collecting feedback on a new release, etc. are all examples of tasks that have to be accomplished by groups of developers. Each such task normally creates a focus point, or a *center of interaction*, for the developer(s) responsible for performing the task[3]. Each center of interaction typically consists of a number of product objects, tools, and developers. The configuration of these objects, tools and developers changes continuously. Focusing on a task means that the group involved in the center of interaction wishes to neglect to some degree the periphery of the task. This focusing is important in order to reduce the cognitive load on the participants of the center while they are solving specific problems.

A center of interaction is thus used for defining a *boundary* for a task, in order to increase focus and preserve privacy (e.g. by not exposing intermediary work results to the outside world). At the same time, the center provides easy access to its *periphery*. This is an important property because it makes it easy to dynamically reconfigure the objects, the tools, and the people involved in the center. Such reconfigurations are often necessary because of the vague nature of the task,

[3]Centers of interaction are in this sense very similar to what Suchman (1997) calls *centers of coordination*, and Fitzpatrick, Tolone and Kaplan (1995) call *locales*. We use the term interaction in order to emphasize the importance of interacting with a product, while Suchman's emphasis is on coordination, and Fitzpatrick et al.'s on interactions in a social world.

or because of the changes in the periphery of the task. In the words of Agostini et al. (1996), the boundaries of a center of interaction provide *transparency* (by hiding unnecessary organizational information) and *visibility* (by allowing access to organizational information when needed).

Shared physical space provides the *medium* for a center of interaction. Physical space provides strong social and architectural resources for creating centers of interaction. From an architectural point of view offices, commons, "war rooms", cafeterias, hotel rooms, dedicated project rooms, etc. are used to give focus to a task (Covi, Olson, Rocco, Miller and Allie 1998). However, social conventions may play an equal role in dividing the space into centers of interaction. An example is provided by De Michelis et al. (2000). In their study of a design studio, De Michelis et al. found that the same room was used for supporting several centers of interaction. In this case, the room was divided through social conventions and physical artifacts (such as interior walls).

In addition to seamless support for configuring and reconfiguring centers of interaction, physical space has other properties that improve the cooperation in a center of interaction (Covi et al. 1998): Physical space increase *awareness* of overall work. By sitting close to each other, developers can see over each other's shoulder, and in this way be aware of what is going on in the center of interaction. Common rooms can strengthen *implicit learning* by allowing developers learn by looking at how other more experienced developers work. Common rooms support *easy transitions from individual to group work*. Being co-present during individual work provides opportunities for interruption in case of emergency. Common rooms also increase the *motivation* of the developers.

2.3 Impact of Geographical Distribution: A Case Study

With more and more product development projects distributed in different countries and continents, it becomes important to investigate the affordances of network-based collaboration technologies and to uncover the problems that geographically distributed product development teams are likely to encounter. In this section we investigate the disruptions and break-downs in the natural (face-to-face) cooperation which may arise when a project is distributed geographically. The main material for the following analysis is from our own observations of a real world project. During a two-year period from 1996 to 1998 we were involved in an EU-sponsored project for developing a large multi-media information system for knowledge sharing among the members of the European aquaculture community. The project team consisted of approximately 40 developers (a combination of zoologists, fish farmers, computer scientists, students, and project managers) who specified and developed a new computer-based product through a collaborative process mainly supported by mailing lists and the WWW. The project is called ALPHA and is used as an example to demonstrate common problems in distributed product development projects when using Internet-based collaboration technologies. The focus of this analysis is to investigate the role of product as a resource for cooperation in geographically distributed projects. In particular, we want to see how lack of access to a shared physical space affects knowledge creation, cooperative learning and coordination processes related to cooperative product development.

There are several reasons why this specific case may be useful. First, the project team and the product they developed were large enough to require a considerable amount of cooperation at the project level, while the vague nature of the product necessitated tight collaboration and negotiation on the small group level. Second, the project team was geographically distributed among four countries, so cooperation using Internet was the only option for almost all of the developers in the project. Third, the project team was intellectually distributed, involving many types of experts

with their own views of the problem domain. Fourth, the control within the project team was decentralized, with a high degree of local organizational and technical autonomy in each site.

The last point, i.e decentralized control, needs more elaboration. ALPHA is a case of a decentralized organization, with several autonomous units (universities, research centers, companies) with varying motivations and goals influencing their participation in the project. In addition, the involvement and the motivation of the people within each unit were highly varied, with some participants working full time with ALPHA-related tasks and some working only a small number of hours every week. This decentralized nature of the project made it extremely difficult for the management to enforce any standard routines or development tools across the sites. This difficulty of "controlling" the progress of the project comes in addition to the already difficult and wicked nature of product development. As also noticed by Button and Sharrock (1996), the progress in such projects is mainly guided by overall heuristics known to the developers, such as the existence of deadlines, the pressure from the management to produce results, the balance between good practice and "quick-and-dirty" fixes, etc. rather than detailed scripting of what should happen or what each developer should do. It is argued that this type of cooperation, i.e. cooperation among highly autonomous agents with varying motivations and goals, will be more and more common in the virtual organizations of the future (Faucheux 1997).

The rest of Section 2.3 is organized as follows. We will first describe the settings for the project. In Section 2.3.2 we describe some general observations from ALPHA. Section 2.3.3 provides a discussion of the observations.

2.3.1 Settings for the study

We have studied a research and development project involving partners in different European countries. The project consisted of both academic and non-academic partners. The project's overall aim was to create a pan-European network of competencies in the aquaculture domain[4]. The goal of the project was to develop an Internet-based multimedia information system for storing and accessing specialized and high quality information, to provide a communication forum for the aquaculture community, links to the international arena, as well as access to on-line courses and learning opportunities from any multimedia desktop. The official communication and documentation language was English. All but one of the developers in the project were non-native English speakers. The project was organized in a participatory manner. Several development and user groups worked closely together in order to outline requirements and test intermediate prototypes during the development. The organization of the project was in a traditional waterfall fashion, consisting of phases for planning, user needs analysis, requirements specification, design, implementation, and evaluation.

Project participants

The project team consisted of approximately 40 members, equally distributed among 4 geographical sites in 3 European countries. Four functional teams were officially recognized within the project, including members with both technical and non-technical background:

[4] Aquaculture is "*The science, art, and business of cultivating marine or freshwater food fish or shellfish, such as oysters, clams, salmon, and trout, under controlled conditions.*" (American Heritage Dictionary).

- *Scientific staff team*: Constituted of ca. 15 members. This team was in charge of defining the requirements for the target information system. The members came mainly from the aquaculture community.

- *Technical support team*: Constituted of ca. 16 members, all computer scientist or software engineers. These were computer science researchers, last year MSc. students, and administrative personnel. They worked both as developers of the target information system, and as support staff for the cooperation infrastructure used by the project team.

- *Daily management team*: Constituted of 8 members. This group was mainly responsible for the daily activities of the project, such as meeting release dates and supervising development work.

- *Project management board*: Constituted of 14 members. These were responsible for contact with sponsors, for overall management of the project, and for making sure that the deadlines were met. They also played an essential role in finding new users and future customers for the information system being developed.

All the teams contained members from different geographic sites. This means that everybody involved in the project had to face the problem of geographical distribution. First site was the main management site, which initiated the project and was the contact site for the sponsors. Second site was the main development site, located in the computer department where the author resides. Third site was the main scientific site, one of the major European universities in the area of aquaculture. The fourth site was involved in the requirements definition and the overall evaluation of the project. Both first and third sites had available technical staff dedicated to the project, while the second site had members from the aquaculture community and members in the management site. There was a high amount of turnover among the members, especially in the technical support team. This was partly due to the involvement of students in the programming activities.

The project team was distributed not only geographically, but also with respect to skills and roles of the members. Most of the members from the aquaculture community had very low knowledge of computers. They could use e-mail and WWW, while some of them had for instance difficulties setting up their e-mail accounts, using e-mail lists, editing Web pages, and setting up a desktop video conference session. Moreover, the technical infrastructure at their disposal was minimal. At the other end, members with a good background in computer science had a low-level competency in aquaculture.

Cooperation infrastructure

The project management board had regular face-to-face meetings every second month. Technical and scientific groups had face-to-face meetings and workshops in connection with each phase of the project. These meetings involved only a few of the developers. All other communication was done using Internet-based communication tools. There were no advanced groupware tools used during the project. Mailing lists and WWW were the only widely available technology. The prototype that was developed and used during the project (the “end-product” of the project) was also WWW-based. Occasionally video conference tools were used, but mainly for demonstration purposes.

Electronic mail and mailing lists– As is the case in many distributed groups, mailing lists quickly became the standard communication medium. In order to reduce the effort of using email a list server was set up in an early stage, and several mailing lists were established. These lists were created by the management in an ad hoc manner according to the needs of each phase. At the end of the project there were 13 active lists being used by the members for different purposes. The mailing lists were intended to be used as discussion fora for developers, managers, and user representatives. Table 2.3 shows some of the lists that are relevant for the study.

Table 2.3: Some of the mailing lists used by ALPHA.

Mailing list name	**Users**	**No.of members**	**No.of msgs.**	**Total msgs. per member**	**No.of threads**	**No.of active months**
All	all project members	41	61	1.5	48	19
Technical	discussions about technology to be used, related and similar systems, products etc.	17	35	2.1	24	9
Scientific	user representatives and people with knowledge about aquaculture	15	1	0.01	1	N/A
Daily management	people engaged in the daily administration of the project	15	428	28.5	209	34
Project management board	project managers involved in the overall management of the project	6	5	0.8	4	5
External actors	people not involved in the project directly, but who were interested in being informed about the project and its results	N/A	24	N/A	11	2
Video conferencing	the members of an interest group in the area of video-conferencing	20	87	4.4	55	14
WP5: Specification phase	people involved in work package 5	5	5	1	5	1
WP6: Design phase	people involved in work package 6	10	133	13.3	73	5

Continued on next page

Continued from previous page

Mailing list name	Users	No.of members	No.of msgs.	Total msgs. per member	No.of threads	No.of active months
WP678: Design/ testing/ evaluation phases	people involved in work packages 6, 7, 8	24	505	21.0	218	12
User interface	user interface design team	12	202	16.8	90	9
Development	people at the main technical development site, and programmers from other sites	11	296	26.9	109	8

The World Wide Web– Throughout the project the developers used WWW for sharing project information, which was organized in product objects. Moreover, the Web was the platform used for developing prototypes of the product. The main Web server was maintained by one site (management site), but each of the other sites had their own servers with additional information and local installations of the various prototypes. The mailing lists were also integrated with the main Web server in that all the messages were made available on the server. It was, however, not possible to send email using the Web interface. Table 2.4 shows some of the product objects used by different people within the project.

Table 2.4: Some of the product objects created and used in ALPHA.

Product object type	Product object contents	Intended users	Frequency of use	Duration of use
Project Web site	practical information about the project, contact information, entry point to the online project archive and developed prototypes	all project members, plus external visitors wanting to learn about the project	high	all project lifetime
ICE	Web-based prototype developed during the project. Tested and commented on by the developers. Also used a period as a cooperation tool among the developers	all project members	high	last 20 months

Continued on next page

Continued from previous page

Product object type	Product object contents	Intended users	Frequency of use	Duration of use
Scientific contents	scientific content files uploaded to the online knowledge base (ICE). Used as test data to test ICE	aquaculture scientists	medium	mostly in the later phases of the project
Mailing list archives	online Web-based list archives containing messages sent to the mailing lists	all project members	N/A	all project lifetime
Project documents	natural language documents and notes. Used to documenting requirements, progress, achievements, etc.	project members and external reviewers	medium	varying
User interface mock-ups	mainly Shockwave-based demonstrations of user interface design. Used for evaluating the designs before implementation	all project members	high	mostly in the later phases of the project
Visual diagrams	used for documenting the design and architecture of the prototypes. Mainly used by technical staff and as illustrations in project documents	technical staff	low	all project lifetime
Source code	Java, Perl, and C source code files. Mainly used locally by one site, but sometimes customized by other sites.	technical staff	high	mostly in the later phases of the project
Bug report form	A Web-based form with specific fields used for registering bugs and issues related to the prototypes. The users would fill in the form and push a send button	all project members	low	short

The product

ALPHA was set out to develop a new product based on a set of high-level requirements. The product consisted of a large informal part during all project lifetime. During the first 8-10 months the product consisted mainly of various documents, meeting minutes, natural language specifications of requirements, etc. Short time after the main design document was produced (September 1996, written in natural language) the project team organized itself according to the architecture of the designed system. Also in the period with a high level of technical development, a large part of the product remained informal and consisted of natural language documents, notes, sketches, drawings, etc. ALPHA produced various prototypes of a specific end-product called ICE (see bel-

low). ICE was programmed in Perl, C and Java programming languages. Source code belonging to these prototypes, and the executable prototypes themselves were the only formal part of the product being developed.

An overview of the the end-product of the project is shown in Figure 2.5. All parts of this end-product were developed in form of executable prototypes. An integration into one system however did not happen during the project. The connecting link was a dynamic information exchange engine called ICE (Internet Collaboration Environment, see also Appendix A). ICE was used for sharing different knowledge objects (Olsrød and Isaksen 1996). It allowed the aquaculture community to share their knowledge by uploading different types of content such as research results, courses, diagnoses of diseases, trends in fish prices, etc. *Content providers* were from academia, aquaculture organizations, and individual fish farmers. They were given access to a "back-office" interface that would allow them to upload different types of *knowledge objects* into their *shared workspaces*. Some of these objects were single files (such as images illustrating the symptoms of different diseases), while others were composite objects (such as aquaculture-related courses with modules and course material). Once these objects were uploaded by the content providers, they were available to *content consumers* through a "front-page." The front page consisted of a number of services such as search, browsing, and collaboration tools. The whole system was available through a WWW interface.

Although the project worked most of the time in a highly integrated manner, different sites were made responsible for different parts of the end-product. Our local site was responsible for ICE, its basic interfaces to content providers and consumers, and some of the front-page tools. A second site was responsible for developing the interface towards fish farmers. This interface would allow them to upload and exchange information about their daily catch, fish prices, and other market information. A third site was responsible for developing a telecommunication infrastructure including video conference through Internet and ATM networks. There was also a group of user interface designers who worked with the Web-based interface towards content consumers.

The method of the study

The case study presented here reports on a period of approximately two years, from April 1996 to January 1998. This period coincides with the development and testing of different versions of ICE by the project team. The study is based on data collected during the project and analyzed afterward. This data includes project deliverables, meeting notes, email messages sent to the various mailing lists, and other product objects developed by the project team. Moreover, the author participated in the project as a senior technical staff person from August 1996 to the end of the project, with heavy involvement in the development of various prototypes in cooperation with other project members. This involvement provides a deep insight into the technical aspects of the project, and hopefully compensates the possibly negative effect of the lack of interviews with the developers.

The analysis of the data has been based on the framework used in Section 2.2. The analysis is concerned with the project team together with the product they developed. We have mainly focused on cooperation problems caused by geographical distribution and the support technology used by ALPHA. Many contextual factors such as reward systems of the local organizations, motivations and goals of the involved people, pressures on the project from the sponsoring organization, etc. are not considered because data about these factors have not been available. In addition, the analyzed communication is limited to the mailing lists and the few meetings and

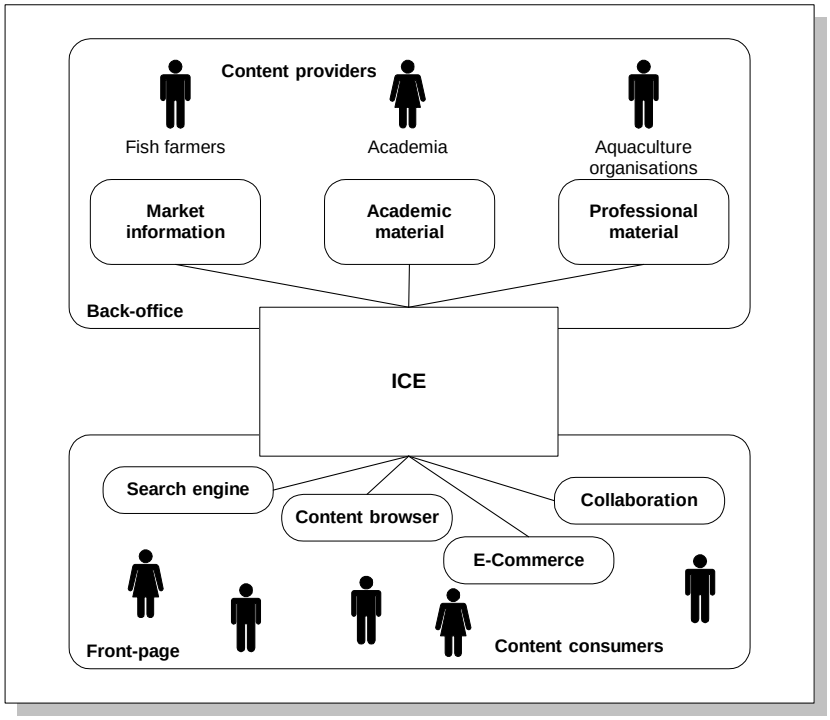


Figure 2.5: An overview of the end-product of ALPHA. ICE worked as a link among the different parts.

workshops held during the project. In particular, the communication among the developers and their communities of practice (e.g. communication among Java programmers in ALPHA with other Java programmers) is not accessible to the study. The absence of these factors may reduce the generality of our study. However, there is a growing body of empirical studies of distributed team performance that supports many of our findings. We occasionally refer to these studies.

2.3.2 Observations

Four kinds of observations are presented. First, we see how the developers cooperated implicitly through exchange of product-related information. Second, we will see how the explicit communication through mailing lists proceeded. Third, we will see how specific tasks involving small groups of developers were performed. Fourth, we will see how the diversity of skills and the decentralized control in the project influenced the cooperation across geographical sites.

Sharing product-related information

A critical advantage of ALPHA was recognized by the project management to be its diversity and multi-disciplinarity. As a result of the varied backgrounds of the involved developers, rang-

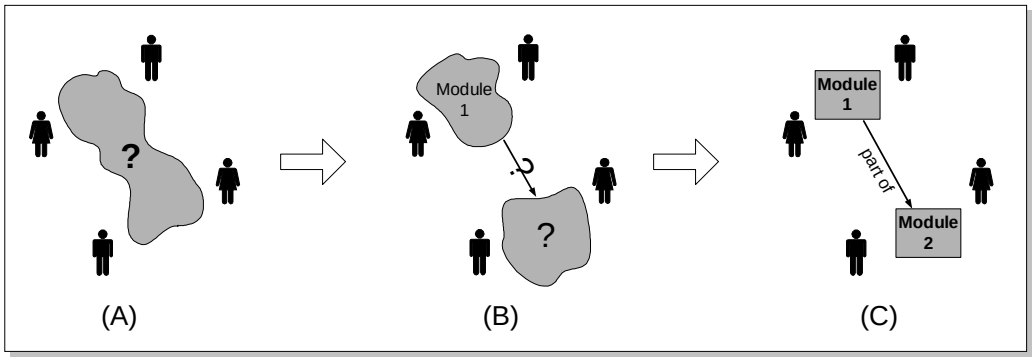


Figure 2.6: The creation of product structure in ALPHA.

ing from computer scientists and software engineers to aquaculture scientists, the product itself worked very much as a unifying link among the developers. Everybody contributed to the product from his standpoint by bringing his unique knowledge to the project. In addition to this diversity in the backgrounds, the fact that most of the developers did not know each other well, and that many worked only a part of their time with ALPHA, further amplified the value of the product as a vehicle for communication and cooperation.

Figure 2.6 shows how the structure of the product emerged over time. Initially all the product was one single artifact (the project plan) with a main goal, that of supporting knowledge creation in the aquaculture community (Figure 2.6.A). We saw a clear mapping between the structure of the product being developed and the way the project was gradually organized and re-organized (Figure 2.6.B). This structure was not clear in the beginning, but emerged as a result of a number of face-to-face meetings and workshops, as well as long discussions in the mailing lists. The structure was also a close mapping of the architecture of the end-product that was gradually being implemented. Once prototypes of the end-product were developed, they strongly guided the organization of the project. When the different components of the end-product were defined more or less clearly, each site was made responsible for developing one or several components (Figure 2.6.C).

Most product objects were informal, natural language documents. This had the advantage of giving everybody the opportunity to get involved in the development, e.g. by reading the requirements document. In addition, having an informal product provide room for creativity, discussions, and critique. On the other hand, one major disadvantage of not having a formal product from a cooperation point of view turned out to be the difficulty of accessing product information. E.g. it was difficult to know which objects belonged to the product, and consequently which objects should be shared with other sites. The majority of the developers were not trained software engineers, and it would have been too costly to educate them in structured methods. Moreover, training in formal or structured methods was not a goal of the project.

One of the major problems facing ALPHA turned out to be the difficulty of exchanging dynamic product-related information. The project Web sites were used for sharing different types of product objects. However there were a number of problems related to the process and the technology of sharing. The control of these Web sites was highly centralized, i.e. often one person

was responsible for maintaining each site. The sites were mainly used for representing the project to the outside world, and not for supporting cooperation internally. They therefore containing information about the results of the project (e.g. deliverables) and little information about the day-to-day activities of the developers. In addition, the sites were quite static; once a product object made it to a Web site, it was difficult to change it.

As a result, information exchange suffered from a constant delay, and the developers normally knew little about each others' *ongoing* product development efforts. This delay applied both to informal product objects, and to the different prototypes of the end-product that were being developed by the different sites. In the following message taken from a mailing list we can clearly see the long term continuous lack of information about remote parts of the shared product:

> "*If I am well informed, [site A] is developing an image archiving system based on Paradox. Unfortunately I didn't have the occasion to get a demonstration of this system so far, so I have no clear view of what is possible and what is not. Can anyone inform me? Maybe I can get remote access to this database under construction?*"

The author of this message has been partly informed about a long term development activity in a remote site involving a part of the product. He has little information about this development, let alone the detailed functions or interfaces of the developed part.

The example above demonstrates the kind of problems that arose as a result of not having access to remote product parts. However, merely making available product parts on the Internet was not enough as it is shown in this message sent to a mailing list:

> "*I took a look at [prototype X] but everything looked confusing to me without documentation, so I can not (yet?) share [Person A]'s enthousiasm. I want to know how this system works and what we in [Site A] can do with it. I need more information before I can say if the proposed system and time schedule is acceptable for [Site A] or not. [Person B] or [Person C], please send me detailed instructions now for evaluation, use, implementation... Or is there on-line documentation somewhere?*"

As this message shows, even in those occasions when product-related information was accessible to everybody, the information was discontinuous, incomplete and static. Incomplete and discontinuous information made it difficult to learn about the remote parts of the product while they were being developed. Information about the ongoing activities in the remote sites was normally many days old and often outdated. In addition, product-related information was often withheld until it was in a "presentable" form. Intermediate results and artifacts were not shared until they conformed to some level of quality.

Besides the difficulty of knowing and learning about what others were doing, coordination of work across sites turned out to be a problem for ALPHA. The symptoms of this lack of coordination were duplication of efforts in multiple sites, and problems during the integration of the prototypes. Duplication of work was extreme in at least one case, when two different sites developed each their own version of a large component. The existence of these two versions was not noticed for a long time. When they were finally discovered, the project management decided to keep them as two alternative forms of the same service, one for supporting older versions of WWW browsers and one designed for newer browsers. This decision was not based on the stated requirements of the project, and was mainly taken as a compromise for not choosing one site's version above the other. As a result of this and similar problems, the prototypes never passed an

integration phase, and the end-product as it was design was never finished. The only integration was realized in the project Web site, where different Web pages gave access to the different prototypes. This was clearly against the requirements of the project, where a uniform representation of contents was a central requirement for making the system user friendly.

Although most product objects were natural language documents, the fact that they were used by developers with diverse backgrounds required a lot of communication related to explaining and negotiating shared understandings. This type of communication was time-consuming and difficult to follow in many cases. An interesting example was a classification scheme that was developed by the aquaculture scientists for classifying the contents of ICE (the contents were mainly aquaculture-related material such as information about different species, diseases, professional journals and other material). This classification scheme had different meanings for the computer scientists and the aquaculture scientists. The computer scientists were interested in the technical feasibility of implementing the scheme into the prototype in form of a menu and search functionality. The aquaculture scientists were more interested in having a consistent classification from an aquacultural point of view (apparently without any concern about its technical feasibility!). The process of arriving at a solution required a surprisingly large number of email messages, and the discussion continued for a number of months.

Breakdowns in learning and coordination did not exist internally within each local site, where the members were normally in the same building. In fact, two of the prototypes that were developed successfully and used also after the project was over were each defined and developed almost completely by one single site. Each of these two sites had access to designers, programmers, and end-users under the same roof or at least within the same city. Although easy access to product objects and other developers was of crucial importance, we believe that the reason why these local sites functioned so well is also related to the *kind* of product objects they used, and the *process* through which these objects were used in co-located cooperation.

First, product objects in a variety of physical and digital forms were used to support co-located groups. A quick look at the desktop of a programmer in our local site would reveal a vast range of physical objects that were used as product objects, and served as strong facilitators of cooperation. Moreover, our group meetings made use of specialized artifacts, e.g. blackboards, for collaboration. Across the geographically distributed sites there was a strongly reduced range of product objects available to the developers. In addition, these product objects were quite limited in their support for collaboration and unanticipated use (Robinson 1993) because they were not explicitly designed for supporting cooperation among developers. They were merely information containers.

Second, in co-located groups product objects made their way into the product in a gradual and evolutionary manner. We are all familiar with diagrams that are drawn on a napkin during a lunch break, to become an important part of the product later. Across the sites, product objects become part of the product only when they were in an acceptable (meaning almost final) form. A "virtual napkin" did not exist in ALPHA, even though physical napkins were used very often by at least our local site. Waiting for an almost final version of a product object meant at the same time an inevitable absence of awareness of the existence of such an object in the meanwhile. In the message quoted above, prototype X is an example of such a product object that is just put on the Internet (in an operational form). The prototype has been under development for some time, but details of the evolving artifact has not been available before the final release[5].

[5]What is not shown in this message is that while prototype X was being developed the author of the message was

Communication about the product

One would expect that the lack of access to dynamic information about the product would be compensated by explicit communication. In ALPHA, the main means of explicit communication were the different mailing lists. The frequency of use of each mailing list depended on the purpose of the list and on the number of its members, but in average approximately 40 message from each developer were sent to the lists during a period of 34 months. This means that each developer communicated approximately one message every month. Even if we assume that each developer also sent the same number of ALPHA-related messages personally to other developers[6], the amount of explicit communication was still much lower than in co-located settings. This issue was fully recognized by the management, and face-to-face workshops were utilized in critical periods during the project in order to cope with the lack of communication.

In addition to the low bandwidth, there were a number of problems connected to using email in the long run (we believe these problems apply also to audio and video communication). First, although email was used extensively for resolving technical issues, explaining things to others, and informing about specific events, email was not used as a regular information channel for informing about ongoing local activities. Email communication provided in this way a discontinuous picture of the development process in the long run. Second, email lists did not manage to create the proper context for the different tasks that existed at any time during the project. It was difficult in many cases to keep an overview of what issues were discussed.

However, there were a number of situations where email communication turned out to be invaluable:

- Providing feedback: Mailing lists played an essential role in collecting feedback on different proposals and prototypes, allowing for the incorporation of different perspectives.

- Access to experts: Mailing lists provided easy access to different kinds of tacit knowledge. This would have been extremely difficult without mailing lists due to the geographical distribution of the participants.

- Resolving design issues: Mailing lists were extensively used for sharing issues and problems, discussing and solving them, and making decisions among alternative designs. However, this worked as long as the number of the ongoing issues was low (see next section on support for tasks).

- Accessing the product: A large number of email messages sent to the lists contained links to product objects, or contained the objects as attachments. Including WWW links was the main method of informing about new releases of prototypes.

One major communication topic in the mailing lists turned out to be user interface mock-ups and prototypes that were developed and made available on the WWW. These mock-ups and prototypes had two important properties: they were *intermediate* and *interactive*. This means that they could be tested and criticized by all the developers, as opposed to other product objects that could only be accessed and "downloaded" (e.g. project deliverables and meeting minutes). The user

developing a duplicate prototype.

[6]The author, being active participant during the last 24 months of the project, received approximately 400 personal email messages related to ALPHA.

interface mock-ups were in form of Shockwave[7] demonstrations that could be run from a WWW browser. Also the prototypes were web-based and easily accessible by all the participants. Communication in the mailing lists increased dramatically whenever new versions of these product objects were released. This might indicate that artifacts, in particular intermediate and interactive product objects, should be used more frequently as accelerators of communication in distributed settings.

Cooperation in performance of specific tasks

Despite the low level of explicit communication as discussed above, it was often noticed that developers believed they received far too many ALPHA-related email messages than they needed. This was in particular the case when resolving detailed design issues. These issues were often raised in the proper mailing list. Arguments and opinions were sent as replies, with a possible resolution of the issue following the discussion. The process of resolving issues related to the product constituted the main type of cooperative tasks that the developers were involved in. The problem with performing these tasks was the difficulty of creating a center of interaction for each task. The lack of these centers of interaction resulted in everybody being involved in all the ongoing tasks, with the resulting information overload.

Information overload was most visible in the way the mailing lists were used. Each issue that was raised in a mailing list normally gave rise to related sub-issues, and the developers discussed these sub-issues while still under the heading of the more general initial issue, which normally was not anymore the proper heading for the sub-issues. A typical example of this kind occurred when a member of the technical staff posted a message to one of the mixed mailing lists, sketching a solution for the database sub-system. This message gave rise to several discussion threads about access control, concurrency control, version control, etc. Some of these discussions continued for weeks under the heading of the initial message, e.g. “database functionality.”

This became more complicated when the number of threads increased to over 2-3. When having too many concurrent issues, most of the issues were never handled or resolved, simply because they were posted to the “wrong thread” in the mailing list. In addition, retrieving information about a specific issue was laborious because of the lack of distinction between the discussion threads.

Due to the seriousness of this problem, one of the members of the technical staff made a simple application for “submitting” issues. This application was very similar to a bug report system. A Web-based form containing a number of fields was used for submitting issues. Information about an issue, e.g. explanation, related product objects, etc., could be filled in and submitted to an issue database. All the submitted issues were available from another Web page, along with information about each. This application was an attempt to support the creation of centers of interaction. Despite the need for such a solution, the application was never used.

Another way of creating centers of interaction, which was much more successful than the issue submission system, turned out to be through shared workspaces in ICE. ICE was initially developed as a part of the end-product. ICE allowed developers to create shared workspaces and insert different objects (e.g. documents and images) in these workspaces. All this was done through an intuitive form-based Web interface. ICE did not integrate any form of explicit communication inside a workspace, so explicit communication had to be supported through mailing lists. ICE was used mainly for uploading different content files, and for making this content available on a sepa-

[7] Shockwave is a program for viewing computer animations.

rate Web page through a menu. After ICE was tested by all the developers, the developers started to use it also for sharing product objects. A number of developers created shared workspaces for different tasks that they were working on, and used these workspaces for exchanging updated product objects with other developers.

We believe this way of creating centers of interaction was more successful because ICE provided more freedom of action. As opposed to the issue submission application, ICE allowed the developers to create empty workspaces, add or remove objects as they wanted, and leave notes (text files) to each other. However, ICE had a number of limitations. For instance, it did not actively inform the developers about the changes to a workspace, so developers often forgot to check for new versions of product objects. In addition, ICE did not support dependencies among tasks or product objects.

Decentralized control and diversity of skills

The technological infrastructure at the different sites were often different and sometimes incompatible with each other. Despite the fact that the project was involved in the development of an Internet-based product, episodes of technical incompatibility were present during the project. One main problem turned out to be the existence of different versions of Linux operating system. The main technical site and two of other sites decided to use Linux for installing and testing the prototypes. However, they each chose a different version of Linux. These were decisions that were taken locally and were not challenged by the project management. The result was a considerable amount of overhead work connected to maintaining at least two parallel versions for each prototype.

Demand for simplicity of collaboration technology was high in ALPHA. Due to the highly variable level of computer- and network-related knowledge, the project as a whole was highly sensitive to the type of technology used for sharing and communication. The following excerpt from an introductory email message sent by the mail server administrator demonstrates well the assumptions one could make regarding the participants' prior knowledge of network tools. The message shows clearly that even mailing lists were unknown technology for some of the participants:

> "*To facilitate the use of e-mail a list server will be set up. What is the ease of using a list server? Within e-mail software address books can be created, containing the e-mail addresses of groups, to which e-mail is sent frequently. The disadvantage of local address books is that they might differ from books used at other sites. A list server is an 'address book tool' at network level. It contains one ore more lists with e-mail addresses. If people want to mail to the members of one of the lists, they can simply mail to the name of that list. . .*"

And a message sent by the same person one month later:

> "*Dear all,*
> *With great pleasure I see that the lists of our server are used frequently! A listserver is a tool to ease communication, and as far as I can overlook it does. However I have a small remark. If you use the reply or answer function of your E-mail software, after having received a message, not only the sender will receive your answer but all the members on the used list. This might not be your intention. So if you want to answer the sender personally, compose a new message!*"

Complicated technology for communicating ideas and results was not always successful. An example occurred when the user interface design team started using animations for visualizing their screen shots. These animations required the installation of new "helper" software on developers' computers. Since some of the developers did not have access to necessary technical assistance for installing these helper programs, they could not view the screen shots that were important for the development of the prototypes. This resulted in a breakdown in the feedback cycle, and increased dissatisfaction among developers with low technical knowledge.

2.3.3 Discussion

In Section 2.2 we saw how product played an important role in supporting cooperation in co-located groups. We also saw which aspects of physical space were important for allowing this to happen. Physical space makes it easy to have continuous, flexible and customizable access to the product, which allows the developers to externalize their product knowledge, to use the product as boundary object, and to coordinate their day-to-day activities with the help of the product.

Our observations show that in ALPHA the product played an even more important role in supporting cooperation. The product was central in ALPHA because of many reasons. First, the developers belonged to different communities of practice, and the product was mainly the only thing connecting these communities. Second, many of the developers worked only part time in ALPHA, and in many cases the only information they were interested in regarding ALPHA was the status of the product (i.e. the result of the project). Third, there were no central routines or common norms or development cultures uniting the different sites. In addition, geographical distribution reduced the social relations among the developers compared to what we normally see in co-located groups. We believe these conditions are not specific to ALPHA and can be observed in many cross-organizational distributed development projects. Access to a shared product, i.e. a shared context, becomes crucial in such settings also because the process of developing new products is highly uncertain and emergent, and one cannot predefine the details of the activities of the developers in advance.

Despite its importance, it turned out to be difficult to use the product as a resource for cooperation. Not unexpectedly, many of the problems can be related to the lack of a shared physical space among the developers. Not being embodied in a physical shared space the developers could not have continuous access to the product, they could not interact with the product in a flexible way, and they could not customize their interaction with the product in an effective manner.

Lack of continuous access to the product

The project suffered from a *long term lack of visibility of work*. Developers did not know what was happening in the remote sites. They did not have continuous access to the remote sites of the product, and they could not use explicit communication to keep informed about the product in the long run. This resulted in break-downs in knowledge creation, cooperative learning, and coordination.

The informal nature of the product, and the lack of proper technological support resulted in many product objects not being shared across the sites. The wide range of product objects that were shared and used for cooperation in the individual sites were never made available online, or in the best case were shared only when they were in a final form. The additional division of the product into parts made it further difficult to know about the remote parts of the product.

Explicit communication in ALPHA was directly related to and shaped by the product. The product created the context for communication. The possibility of access to the product strongly affected the amount of explicit communication, and vice versa. People did not communicate about what they did not know about. Explicit communication through mailing lists alone was not suited for sustaining a long term awareness of the whole product. A developer who changed a product object or created a new product object didn't know who was interested in knowing about the changes. In addition, explicitly providing information about such changes was overhead work with no direct benefit for the informant.

Co-located groups do not suffer from these problems of invisibility of work because they are all the time embodied in a physical space. Also in ALPHA physically co-located teams cooperated much better than distributed ones, even if these co-located teams contained members from different communities of practice and different institutions. The lack of a physical space, and its consequences for the visibility of work, suggests that *explicit efforts* in form of technical or other solutions are needed for setting up a *virtual shared space* that can simulate some of the properties of physical space. In particular, support for consequential and opportunistic communication are crucial. Such a virtual shared space should not only give continuous access to the shared product. It should *actively* inform the right developers about accesses to the product, and in this way create *opportunities* for communication and cooperation. The existence of such a virtual shared space can help the developers to utilize the product as a resource for cooperation, and not just as decontextualized information. This support can be provided by employing more active information delivery mechanisms, in particular for delivery of product-related information.

Difficulty of flexible interaction with the product

The second group of problems arose as a result of the difficulty of interacting with the product. Product objects that were made available through the project Web server were not used in cooperation because they were not easily modifiable. Also, the range of shared product objects was very limited compared to what developers in co-located groups shared and used as resources for cooperation. Product objects were very often held back until they were in an almost final form. As a result, the developers cooperating across geographical sites had access to only a limited range of product objects (e.g. natural language documents) and they were often not in the position of changing or annotating these objects. The interaction between the formal and informal was hampered considerable with respect to what is normal in physically co-located groups.

There were a number of occasions where product objects were used actively as resources for cooperation. One case was the use of prototypes and user-interface mock-ups. These were not modifiable by all the developers, but could be tested and commented on. They were *intermediate* product objects. Another case was the use of ICE and its shared workspaces for cooperation. ICE provided a tailorable interface through its shared workspaces. The workspaces in ICE were *modifiable* by the developers. The developers could create any number of nested workspaces, and add any form of objects (although from a predefined set of object types) into these workspaces. These observations suggest that the traditional "centralized control" model supported by WWW is not suited for supporting flexible sharing of information. Physical space supports a much higher level of flexibility in use of artifacts. Any artifact that is considered useful can be inserted into the shared space. In addition, the transfer of artifacts from outside the product into the product is seamless and can happen gradually. Computer support for sharing arbitrary objects is necessary. In addition, the transition from informal to formal and from non-product artifacts to product objects

should be supported in a seamless way.

Difficulty of customized interaction with the product

We observed problems in resolving misunderstandings and creating a common and agreed-upon picture of the product. There was a need to use the product as a boundary object because explicit communication through mailing lists was not enough for creating the needed common understanding. The product's role as a mediator of meaning across communities of practice was increased, but the product was not able to live up to the expectations. Ideally the product should have supported a shared view across the communities, and at the same time incorporated different types of details for each of the communities. Mixed mailing lists were used occasionally for explaining things from different perspectives, but this was limited and time-consuming. In addition, the rapid pace of changes in the product made it even more difficult to maintain a shared understanding. The lack of effective communication channels in ALPHA increased the value of the product as a boundary object. This suggests that the product and its representation to the different communities of practice should be enhanced to support both a shared view, and a number of specialized and locally customized views.

Moreover, even if only a limited part of the product was shared, and even if the amount of explicit communication was low, the developers had problems dealing with this shared information. This is because the available technology did not support them in creating *boundaries* for the tasks they were involved in. Sharing has to be balanced in order to allow both the project as a whole and the individual developers to proceed with their development activities. A shared space where all the interactions happen has to be balanced with a number of centers of interaction where the specific interactions can happen uninterrupted. Simply increasing the amount of sharing, by for instance putting everything in a shared repository, would have resulted in confusion and increased information overload.

These observations suggests that the support system should provide a *customizable interface* to the shared product. This interface would allow the developers to create arbitrary centers of interaction for limiting their interaction with the product. The interface should allow the developers to customize or annotate their views of the product in order to support local understandings of the product, and at the same time be able to share the product and its meaning across different communities of practice.

Advantages of using WWW and mailing lists

Despite the problems that occurred during the project, mailing lists and WWW showed some strong aspects that we believe should be noticed and followed by developers of more specialized support technologies. This combination of technologies showed to be highly open and accessible. Virtually all the developers were able to use the mailing lists and the product objects that were made accessible through WWW. This is a privilege that many kinds of advanced support technologies have not enjoyed, i.e. being accessible from every desktop through a couple of clicks. In addition, mailing lists and WWW are highly flexible. By flexible we mean they do not impose any explicit or implicit cooperation policy on the developers. Email can be used for different kinds of tasks such as brainstorm, planning, coordinating, discussing, etc. The users decide the usage. In the same way, WWW and related technologies can be used to structure an information space in many different ways. This flexibility can be an advantage when the tasks are uncertain and require

flexibility, such as upstream product development activities. On the other hand flexibility requires efforts from the part of the users in order to structure and articulate their work.

Distributed projects often cross the boundaries of various organizations, or involve virtual organizations. In these cases it is difficult to centrally decide upon global processes, standards, and tools. Moreover, it often becomes unrealistic to look for advanced and technically demanding collaboration infrastructure. The participation of people from different organizations (or outside any organization) implies that it is highly probable that different local development tools are used. Though advanced technical solutions could be found locally, in many distributed settings what is needed and usable is a *collaboration infrastructure* that brings people together and coexists with existing heterogeneous tools.

2.4 Requirements for Support Environments

The analyses in the previous sections will be used in this section to derive a set of requirements for product development environments. These requirements are in particular important for supporting geographically distributed projects, but are also useful in co-located projects. The requirements are tailored in a way that they can also be used as basis for extending conventional CASE tools with cooperation functionality. An overview of these requirements is shown in Table 2.5.

In general, product development environments should support sharing of relevant information about the product (REQ.1 in Table 2.5). Note that this sharing might be different from sharing through a central repository. Here, the focus is on sharing information that is of importance for day-to-day *cooperation* among the developers. This includes details about the product that are understood and used by a group of developers, but also information about the activities of the developers involving the shared product. A product development environment should enable sharing of information that is important for using the product as a resource for cooperation, i.e. as an enabler of knowledge creation, cooperative learning and common understanding, and coordination. It is also important that this sharing is continuous, which might mean that the support should be integrated with the tools that the developers use in their daily work.

Another basic requirement for product development environments is support for *flexible access to the product* (REQ.2). Flexible access requires that the developers, regardless of their geographical location, can easily update the product, and that they can access different types of information about the product in an ad hoc manner based on their emerging information needs.

From a knowledge externalization perspective a product development environment should allow the developers to share any type of product object and relation (REQ.3 and REQ.4). In addition, product development environments should support *incremental refinement* of the product through transfer of knowledge from developers and their social interaction into the product and back (REQ.5). This transfer should be gradual and seamless. The shared product space should not force developers to only share “perfect” product objects and relations, but allow them to gradually refine intermediate objects and relations.

A product development environment should support the viewing of the same product from different perspectives. A product should be represented as a boundary object (REQ.6). It should support customized local views to be used by different communities of practice, and at the same time provide a view that is shared among all the developers in a development project.

A product development environment should support *active delivery* of information, without requiring each developer to explicitly browse or query the environment for information (REQ.7).

In situations where developers are physically co-located, they normally get "hinted" about recent updates to the shared product through opportunistic interaction, e.g. talking in the hallways during chance encounters. The lack of these opportunistic interactions in geographically distributed teams implies that the development environment should take a more active role in "hinting" the developers about changes. They should provide more *awareness information* to the distributed developers. What awareness information is necessary is hard to guess for a computer system. Therefore, the information that is provided actively should be tailorable by each developer or group of developers (REQ.8).

In order to be able to use sharing as a means for cooperative learning and coordination, information about accesses to the shared product should also include information about who did the changes. This requirement is important for supporting cooperative learning, but becomes crucial in case of coordinating the day-to-day work. In particular, one needs to have access to the person in order to repair the immediate breakdowns in cooperation through direct communication with the person. In general, every piece of information in the shared product space should be traceable to a specific developer or a group of developers (REQ.9).

A user or a group of users will normally work and cooperate within a center of interaction and through the resources provided by the center of interaction. A center of interaction may include people, artifacts, tools for modifying the artifacts, different media for supporting the cooperation among the members, etc. Emergence, external visibility and fluid boundaries are important properties of a center of interaction. People may enter and leave a center, artifacts may be added and removed, and different tools and media might be employed. A product development environment should provide mechanisms for creating center of interactions that can bring together developers, objects and tools regardless of geographical locations (REQ.10). These centers of interaction should be easy to create and modify, in such a way that developers can create and use them as cooperation emerges (REQ.11). They should provide a focused area for cooperation, and at the same time not isolate the developers from the activities outside the center, i.e. they should have fluid and emergent boundaries (REQ.12). In addition, centers of interaction should provide dynamic interaction mechanisms for the involved people (REQ.13). Cooperation in centers of interaction, because of being focused, often has a higher pace and requires richer interactions among the members.

An important part of a center of interaction is the parts of the product that are made available within it. Interaction among the members of the center of interaction are important from a product development perspective because they will eventually modify these parts. Therefore the creation and the usage of a center of interaction can be seen as a step in the process of refining the product. A center of interaction should not only provide mechanisms to allow its members to easily modify the contents, but also to allow the creation of local annotations and customization of these contents to the local context (REQ.14).

Our experience from ALPHA shows that one major advantage of using mailing lists and WWW is that these tools are already integrated into the desktop of many developers. The developers could get involved in the ongoing cooperation easily because the tools they used to cooperate (i.e. email clients and Web browsers) were already integrated with their daily activities. Their involvement in ALPHA did not require much explicit effort, or large cognitive and technical shifts. Therefore, an important part of our requirements is concerned with increasing the ease of use and technical openness of development environments. Our experience is that it is very hard to ask users to abandon their favorite tools in favor of a new tool or environment with

similar functionality. We suspect this will be even harder in case of highly specialized tools such as CASE tools. Multiple interfaces to the system are important in order to allow different groups of users access the functionality of the product development environment through their familiar tools (REQ.15).

Tailored functionality (REQ.16) is needed to allow for *incremental integration*. Any product development environment will include a host of functionality, in particular if the environment is designed to support the whole life cycle of a project. For specific (and often critical) types of cooperation, such as user involvement, one will need only a subset of the functionality provided by a product development environment. Taking into account that the difficulty of using CASE and similar tools is often quoted as the most important reason for these tools not being used (Iivari 1996), it is an advantage if the users can use the part of the environment they really need.

The last requirement (REQ.17) is motivated with the fact that a large part of the communication in a product development project is informal and opportunistic (Kraut and Streeter 1995). This communication is crucial for the quality of the resulting product, in particular in the initial phases of a project. In geographically distributed groups this form of informal and opportunistic communication is almost totally absent. Opportunities for communication can be related to the product, or to completely unrelated issues. The important thing is that developers should be able to initiate contact in an easy and low-cost way. Opportunistic communication is promoted by continuous awareness of what others are doing, i.e. *participant awareness*. Examples of computer-based systems are media spaces (Mackay 1999)

Table 2.5: Requirements for product development environments.

Req. ID	Requirement description
REQ.1	**Shared product space**: A product development environment should provide a shared space for embodying the product and the interactions of the developers with the product.
REQ.2	**Flexible access to the product**: A product development environment should provide flexible mechanisms for accessing and updating the product.
REQ.3	**Unrestricted product object types**: A product development environment should allow the developers to share any type of object that they might find useful for supporting their cooperation.
REQ.4	**Unrestricted relation types**: A product development environment should allow the developers to create any type of relation between any two product objects.
REQ.5	**Incremental product refinement**: A product development environment should provide the developers with flexible mechanisms for incrementlly refining the product. The developers should be allowed to start with vague products, and to refine them into more complete and formal ones.
REQ.6	**Support for boundary objects**: A product development environment should allow the developers to view the product from different perspectives. The environment should in addition support a global view of the product.

Continued on next page

Continued from previous page

Req. ID	Requirement description
REQ.7	**Active delivery of information**: A product development environment should take an active part in delivering necessary information to the developers. In particular information about changes to the shared product should be delivered continuously to the interested developers.
REQ.8	**User-defined information delivery**: Active accessibility should be primarily based on user-defined criteria about the relevance of the information.
REQ.9	**Representation of developers**: A product development environment should support a representation of the developers and their activities. This representation should be used to support opportunistic communication among developers, and to connect product-related information to individual or groups of developers.
REQ.10	**Centers of interaction**: The product development environment should support focused cooperation among the members of a group working on a common task. Cooperation on specific tasks should be supported among developers regardless of their geographical location.
REQ.11	**Emergent creation of centers of interaction**: The product development environment should provide easy mechanisms for creating centers of interaction as they emerge.
REQ.12	**Emergent boundaries for centers of interaction**: The product development environment should allow dynamic reconfiguration of centers of interaction with respect to their members, their contents, and their supported interaction mechanisms.
REQ.13	**Dynamic and rich interaction in centers of interaction**: The product development environment should support the members of center of interaction with mechanisms for flexible, rich and dynamic interaction.
REQ.14	**Local customization of contents**: The contents of a center of interaction should be customizable based on the preferences of the members.
REQ.15	**Multiple user interfaces**: The functionality of the product development environment should be available through different, unanticipated user interfaces and interaction devices.
REQ.16	**Tailored functionality**: The developers should not be forced to use all the functionality of the environment. The users should be able to decide what parts of the provided funtionality they want to use.
REQ.17	**Support for oppotunistic communication**: The product development environment should support its users in getting involved in chance encounters and opportunistic communication.

2.5 Summary

In this chapter we have investigated the role of product as a *resource for cooperation*. We have seen that the product has a number of implicit properties that are used by co-located developers in order to support their day-to-day cooperative activities. Products are used intensively for recording

accumulated product knowledge, creating common understanding, and coordinating the activities of the developers. These properties of products are often implicitly and transparently supported by the shared physical space that embodies the developers and the product. Physical space provides an efficient medium for both intentional communication, and opportunistic consequential communication. It provides seamless access to up-to-date information about what others are doing with the product. The space also allows developers to easily create centers of interaction where they can customize their interaction with a large product. We have seen, through a case study of ALPHA, that geographically distributed projects suffer from the fact that the product can no longer be used as a resource for cooperation. Difficulty of access to the product, and to the activities of the developers related to the product, causes breakdowns in knowledge externalization, cooperative learning, and coordination. In addition, it becomes difficult to customize the interaction with a large product and information overload occurs easily.

The requirements for computer systems that we have derived from our analysis are crucial for using the product as a resource for cooperation, in particular in a geographically distributed project. In the next chapter we will investigate a number of existing tools and environments that are either explicitly designed for product development, or have a potential for supporting cooperative product development.

Chapter 3

Cooperation Support in Product Development: State of the Art

3.1 Introduction

The aim of this chapter is to provide a state of the art survey of tools and environments for supporting cooperative product development. Although CASE is a mature technology and has existed since the middle of 70's, we will see that cooperation support is often quite limited in existing CASE tools. Advancements in cooperation technologies, in particular those originating from the CSCW research field, have not influenced CASE research and development to a degree that one would expect. This is unfortunate because more and more product development projects are being conducted by geographically distributed groups. If CASE tools cannot support the cooperation among these groups, the use of CASE will be marginalized.

A short overview of CASE research issues is presented in Section 3.2. This is the main strand of research that is concerned with supporting the upstream activities of software product development in large groups. The goal of this overview is to show how CASE deals with the problem of supporting a large number of people interacting with a complex artifact. We will examine the strong and the weak aspects of CASE from a cooperation support perspective. Section 3.3 gives an overview of cooperation technologies for supporting groups of people in their cooperative activities. This type of technology is also called *groupware*, and originates mainly from the CSCW research field. In Section 3.4 we discuss some existing systems, originating both from CASE and CSCW research. We present one configuration management system (ClearCase), two CASE tools (MetaEdit+ and TDE), and four shared workspace applications with support for shared interaction with artifacts (BSCW, CBE, TeamWave and Orbit). We evaluate each system according to the requirements of Chapter 2, and present the evaluation results in Section 3.4.4.

3.2 Computer Aided Software Engineering: An Overview

Systems for supporting the process of software development are as old as computers. The first such systems were assemblers, compilers and linkers. These early examples were mainly tools used to give instructions to computers. High level programming languages were developed later in order to make it easier for programmers to write computer instructions. The source code files that were created using these high level languages served also another important need: communication among programmers. Readability of source code was soon considered a most important property (Dijkstra 1968). Natural language annotations were added to computer instructions written in high level languages. These "comments" turned out to be of crucial importance, both for communication among the programmers who developed the software, and for those who maintained the software later.

As the size of software products grew, a host of supporting methods and tools were developed. These tools and methods were developed not only to cope with the increased size of the products (which by now consisted of millions of lines of source code), but also, as a consequence and often more importantly, to cope with the process of creating those products. These large products could not be developed by single or few programmers. Coordinating the work of tens or hundreds of developers working together to develop software products became a central issue (Brooks Jr. 1975).

The terms *Software Engineering Environments* (SEEs) and *Computer Aided Software Engineering*[1] (CASE) denote tools and groups of related tools that are used to support the work needed to develop a software product. Tools are the building blocks in CASE and SEE, and may range from assemblers and compilers, to graphical editors for creating visual diagrams, to project and process management tools. Tools are often grouped according to various criteria, such as the project phase they are used in, the type of activities they support, the information in form of product objects they exchange, the target user group, etc. Different types of CASE and SEE may support different aspects of work, e.g. creation and sharing of product objects, transformation of product objects to machine-readable formats, definition of work processes and methods, etc.

There exist different definitions of these terms in the literature. In terms of Sommerville (1992) SEEs are regarded as consisting of *CASE building blocks*, where these building blocks are *integrated* in different forms of *environments*. Sommerville also distinguishes between the types of environments that can be created, based on the type of activities they support: *Programming environments* (for coding activities), *CASE workbenches* (for analysis and design activities), and *Software-engineering environments* (for whole life cycle).

A more comprehensive classification is provided by Fuggetta (1993). This classification distinguishes between a *production process* and a *metaprocess*. The production process includes "all the activities, rules, methodologies, organizational structures, and tools used to conceive, design, develop, deliver and maintain a software product." A production process is "defined, assessed, and evolved through a systematic and continuing metaprocess." (Fuggetta 1993, p.26)[2]. These two processes are supported by an *infrastructure* (a combination of operating systems, advanced databases, process technology, etc.), which is implemented using the *enabling technology* (standards that allow tools to be physically distributed and still cooperate with each other, e.g. *integration platforms* such as network file systems). The infrastructure, production process support,

[1] Or Computer Assisted Software Engineering, or Computer Aided Systems Engineering.

[2] Note that this distinction is very similar to the distinction between work and articulation work of Schmidt and Bannon (1992).

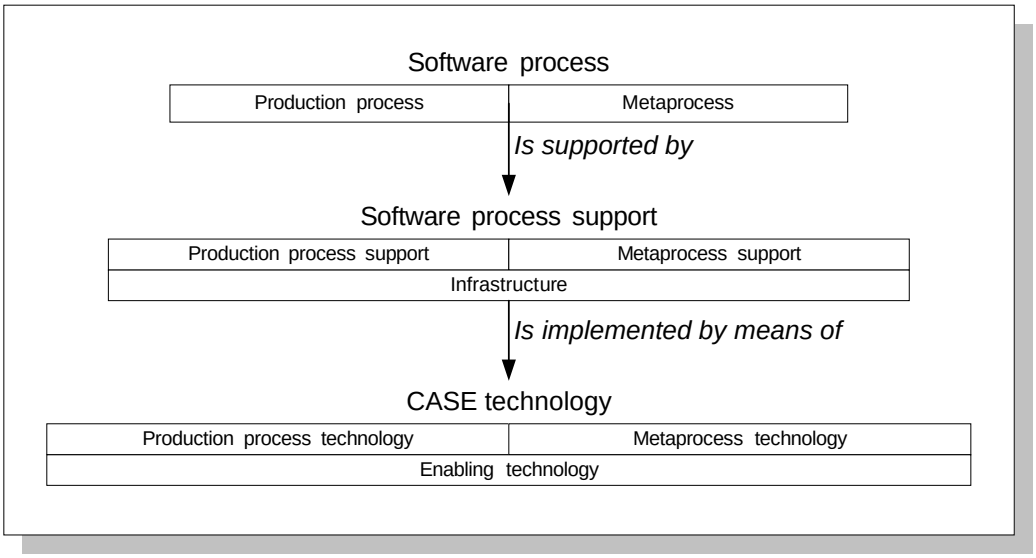


Figure 3.1: Fuggetta's (1993) CASE classification framework.

and metaprocess support together constitute the *software process support*. Figure 3.1 shows the distinction between production and meta processes in process, support, and enabling technology levels.

Fuggetta's definition of CASE is more generic than that of Sommerville's (1992). In Fuggetta's terms, CASE is considered to be any combination of enabling technologies and software process support technologies. This means that CASE may support both the production process and the metaprocess. Fuggetta further classifies CASE used in the production process as: *CASE tools* (used to support single tasks), *CASE workbenches* (used to support activities consisting of tasks) and *CASE environments* (used to support a possibly large part of the process). Parts of Fuggetta's further division of these three levels of support are shown in Table 3.1. For each subclass of CASE in this classification, we have added their importance for supporting cooperation among product developers[3].

Though a CASE tool is defined by both Fuggetta (1993) and Sommerville (1992) to include all types of tools ranging from programming to project management, it is recognized by both of them that the term CASE is often used to denote tools and environments that support analysis and design phases of a development project. These tools are also called "Upper CASE" as opposed to "Lower CASE." In this thesis we use CASE to denote these upper CASE tools, while the term SEE is used to denote both upper and lower CASE. Sølvberg and Kung (1993) focus on

[3]There are other classifications. Forte and McCulley (cited in Fuggetta 1993) distinguish between *vertical* and *horizontal* tools. A vertical tool supports one type of activity in one specific phase of the project, while a horizontal tool supports one type of activity across project phases. For instance, a project management tool can be seen as a horizontal tool, while a programming tool is a vertical tool. This classification becomes very limited for iterative design, where a compiler, for instance, may be used all over the project. Perry and Kaiser (cited in Fuggetta 1993) make a distinction between *structures*, *mechanisms* and *policies*. Environments can be individual, family, city and state.

Support level	Class of products	Subclass of products	Importance for cooperation
Support for tasks: *CASE tools*	Editing tools	Graphical editors; Text editors	Modification of product objects shared by several developers.
	Configuration management tools	Version managers; Configuration builders; Change managers	Management of dependencies among developers, and articulation of dependencies in cooperation.
	Project management tools	Project planners; Conference desks; Email; Bulletin boards	Articulation of project-related aspects of cooperation, and support for communication among developers.
Support for activities: *CASE workbenches*	Business planning and modeling workbenches	Graphical editors; Report generators	Articulation of organizational aspects of cooperation.
	Analysis and design workbenches ("Upper" CASE)	Structured analysis and design tools; Graphical editors; Analyzers	Manipulation and exchange of product objects shared by several developers.
	Programming workbenches	Suite of editor/compiler/debugger	Manipulation and exchange of product objects .
	Configuration management workbenches	Suite of version/configuration/change control tools	Articulation and management of dependencies, exchange of representations.
Support for processes: *CASE environments*	Toolkits	Groups of (loosely integrated) tools, often supporting programming and CM	A common set of tools to be used by the developers to manipulate and share product objects.
	Integrated environments	Repositories; User interface integration	Integrated data and user interface mechanisms to be used by all developers.
	Process-centered environments	Process definition tools; Process execution engines	A common software process to be followed by all developers.
Meta-environments	Process definition and enactment tools	Process modeling tools; Process enactment tools; Process evolution tools	Definition of production processes to be followed by all developers.

Table 3.1: Some CASE product types supporting the production process in software engineering (adapted from Fuggetta 1993) and their importance for cooperation.

upper CASE and classify tools used for analysis and design into *individual tools*, *workbenches*, *ICASE* (Integrated CASE), and *IPSE* (Integrated Project Support Environment). The difference is mainly along repository integration (see next section for more on integration). Single tools have each their own repositories, with data representations that are often incompatible with other tools. Workbenches consist of groups of incompatible tools, which however provide a common user interface and a common data dictionary. ICASE has the highest level of integration among tools, with common or compatible data representations such that the output from one tool can be used as input for another tool without manual changes. IPSEs are more focused on the project management aspects. An IPSE is concerned with engineering practice, and takes the management of this engineering practice as the starting point for supporting product development (Sharon and Bell 1995).

3.2.1 Integration in SEE and CASE

Development tools are often single-user. This can be because of many reasons, for instance the fact that these tools existed long before collaboration and network technologies were commonplace, and the fact that these are often complex tool with priorities other than supporting cooperation. However, the need for supporting cooperation among developers has emerged in the recent years. The traditional approach to provide this support has mainly been to *integrate* the single-user tools into *environments*. Integration often means increased sharing, which is one of the main underlying presumptions for cooperation as we have seen in Chapter 2. Integration of tools into environments or frameworks affects cooperation among developers. SEEs, by trying to integrate single tools into coherent wholes, implicitly introduce dependencies in terms of both data and process among developers using these tools. In addition, SEEs often try to support cooperative aspects of software development by introducing explicit cooperation models. The nature of an environment or framework, and the mechanisms it uses for integrating tools are therefore of crucial importance for the cooperative work of the developers.

There are different definitions of integration. However, there seems to be an agreement about at least the following integration policies, with varying influence on cooperation (Brown, Earl and McDermid 1992, Sommerville 1992):

- *Data integration*: All tools are integrated into an environment through a database infrastructure that allows product objects created by one tool be shared with other tools. This is the traditional "central repository" or "data dictionary" approach. Developers have to adhere to the standards and models used by the repository for codifying the outcome of their work. In addition, mechanisms for version and concurrency control may affect the way developers use the product objects.

- *User interface integration*: All tools are integrated into an environment through a user interface infrastructure that provides consistent user interfaces to all the tools. Examples are X Windows (for UNIX) and Microsoft Windows (for PCs). User interface integration may restrict access to the environment, in particular if the employed paradigms or metaphors are not familiar for a group of users. For instance, a WWW-based user interface to a SEE might be preferred by some developers over a UNIX-based one.

- *Control integration*: A uniform mechanism is provided for invoking and controlling tools, possibly with mechanisms for invoking tools from other tools. An example is the invoca-

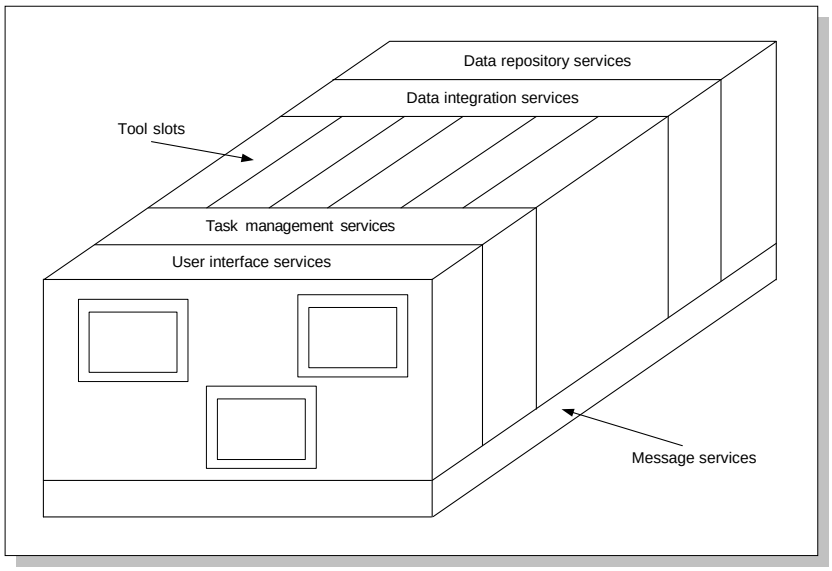


Figure 3.2: Reference model defined by ECMA. Adapted from (Brown et al. 1992)

tion of configuration management tools from within a text editor (e.g. emacs and CVS) or modeling tool (e.g. Rational Rose and ClearCase).

- *Process integration*: All the tools operate as parts of a common work process (a software process). The process explicitly defines the dependencies in work (often in form of a workflow model) and partly automates access to information and tools. Examples are workflow management systems, and software process modeling and enactment systems.

Integration is often seen as a challenge for SEE and CASE (Sharon and Bell 1995, Sølvberg and Kung 1993). Tools are developed by different companies, adhering to different standards, methods, etc. Forte and Norman (1992) argue that the lack of flexible tool integration in order to support different processes is the main shortcoming of CASE. The aim of CASE, according to Forte and Norman, is to prevent defects in software, and to eliminate unnecessary clerical work. While defect prevention has succeeded to some degree (and even become saturated in some cases), there is a general lack of knowledge about the software process that slows down the development of process support (Forte and Norman 1992).

As an attempt to promote integration among tools from different providers, the widely acknowledged ECMA (European Computer Manufacturers Association) reference model (Brown et al. 1992) has been developed in order to standardize a set of services that an ideal "universal" SEE should provide. Adhering to the defined set of ECMA services may facilitate the integration of incompatible tools into environments. This reference model is shown in Figure 3.2. The services are divided into five groups (Brown et al. 1992):

- *Data repository services*: Services for the maintenance, management and naming of data entities or objects and the relations among them.

- *Data integration services*: Services for enhancing the data repository services by providing higher level semantics and operations with which to handle the data stored in the repository.
- *Task management services* (or software process management services): Services for providing a layer of abstraction which allows the user to deal with tasks, as opposed to accomplishing each job by a tedious series of invocations on individual tools.
- *Message services*: Provide a standard communication service which can be used for intertool and interservice communication.
- *User interface services*: Services for developing standard user interfaces that are consistent across tools.

According to the ECMA model, a new tool is integrated into an existing SEE by registering itself as a client of the message services. These message services allow the tool to use the other services and to communicate with the other tools and the users of the SEE. Data storage and integration services and task management services in particular affect the cooperation in a project team. Data storage and integration services define how product objects are created, accessed, and modified by a group of developers. Task management services define the process that has to be followed by the developers, and defines how developers access the other services, including data access, in the context of a task.

3.2.2 Supporting vs. controlling cooperation

One look at existing SEEs, and one can clearly see that these systems are built with multiple users in mind. Integration frameworks are developed because there are often a high number of developers involved in a product development project, and because these developers work in a network of dependencies. The prime result of the activities of developers is the production of artifacts, e.g. product objects that make up the final product. Data integration efforts acknowledge the importance of the fact that developers depend on the result of each others' work. Data interchange formats such as CDIF (Gray and Ryan 1997) allow the result of the work done by one developer to be used by another. CASE repositories try to provide mechanisms for viewing a product object or a group of product objects from different perspectives, depending on the context and the background knowledge of the particular developer group. Mechanisms such as transaction control, version control, and concurrency control try to prevent one developer from destroying the result of the work done by another. Process integration efforts acknowledge the fact that a software process is a complex artifact that involves many activities and affects many developers. An integrated software process tries to bring together developers and their data in a way that the coordinated efforts of single developers can result in the large product.

Unfortunately, seen with another pair of glasses, a large part of these services are mechanisms created for *controlling* cooperation, rather than *supporting* it (Sommerville and Rodden 1993). For instance, as we saw in Chapter 2, three important properties of products are their central role in supporting knowledge creation, cooperative learning, and coordinating. In addition, we saw that utilizing these properties of the products requires flexible interaction among those who use the products. However, data and process integration services found in many SEEs (Brown et al. 1992), and in DBMSs in general, are often perfectly designed to create "invisible walls" between the developers (Rodden, Mariani and Blair 1992). Even though many developers might

be using the same product objects, they are hardly aware of what the others are doing with these objects, e.g. if the objects are being read, accessed, changed, discussed, etc. by others. Many CASE repositories are developed as "time sharing" systems: each developer is given the feeling of being the "only user" of the system. In addition, the format of the product objects are often controlled by strict consistency checks, making the evolution of product objects from informal ideas to formal constructs difficult. As a consequence, CASE tools might be reduced to tools for documenting products that are developed outside the CASE tools, which is clearly not what CASE tools are built for.

There are of course good reasons for enforcing this "controlling" principle. Sommerville and Rodden (1993) give three reasons: 1) SEEs were developed long before collaboration technologies became common, and their design conventionally does not support cooperation, 2) there is little knowledge of how developers cooperate in order to create systems to support this cooperation, and 3) the specific ideas behind cooperation technologies, such as sharing and communication, are not familiar to developers, who are used to information hiding and minimizing interactions. We add two related reasons: 4) time is a precious resource in product development projects, and communication must be controlled in order to save resources, and 5) the prime result of a development effort is the product, and this product's consistency and completion are the prime concerns of a SEE. The main mission of SEE as an information system is to provide means for developing software products. The obvious approach is therefore to provide services that are centered around the product. Data and process integration services seen in many SEEs are therefore centered around mechanisms for developing a technical construct, e.g. the product, an approach that is logical enough.

3.2.3 Limited cooperation support in contemporary CASE tools

Upstream activities of analysis and design are highly cooperative, and integration efforts are only one initial (and highly necessary) step in supporting cooperation. However, more advanced support, for instance direct support for interaction and communication among the developers, is hard to find in contemporary CASE tools. Vessey and Sravanapudi (1995) performed a survey of existing CASE tools and their support for cooperation. The model they used to underlie their survey classified cooperation support into three groups[4]:

- *Taskware*: is concerned with automated support to help the user perform the task at hand. A methodology companion is an example. Other examples are syntax and semantics checks in CASE editors. All these mechanisms help to provide a common framework for the developers. For instance, the fact that a tool supports a development method means that one can rely on all the developers following the method.
- *Teamware*: is concerned with coordinated access to the product objects. From an organizational perspective this means controlled access to the product and its parts. From a group perspective it means flexible sharing of the product (e.g. data sharing, consistency enforcement, concurrency control), and monitoring (of both the product and the activities of the other developers).

[4]This model is in our opinion too restrictive. In particular, the main focus is on teamware, and groupware functionality is highly marginalized (mentioned only as email and calendar functionality). The shared workspace applications that we investigate later in this chapter demonstrate some advanced groupware functionality, and show what an important role such functionality can play in for instance supporting the performance of a tasks by multiple developers.

- *Groupware*: is concerned with direct communication among the developers. This communication is supported through communication media such as email, and through "timing/meeting management."

Vessey and Sravanapudi present a detailed set of "desired features" (35 of them) regarding teamware and groupware support (33 for teamware and 2 for groupware). They investigated four multi-user CASE tools[5] and found out that only a total of 39.3% of the features (24 out of 35) were implemented in these tools. Support for data sharing was strongest (implementing 54.4% of the total number of features) followed by control (28.6%), monitoring (25.0%), and cooperation (16.7%).

Vessey and Sravanapudi's (1995) study shows a picture of CASE tools with very little support for communication among developers. Communication is assumed to happen using third-party tools such as email. This is alarming since upstream development activities that are to be supported by these tools are communication-intensive. We did a study similar to that of Vessey and Sravanapudi in order to verify their results. We used a framework similar to theirs, but extended the set of desired features with features for supporting availability on different platforms, flexibility of tasks, and user-friendly interface[6] (see Sande 1998, for details). These features are all of importance in distributed development projects, in particular when developers with varying background and skills are involved in a project. Our study included more than a hundred tools, compared to four in Vessey and Sravanapudi's (1995) study. 21 of these tools were chosen for closer investigation. The tools that we investigated were naturally more modern than the ones studied by Vessey and Sravanapudi (by at least 7 years). In particular, cooperation technologies had been widely available for 3-4 years by this time. In addition, we used questionnaires in order to investigate CASE companies' intentions and future plans regarding collaboration support. Seven companies responded to these questionnaires. Our study was not based on the testing of the products, as was Vessey and Sravanapudi's (1995).

The results of our survey are based on the 21 tools that were chosen for closer investigation (this gives an initial screening of as large as 75% of the considered tools). The obtained results confirm to a large extent those of Vessey and Sravanapudi (1995), however showing a weak tendency to support real-time conference (sharing of windows), integrated email support, and accessibility through WWW. 80% of the tools supported accessibility (mainly WWW-based views of the tasks and the product), while task flexibility and user-friendliness were low (bellow 20%). Among the features related to teamware, control was strongly supported (70%), with data sharing (30%) and user monitoring (10%) being the weakest. Regarding groupware, Synchronous communication was supported by 20% of the tools, while asynchronous communication was integrated in only 8% of the tools.

These studies show that CASE tools assume a model of cooperation in development processes that is quite unrealistic. Many CASE tools are built out of an implicit assumption that product development activities always happen in physically co-located groups. Therefore, these tools assume the availability of a shared space similar to the physical space. For geographically distributed groups, CASE tools in the best case confine themselves to offering a central repository where information about the product can be accessed regardless of geographical location. The underlying assumption for these repositories is that developers will communicate implicitly through "perfect" product objects. Direct communication is seen as "exception." The reality, however, is that CASE

[5]These tools are Deft 4.0, Iconix 4.0, System Architect 2, Visible Analyst 3.0.

[6]The study was done by Sande, a diploma student who was supervised by this author.

GROUPWARE	=	intentional GROUP processes and procedures to achieve specific purposes + softWARE tools designed to support and facilitate the group's work

Table 3.2: Groupware equation, defined originally by Johnson-Lenz and Johnson-Lenz (1982)

users, and developers in general, talk to each other very often. They discuss, criticize, refine each others' ideas. They socialize. Perfect artifacts are developed *as a result* of this socialization and not independently from it. For groups of developers who are co-located this lack of support for cooperation is not a problem because they will manage to socialize in a face-to-face manner anyhow. For geographically distributed teams, this lack of support may become a real problem.

3.3 Cooperation Technologies: An Overview

Multi-user applications have existed for decades. Database management systems have long demonstrated multi-user functionalities, and multi-user operating systems have been commonplace for a long time. Normally, the focus of these systems has been to enable the sharing of scarce resources among a group of users, while giving each user the impression of being in charge of those resources. *Cooperation technologies*, or *groupware*, are technologies developed with the main goal of supporting cooperation among people. Cooperation technologies are often based on the ideas developed from multi-user systems, and from a technical point of view are quite similar to them. There are, however, some major differences between multi-user, "timesharing" systems and cooperation technologies. In particular, a groupware application is designed to provide a *context for cooperation* as opposed to isolating its users from each other. This context is normally provided using a *virtual shared workspace*, where group members can meet, communicate, and access common resources. In addition, the group processes unfolded in this shared workspace, as a consequence of the facilitated interaction among its users, are to be supported by means of the technological apparatus offered by the groupware application.

A groupware is thus defined to be not only the technology but also the group processes and their support, as defined originally by Johnson-Lenz and Johnson-Lenz (see Table 3.2). The group processes are normally represented in a groupware application by a *common task* and a *common environment* provided to the users, as stated in another well-cited definition of groupware by Ellis, Gibbs and Rein:

> "Computer-based systems that support groups of people engaged in a *common task* (or goal) and that provides an interface to a *shared environment*" (1991, pp. 40).

Early experiments with groupware applications have resulted in different classes of tools. These early groupware examples are normally specialized in supporting one or few types of tasks, such as decision making (Nunamaker, Dennis, Valacich, Vogel and George 1991), argumentation (Rein and Ellis 1991), etc., or specific application domains such as co-authoring (Sharples

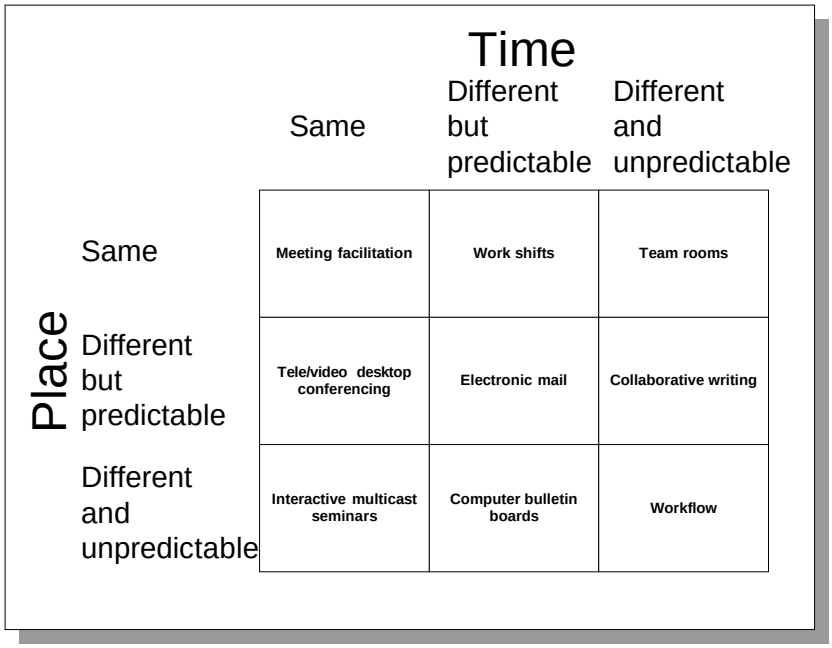


Figure 3.3: Grudin's (1994a) extension to time/space taxonomy

1993), co-drawing (Bly 1988), programming (Hu and Wang 1998), etc. These tools have provided us with valuable knowledge about the group processes they try to support, and in particular about the effects of technological support on these processes.

There are a variety of classifications of groupware applications. Maybe the most known classification is the time/space taxonomy proposed by DeSanctis and Gallupe (1987). The two dimensions of this classification are *time* (synchronous/asynchronous collaboration modes) and *space* (same/different geographical locations). The four classes of groupware defined by this classification have guided the development of groupware applications for years. The classification has been extended many times by other researchers. Nunamaker et al. (1991) have extended it with indications for group size, and number and nature of collaboration sites. Grudin (1994a) has added a third measure along each dimension, indicating whether the collaboration type and the location of participants are predictable or not (see Figure 3.3). Rodden (1991) also uses the time/space taxonomy as the basis for his application-level classification shown in Figure 3.4.

The time/space taxonomy has been criticized by some researchers (Greenberg and Roseman 1998, Schmidt and Rodden 1996). According to some researchers, basing the development of groupware applications on one or few of the classes introduced by these taxonomies will eventually lead to conceptual and cognitive gaps in different modes of cooperation supported by each application. Group processes are fluent and shift seamlessly from synchronous to asynchronous, and from remote to co-located[7]. In addition, the concept of a group as a canonic entity that can be

[7]This fact is already recognized by Rodden (1991). In his classification of groupware applications (Figure 3.4) many

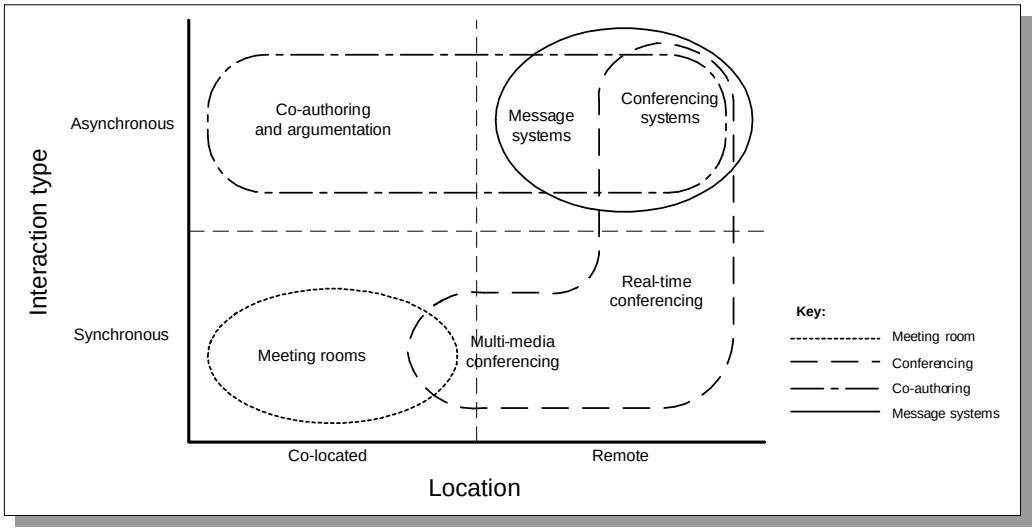


Figure 3.4: Rodden's (1991) application level classification based on the time/space dimensions.

supported by a tool is also debatable (Schmidt and Bannon 1992). The challenge for a groupware application is to support many types of processes and modes of cooperation seamlessly, and to ease transitions from one type of cooperation to another (Greenberg and Roseman 1998).

Another more interesting classification from this perspective is presented by Ellis, Gibbs and Rein (1991). This classification is based on two (continuous) dimensions of *common task* and *common environment* (see Figure 3.5). The classification does not distinguish between synch./asynch. cooperation, and is also orthogonal with respect to the location of the participants. Timesharing systems, such as DBMS, support many concurrent users, but they normally don't have any notion of a common task to be shared among these users and lie therefore to the low end of the common task dimension. On the other hand, WYSIWIS (What You See Is What I See) shared window systems (Lauwers and Lantz 1990) support to a much higher degree the notion of a common task. The shared environment dimension focuses on how much of the environmental context of a common task is preserved by the system, and is accessible to the cooperating parts when needed. Electronic mail systems based on single textual email messages do not support the notion of a shared environment despite the fact that they support a high degree of common task.

Other classifications exist. Ngwenyama and Lyytinen (1997) classify groupware as different types of resource for social action. McGrath and Hollingshead (1994) classify groupware depending on whether they modify the group's internal communication system, information base, external communication system, or performance processes. Ellis and Wainer (1994a) classify groupware based on if they are *keepers*, *synchronizers*, *communicators*, or *agents*. Moran and Anderson (1990) divide groupware development into three paradigms: shared workspace paradigm, coordinated communication paradigm, and informal interaction paradigm. Recent groupware re-

applications cross the boundaries of time and space. For instance, conference systems span almost all the four classes of groupware.

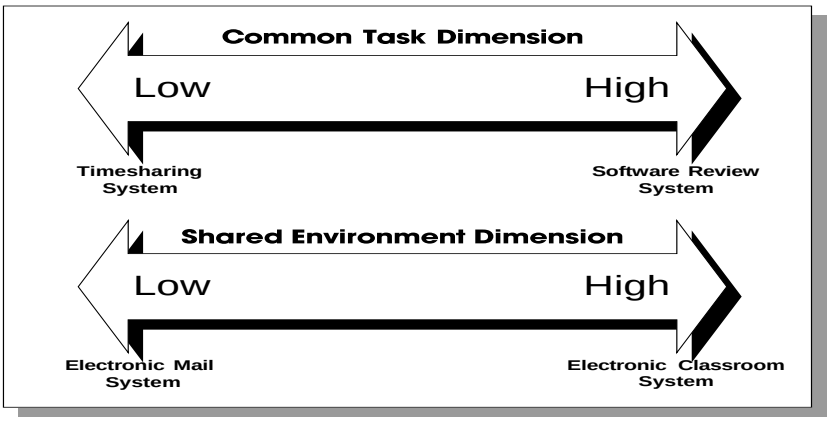


Figure 3.5: Ellis et al.'s (1991) classification dimensions.

search is more and more focused on providing generic services that are independent of the type of groupware application or the supported task (Schmidt and Rodden 1996).

3.4 Systems for Cooperative Product Development

In this section we will describe a number of systems that can be used for supporting cooperative product development. The systems demonstrated here are chosen because they demonstrate some potential for satisfying the requirements of Chapter 2. We will look at one configuration management tool, two CASE tools, and four shared workspace applications. For each tool we provide a short evaluation. An overall evaluation of these systems according to the requirements of Chapter 2 is presented in Section 3.4.4.

3.4.1 Configuration management tools

Software products are frequently modified, and there is a need for keeping track of the changes. Knowing what is changed by whom is crucial for controlling the development of software, for tracking problems, and for creating successive versions of the product. In addition, for each product there will often be different co-existing variants, for instance for different operating systems and different languages. *Configuration Management* (CM) is a common term for tools that allow developers to keep track of changes, and to choose the right versions of the right parts for a specific release of a product (Babich 1986). CM tools therefore have two main functions: to manage the changes to the product parts, and to manage the build of a complete product. Both these functions have important effects on cooperation. In this section we will investigate CM technology in general from a cooperation support perspective, using ClearCase (Allen et al. 1995) as an example of a specific CM tool[8].

[8]ClearCase from Rational is the leading CM tool in the market. ClearCase includes also other functionality than configuration management, such as software process support. Here we consider only conventional CM functionality of

Version control is the ability to create different versions of the same product object. Version control is not only important for source code, but also for documentation and other types of product objects. Sølvberg and Kung (1993) define version control (and CM in general) as a fundamental requirement for CASE tools. However, for source code there is an immediate need for version control: to track unexpected side-effects. Updating a source code file may introduce a new error or may influence other developers because of various formal dependencies. It is important to be able to compare the old and the new versions in order to find the cause of possible problems. An early example of a version control tool is RCS (Revision Control System, Tichy 1985). RCS is a part of the UNIX operative system, and has played an important role in making CM popular in the commercial world. However, the model used in RCS is based on check-out and check-in combined with locking, which does not scale to large groups of developers. In RCS, once a file is checked out it is also locked, and cannot be changed by other developers until the lock-holder checks the file into the repository. This disables parallel development of the same file by several developers. The problem becomes more severe in case of long transactions, where a file can be checked out for weeks or even months, and in some cases not be changed at all. More advanced version control tools solve this problem by allowing multiple copies of the same file to be checked out at the same time. An example from the ClearCase (Allen et al. 1995) is shown in Figure 3.6. In ClearCase, single files or groups of files can be checked out from the main *branch* into local *subbranches*, and developed in parallel with the main branch. Once the development is finished in a local branch, the files in the local branch may be merged with the corresponding files in the main branch.

Different configurations of a product normally co-exist, as for instance ports into several operative systems, product versions in different languages, etc. Each configuration may include older or newer versions of varying product objects. Another important function of a CM tool is therefore the ability to keep track of the co-existing configurations, and to automate the build of a complete product based on any specific configuration. In addition, the CM tool may help in automatically including other files in the configuration according to some dependency rules. A build is created (semi-)automatically by the CM tool. Frequent creation of builds guarantees that the work done by different developers or groups can be integrated. Daily builds are in fact common in many software houses (Iansiti and MacCormack 1997).

CM tools have gained great popularity in the software industry, and one can hardly find a software development project that does not use one or another CM tool (Radding 1999). According to Ovum (Ovum 1999) the market for CM tools reached $1 billion in 1998. Most popular are those CM tools that also support geographically distributed projects. For instance, ClearCase MultiSite (Allen et al. 1995) allows each site in a distributed project to have a replica of the repository. Replication mechanisms used in MultiSite are based on the existing branches: a site is made responsible for a group of branches, and is consequently the only site that can change these branches. Updates to the branches are then replicated to the other sites periodically.

From a cooperation support perspective, a CM tool is a good example of an application that gives all its users direct benefit in return for the cost of using it (Grudin 1994b). For a project manager, the proper use of the tool guarantees control over the various products of the project. For a build manager (the person who is responsible for building a specific configuration) the tool largely automates the tedious task of manually getting hold of the latest versions of all the files belonging to the configuration. From a developer perspective, the tool helps protect the work

ClearCase, e.g. version and configuration control.

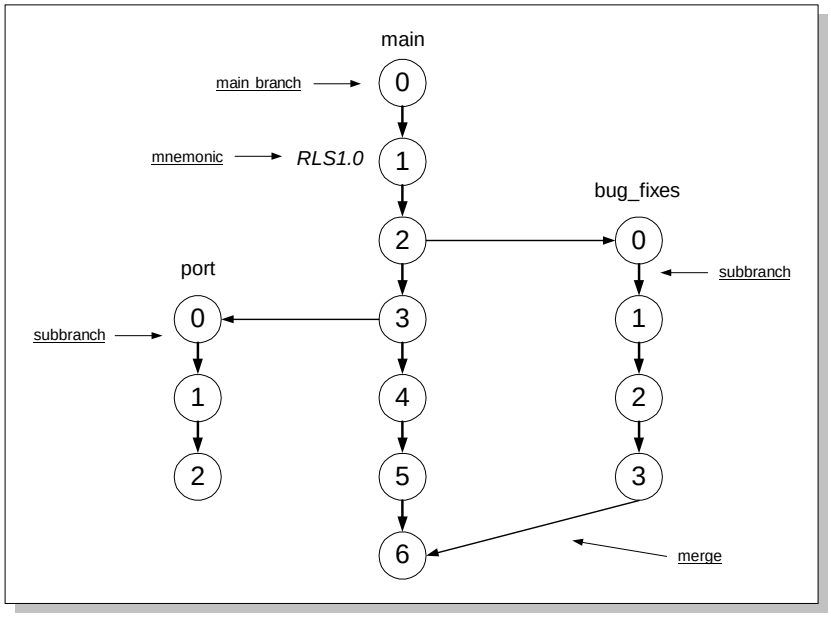


Figure 3.6: ClearCase branching model for parallel development. Adapted from (Allen et al. 1995).

done by a developer, and as a central repository it also gives access to the latest versions of other developers' files. In addition, many CM tools make visible the status of the different files in the repository (e.g. if they are checked out, changed, etc.) so that the developers can coordinate the work among themselves[9].

Despite their popularity among programmers, CM tools are not without problems. CM tools are mainly based on isolating the developers from each other, and delaying direct communication among them as much as possible. An example can be seen in Figure 3.6. Here, the developer or the group of developers working with the branch called "bug_fixes" are effectively isolated from those working with the "main" branch. This policy is implemented into the tool intentionally in order to allow focused work to be carried out in each branch. However, the result is that "merging" becomes a complex problem (Grinter 1995). The two branches can develop in quite different ways, and include updates that are incompatible with each other. The tool does not take this aspect into consideration, and assumes that the two groups do not need to talk or be aware of each other's work. In cases where the two groups working on the two branches are physically co-located, (offline) awareness and communication among them helps them to keep updated about each other's work (Rogers 1993). However, if the two groups are geographically distributed they will not have this opportunity, and in fact merging and integration problems are quite common in these situations (Herbsleb and Grinter 1999). Frequent replication of updates among geographically distributed groups is suggested as a solution to the merging problem by

[9] More data on the usage of CM tools can be found in form of empirical studies done by Grinter (2000, 1996, 1995) and Tellioğlu and Wagner (1997).

Allen et al. (1995). This might be a partial solution only in case of well-defined development tasks that require minimum communication among developers.

Merging is a much "feared" problem even among co-located developers. Merging facilities offered by existing CM tools are quite limited, and merging almost always has to be done manually by the developers. Therefore, most developers will prefer to avoid situations where they have to do merging. In fact, many tools either forbid the occurrence of these situations (such as RCS that does not allow more than one check-out for each file) or warn the developers about conflict situations before they occur. It is often the case that if a developer knows that a file is checked out by another developer, he or she will wait for this other developer to be finished, or will negotiate with the other developer about the changes he or she intends to do (Tellioğlu and Wagner 1997). This is a contradiction with respect to the "information hiding" model of cooperation that many CM tools, including ClearCase, support. This problem of invisibility is not limited to merge situations. Generally, there is need for more organizational visibility of work in CM tools, as observed both by Grinter (2000) and Tellioğlu and Wagner (1997).

Direct communication support is very limited in CM tools, and is often provided through a "hook" (such as email address) of the developers. Co-located programmers might be able to deal with the lack of communication support, both because they are co-located and because programming tasks often have well-defined interfaces to each other. Using a CM tool in all the phases of a geographically distributed project will require more advanced support for communication than what is currently provided. This lack of support for communication might in fact be the main reason why CM tools are primarily used for supporting programming tasks. An attempt to use CM in an upper CASE tool is reported by Andersen and Sølvberg (1993). The model of cooperation reported here is very similar to the one used in ClearCase. In particular, communication is still limited to merge situations. Merging of high-level conceptual representations, such as graphical conceptual models used by many CASE tools, is even harder to automate than merging source code. Andersen and Sølvberg therefore suggest using groupware technology, such as multi-user editors and shared whiteboards, for supporting the manual merging activities. One may also imaging a solution where communication is initiated before conflicting situations occur. This can be done for instance by notifying both developers about potentially conflicting check-outs *while they occur* instead of delaying this notification until a merge is needed. In addition, more general information about the status of the different parts of the product can be provided in order to keep the developers up-to-date about other developers' changes or intentions for change.

Evaluation– Shared interaction is not supported well in conventional CM tools. Although most CM tools provide support for a shared product in a central repository or file system, developers' interactions with this product are not fully shared. This is because of the check-out mechanisms, and the resulting information hiding focus of CM tools. This problem is being addressed by some recent CM tools, such as CVS (Fogel 1999), where developers can "watch" different files and be notified of changes to them. Most CM tools provide flexible access to the product through easy-to-use graphical tools, or through seamless integration with common tools of developers. CM tools are often capable of operating on files of arbitrary type. This is very practical because even the most sketchy and informal drawings can be put under version control in the central repository. CM tools support formal dependencies among files, such as import relations among program source files. Some tools, such as ClearCase, allow users to add arbitrary metadata to the files, and to search for files based on these metadata. CM tools associate all events with individual developers,

often through their user name or email address. Their accessible user interface, their notification facilities, and their often tight integration with the underlying operating system make CM tools good candidates for being integrated into the desktop of the users. Many CM tools provide a command line or network-based interface that allows them to be invoked from the developers' familiar tools in a user-friendly manner.

CM tools are very weak in providing support for centers of interaction. They often don't have a notion of center of interaction at all; all CM tools to our knowledge support a notion of a "private workspace" instead of a shared workspace. This means that they are fundamentally incapable of supporting tight interaction among a group of users. The cooperation model supported by CM tools is that of a lonely programmer. However, CM tools have a great potential for being enhanced with functionality needed for centers of interaction. Research in the area of cooperative transactions (Ramampiaro and Nygård 1999) might be a step forward.

3.4.2 CASE tools

There are only few CASE tools that have taken an active stance in supporting cooperation. In this section we describe two CASE tools in order to illustrate the kind of cooperation support that might be provided. The first tool is a metaCASE tool called MetaEdit+ (Kelly 1998, Kelly and Lyytinen 1996), and the other is a CASE tool called TDE (Telecom Design Environment, Taivalsaari and Vaaraniemi 1997). Both tools are fully implemented. MetaEdit+ is a commercial tool, while TDE is used internally by Nokia designers. An important difference between these CASE tools is that MetaEdit+ was developed as a traditional CASE tool and cooperation support is added to it recently, while TDE has been developed with cooperation support in mind and as its main priority.

MetaEdit+

MetaEdit+[10] (Kelly 1998, Kelly and Lyytinen 1996) is a metaCASE tool that provides a full multi-user environment where different users can access a central repository. The user interface of a MetaEdit+ client is shown in Figure 3.7. The basic concepts of MetaEdit+ from a user perspective are *sessions*, *transactions*, and *locks*. A session is the time from when a user logs in to the repository to when he or she logs out. Each session is composed of one or more transactions. A transaction is an atomic unit of work. A user does his or her work in a transaction, and changes that are made by the user to the repository are not visible to others before the user commits the transaction. A read, write or exclusive lock can be obtained by a user for an object in the repository. Locks can be obtained for single transactions or for whole sessions. MetaEdit+ employs a pessimistic locking mechanism in that a lock has to be obtained by a user for any information that is to be changed by that user. Transactions can be very short, for example when changing a single object, and the results of a transaction are immediately visible in the repository. In addition, the files that are locked by a transaction always provide information about who has locked them. This means that the repository provides almost full visibility of work (except the visibility of work that is being done inside an ongoing transaction).

Locking is performed automatically by MetaEdit+ on behalf of the user, based on the user's actions. The actions of the user can be starting and ending sessions and transactions. For instance,

[10] MetaEdit+ was one of the 21 CASE tools evaluated in our survey. See (Sande 1998).

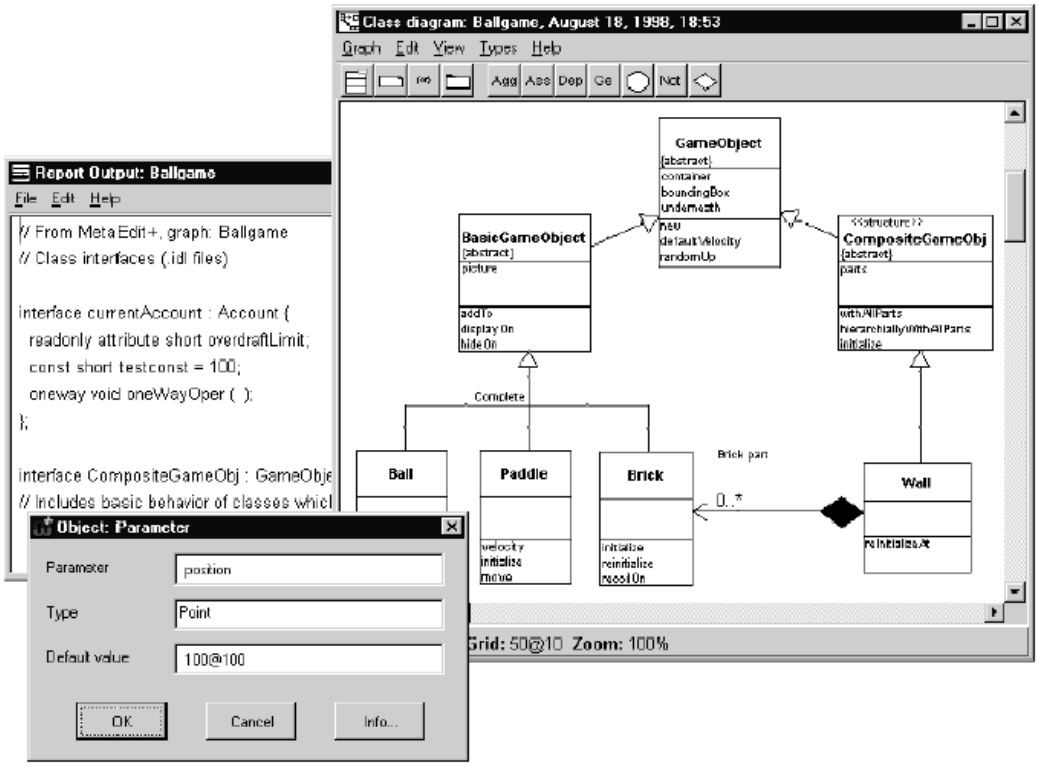


Figure 3.7: MetaEdit+ client user interface.

when a user opens a diagram in an editor, MetaEdit+ will try to get write or other type of lock for the diagram's graphical and conceptual properties. The actions available to the user after the diagram is opened depend on the type of obtained lock. If write locks are obtained for both graphical and conceptual parts of the diagram, the objects in the diagram can be changed freely. If write is obtained for only graphical part, the user can move around objects and change their representation. If no lock was obtained, the user can still view the diagram but is not allowed to change anything[11].

Evaluation– MetaEdit+ supports cooperation by 1) following a fine-grained pessimistic locking mechanism, 2) minimizing lock durations by getting and releasing locks automatically, and 3) using identified locks, e.g. a user can see who has locked an object. This is clearly a step forward regarding the information hiding principle that is common to many CASE tools; a developer can see who is locking an object, and can possibly contact that person for negotiation. In addition, the results of transactions are immediately available after commit. Developers can in this way coordinate their work with each other more easily based on the information the repository pro-

[11] It is not clear from available information whether opening a diagram requires locking all the conceptual objects in that diagram, or only those diagrams that the user intends to work on.

vide about other developers' activities. A shortcoming might be that the activities of a developer within a transaction are not visible to other developers before they are committed to the repository, but this is difficult to avoid with a pessimistic locking approach, and is probably even a desirable "limitation" because of privacy reasons. Further, Kelly (1998) takes an active stance in excluding support for direct communication and for synchronous cooperation in MetaEdit+. This means that a diagram or single objects cannot be shared in a shared workspace, and communication among developers has to go through external tools. All in all, MetaCase+ repository solves some of the problems of invisibility in the shared repository that we see in CM tools, but provides little support for constructing centers of interaction. MetaEdit+ provides a Web-based interface to some of its functionality.

In addition, metaCASE functionality of MetaEdit+ (Kelly and Lyytinen 1996) in itself can be used for supporting cooperation by allowing the construction of product object types of different formality. MetaEdit+ supports the definition of any product object or relation type, but these types do not provide different views of the same repository; they are merely developed for different projects using different methods and formalisms.

TDE

TDE (Telecom Design Environment Taivalsaari and Vaaraniemi 1997) is a CASE environment developed at Nokia Research Center (see Figure 3.8). TDE has been used by Nokia staff for designing telecommunication products. The reason for including TDE here is because TDE addresses some of the limitations of CASE tools that has been pointed out by many researchers, e.g. rigidity in how data is represented (Jarzabek and Huang 1998) and formality in processes that are allowed performed by the developers (Orlikowski 1993). In addition, TDE explicitly includes the developers in the system, and supports communication and awareness among them.

The main object for supporting cooperation in TDE is the *workbook*. Workbooks are also the main user interaction mechanism in TDE. Workbooks are "large, flat, shared, graph-structured, versioned work areas" used for collecting, representing, and modifying information of different types (Figure 3.8 shows three workbooks as windows within the main window). Workbooks are highly flexible in that they can contain any type of information, can be freely annotated using text, and both workbooks and their contents can be freely linked to each other. In addition, a workbook can be shared and modified by more than one developer simultaneously. Workbooks have an intuitive graphical and icon-based representation. The basic object types that can be in a workbook include other workbook objects, diagram objects (such as DFD and use case diagrams), user objects (objects containing "business cards" that users can leave in a workbook), host system objects (programs existing on the client, such as a word processor), objects in external CM tools (objects that are fetched from a CM tool upon clicking on them), WWW objects (links to WWW), text comments, and pictures. There are generic linking capabilities that apply to all these objects. Links can be made between any two objects in a workbook, or between two objects in two different workbooks. In addition, *aliases* can be made of the same object, allowing an object to be inserted in more than one workbook with changes to it being propagated to all the referring workbooks. Aliases also allow users to view an object in different contexts (i.e. workbooks). According to the authors aliases are a powerful and much used feature of TDE.

The more traditional CASE functionality is provided through the diagram objects. TDE takes a metaCASE approach to diagram support, in that it provides a meta-model that can be used to derive different types of graphical languages. The tested version provides support for creating

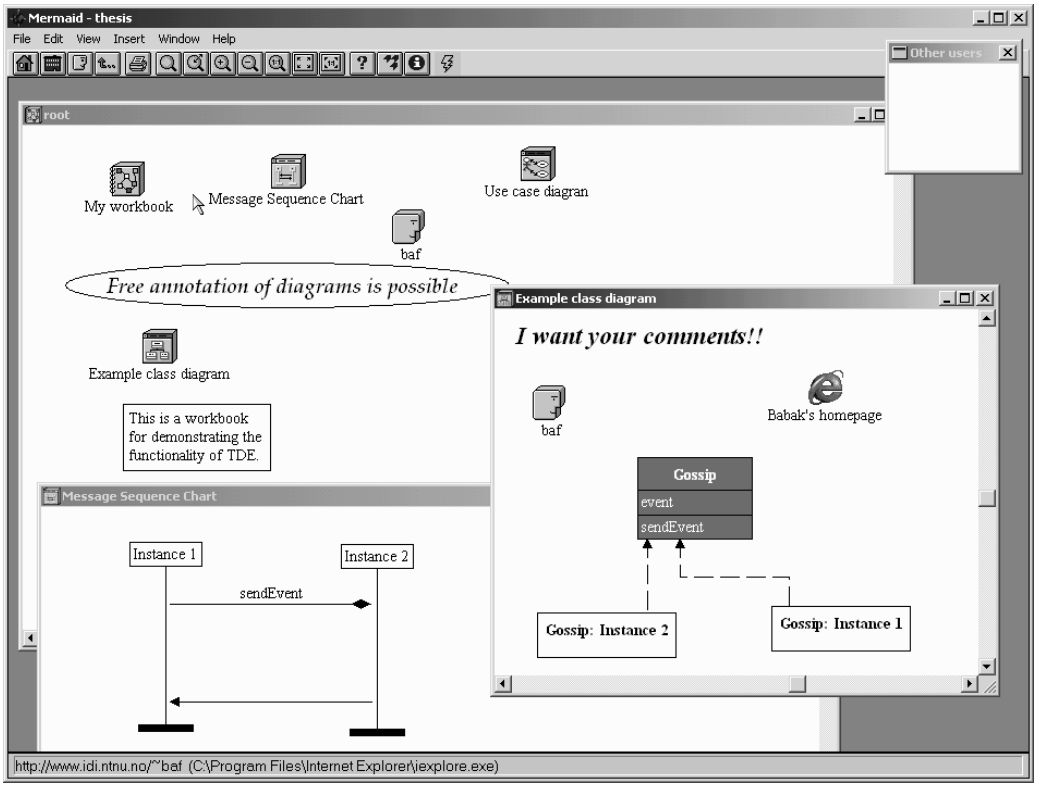


Figure 3.8: TDE client user interface.

class diagrams, object interaction graphs, message sequence charts, state charts, and use case diagrams. A diagram is basically a special type of workbook, which can be shared, linked to, versioned, etc. similar to a workbook.

In short, TDE provides a highly flexible and adaptable shared interaction framework that can be extended using the simple concepts of *places* (workbooks), *things* (objects in the workbooks) and *people* (user objects). TDE might seem very limited regarding traditional CASE functionality, and might even be regarded by some to be only a graphical editor and not a CASE tool. However, it is important to take into consideration the underlying principles for developing TDE, as formulated by its developers:

- *Focus on design rather than programming*: Rather than focusing on programming-oriented features such as code generation, integrated compiling and debugging facilities, syntax-oriented editing, and simulation capabilities, the main emphasis is on facilitating communication among designers.

- *Focus on representation and communication rather than formalization*: A goal of TDE is to serve as a collaborative media that allows designers to effortlessly and clearly express what they really mean, regardless of whether the information is formal, semiformal, or informal.

- *Going away from document-driven design*: TDE aims at replacing the document-driven design approach with a more interactive, collaborative approach in which design information –whether contained in TDE itself or in legacy systems– is represented primarily visually, and all information is accessible to the designers location-transparently.

- *Supporting location-transparent teamwork and informal collaboration*: The often formal and restrictive models of cooperation used in contemporary CASE tools do not foster true designer collaboration and is often counterproductive, since the designers have a tendency to keep their designs in their own private work areas until the very last minute.

Evaluation– TDE supports cooperation among its users by 1) supporting a shared product space and allowing full sharing of the content of this space (i.e. everything is shared and can be modified freely), 2) allowing high flexibility both regarding the type of product objects and the processes to be followed. Any type of product object can be inserted into in a workbook, different development methods can be supported as specializations of diagram object and its graphical languages, and workbooks can be used in any type of activity. The same product object can be used in different contexts. These are important steps in supporting cooperation. The product itself is protected by version control mechanisms integrated in each workbook (though version control is provided by a third party tool). It is also interesting to see that TDE is the only CASE tool we know of that includes an extended representation of the developers in the system. Each developer can leave his user object in any workbook (possibly together with a note), indicating that he has been visiting the workbook. Also, user objects can be used to indicate "membership" in a workbook, and will allow others to regard the developer as a member of the workbook. A list of other connected developers is shown in a dedicated window, in this way supporting opportunistic communication among developers. A workbook can be shared by a group of developers, which provides a simple and flexible mechanism for creating centers of interaction.

TDE is rather weak from the point of view of traditional CASE tool functionality (configuration management, formal methods, transformation of specifications, etc.). According to TDE's developers, when constructing a collaborative CASE tool the right approach is to construct a completely new tool with cooperation support in mind. This approach will provide more freedom (to provide cooperation support) than an approach that is based on extending an existing conventional CASE tool with cooperation support functionality.

3.4.3 Shared workspace applications

In this section we will look at some example shared workspace applications that are developed as generic cooperation technologies within the field of CSCW. What is common for these applications is that they focus on technical support for creating *shared workspaces*, i.e. virtual rooms that allow a group of distributed people to cooperate in solving a problem or performing a task. These applications are not specialized in supporting cooperative product development. However, they demonstrate functionality that can be useful also in a product development context. In particular, they are all based on shared artifacts, and support cooperation through shared manipulation of artifacts within shared workspaces.

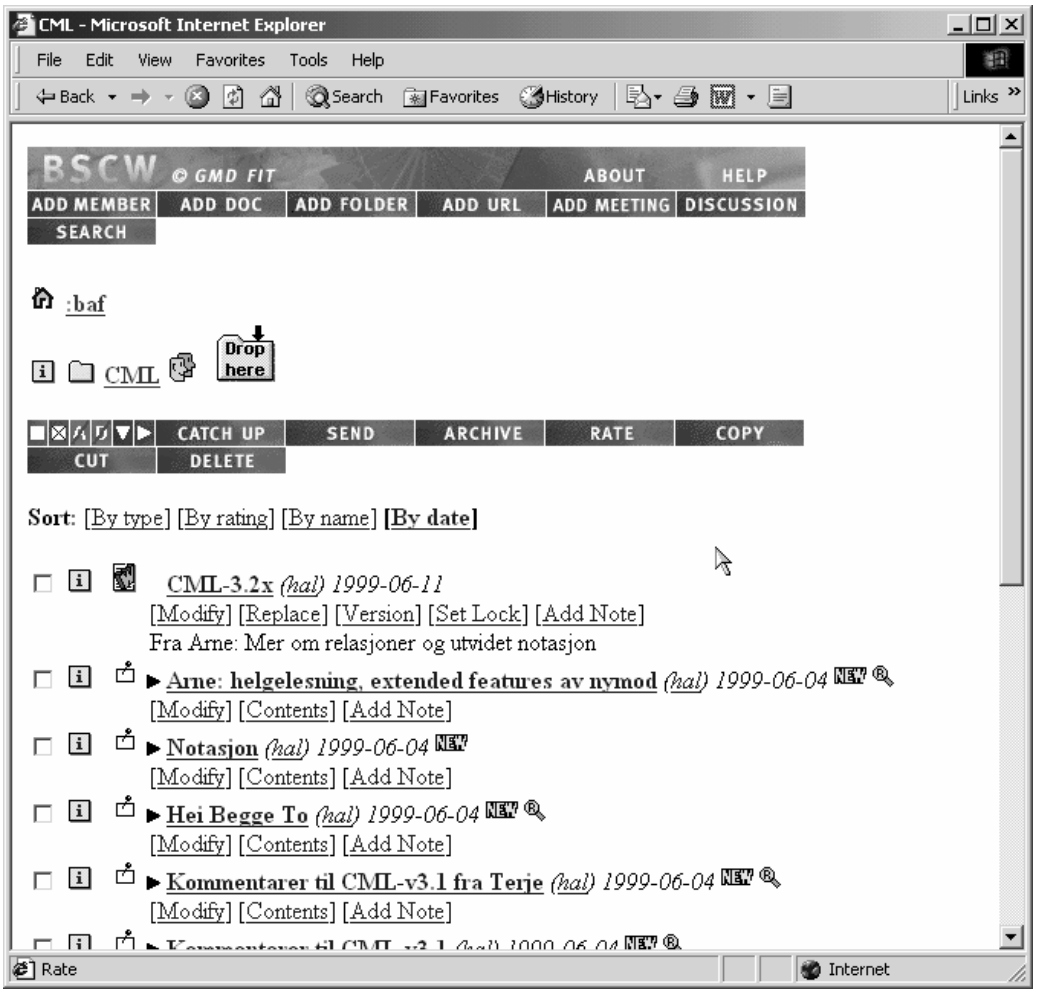


Figure 3.9: BSCW user interface as seen in a WWW browser.

BSCW

BSCW (Basic Support for Cooperative Work, Bentley, Horstmann and Trevor 1997) was initially developed as a research prototype at GMD (German National Research Center), and is now available as a commercial system. BSCW is a web-based shared workspace application with a focus on sharing of artifacts. Using BSCW does not require the installation of any programs, and a standard WWW browser suffices in order to access BSCW's full functionality. BSCW can be characterized as a system which is easy to use, and which provides basic and essential support for cooperation. The user interface of BSCW as viewed in a WWW browser is shown in Figure 3.9.

The model of cooperation supported by BSCW consists of *folders*, *shared workspaces*, *objects*,

awareness events, and *users*. Each user is provided with a "homepage," which is a web page that provides access to the functionality of BSCW. In this homepage, the user can create folders and invite users to these folders. Each folder has its own web page. Once a user invites somebody to a folder, the folder becomes a shared workspace. This is shown in the homepage with a special icon in order to help the user distinguish folders from shared workspaces. The invited user gets notified through email, and is provided instructions on how to register to BSCW in case he is not already a BSCW user.

A folder or a shared workspace can be used to upload files. BSCW is indifferent regarding the type of the uploaded files. Also, beside some simple conversion mechanisms (for instance converting a MS Word file into HTML), BSCW does not process the contents of the uploaded files in any form. This is done by the users' local applications. However, BSCW supports the addition of various metadata to each file in order to allow the users to explain the contents and the purpose of the files they upload to a shared workspace. BSCW uses an advanced access control mechanism for the uploaded files, supporting negative rights and delegation of rights (Sikkel 1997). The later versions of BSCW also include simple version control with support for variants.

A shared workspace in BSCW offers a number of functions. These include operations on objects and folders (such as adding, deleting, modifying, putting under version control, attaching notes, etc.), operations for adding/removing members, and operations for creating discussions. Discussions can be connected to single objects, and can have an IBIS-like structure consisting of issues, replies, arguments, etc. There are a number of navigation mechanisms, allowing a user to see where he or she is currently, and allowing the user to move to other workspaces easily.

BSCW supports awareness of others' activities related to the shared objects. BSCW logs all access to the existing objects. This means that a log is created every time a new object is created, or every time an existing object is read, changed, revised, deleted, moved, or accessed otherwise. This log is used to notify the users about other users' activities. Each time a user visits his homepage, various "awareness icons" in his homepage show an overview of the recent activities in the shared workspaces the user is a member of. This awareness information can also be sent to the user by email. The awareness icons will be displayed in a user's homepage until the user "catches up" with the events, an action that tells BSCW the user has seen the awareness information. Catch-up function is object-based, meaning that a user can catch up a single file, all the files in a single shared workspace, or all the files in all the workspaces.

Members of a shared workspace can be viewed easily by clicking on the "members" icon. Users have a rich representation in BSCW. A user can enter detailed contact information about himself, including a picture. In addition, hooks to other communication media such as NetMeeting (a popular video conferencing tool) and ICQ (an instant message tool) can be added in order to allow others to contact the user using NetMeeting or ICQ.

Due to its WWW-based nature, BSCW is weak at supporting synchronous cooperation. Once in a shared workspace, a user cannot see the other members who may be viewing the same workspace at the same time. There has been some attempt to extend BSCW with functionality to allow this (Trevor, Koch and Woetzel 1997). The current version of BSCW includes a Java applet that shows a list of currently connected users. Users shown in this applet can be contacted through email.

Evaluation– As a system for supporting product development, BSCW has the advantage of being a highly flexible system that provides some basic support for sharing, and no advanced

support for specific tasks. BSCW allows the users to create an arbitrary structure for their cooperation in form of nested shared workspaces. It allows an arbitrary set of product object types to be shared in a uniform way. It supports the creation of centers of interaction (in form of shared workspaces) that combine product objects, people, and communication tools. The visibility of work is supported by sharing of product objects and the structure of the shared workspaces, and by the integrated awareness mechanisms. An important advantage of BSCW is of course its being WWW-based, which makes it universally accessible. BSCW also supports tailoring of its user interface to different levels of skills.

The weakness of BSCW regarding product development is the lack of an underlying shared product space to support shared interaction in the large. Despite the fact that BSCW employs a shared object system in its architecture, an object in BSCW can exist in only one shared workspace. This is very limiting since objects in a shared product space have to be accessible in different centers of interaction, possibly visualized in different forms. An object may be used by many groups in different phases of a project. In addition, BSCW does not support any notion of dependencies among objects (except through putting them in the same shared workspace). This is again a strong weakness that limits the use of an object to only one context. Another weakness of BSCW is connected to its being an asynchronous system. BSCW falls short in supporting interactive and dynamic cooperation in a center of interaction. In particular, communication is limited to asynchronous text messages, which can be quite slow and not so interactive.

CBE

CBE (Collaboratory Builder's Environment, Lee et al. 1996) is a system developed at the University of Michigan, An Arbor, for supporting cooperation among scientists. CBE is a part of a larger scientific testbed called UARC (The Upper Atmospheric Research Collaboratory, Olson et al. 1998) developed for supporting research in upper-atmospheric physics. UARC aims to support a community of researchers cooperating with each other over the Internet, and accessing remote instrumentations (often located in the Arctic areas) for viewing collected data about the earths ionosphere[12]. CBE is interesting for us because it is an example of an advanced cooperation technology used to support cooperation among experts in a complex domain. CBE is designed to integrate already existing domain-specific tools of the scientists. In this way it allows the scientists to access and view large quantities of collected data from different perspectives, and to base their cooperation on this shared data.

Figure 3.10 shows the user interface of CBE. CBE is a synchronous shared window application. The main concepts are *shared/private rooms*, *applets*, *URLs*, and *users*. CBE uses a room metaphor for organizing the activities of the users. Users can create rooms according to the activities they want to be involved in. The window to the top-left corner in Figure 3.10 is a room navigator. All the existing rooms are shown in this window. There are two types of rooms, private and shared. For each user who logs into the system, a private room is created automatically. Other users cannot enter a user's private room, but are made aware of its existence. In this way all users have an overview of who is currently logged into the system. Users can send messages to a private room, which will be displayed to the owner of the room. For shared rooms it is possible to view who is currently inside a room by clicking on the name of the room.

[12]Data collection instruments are radars that are located in Sondrestrom in Greenland, Tromsø in Norway, Millstone Hill in Massachusetts, USA, and Saskatoon, Kapuskasing, and GooseBay in Canada. However, researchers consuming the data collected by these radars (i.e. the users of UARC) are in a number of different universities around the world.

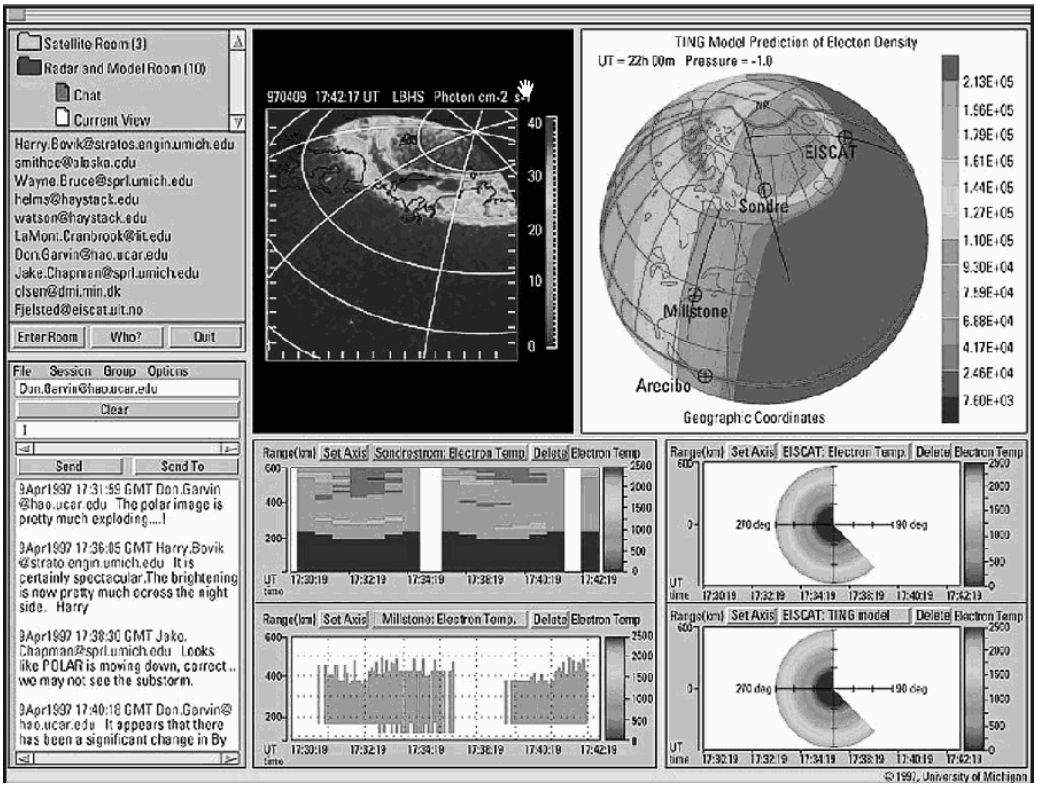


Figure 3.10: The user interface of a CBE collaboratory.

Cooperation in CBE is supported by real-time sharing of windows. A room may contain an arbitrary number of applets and URLs. Applets are CBE-specific single-window applications. In the context of scientific research, an applet is normally an application for viewing or analyzing large amounts of data. In product development, an applet might be a graphical editor or a programming tool. Each applet runs in its own window, and is shared in real-time among all the users in a shared room. Sharing means that the contents of the applet are synchronized in real time on the screens of all the users in the room containing the applet. In addition, each applet provides a telepointer (shared mouse pointer) to each user, which can be used to simulate the user's gestures. All the URLs in a room are shared, but shared browsing of the objects they refer to is not supported.

Applets can be moved back and forth between private and shared rooms. For any applet in a shared room, the user can easily move the applet back to his private room. After the applet is moved to the private room, the user's actions regarding the applet will not be visible to the other users in the shared room[13]. For the other users in the shared room this move will not make any difference, i.e. the original applet will still be shared in the shared room. In the same way, an

[13]Technically this means that a new copy of the applet is created which is not shared by the other users.

applet in a private room can be moved into a shared room[14].

Communication among the users in a shared room is supported by the integrated chat tool which allows the users to send text messages to each other. In addition, specialized communication applets can be imported into a room to allow for more advanced communication.

CBE has many useful aspects. Maybe the strongest aspect is the flexibility in switching between private and group work. A user can easily move an applet back and forth between a shared room and his private room[15]. Another strong point is that already-existing single-user applets can be made multi-user by sharing them in a room. However this requires that these applications are written in Java (which is the programming language used to implement CBE) and their source code is available for modification. Also, CBE is implemented to be used on the Internet, and can be accessed through a Java-enabled browser. CBE uses advanced object-sharing and data management technology for keeping the different replicas of the applets consistent in a chaotic Internet environment with varying quality of service (see Prakash, Shim and Lee 1999, for an overview of the technical merits of CBE.).

Evaluation– From a product development point of view, CBE is interesting because it is designed to support scientists in accessing large amounts of data. As we have seen in Chapter 2, this feature is also of central importance for product development projects dealing with large products. The solution that is chosen in CBE is to integrate existing domain-oriented data processing tools into a cooperation support environment (i.e. a shared window system), rather than developing these tools within the system. An applet in a room can be a complicated data visualization tool that is shared among the inhabitants of the room. This can be seen as a strong point because a weakness of cooperation technologies is that they often do not provide advanced domain-oriented support. However, this approach has the disadvantage that CBE does not support the concept of shared interaction in the large. A shared product space is considered as "external" to CBE, only to be accessed through the domain-oriented tools of the users. The consequence is for instance that the same data can be manipulated by the inhabitants of two different rooms only if the specific applets that are used for modifying the data are capable of concurrent access to data. This means that CBE by itself does not provide any support for shared access to a product unless within the same shared room. Rather, CBE can be seen as a "session manager," used for keeping track of the created rooms and the objects within each room, and for keeping consistent the different replicas of the applets that are shared within a room[16]. This means that CBE by itself (without the external domain-oriented tools) does not support any of our requirements for interacting with a large product.

From a center of interaction perspective, CBE supports focused work by an explicit emphasis on rooms containing people and objects. The contents of a room can be changed in an ad hoc manner. Users and applets can easily enter and leave a shared room. Communication in each

[14] This flexible window sharing mechanism is provided by the underlying object-sharing toolkit called DistView (Prakash and Shim 1994), which allows for sharing of applets and provides telepointers for each applet. DistView enables the sharing of applets that are initially single-user, though some changes to the code of the single-user applet is necessary.

[15] Though the metaphors can be somehow confusing for new users. For instance, moving an applet out of a shared room creates a new copy of the applet. This means that when the applet is moved back to the shared room it will be a completely new applet seen from the other users' points of view. In a face-to-face situation, moving a document out of a meeting room will normally remove that document from the meeting room, and returning the document will not create a new copy of it.

[16] In fact, the central concept in CBE is that of an *applet group*, i.e. a group of applet replicas that have to be kept consistent. The consistency is guaranteed through a notification server called Corona (Shim, Hall, Prakash and Jahanian 1997)

room is supported by an integrated chat tool, and any other "communication applet" that might be in the room. Inhabitants are made aware of each other through a window showing their names and email addresses. Interaction with data, and in particular customization of data in a center of interaction is allowed only if the specific applets used in the room allow it. Sharing is on a window-based WYSIWIS basis, i.e. all the users see exactly the same information in each window, but they can re-arrange the windows on their desktop independently from others. Overall view of work is visualized in the room navigator as a list of the existing rooms. There are no relations among either rooms or data.

TeamWave

TeamWave is the commercial successor of TeamRooms (Roseman and Greenberg 1996), initially developed as a research prototype at the University of Calgary, Canada. TeamWave is a shared workspace application based on the room metaphor. TeamWave is a *generic* shared workspace application. This means that TeamWave is not developed to support a specific domain. TeamWave is a tool that can be used for supporting a wide range of cooperative activities that can happen in a normal meeting room. The user interface of a TeamWave client is shown in Figure 3.11. Central concepts of TeamWave are *users*, *rooms*, and *tools*. Users create rooms, and place different types of tools in each room. A room in TeamWave is a two-dimensional surface. In Figure 3.11, the large white surface is a room that contains a number of tools.

A room in TeamWave can be used as a shared desktop where the users can leave tools and artifacts of different types, or use the surface for annotations and sketching. The toolbar to the left contains different tools for drawing freehand sketches on the surface of the room. In addition, a collection of simple tools, including file holders, URL references, concept maps, address books, post-it notes, to-do lists, and voting tools are provided. The default tools in TeamWave are mainly simulations of tools that are used in conventional meeting rooms. TeamWave also provides a programming interface for developing new tools, for instance domain-specific tools. Tools in TeamWave are in form of objects that can be left on the desktop and shared with others. All the changes to the surface of a room, including changes to the tools, are replicated in real time to all the users in the room.

Users can create new rooms and doors between any two rooms. Clicking on a door will take the user to another room. A room navigator window on the top-left corner (see Figure 3.11) shows an overview of all the existing rooms together with the name of the current users in each of them. Users can enter a room by clicking on the name of that room. The list of the rooms in the room navigator can be sorted chronologically. Grouping of the rooms in the room navigator is however not possible.

A room can be "inhabited" by a single user or simultaneously by several users. The contents of a room, i.e. tools and artifacts left there by the users, are shared in real-time among all the users inhabiting the room. TeamWave allows the users in the same room to be aware of each other in a number of different ways. The window above the room navigator in Figure 3.11 (top-left corner) shows a list of all the users currently in the room. This list can be augmented by a small portrait of each user. By clicking on a portrait one can send email or instant text message to that user. The list of users is updated as users enter and leave a room. The window on top-right corner shows status information about other users in the room, and also indicates their level of activity, i.e. if they are active, and if not how long they have been idle. There are two other important information about the users in a room. A *telepointer* (a special mouse pointer) is given to each user, and the

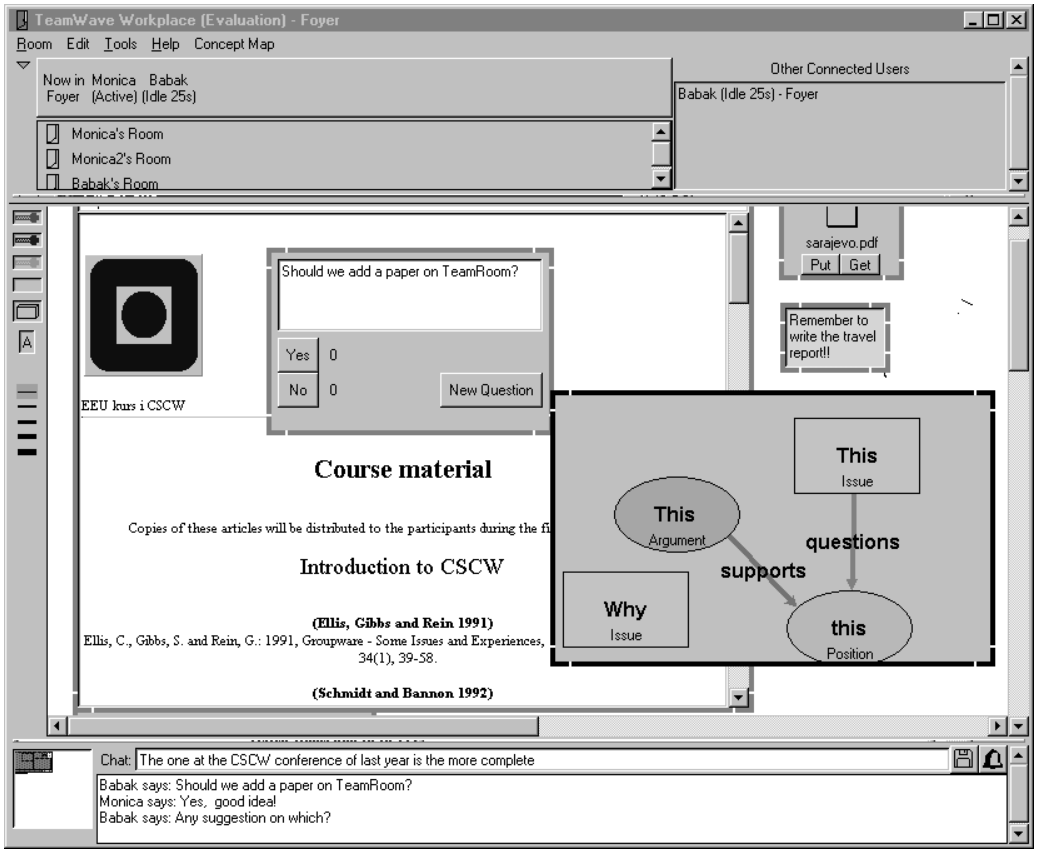


Figure 3.11: The user interface of a TeamWave client.

movements of each telepointer are visible to all the other users in the room. Each user's initials are written on his telepointer, in this way allowing others to identify the user easily. Telepointers are the main gesturing tool within a room.

The other important information is about the location of the users in a room. TeamWave follows a relaxed WYSIWIS (Lauwers and Lantz 1990) approach. This means that not all the users need to see exactly the same view of the room. A room is typically much larger than what can be viewed on the screen, and each user can be in a different corner of the room. The small window on down-left corner is a *radar view* (Gutwin et al. 1996) of the current room, showing the location of each user in form of a rectangle. The users' telepointers are also shown in this radar view, which can be used as an indication of activity levels. As users "move" within a room, their associated rectangle also moves to indicate their new location in the room. Both telepointers and the rectangles in the radar view assign a separate color to each user.

Direct communication in a TeamWave room is supported by a chat tool (the lower-most window). A "pager" tool is available for direct communication among any two users.

Evaluation– The strong aspect of TeamWave is the advanced interaction mechanisms provided within a room. Combined with an audio connection, the users can be involved in a rich and dynamic interaction. This is mainly thanks to the advanced shared workspace awareness mechanisms and the groupware widgets provided within a room (Greenberg and Roseman 1999, Gutwin et al. 1996). Virtually any activity in a room is visible to all the other users in the room in real time. A TeamWave room can be used for holding interactive design meetings. The tools provided in the room such as telepointers, brainstorm and voting tools, free-hand sketching tools, etc. can be used to support interactive meetings. TeamWave in this way supports the ad hoc creation of advanced centers of interaction that combine users, objects and communication, and that support the interaction through a spatial medium.

However, seen from a product development point of view, TeamWave is weak in supporting shared interaction in large shared spaces. Although visibility is supported to a high degree within a room, it is almost non-existent across the rooms. In order to know what is happening within a room a user has to be inside that room. In addition, a user is not allowed to be in more than one room. These limitations were also reported in a study of a group of developers using TeamWave for requirements engineering (Herlea and Greenberg 1998), where the different groups involved in the task had to divide the task into activities to be performed within different rooms. In this study, although the rooms where the developers resided were connected through doors, and a common foyer was provided for plenary discussions, the level of isolation caused problems for the overall task of requirements engineering. In particular, objects within one room (such as documents and other data objects) were not accessible in other rooms unless they were moved or copied to those rooms. This indicates that TeamWave does not support the concept of a shared product space.

From an integration perspective, TeamWave is again quite weak. The only interface to the system is through the TeamWave client shown in Figure 3.11. TeamWave is written in Tcl, which also makes it difficult to run the client directly from a WWW browser.

Orbit Gold

Orbit Gold (Mansfield et al. 1999), and its predecessors Orbit Mercury, Orbit Light, and WORLDS, are all cooperation support systems that are based on the *locales framework* for CSCW system design (Fitzpatrick, Mansfield and Kaplan 1996), influenced by the work of sociologists Anslem Strauss and Antony Giddens. All these systems (and the locales framework itself) are developed at the university of Queensland, Australia. Here we discuss Orbit Gold. However, a short introduction to the main concepts of the locales framework is given first (based on Fitzpatrick et al. 1996).

Maybe the most distinct feature of the locales framework, compared to the room-based systems we have seen so far, is the abandoning of "space" as the basis for supporting cooperation. The spatial metaphor is seen as limiting because the virtual world is as much conceptual as it is physical. For instance, a group of people can cooperate in performing a task regardless of the geographical location of the members. It is the concept of a "common task" that binds the members together, not that of a meeting room. In the real world, people live in *social worlds*, i.e. "a group of people bonded by a common, sometimes implicit, goal." Membership in social worlds can be considered along dimensions such as size, duration, and the formality of the membership. The notion of a *locale* is introduced, instead of space, as the "site and means" of shared interaction in social worlds: "A locale is not simply the environment in which interaction occurs but it is the environment as part of the interaction. Hence, we purposely use the term locale to move the fo-

cus away from space itself to capture some of the purpose for which space is used" (Fitzpatrick et al. 1996, p.35)[17]. This means that a locale is independent of any domain-specific instances, such as physical space. For instance, a task can be performed both in a meeting room and through telephone calls. Here, both the meeting room and the telephone calls are examples of locales. According to Fitzpatrick et al. (1996), the reason why space has been used so often as a metaphor for supporting cooperation is because it is so obvious. We all live in space and interact within space. However, space is not so meaningful in the virtual world, where moving "can occur via a simple mouse click or a command."[18]. For this reason, a locale is seen as consisting of a *centre* rather than a *boundary*. A center allows varying levels of membership ranging from core to peripheral, while a person can only be either within or outside a boundary. In addition to the notion of a locale, the locale framework supports the notion of presence and awareness (in combination called *mutuality*), and individual views over multiple locales. A person is often involved in many social worlds, where involvement in each of these worlds migh have different intensity. A person has thus a view of all the locales he involved in, with an intensity that is specific to that person.

Orbit Gold (hereafter called Orbit) is a system based on the notion of locales. A locale is a "conceptual center" created for performing a task. Each locale has a number of members and a collection of objects associated to it. As opposed to a room in TeamWave, a locale does not have any notion of physical space or spatial relations. A user is allowed to be a member of several locales concurrently. The user interface of an Orbit client is shown in Figure 3.12. A list of all the locales that a user is a member of are shown in the locale navigator window (the "Orbit navigator" window to the left in Figure 3.12). The larger window to the right is the private workspace of the user. The icons on the workspace denote a selection of objects belonging to the different locales the user is a member of. As indicated in the figure using circles and arrows, each group of objects belong to a specific locale. Objects can be documents, source code, etc. For each locale, the user can decide what objects from that locale are to be displayed in the private workspace. All the objects from the same locale are shown with the same color in order to separate them from each other. The user can in addition arrange the objects into groups according to the locales they belong to (this is the only notion of spatial relations used in Orbit). The workspace is in this way tailored to the needs of its user.

The contents of the objects are not stored within Orbit, but reside in external repositories or on the Internet. The processing of the contents of the objects is neither handled by Orbit, but by helper applications from the user's desktop. Orbit supports access to several types of repositories, which allows the users to use their familiar objects from these repositories. This makes it also possible to reuse the same object in different locales. When an object is accessed by a user, notifications are sent to all the members of the locale the object belongs to.

A locale is a centre shared by all its members. Each member can be involved in the locale with varying *intensities*. The intensity of a user's membership in a locale can be changed in the locale navigator window by choosing among high, medium, and low intensity, and by turning on and off video and audio connections. For instance, if a user is currently focused on the locale called "power supply" (the lower most locale in the locale navigator window in Figure 3.12) the

[17] This is very similar to the distinction made by Harrison and Dourish (1996) between "space" and "place." According to their definition, space is independent from its use as a place for performing some activity. For instance, a meeting room can be used for holding a variety of meeting types, or for having a Christmas party.

[18] This is highly relevant to product development, because a product is a conceptual and not a spatial construct. "Distances" between modules in a software system are not spatial. In fact, as we have seen in Chapter 2, these distances are often socially and organizationally determined.

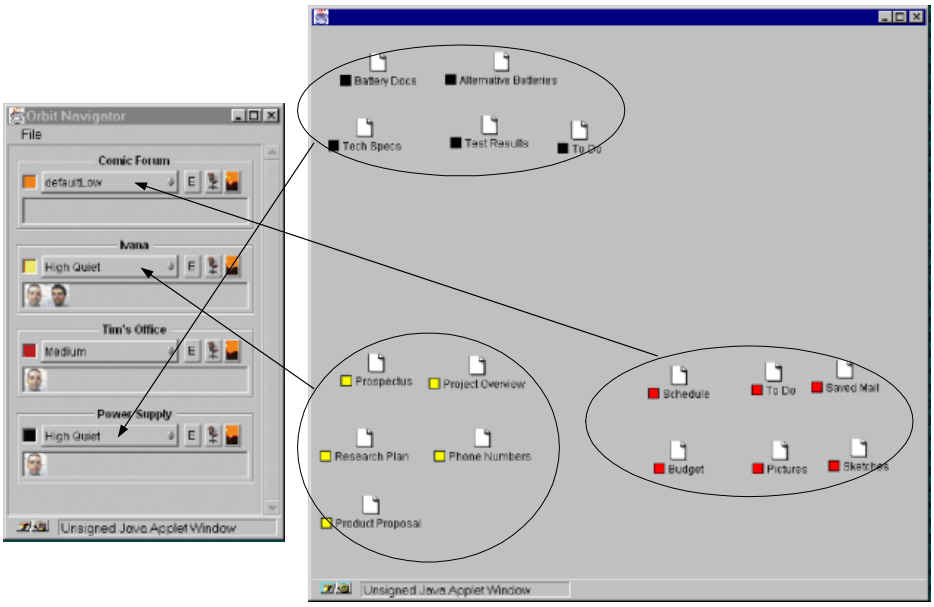


Figure 3.12: The user interface of an Orbit Gold client, with navigator (to the left) and workspace windows.

intensity of membership in this locale can be increased by turning on video and audio. In this way, the user shows his presence and "readiness" for cooperation with the other members of the locale. This will allow the other members, who might also be intensively working in the same locale, to start communicating with the user through audio and video. In this way, Orbit allows a range of "cooperation intensities" from pure asynchronous to total synchronous. For instance, if all the members of a locale increase their membership intensity to high, they can start having a meeting. However, it is not clear how a user who is not currently focusing on a locale is made aware of another user's increased focus on that locale. This would be useful for supporting opportunistic interaction.

Evaluation– Orbit demonstrates only a subset of the ideas from the locales framework. Since we are evaluating specific systems, our evaluation here will necessarily be based on the functionality of Orbit as a system by its own and not of the locales framework.

From a shared product space perspective, Orbit integrates different third-party repositories, but does not store any objects internally. When an object belonging to a locale is changed, notifications are sent to the members of that locale only. This has the same isolating effect as TeamWave, and can be problematic for large groups working with large composite products. Orbit creates a space that is shared only among the members of the same locale. This means that Orbit will not support a large group interacting with a large product.

Orbit takes a novel approach to supporting centers of interaction. A locale corresponds to a center of interaction. The strong aspects of Orbit regarding centers of interaction is that it allows

a (near) continuous involvement intensity in several locales. A user can be working intensively in one locale, and still be aware of what is happening in the other locales. This is a strong improvement compared to for instance TeamWave, where a user is either inside or outside one single room. However, interaction in a locale is not supported as well as for instance TeamWave. Users have access to audio and video for communication and awareness of others. It is reported by Mansfield et al. (1999) that Habanero, a toolkit for sharing windows, is integrated in Orbit. This can strongly increase the support for rich interaction within a center of interaction through the integration of Habanero tools such as chat and application sharing.

3.4.4 A comparison of the studied systems

Table 3.3 shows an overview of how each of the studied tools supports the requirements of Chapter 2. The support for each requirement has been evaluated to be either high, medium, or low for each tool. What we see is a clear division into two types of tools.

Genuine product development tools, i.e. CM tools, MetaEdit+ and TDE, are good at providing a space for the product, and for allowing a potentially large group of developers to interact with this product. All three tools provide a central repository where product objects can be stored. They also support relations among different parts of a large product. Along with the product space comes often methods for customizing the interaction to some degree. For instance, both MetaEdit+ and TDE allows the users to create ad hoc diagrams or workbooks. CM tools often provide multiple user interfaces that allow the users to integrate CM functionality into their desktop tools. CM tools and MetaEdit+ have no support for creating centers of interaction where many people can participate in the performance of a task. However, TDE workbooks can be shared by many developers, and in this way provide support for centers of interaction.

The four shared workspace applications show a clearly different functionality. All of them are specialized in supporting small groups of workers. For this they provide support for creating sophisticated centers of interaction, with mechanisms for supporting rich interaction within a center of interaction. However, none of the tools support the notion of a central product that has to be shared among many developers, possibly through a large number of centers of interaction. One major disadvantage is that the different workspaces are often isolated from each other. This is problematic for product development because a product often consists of complex relationships among its different parts. As we saw in the case of TeamWave, not being able to connect several workspaces together poses serious limitations on cooperation involving a large product.

Table 3.3: Overview of which systems support what requirements to what degree.

Req. ID	Requirement description	CM tools	Meta-Edit+	TDE	BSCW	CBE	Team-Wave	Orbit Gold
REQ.1	**Shared product space**	medium	high	high	medium	low	low	low
REQ.2	**Flexible access to the product**	high	low	high	medium	low	low	medium

Continued on next page

Continued from previous page

Req. ID	Requirement description	CM tools	Meta-Edit+	TDE	BSCW	CBE	Team-Wave	Orbit Gold
REQ.3	**Unrestricted product object types**	high	medium	high	high	medium	medium	high
REQ.4	**Unrestricted relation types**	medium	medium	high	low	low	low	low
REQ.5	**Incremental product refinement**	medium	medium	high	medium	medium	high	medium
REQ.6	**Support for boundary objects**	low	low	medium	medium	low	low	low
REQ.7	**Active delivery of information**	medium	low	low	medium	low	low	low
REQ.8	**User-defined information delivery**	medium	low	low	medium	low	low	low
REQ.9	**Representation of developers**	medium	medium	medium	high	high	high	high
REQ. 10	**Centers of interaction**	low	low	medium	medium	high	high	high
REQ. 11	**Emergent creation of centers of interaction**	low	low	high	high	high	high	high
REQ. 12	**Emergent boundaries for centers of interaction**	low	low	medium	medium	medium	medium	high
REQ. 13	**Dynamic and rich interaction in centers of interaction**	low	low	medium	medium	high	high	high
REQ. 14	**Local customization of contents**	medium	medium	medium	high	medium	low	medium
REQ. 15	**Multiple user interfaces**	high	low	low	high	low	low	low
REQ. 16	**Tailored functionality**	high	low	low	high	low	low	low
REQ. 17	**Support for oppotunistic communication**	medium	low	medium	medium	medium	medium	medium

3.5 Summary

In this chapter we have investigated some existing technologies that we believe have the potential for supporting geographically distributed product development projects. We reviewed some systems that are specifically designed for supporting product development. We have also looked at some shared workspace applications that are designed to support cooperation in distributed teams in general. Our study shows that there is a separation of concerns between these two groups of technologies. Product development tools support the creation of a large composite product, and allow global access to this product. Shared workspace applications on the other hand have a weak notion of a large composite product, but are good at supporting cooperation in small groups. As we have seen in Chapter 2, for supporting geographically distributed product development teams we need both types of support. We need tools that allow the creation and access to a central composite product, and at the same time support cooperation within and across centers of interaction. In the next section we will review the underlying cooperation models used in some of the investigated tools, and we will develop a new model for product-based shared interaction. This model will be used later as the basis for IGLOO framework. IGLOO framework solves some of the problems of the investigated systems.

Chapter 4

A Model for Shared Interaction in Product Development

4.1 Introduction

In this chapter we will discuss the concept of *shared interaction*, and we will explain how shared interaction can be used to support cooperation in dispersed groups. We will review some shared interaction models that are used in existing cooperation technologies, and will evaluate them according to the requirements for collaboration support in product development groups. Based on this evaluation we will develop a shared interaction model for supporting product development.

4.2 Shared Interaction: A Definition

Shared interaction is, for a group of people, to be able to interact with each other and with the subject of their work, i.e. the artifacts that they are using or producing. Shared interaction is interaction within a shared context. As opposed to individuals interacting with their private field of work, shared interaction must be visible to a group of individuals. Developing a product is an example of shared interaction. A group of developers share a set of artifacts, tools, information, etc. They interact with these artifacts and with each other in a public manner, i.e. in a way that the interactions are visible to the other group members. Interdependencies make it necessary that changes to these artifacts, tools, information, etc. are made visible to the involved people. Cooperation depends on shared interaction. A group of developers cannot cooperate if they are only involved in interactions that are not visible to others.

Probably the most common form of shared interaction is being within the same room with others. A group of people in a meeting room are engaged in shared interaction because they can see each other, they can see what the others are doing in the room, they can see the changes to the artifacts in the room, and they can themselves access and change the artifacts. In this way shared interaction creates a basis that supports cooperation among the meeting participants. An example

of shared interaction in a larger scale happens among the members of an organization. People in an organization share a view of the organization, e.g. which departments exist, who works where, who has power, who works with whom, etc. Efficient operation of an organization depends among other things on providing the basis for shared interaction among its members.

Shared interaction happens within a *shared space*. The shared space contains the people and the artifacts that are involved in shared interaction. In a meeting room, the shared space consists of the meeting participants and the artifacts within the room. In an organization, the shared space consists of people, processes, procedures, organizational maps, artifacts, etc. A shared space can take on different *structures*. The structure of a meeting room is spatial, meaning that the artifacts are located in spatial positions and have spatial relations among themselves. The shared space within an organization has an organizational structure, representing organizational boundaries, relations, norms, etc. In addition to having different structures, shared spaces have varying *size*. In a meeting room, the shared space is small enough for its contents to be fully perceived and understood by an individual. In an organization the shared space will typically be much larger than what any individual can keep an overview of. Moreover, the *boundaries* of a shared space are often not defined well. This is in particular visible in shared spaces that have a more conceptual nature. For instance, while it is easy to point out the boundaries of a meeting room, it is not clear where the boundaries of an organization reside.

The contents of a shared space undergo changes, and there is a need for the involved people to keep up-to-date. In a meeting room, people move around and artifacts get moved around by people, people enter and leave the room, bring to the room new artifacts and remove artifacts from the room, etc. The participants need to keep updated about the changes in the meeting room in order to be able to cooperate with each other. In a similar vein, cooperation among people in an organization depends on how up-to-date they are with regard to the shared state of the organization. The phenomenon of perceiving the changes and understanding their consequences is called *awareness*.

Shared interaction thus involves access to two types of information. *State information* is information about the contents of the shared space (i.e. what is in the shared space that might be of interest to me), while *awareness information* is information about changes to the shared space (i.e. what is going on in the shared space that might be of interest to me). In many cases, information in the shared space (both state and awareness information) is much larger than what any individual can understand and keep track of. *Shared interaction models* are developed in order to estimate the information needs of groups involved in shared interaction within large shared spaces. A shared interaction model (dynamically) structures the shared space according to different criteria (i.e. who might need what information), and facilitates access to state and awareness information based on the needs of the users. A shared interaction model uses different *heuristics* in order to estimate what information (both state and awareness) will be needed by each user or group of users.

Shared interaction is a prerequisite for communication and cooperation. Shared interaction as defined here does not restrict the way people can cooperate with each other (i.e. the process of cooperation) but provides a basis on which different cooperation processes can be built. Shared interaction can be used for providing the information basis for cooperation without enforcing any specific style of cooperation. As we saw in Chapter 2, the processes of product development are highly flexible with a large amount of uncertainty involved. We argued there that a support system should not predefine how people should work. This fits well with the concept of shared interaction.

Of course there is always a possibility that a shared interaction model will not be able to estimate the information needs of its users in an optimal way. A precondition should therefore be that shared interaction models are built based on empirical evidence of how people really work, and that systems implementing specific shared interaction models are flexible and adaptive.

Our interest is in particular related to computer-based shared interaction models. We saw in Chapter 2 that lack of visibility of work was one of the biggest problems facing ALPHA. In the case of ALPHA, the shared space (i.e. the developers, the shared product, and the developers' activities related to the shared product) was divided into isolated sub-spaces according to the geographical distribution of the project team. For instance, a part of the product being developed by one site was not visible to the developers in the other sites. A computerized shared interaction model can connect these isolated sub-spaces using a *virtual shared space*, in this way increasing the visibility of work by giving access to both state and awareness information regardless of geographical barriers.

4.3 Elements of a Shared Interaction Model

In this section we will look closer into the elements of a shared interaction model. We will see how a shared space is created and what constitutes it. We will look at different ways of structuring large shared spaces. We will describe the processes of awareness involved in shared interaction that must be supported by a shared interaction model. We will in particular focus on the challenging aspects of developing a computer-based model of shared interaction.

4.3.1 The shared space

The shared space is the *what* of shared interaction. It relates to the question: "*What do we need to share in order to be able to cooperate efficiently?*" The shared space might consist of artifacts, people, organizational facts, processes, etc. In ALPHA, for instance, the shared space consisted of the developers and the artifacts (e.g. prototypes, notes, source code files, documents, tools, etc.) that the developers were using in their shared interaction. Such a shared space is a resource for shared interaction.

We adopt Rodden's (1996) definition of shared space: "a collection of objects shared by a number of users." In this definition, the users themselves are represented as objects contained within the space. This definition of space makes use of the *containment property* of space[1], i.e. for an object to be shared it has to be contained, or *embodied* in the shared space[2] (Benford, Bowers, Fahlén, Greenhalgh and Snowdon 1995). In case of a computer-based interaction model, being embodied might mean that the object is *represented* in the shared space using a number of attributes, instead of actually being "inside" the space[3]. The embodiment information for each object is used (by the shared interaction model) to *position* the object in the shared space. While embodiment information itself is a property of the object, positioning an object within the space

[1] Physical space has additional properties that are not included in our use of the term space. See (Harrison and Dourish 1996) and (Benford, Bowers, Fahlén, Mariani and Rodden 1994) for a discussion of other properties of space.

[2] Embodiment here does not refer to "body" as opposed to "mind." The term is used in this thesis in a neutral way for denoting containment of any sort in a space.

[3] This is because both people and a large number of artifacts that are used as resources for cooperation are physical, and exist in the physical world. See Chapter 2 for a discussion of the importance of physical objects for product development groups.

happens in relation to other objects or an observer (e.g. a user). Positioning of objects in a shared space decides the *structure* of the space.

Positioning objects in a shared space is a critical task for a shared interaction model. The way the objects in the shared space are positioned in relation to each other or to a user (e.g. "close to" or "far away from" him) often decides how the contents of the space can be accessed. In large shared spaces containing complex objects it is often not an easy task to choose the proper positioning *heuristics*. Benford et al. (1994) have recognized four approaches to structuring what they call "Populated Information Terrains" (PITS):

1. The shared interaction model uses explicitly provided properties of objects as embodiment information for positioning them in the space. The attributes are divided into two groups, following Benedikt's (1992) definition of space. *Extrinsic* properties specify locations in the space, while *intrinsic* properties specify characteristics of the objects, such as their color. This approach is used for instance in virtual reality systems.

2. The shared interaction model uses statistical methods to analyze large collections of data in an attempt to cluster objects according to some measure of "semantic closeness." This is normally done in spaces containing document collections.

3. The shared interaction model uses explicitly provided links or relations between objects in order to position them in the shared space. An example is the WWW.

4. The shared interaction model uses ad hoc user input in order to structure the space. An example is the organization of files and folders in the desktop metaphor.

The first approach is usable in cases where the shared interaction model has access to well-structured and ordered data. An example is the *spatial model of interaction* (Benford and Fahlén 1993). This model is based on ethnographical studies of people using a physical space as a resource for cooperation (Benford et al. 1994). According to this model, objects are embodied in a virtual N-dimensional space using a set of co-ordinates computed based on a spatial frame. In this case embodiment information (i.e. the co-ordinates of the objects) is explicitly provided by the objects, and positioning information (i.e. the distances among the objects) is easily computable by the shared interaction model because a space is defined through "spatial metrics": "*well-defined ways of measuring position and direction across a set of dimensions.*" (Benford et al. 1994, p.656). This is shown in Figure 4.1. This figure illustrates a two-dimensional space where objects and users are embodied using X and Y co-ordinates provided in relation to a shared frame [i.e. the point with co-ordinates (0,0)]. The shared interaction model can in this case position user U close to objects A and C and far from object B (where "being close to" means the user can easily approach or access the objects). It is important here to note the division of responsibilities between the objects and the shared interaction model. The responsibility of the objects (including the users) is to register and regularly update their embodiment information. The responsibility of the shared interaction model is to position the objects in the space (e.g. giving user U access to objects A and C and preventing access to object B).

In most cases it will not be easy for the objects to provide the right embodiment information, or for the shared interaction model to find the right heuristics for positioning the objects. As an example consider a shared space consisting of a collection of documents constituting the documentation of a computer system. In this case, pure spatial positioning will position the documents in the space without any meaningful structure (similar to accidentally spreading the documents on

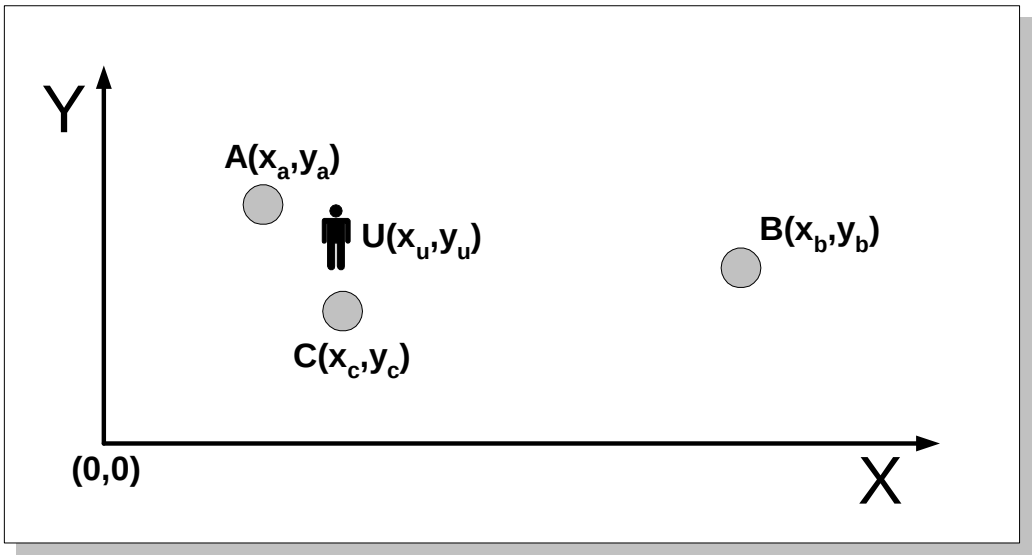


Figure 4.1: Spatial frames, embodiment and positions in a spatial model of interaction.

a desk). For the programmers of the computer system, however, the documents have meaningful relations to each other. They are often placed on the desk reflecting these relations. The other three approaches to structuring shared spaces are suited for such cases.

The second approach (using statistical positioning) is particularly suited for unstructured, often highly textual information spaces. In these cases the relations among the objects are not known to the users, or are difficult to specify in a formal way. In the above example, a model based on this approach will typically analyze the contents of the documents, and will position the documents on the programmers' desk according to "probabilistic distances" among their contents. In this approach the objects themselves provide too little meaningful embodiment information that can be used as a basis for positioning them in the space. The main responsibility of the shared interaction model is therefore to find the right embodiment information and to use the right algorithm to position the objects in the space.

The third approach requires that the relations among the objects in the space are made explicit. An example can be a user manual for a large computer system. In this case, the chapters and the sections of the manual are already structured in a predefined way. This approach puts less responsibility on the shared interaction model and more responsibility on the objects in the shared space. Although the objects may not have registered the right embodiment information, they have already specified how they should be positioned in the shared space in relation to each other.

The fourth approach (ad hoc structuring) is suited for shared spaces where not only the relations among the objects are unknown, but also the information needs of the users are uncertain. An example can be a shared space containing the requirements for a computer system under development. In this case the shared interaction model has too little information (both embodiment information and predefined heuristics) for deciding how to structure the shared space. The objects themselves take the main responsibility for structuring the space, and this structuring happens in

an ad hoc manner often with frequent changes.

There are different trade-offs among these four approaches. These trade-offs are mainly related to two issues:

- The amount of "meaningful" embodiment information that a shared interaction model has available for positioning the objects in the shared space: This information is either provided by the objects themselves, such as in the spatial model (which means more overhead work for the objects) or computed by the model, such as in the probabilistic approach (which means more complicated models, and most probably less accurate embodiment information seen from users' perspective).

- The type of heuristics used for positioning the objects in the space: These heuristics can again be totally provided by the objects themselves, such as in the case of ad hoc structuring (with more overhead work for the objects) or be implemented by the model, such as in the case of spatial models (with less flexibility regarding different contexts of use).

The problem of positioning objects in a shared space is not only related to the technicalities of providing the right embodiment information and using the right positioning heuristics. There is also a problem of multiple representations and understandings of the same object. In the majority of shared interaction models, and in cooperation technologies built based on these models, artifacts are represented in an objective form. For instance, in the spatial model of interaction, objects are represented using objective embodiment information such as spatial co-ordinates (i.e. once the common frame and the metrics are defined, all objects have well-defined co-ordinates). The situation is more complex when objects can be viewed differently (subjectively) by different users. A common example is that of vocabulary problems found in cooperation support systems, where different terms are used by the users to denote the same concept (Chen 1994).

Robinson and Bannon (1991) have discussed these representation problems specifically for (conceptual) models used in software development. One of the problems discussed by them is "ontological drift," i.e. when different "semantic communities" perceive the same objects in different ways. Also in the discussion in Chapter 2 we saw the importance of being able to view the same object across different perspectives, i.e. as a boundary object. This makes the embodiment of objects in a shared space problematic since one cannot predict what properties of the objects are considered important in different situations by different people.

In addition, once we acknowledge that the same object can be viewed differently by different users or groups of users, it follows that the positioning of the object in the space will be different for different users or groups of users. This makes it difficult, if not impossible, to have a shared interaction model that can always provide the right positioning of objects without any ad hoc user input.

These problems are highly visible in product development processes. The uncertain nature of the activities of the developers makes it difficult to provide the right embodiment information or to use the right heuristics for positioning the objects in the shared space. The product and its objects are under development, and change all the time. These changes include both change in specific properties, in the interpretations of the developers, and in how objects relate to each other. The same is true for the developers themselves, whose roles and competencies are changing as the product objects change. Uncertainty in the course of the interactions that the developers are involved in also makes it difficult to know what objects are needed when and by whom.

4.3.2 Awareness

Awareness is concerned with the *how* of shared interaction. It relates to the question: "*How do we keep aware of a shared state?*" Awareness is knowledge about changes in a shared space. In ALPHA, for instance, not only it was necessary to share the product objects, but also to shared the knowledge about changes to these objects and other developers' interactions with these objects. This knowledge was important for supporting cooperative learning, and coordination of day-to-day activities. Awareness information is similar to state information, but differs from it in important ways (Gutwin and Greenberg 1999):

- Awareness must be constantly maintained and kept up-to-date because environments change over time.
- The maintenance of awareness happens as part of the interaction with the environment.
- Awareness is almost always part of some other activity. Maintaining awareness is rarely the primary goal of the activity.

There are three main *processes* involved in achieving awareness in shared interaction. A basic awareness process is that of *producing* awareness information. In a shared interaction model, awareness information is produced as the embodiment information of the objects in the shared space is changed. An *awareness event* is a unit of awareness information if we consider an object's embodiment information as being constituted of discrete units[4]. An awareness event is often produced in a context that helps the receiver to observe the nature of the change, e.g. who did the change, what objects were involved, etc. It is the continuous exchange of awareness information that keeps the involved objects aware of the activities in the shared space.

Consumption of awareness information is another important process that helps the objects in a shared space make use of the produced awareness information. Consuming awareness information includes three sub-processes (Endsley 1995). *Perception* is the process of perceiving the awareness events produced by the environment (here the shared space), *comprehension* is the process of understanding the meaning of the awareness information, and *projection* is the process of predicting a future status for the environment based on the understood meaning of the awareness information. More details on awareness production and consumption processes can be found in (Gutwin and Greenberg 1999).

A third process is that of awareness information *distribution*. For a shared interaction model, an important issue is to provide the right awareness information to the right objects. In a small space, such as a meeting room, it is not difficult to keep oneself up-to-date about all the changes to the shared space. However, in large spaces the users will need only a limited amount of the total awareness information that is produced within the space. In these cases the shared interaction model is responsible for distributing the right awareness information to the right objects. In many shared interaction models, this issue is solved by allowing the structure of the shared space guide the distribution of the awareness information among the objects. In the spatial model of interaction (Benford and Fahlén 1993), objects embodied in the space carry their own *aura* with them. Auras are calculated by the shared interaction model (as part of the task of structuring the

[4] We consider here only *discrete* awareness information. In many groupware applications, such as video communication tools, awareness information is transferred through continuous streams of information (mainly vision and sound). These applications do not support any configuration of awareness information, and therefore are only useful for very small groups of people engaged in close cooperative tasks.

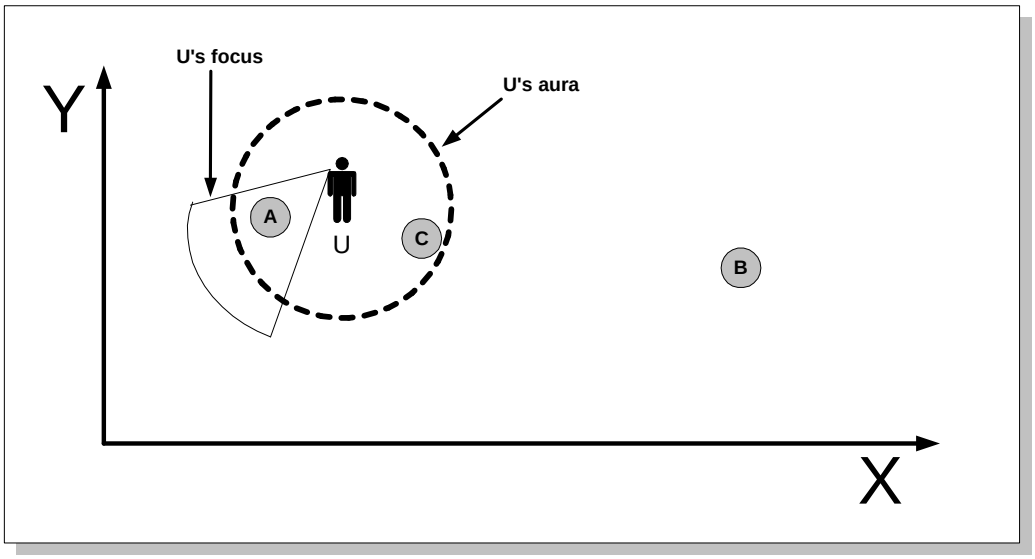


Figure 4.2: Aura and focus in the spatial model of Benford and Fahlén (1993).

space). An aura decides what awareness information is available to an object. Figure 4.2 shows a user U in a two-dimensional space. U's aura is in form of a circle (it can be any shape). U is able to receive awareness information related to all the objects within his aura. U can further decide his own *focus*, i.e. the part of the available awareness information he is interested in. In Figure 4.2, U has chosen to focus on object A and disregard object C, which is also within U's aura.

The spatial model simulates the awareness processes that take place in a physical space, where awareness information is related to spatial relations among objects (e.g. I cannot see objects moving in a distance from me). In a large shared space it might as well be the conceptual relations that are of importance. In case of ALPHA we saw that relations among objects in a shared space are more based on factors such as the conceptual architecture of the product being developed, or social relations among the developers. Spatial relations were of marginal importance for ALPHA. This problem is similar to that of structuring a shared space based on other heuristics than spatial relations. In the previous section we reviewed four approaches to structuring shared spaces, which are also recognizable for the distribution of awareness information.

However, using the same approaches does not mean that awareness information needs always be distributed according to the structure of the space. It is completely feasible, and in many cases necessary, to distinguish between the structure of the space and the scope of awareness information one receives about the space. An example is *peripheral awareness*, i.e. awareness of the part of our surroundings that is not completely in our focus. As also observed by Simone and Bandini (forthcoming), peripheral awareness can be about parts of the shared space that are not at all related to what we are currently focusing on. It is for instance quite normal to be aware of what is happening outside a meeting room (through hearing voices) while being involved in a meeting inside the meeting room. Put by Simone and Bandini:"it is not always the case that focus fully dictates the peripheral attention." This means that the distribution of awareness information needs

to be separated to some degree from the structure of the space.

4.3.3 Support for cooperation

Shared interaction involves additional activities on part of the users, such as communication, cooperation, and coordination. For instance, in order to comprehend a shared situation, the users need to have a shared mental model of that situation (otherwise they may interpret the same situation in different ways, with resulting different decisions for action). This shared mental model is often achieved by communication among the users. For solving complex problems, additional group processes such as decision making may be needed. In larger groups, communication and cooperation among a few people gives way to large scale "articulation" or coordination of communication and cooperation (Schmidt and Bannon 1992). Although these processes are not directly related to a shared interaction model as defined here, a shared interaction model should have a level of flexibility that allow the users to be engaged in different courses of interaction based on the information provided by the model.

4.4 A Comparison of Some Existing Models

In this section we will take a look at some existing shared interaction models. We will see how these different models are implemented in form of systems, and how they function from a user perspective. The term shared interaction model as we have defined it in this chapter is not explicitly identifiable in many collaboration technologies. But as we will see here, it is quite easy to find some notion of a shared space, heuristics for structuring the shared space, and means for embodiment of the objects in the space in all these technologies. The models we review here are used as basis for the collaboration technologies we have reviewed in Chapter 3.

We will evaluate the models according to the definition of shared interaction and its elements as defined in Section 4.3. In short, we will investigate the following points for each model:

- What kind of objects can be embodied in the space? What type of embodiment information is available to the shared interaction model?
- What is the size of the shared space? What kind of heuristics are used for structuring the space?
- How is awareness generated? What is the amount of generated awareness information? What is the type of awareness information that is available to the users?
- What kind of structuring is used for distributing awareness information? Can the model support awareness in a large shared space?
- What type of cooperation is supported by the shared interaction model?

In particular, awareness support is compared along the two dimensions shown in Figure 4.3. The first dimension focuses on the *quantity* of the exchanged awareness information. This quantity is decided by the kind of embodiment information that is available to the model. Larger amounts of embodiment information must be available to the shared interaction model if large quantities of awareness information are needed. The second dimension is the level of support for *organizational*

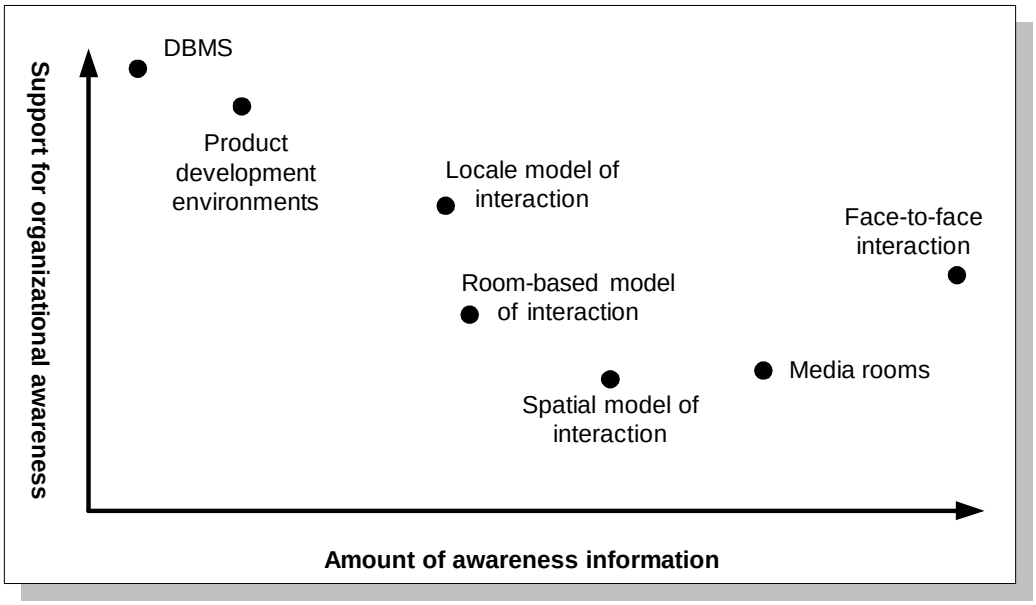


Figure 4.3: A comparison of awareness support in different systems.

awareness provided by the model. By organizational awareness we mean awareness of the overall organization of the shared interaction, and how users can access this information. Factors that increase organizational awareness are the size of the shared space, the way the shared space is structured, and the way awareness information is distributed to the users of the shared interaction model.

As a reference point for our comparison we consider face-to-face interactions (see Figure 4.3). In a meeting room the type of objects include the people and the physical artifacts they use as resources for their cooperation. The quantity of transmitted awareness information is largest in face-to-face interactions, where all the objects are fully embodied in the space (they exist within the space), and where all available social channels are used for shared interaction. The structure of the space is spatial, i.e. objects are located according to spatial relations. Face-to-face interaction is normally limited regarding organizational awareness (normally limited to meeting room settings with a few people and a few artifacts involved). This is mainly because such interactions are focused in space and time, and because there are limitations in our perceptual abilities in processing large shared spaces with lots of information.

4.4.1 The spatial model of interaction

The *spatial model* (Benford and Fahlén 1993)[5] is a shared interaction model that is directly based on observations of people using physical spaces for cooperation. The model is mainly used for realizing virtual reality environments, such as MASSIVE (Benford et al. 1994), but is also identifiable in more familiar applications such as virtual desktops. In this model, each object embodies itself in the shared space using its coordinates according to some spatial measure (e.g. geographical location), and other attributes such as color, shape, etc. The structure of the resulting shared space is spatial, i.e relations among the objects in the shared space are based on spatial metrics. By changing its coordinates, an object can "move" in the shared space, and in this way change its relation to other objects (movement in the space is implemented by the model, based on the changes in the embodiment information of each object). The spatial model can implement shared spaces of arbitrary size. However, each object's access to the contents of the space is limited to the "spatial surroundings" of the object.

As we see in Figure 4.3, systems that adopt the spatial model often result in moderate amounts of awareness information. The amount of embodiment information is often much lower than what is available in the physical space, e.g. in the case of face-to-face cooperation. In addition, in cases where the objects themselves exist in the physical world (which is the case for e.g. human beings) it is often difficult to collect large amounts of awareness information about the objects. This results in lower amounts of awareness information generated by the model. In addition, distribution of awareness information is based on the structure of the space. This means that an object cannot be aware of objects that are "spatially" far away from it, even if these objects are "conceptually" close to it. The spatial model is therefore not capable of providing a high level of organizational awareness in cases where organizational relations cannot be formalized into straight-forward spatial metrics (which is the case in many cooperation types with high level of uncertainty).

4.4.2 The room–based model of interaction

The *room–based model* (Greenberg and Roseman 1998) (also called shared workspace model) is a model of shared interaction that has gained popularity among the researchers and developers of collaboration technologies. Rooms of different kinds are used in BSCW (using folders), TeamWave (using simulations of 2-dimensional physical surfaces), CBE (using collections of windows and applets), and TDE (using workbooks)[6]. In this model the shared space is a room with more or less rigid walls. The walls control the visibility of the objects in the room. These rooms simulate to different extent physical meeting rooms, where people can meet, leave documents and other artifacts for each other, and be aware of who does what within the room (Greenberg and Roseman 1998).

Systems based on the room-based model of interaction often provide a high degree of support for shared interaction, but many of them do not support large shared spaces (Farshchian 1999). Different types of objects (both users and artifacts) can be embodied in a room with varying

[5] This model is not directly used in any of the systems discussed in Chapter 3. We discuss this model here because it is a fundamental model for systems using any form of spatial metaphors. Elements of this model are recognizable in TeamWave, Orbit Gold, CBE, and in many graphical editors in conventional CASE tools.

[6] The original room-based model was developed by Henderson and Card (1986) for single-user applications. While the problem Henderson and Card addressed was that of limited screen space, to be solved by providing the user with a number of rooms, rooms are used in collaboration technologies in order to provide a place for cooperation, i.e. as "meeting rooms."

amounts of embodiment information. Spatial relations among the objects within a room are common (for instance in TeamWave and CBE). The interaction in the room is supported by advanced collaboration support techniques, such as telepointers, specialized user interface widgets, etc. (Gutwin et al. 1996). All interactions that happen within a room are visible to all the users in the room. However, interactions within a room are limited to small numbers of users and artifacts, and are not visible outside the room.

The amount of awareness information in typical applications based on the room-based model can be as much as that for spatial models (see Figure 4.3). This awareness is related to changes in the embodiment information of the objects in the shared space. For instance, TeamWave provides visual clues when objects in a room are changed, moved around, deleted, added, etc., and when users enter or leave the rooms or get involved in some activity. In reality, however, this awareness information is often less than what is available in typical virtual reality systems, and is limited to what is important for performing a specific task in a room. For instance, TeamWave has a quite limited representation of human beings in a room, and awareness information about other objects is limited to the changes being done to them. Room-based models are often quite limited in providing organizational awareness because they separate the inhabitants of a room from the outside. For instance, being within a room in TeamWave is quite isolating because the user is never aware of what is happening outside the room (Kaplan, Fitzpatrick, Mansfield and Tolone 1997).

Applications based on the room-based interaction model normally support the creation and maintenance of several rooms, or shared "sub-spaces." One can argue that the shared space in these applications is the collection of all these sub-spaces, such as all the rooms currently existing in a TeamWave installation. In these cases, each room is normally treated as a shared space of its own, and there are (often loose) connections among the rooms. For instance, in TeamWave two rooms can be connected to each other by creating a door between them. However, doors do not solve the isolation problem because the users still have to enter a room (through a door) in order to see what is happening in the room (Herlea and Greenberg 1998).

4.4.3 The locale model of interaction

The *locale model* used in Orbit Gold (Mansfield, Kaplan, Fitzpatrick, Phelps, Fitzpatrick and Taylor 1997) differs from traditional space- and room-based models in significant ways. In this model, the shared space is divided into *locales*. Locales are similar to rooms in the room-based model in the sense that they partition the shared space. They are however different because they do not have rigid boundaries and might involve different people and objects at any time. In addition, locales are conceptual constructs as opposed to physical rooms which are often spatial. A locale is created in Orbit Gold in order to support a specific activity, and work in this way as a focus point for a group of users involved in the activity. In this sense, a locale does not require that the activity is performed in a physical room, as is the assumption in for instance TeamWave or CBE (see Section 3.4.3 on page 67 for more details on this).

The shared space in this model consists of locales and objects within the locales. Locales are created in an ad hoc manner, and different kinds of objects can be embodied in a locale. A user can be involved in a number of locales at the same time. The involvement in each locale has a different intensiveness according to how much the user focuses on the activities in the locale. The locale interaction model is suited for situations where the shared space cannot be structured using computable relations among the objects. The model is in this way capable of structuring shared spaces in a conceptual way.

The locale interaction model as used in Orbit Gold produces lesser amounts of awareness information compared to spatial and room-based models. This is because awareness information is normally connected to more abstract happenings in the shared room. For example users do not get awareness of objects moving around in a locale (this is in fact not possible because a locale does not have any notion of space), but are made aware if the contents of the objects are changed. However, awareness is connected to the structure of the space in that users get awareness of locales they are involved in at any time. This gives a moderate organizational awareness because there is a cognitive limit for how many locales each user can be involved in.

4.4.4 Shared interaction in product development environments

In this section we will look at how product development environments support shared interaction among their users. There is not a single identifiable model of shared interaction in these tools. We will try therefore to generalize. The generalization is based on the review of the product development tools in Chapter 3, and our own experience from using this type of environment.

All the product development environments that we reviewed in Chapter 3 (i.e. ClearCase, MetaEdit+ and TDE) are implemented to handle large numbers of artifacts and people. They allow their users to create a *repository* consisting of a large number of artifacts. These artifacts are often conceptual, and can have conceptual relations among themselves. The contents of the repository can be inspected by any user, in this way providing the users with information about the shared product. Such a repository, seen as a shared space, has a large size and a conceptual structure. Spatial relations among artifacts are of less importance[7]. Conceptual relations are mainly created explicitly by the users, meaning that the space is structured in an ad hoc manner based on input from the users.

The repository in product development environments is mainly used as a place for *storing* the product. Shared interaction often does not happen within the repository. For instance, in ClearCase, every time a user wants to change a part of the product, he has to “check out” the part he wants to use. In MetaEdit+, interaction with the product is private and happens within the private session of the user. In neither case the users can be involved in shared interaction with each other, and even their private interactions with the product is “disconnected” from the shared repository. The shared repository by itself is thus not used so much for shared interaction. It is mainly used as a “protected” place for storing information about the product.

The situation is similar regarding the awareness support in these tools. Product development environments do not take active role in providing awareness information, and rely on users inspecting the repository in order to keep themselves updated about the changes. Most interactions happen outside the repository, and the tools actually do not have the possibility of informing their users about these changes because they happen somewhere else than in the repository. Also, users are often represented in a minimal form, and awareness information about other users is not available. This implies for instance that users do not have the possibility of getting involved in opportunistic communication resulting from changes to the shared product.

TDE is one CASE tool that has solved some of these problems. In TDE, the repository by default contains the latest version of all the objects. This means that the latest changes to the objects are always visible to all the users, and all interactions with the product happens within the

[7] Spatial relations among objects can be seen in graphical editors used in many CASE tools. Spatial relations among objects in a graphical diagram might be used for making the diagram more understandable.

global repository. TDE still does not take an active role in providing awareness information to its users, and is based on the assumption that users will inspect the state of the repository. TDE puts more focus on representing the users in the shared space by allowing the users to leave business cards in workbooks as placeholders. In the case of TDE, the repository has acquired some of the properties of a shared space as we discussed earlier.

4.4.5 An evaluation of the models

The spatial model, the room-based model, and the locale model of interaction have been developed as generic models for shared interaction and cooperation. The spatial and the room-based models in particular try to simulate face-to-face cooperation among geographically distributed groups of people. The underlying concepts, such as rooms, spatial relations and co-ordinates, are all resources that we normally use in face-to-face interaction and cooperation. The models themselves are developed in close accordance with empirical investigations of interactions among people, and this is the main reason why these models are so powerful and universal. However, each cooperation domain normally poses additional restrictions, characteristics, and ways of doing things that can be different from other domains. The domain investigated in this thesis is that of product development. We have seen in Chapter 2 how groups of developers do things in order to develop a product. Our goal is to develop a shared interaction model that is more suited to this way of working, and that can support developers in product development. Based on the analysis in Chapter 2 we can point out some shortcomings in the reviewed shared interaction models and the systems built on them:

- Product development is an activity that involves a large number of conceptual (knowledge) artifacts and conceptual relations among them. A shared interaction model for supporting it should put more emphasis on the product and its conceptual structure. Spatial and room-based models provide a limited notion of artifacts, with focus on objective properties of these (such as their physical shape and geometric co-ordinates). Product development environments are more advanced with respect to conceptual representation of artifacts, but are limited in providing support for shared interaction.

- Product development involves a large number of people and artifacts, and at the same time requires intense focused cooperation among developers. There is a need for a global *shared product space* and local *centers of interaction* in the same model. The models we have seen mainly focus on one of these aspects. Room-based models support well focused cooperation involving a small number of people and artifacts, while product development environments and the spatial model have limited notion of a center of interaction.

- A typical product has a complex structure, where dependencies among the parts are often critical. Changing one part of the product may affect other related parts, and as a consequence may affect the work being done by others. There is a need for a separation between the part of the product a developer works on, and the part of the product he needs to be aware of. Spatial and room-based models assume that users need to be aware of what they are working with, and nothing more. However, in product development, a developer might be interested in changes to the parts that are not in his current workspace. Separating the structure of the space from the structuring of awareness distribution can help developers to explicitly specify what “peripheral awareness” they need to have.

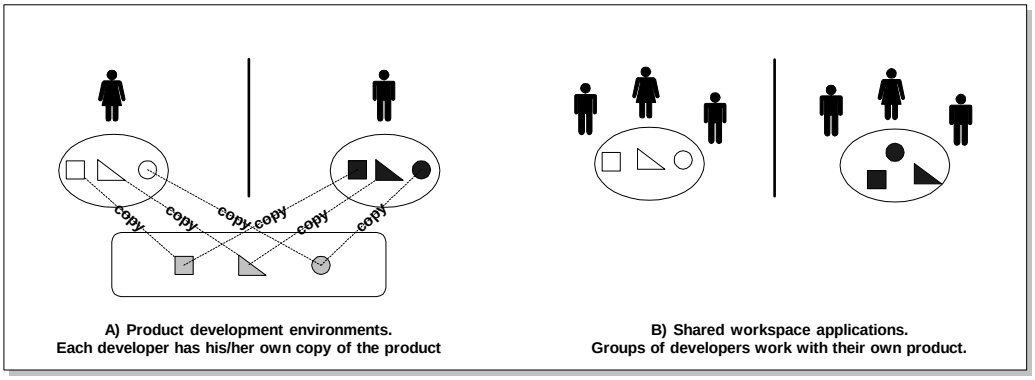


Figure 4.4: Product development environments isolate each developer from the others, while shared workspace applications isolate small groups of developers from other groups.

- Though product development environments provide access to an organizational context in form of a large repository, and they often support advanced conceptual structuring of this repository using conceptual objects and relations among them, the degree of support for shared interaction is low. These repositories are normally developed in form of time-sharing systems. In particular, they are based on the assumption that conflicts among developers should be delayed as long as possible (Jarke, Maltzahn and Rose 1992), with the consequence of isolating developers from each other.

These limitations and strengths of the reviewed models are summarized (at a system level) in Figure 4.4. To the left we see a scenario that is common for product development environments. In this case each developer is given his own copy of the product to work with. Interactions with the product happen in the private workspace of the developer, and interactions with other developers are not supported. To the left, shared workspace applications support interactions among a (small) group of developers. Interaction across these groups is not supported well, and the groups are isolated from each other.

Central repositories provide a first technological step for supporting shared interaction in product development groups, but more is needed. In the next section we will introduce a new shared interaction model that addresses some of the discussed problems.

4.5 A Product–based Shared Interaction Model

In this section we develop a *product-based shared interaction model* that uses the product, in particular its structure, as a basis for supporting shared interaction among developers in a distributed project. In accordance with the definition of shared interaction in the previous sections, our model contains a shared space and supports awareness of events in this shared space. The shared space in our model consists of two integrated parts. There is an underlying *shared product space* that contains the shared product. The structure of this space mirrors the structure of the product being developed. A shared product space has in this way a structure that is similar to a repository

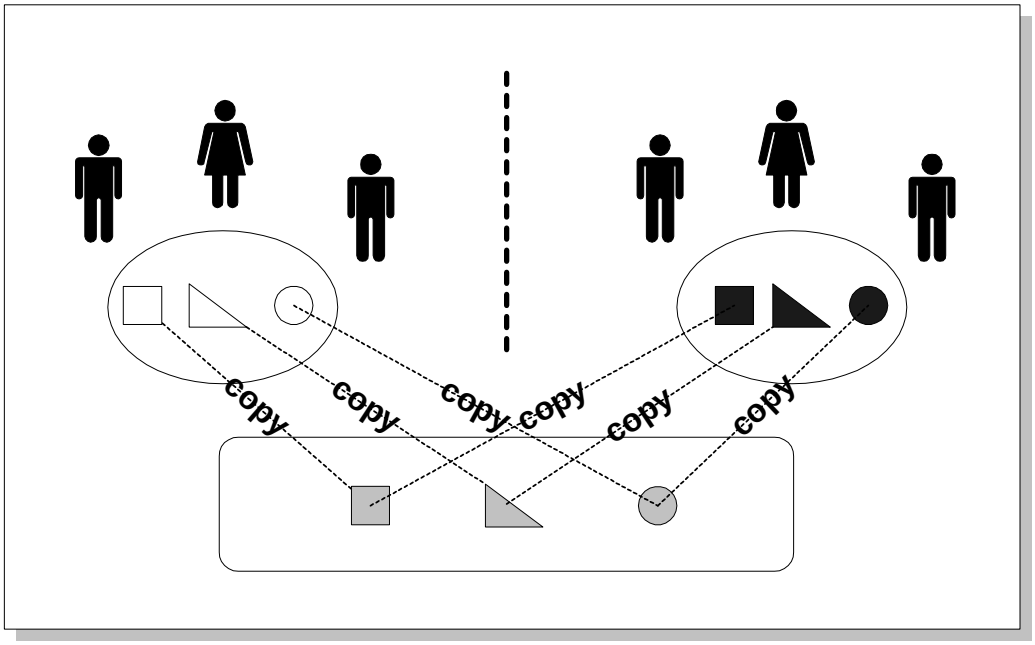


Figure 4.5: The product-based shared interaction model allows groups of developers to work in centers of interaction, and at the same time be integrated through a global shared product space.

in product development environment. However, our focus here is on defining a place for shared interaction, and not on storing different versions of the product. The second part of the shared space consists of *centers of interactions*. These are focus points that allow groups of developers to interact with the shared product space in context of a common task. Centers of interaction are similar to rooms in the room-based model of interaction, but they are integrated with the underlying shared product space. This is shown in Figure 4.5. The shared space in this way integrates the strong aspects of product development environments (i.e. access to a large shared space containing a composite conceptual artifact) with those of room-based systems (i.e. support for shared interaction in a center of interaction).

Awareness in product-based shared interaction model is primarily related to the events related to the shared product, and is called *product awareness*. Product awareness is supported at two levels. First, *product awareness information*, i.e. information about changes to a shared product, is produced according to the contents and the structure of the shared product space. Awareness information is generated according to what objects are within the shared product space, and what relations exist among these objects. In a second level, centers of interaction decide what awareness information will be delivered to which groups. Awareness information is distributed to the developers based on what part of the shared product space they work with, and what other parts of the space they are interested in. Awareness distribution thus need not strictly follow the structure of the shared product space. In this way, the developers in Figure 4.5 can be involved in a common task in a center of interaction, and at the same time be kept informed about what is happening

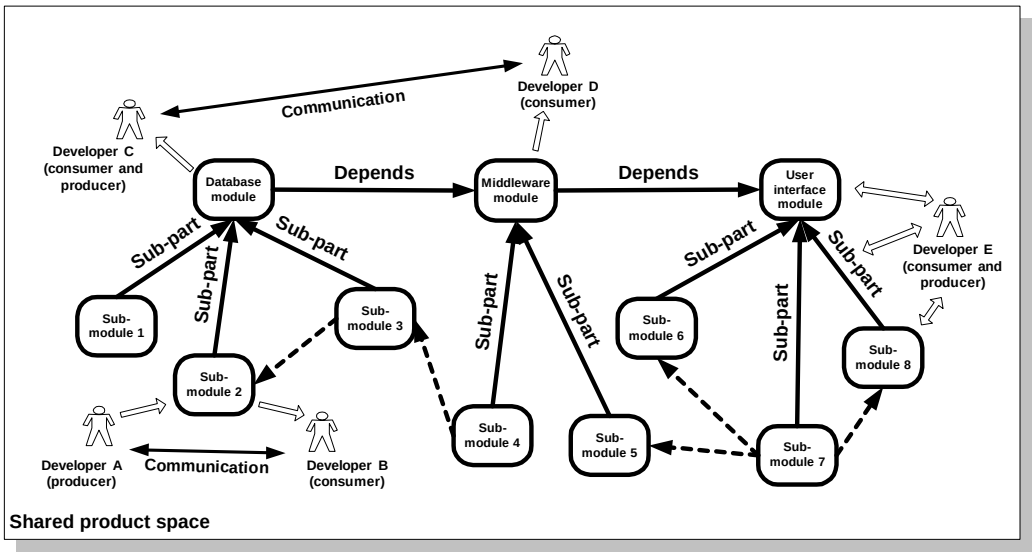


Figure 4.6: An example shared product space. Round rectangles are product objects and arrows are relations. Dotted arrows are awareness relations.

outside the center of interaction (this is indicated by a dashed line separating the two groups, as opposed to Figure 4.4 where the developers were isolated by a solid line).

The following sections describe in more details the shared space, awareness support, and co-operation support as defined in product-based shared interaction model.

4.5.1 Support for shared space

The model introduces a *shared product space* as a shared space within which interactions involving a *shared product* can happen. There are two types of objects that can be embodied in this space, *product objects* and

relations. These constitute the shared product. In addition, developers are represented in the shared product space in form of *producers* and *consumers*. Figure 4.6 shows a shared product space containing product objects (shown as rounded rectangles), relations among these objects (shown as arrows), and developers.

Product objects represent parts of the shared product being developed. Product objects can be documents, diagrams, source code files, etc. The product parts may be stored in a CASE repository, on the Internet, or in a database elsewhere. For each such part, one product object may exist in the shared product space. The embodiment information for each product object contains those properties of the object that are important for cooperation. These properties are not predefined by the model. This means that product objects may be embodied with the simplest embodiment information (e.g. geometric co-ordinates) or more complex conceptual information (such as semantic properties), based on the needs of developer groups.

In addition to product objects, the shared product space may contain relations among product

objects. Relations are unidirectional, and have a *source* and a *destination* product object. Relations have two main functions. First, they represent conceptual relations of different kinds. For instance, the relation from product object "Database module" to product object "Middleware module" in Figure 4.6 is a "Depends" relation, which may mean (for a group of users) that "Database module" depends on "Middleware module." The second function of a relation is *product awareness mediation*. Relations may mediate product awareness from their source to their destination. Product awareness information can in this way propagate from one part of the product to another, possibly through several intermediate parts. This mediation property of relations is not completely connected to their function as conceptual constructs. This means that a relation can be only an awareness mediator, only a conceptual relation, or both. This separation, as we will see in Section 4.5.2, allows more control over distribution of awareness information. For making it easy to distinguish between these two properties of a relation, we will in the following use the term *awareness relation* when referring to a relation as an awareness mediator, and *conceptual relation* when referring to it as a conceptual object.

In the shared product space shown in Figure 4.6, the shared product consists of three main modules. These are represented as product objects "Database module," "Middleware module" and "User interface module." In addition, each module consists of other sub-modules also represented in form of product objects. There are in addition a number of relations created among these product objects. Some of these relations are conceptual relations of type "Sub-part," and "Depends," and some are awareness relations (dotted arrows).

Both product objects and conceptual relations are represented using property sets. The model does not predefine what types of product object or conceptual relation might exist in a shared product space. This is done by the users defining what properties each product object or conceptual relation type may have. In addition, the model does not require that all object and relation types are defined a priori by the users. Users might introduce new types any time, or redefine existing types by adding or removing properties. The configuration of product objects and conceptual relations constitutes the *structure* of the shared product space. The heuristics for creating this structure are not defined by the model. This means that product objects and conceptual relations might be created using any computer-based heuristics, or they may be created in an ad hoc manner by the users.

Once product objects and conceptual relations are embodied within the shared product space, they are subject to shared interaction initiated by producers and consumers. Producers and consumers represent the developers within the shared product space. Producers are those developer who access the contents of the shared product space in order to read, update, or otherwise manipulate them. Producers may also insert or remove product objects and relations from the space at any time. Consumers are those developers who use the awareness information that is produced as a result of producers accessing the shared product space (see next section on awareness). The distinction between producers and consumers is analytical; each developer can be a producer, a consumer, or both. Many developers, such as developer C in Figure 4.6, will be both consumers and producers.

The shared product space implements a global space where all the components of the shared product and all the developers can be involved in shared interaction. The second part of the shared space is *center of interaction*. A center of interaction is a center for a specific activity involving a small number of developers and a small part of the shared product space. A center of interaction has two main functions. First, it allows a group of developers to define a *view* into the

shared product space. Second, it defines the *medium* through which interactions in the center of interaction happen.

The view in a center of interaction gives access to a part of the shared product space without overloading the developers with too much information about irrelevant parts. As opposed to rooms in the room-based model of interaction, or private workspaces in product development environments, views in centers of interaction are not disconnected from the shared product space. all interactions with the shared product space happen *through* the view, and not *within* the view. This means that all the interactions of the developers in a center of interaction are visible to other developers who use the shared product space. In addition, those developers interacting through a view are integrated into the shared product space in that they can see, through the view, what is happening in the shared product space.

A view is in this way a focus point similar to a locale in the locale interaction model used in Orbit (see Section 3.4.3). It gives access to those artifacts and people that are related to each other because they are part of the focus point. As opposed to a locale, a view is connected to the shared product space and is in this way a part of the structure of that space[8]. Another difference between a view and a locale (or a room) is that views are *cooperative*, i.e. they can exchange information. This is illustrated in Figure 4.7. This figure shows two views into a shared product space. Each view contains a small group of product objects and relations. View A is connected to view B through the existing relations among the product objects. This means that developers using A will be notified of some of the activities of the developers using B. In addition, both views will notify their users of other users' activities involving the shared product space.

Views are *customizable*. Product objects and relations within a view can be annotated by the users of the view. We distinguish between *deep* and *shallow* properties of product objects and relations when they are part of a center of interaction. Deep properties are those properties that exist within the shared product space and are independent of how the objects or relations are viewed. Deep properties are the embodiment information of each object or relation in the shared product space. Shallow properties are specific to a view and are visible only when accessing the shared product space through that view. Shallow attributes can be used for annotating the contents of a view, and for customizing the view for different audiences. The shared interaction model does not predefine any shallow properties. This means that the users can decide what shallow properties they want to add to each object or relation, in the same way that they can decide what embodiment information (i.e. deep properties) each object or relation should have.

Beside providing a view, a center of interaction also contains a medium for supporting the interaction with the shared product space. The medium is responsible for supporting the communication among the people involved in the center of interaction. A medium is also responsible for providing proper tools for interacting with the product. A medium can be in different forms. In TeamWave, the medium is a two dimensional room that supports informal cooperation among the users in the room. In MetaEdit+ the medium is the surface of a graphical diagram and the tools used to edit the contents of the diagram. Separating the view from the medium has the advantage that the interacting with the the same view can be supported using different mediums. A view can for instance reside in a document, in a meeting room (through electronic whiteboards), in a Web page, etc. This is very similar to the concept of a locale in that a locale is also disconnected from

[8]Locales in Orbit Gold are also structured, but this structure is very limited and takes the form of a linear list of existing locales. In addition, the activities in different locales are disconnected from each other. Later developments in Orbit Gold may have solved some (but not all) of these isolation problems by employing one global space underlying all the locales (personal communication with Tim Mansfield).

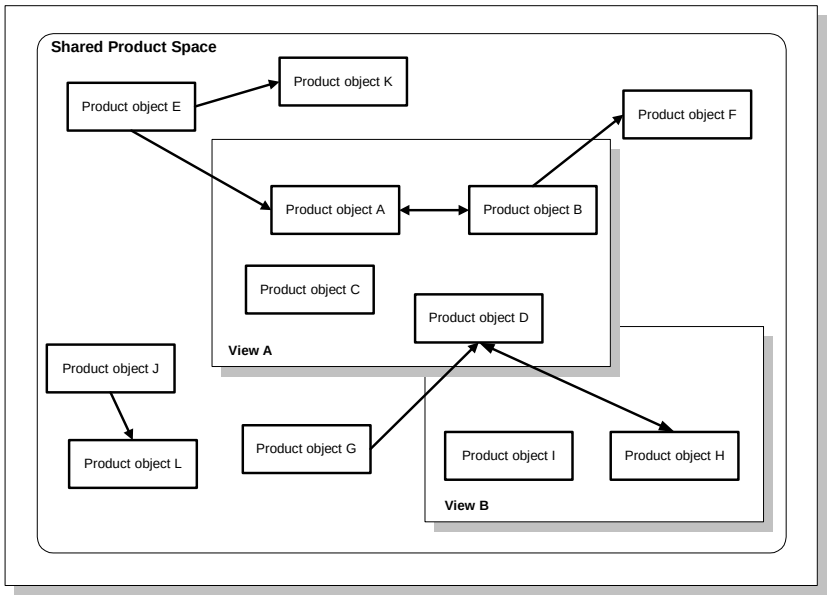


Figure 4.7: Views into the shared product space.

the specific medium (Mansfield et al. 1997).

4.5.2 Support for awareness

The shared product space contains information that is of importance for cooperation among developers. As product development activities proceed, the space is accessed by the developers in different ways. The developers may access the space for reading or updating its contents. All accesses to the shared product space will result in awareness information being produced. The model is responsible for producing this awareness information. There are two types of awareness information. *Product awareness information* is information about access to the product objects and relations in the shared product space. *Participant awareness information* is information about the developers themselves. Product awareness is the main focus of the model.

Product awareness information is produced as a result of any access to the product objects and relations within the shared product space. All product awareness information is discrete and consists of *awareness events*. Awareness events belong to one of two types, *direct* or *mediated*. A direct awareness event is produced by the model whenever a developer accesses a product object or a conceptual relation. This access can be read, update, or any other kind of interaction. For instance, if developer A in Figure 4.6 accesses product object "Sub-module 2," developer B will receive a direct awareness event. A direct awareness event indicates what kind of access was performed on the contents of the shared product space. In addition, the event contains information about who did the access. Mediated awareness events are used by the *awareness mediation* mechanisms for propagating awareness through a product.

An important property of the model is that it supports the propagation of awareness information in the shared product space. The majority of the shared interaction models we have seen supports awareness of what each user is currently focusing on. Peripheral awareness is in this way limited to the current focus of the user (Simone and Bandini forthcoming). As we have seen in Chapter 2, products are often large and have complex structures. Often, the part of the product that each developer is working with resides within a web of interdependencies. Propagation of awareness information guarantees that users can receive awareness information about remote parts of the product. Awareness information can propagate from one part of the product to another following a path containing awareness relations among product objects. In this way, a group of developers working on one part of the product can get the proper awareness information from any other part without explicitly focusing on that part.

Developers sharing the same product objects and conceptual relations can exchange direct awareness events related to these objects. Moreover, direct awareness events can be propagated to other product objects. Propagation of awareness inside the shared product space happens through the awareness mediation mechanism . A mediation starts as a product object X broadcasts its awareness information (i.e. direct awareness event that is produced as a result of a developer accessing that object) to all the product objects in the shared product space. A product object Y that is interested in X will produce a *mediated awareness event* based on the broadcasted event (Y will be interested in X if there is an awareness relation from X to Y.) The mediated awareness event produced by Y contains the same information as the original direct awareness event, except that its originator product object is changed to the new product object, i.e. Y. However, the mediated awareness event also contains a pointer to the original product object X. Mediation from a product object X to a product object Y is possible only if a *mediation path* consisting of (possibly several) product objects and awareness relations exists between X and Y. *Recursive mediation* happens when an awareness event is mediated more than once.

As an example consider developers A and E in Figure 4.6. As developer A accesses product object “Sub-module 2,” the access will result in a direct awareness event (which is sent directly to developer B). This direct awareness event might be mediated through a mediation path consisting of product objects “Database module,” “Middleware module,” “User interface module,” and the relations among them. Developer E will then receive a mediated awareness event that not only tells him who accessed which product object, but also which product objects that were in the awareness path from developer A to him.

Not all awareness events need to be mediated by all awareness relations. There are two ways to control the mediation mechanisms without explicitly removing awareness relations. First, each awareness relation is assigned a set of *interaction types* that it is allowed to mediate. This makes it possible to have an awareness relation that only mediates update accesses to its source product object. For instance, in the example above developer E may not be interested in knowing who is reading “Sub-module 2” but may be interested in changes to it. The users can define and change the set of access types for each awareness relation in order to configure the production of mediated awareness events.

As a further means for configuring the amount of mediated awareness information generated by the model, each mediation in a mediation path is checked against a *strength factor*. Each awareness relation has a strength factor that can be changed by the users. Before a mediated awareness event is mediated further, the awareness relation’s strength factor is compared with the number of times the mediated awareness event is already mediated. If this number is equal to

the strength of the relation, the mediation stops. This feature, in combination with the interaction types mentioned earlier, can be used to configure the generation of awareness information to fit the different needs of the users. For instance, in Figure 4.6, user E might not be interested in knowing every access to "Sub-module 1." In this case, the relation from "Middle-ware module" to "User interface module" can be given a strength of 2. This will prevent "User interface module" from producing any mediated awareness related to "Sub-module 1."

Criteria for deciding how awareness relations should be created among product objects is not defined by the model. In some occasions the conceptual structure of the product may be suited for awareness mediation. In such cases all relations in the shared product space may be both conceptual and awareness relations. An example is dependency relations among source code files in a software product. In this case, the structure of the product (defined by the conceptual dependency relations) strictly defines the dependencies among the developers. This means that if a source file A imports another file B, A not only is conceptually related to B, but also the developers working on A will need to be aware of changes to B in order to coordinate their work.

Relations might as well be created based on social relations among the developers themselves. This will be more common in earlier phases of a project, where the product itself and its structure are not defined in details. In such cases it may not be possible to define conceptual relations, but one might need awareness relations for keeping oneself aware of what others are doing with their parts of the product (see Figure 2.6 on page 32). It is not possible for a model to predict all the possible criteria since they will be influenced by the way different groups do their work. These criteria are therefore left to be defined by the users of the model. The shared interaction model and its implementation focus only on providing easy mechanisms for modifying the structure of the shared product space, and for configuring the mediation of awareness as needed.

Centers of interaction make use of the product awareness (both direct and mediated). When a group of developers are working in a center of interaction, the model supports the delivery of exactly the same product awareness to all the members of the group. This is done through the view part of the center of interaction. For instance, in Figure 4.6, developers A and B can define a view that contains "Sub-module 2." In this way, they can interact with the shared product through this view. The view is responsible not only for providing the same picture of the product to both A and B, but also delivering the same awareness information. This is useful in a shared workspace (e.g. in a shared room) where an important prerequisite for cooperation is having access to the same information and being aware of the same events.

Participant awareness is not the focus of the model. However the model supports combining participant awareness with product awareness in two ways. First, all product awareness events that are produced by the model contain information about the developer who was responsible for the event. Second, developers can be embodied in the shared product space by becoming a producer or consumer. Each developer who is already a producer or consumer will get participant awareness about other developers becoming producer or consumer, or leaving the shared product space.

4.5.3 Support for cooperation

The product-based shared interaction model supports opportunistic communication among developers by allowing them to see who is doing what to the shared product. This is done by explicitly informing each developer about the events related to the product, and by connecting each product awareness event to the developer who was responsible for it. In addition, the model supports a limited notion of participant awareness by producing awareness events when developers enter or

leave the shared space. These features of the model can be used in different ways to support a variety of communication patterns among the developers.

Moreover, cooperation in small groups is supported by centers of interaction. These centers provide a shared context for a group, and give access to shared information and a medium for interaction. However, the model does not predefine the overall course of the interactions. For example there is no coordination among the different centers of interaction besides what is promoted by the structure of the shared space. Coordination in terms of workflow and software processes is external to the model, and can be done in different forms according to the preferences of the users. What model provides is the information basis for supporting different forms of coordination.

4.6 Summary

In this chapter we have defined a shared interaction model for product development. The model combines the strong aspects of existing shared interaction models and product development environments. It emphasizes the importance of the product as a resource for cooperation in the large project team, and at the same time acknowledges the existence of close and dynamic cooperation within groups of developers. The combination of a shared product space and centers of interaction is in line with the requirements developed in Chapter 2 and shown in Table 2.5 on page 43. Locating the product in a shared space allows the developers to access the product easily, and to be aware of the other developers interactions with the product. This can facilitate both short term coordination of actions, and long term cooperative learning in distributed product development projects.

The product-based shared interaction model is realized in form of a framework for shared interaction in product development environments. In the next chapters we will introduce this framework in details, and will see how it can be used to build cooperative systems for supporting distributed product development projects.

Chapter 5

The IGLOO Framework: Overview and Examples

5.1 Introduction

In the previous chapter we developed a product-based shared interaction model for cooperative product development. This model consists of a shared product space and centers of interaction. The model was developed based on requirements posed from several sources, including our experiences from the ALPHA project and our earlier prototyping efforts. This and the following chapters describe an architectural framework that realizes the product-based shared interaction model. The framework is called *IGLOO*. The relations between IGLOO and the shared interaction model are shown in Figure 5.1. The round rectangles in the upper part of the figure denote the shared interaction model and its sub-parts. The middle part of the figure (the white rectangles) shows the different parts of the IGLOO framework. IGLOO framework consists of three parts called *service layers*. *Product Layer* is a detailed specification of the shared product space, while *Cluster Layer* and *Workspace Layer* together provide a specification of the centers of interaction. Workspace Layer in particular provides the medium for a center of interaction in form of shared workspaces, while Cluster Layer is the provider of product-related information to a center of interaction. This is in accordance with the product-based shared interaction model, where a center of interaction contains a *view* into the shared product space and a *medium* for supporting the interaction in the center of interaction. The advantage of separating the medium from the informational contents of a center of interaction, as we have done here by having Cluster Layer and Application Layer specify each of them separately, is that contents and medium for the center can be applied separately. For instance, a center may need to use a different medium than what an implementation of IGLOO supports, and still use the contents provided by IGLOO. This makes it possible to use IGLOO in combination with other existing advanced shared workspace application.

For each of the service layers in the framework we have developed a *generic implementation* in the Java programming language. These implementations are in form of stand-alone network servers with their own well-defined protocols and services. The implementations are shown as

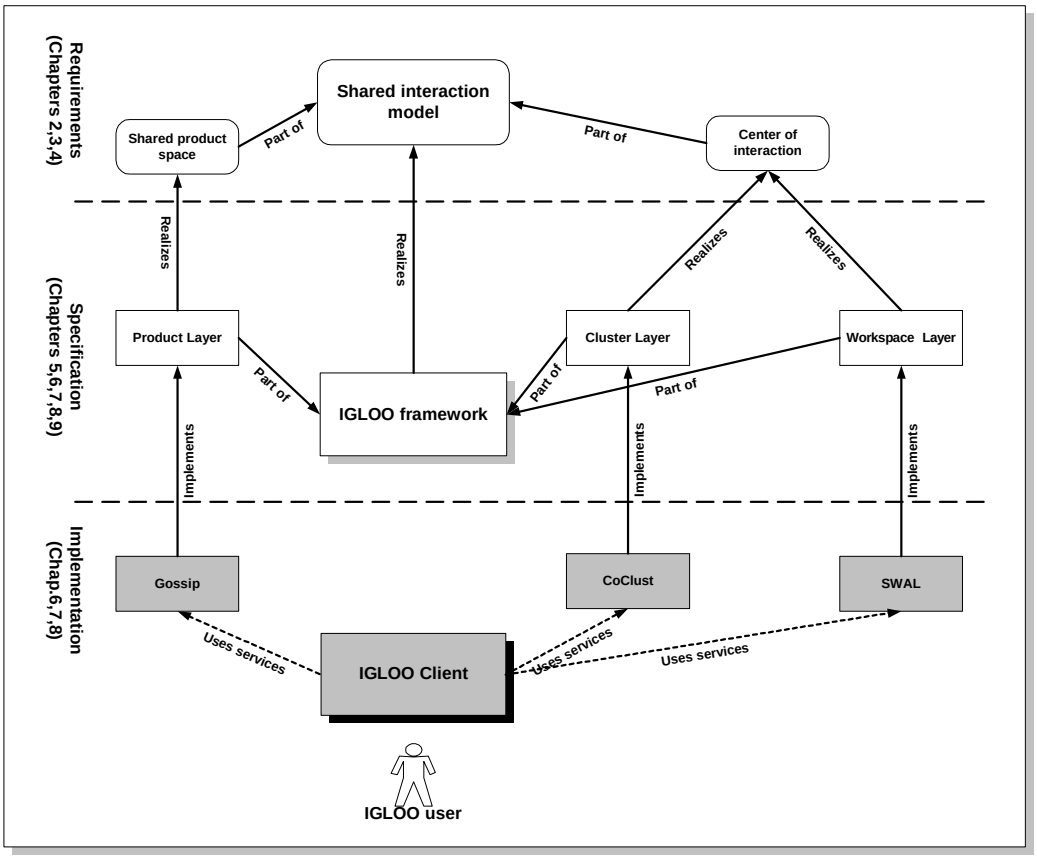


Figure 5.1: The relations between concepts, specifications, and implementations underlying the IGLOO framework.

gray rectangles in Figure 5.1. The services offered by each implementation are used by different *IGLOO clients* (lowest part of Figure 5.1). IGLOO clients are software applications that cooperate with each other through the IGLOO framework. An IGLOO client does not need to make use of all the services provided by IGLOO, and may choose to use any combination of the services provided by the three service layers. This is indicated in Figure 5.1 in form of dashed lines.

In this chapter we provide an overview of the IGLOO framework and its service layers, and describe an example IGLOO client. The structure of this chapter is as follows: Section 5.2 provides an introduction to the different service layers of the framework, shows how the layers are connected to each other, and describes their relation to the shared interaction model. In Section 5.3 we introduce *MultiCASE*. MultiCASE is a shared graphical editor for creating architecture diagrams. MultiCASE demonstrates the usability of the framework, and how the different layers of the framework can be used for developing group support applications. The service layers of the framework are described in more details in the chapters following this chapter. Section 5.5 provides a map for reading these chapters.

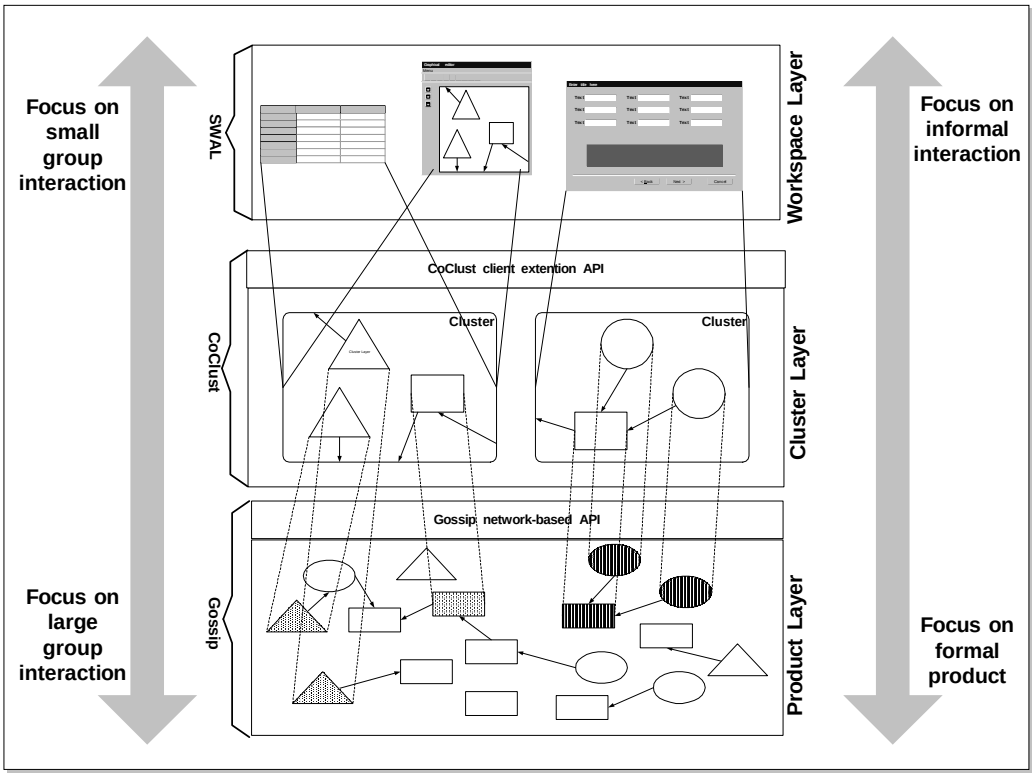


Figure 5.2: The three layers in the IGLOO framework, each implementing a set of service that are provided to the layer above.

5.2 The IGLOO Components

IGLOO framework employs a layered architecture. The core services of the framework are provided through three *service layers* (see Figure 5.2). Each layer provides a set of *services* to the layers higher up in the framework. The lower part of the framework is concerned with the creation and maintenance of a large shared product space used by all the developers, and in this way should be seen as a common information space (Bannon and Bødker 1997). This means that the shared product space should be understandable and predictable, i.e. be formal. This formality is not the same as the formality required for a product as the basis for code generation. It is merely a formality that is called upon because a large number of people with different backgrounds and contexts will use the product (see also Section 2.1 on a definition of product). The product objects within the shared product space should therefore be packaged for a larger audience, their contents and changes to them should be understandable to all the users, and the changes to the shared product space should be understandable to all the involved people (Bannon and Bødker 1997). The shared product space should support the articulation of work within large groups (Schmidt and Bannon 1992).

The higher levels of the framework, on the other hand, are concerned with small groups and their local needs. They support the creation of centers of interaction. Formality in the sense discussed above is no longer the main priority. What is needed is a focus point within the shared product space, and a medium for supporting rich interaction among a group of people involved in solving a specific task. Formality is not as important because each local group will eventually develop their own local understanding of the shared product space. Formality is even restricting with regard to cooperation because it will hinder the emergence of local understanding that is important for the focused task of the small group. The speed of changes made to the product objects by a small group will inevitably be very high due to the focused work on these objects, making the achievement of formality further impractical. What is needed here is mechanisms for customizing the common information space to local needs and interpretations (Bannon and Bødker 1997). Cluster Layer provides local and customized views of the shared product space, while the Workspace Layer provides the medium for interaction among the developers and for the interaction between the developers and the shared product.

The unifying factor between the small group and the large group, between the formal and the informal, is *product awareness*. Product awareness is information about user access to the shared product space. This information might be needed in different forms depending on the context of cooperation. For example, for two groups working with two dispersed parts of the product what may be of importance is the changes done to the product by the other group, i.e. the *what* of awareness. Knowledge of *who* in the other group did the changes may be of secondary importance. However, this knowledge of who did the changes may be of primary importance when the changes are done by the members of the same small group during the course of a task. The knowledge of who may for instance be used as a resource for fine-grained coordination of the work within a shared workspace (Gutwin and Greenberg 1999). In IGLOO, each unit of product awareness information is connected closely to the developer who produces it. As we will see later in this chapter, IGLOO clients can customize their use of product awareness information based on their local needs.

The three service layers of IGLOO framework are developed to support both large and small groups cooperating to develop a product. A short description of each layer and the types of services they provide is given here:

- *Product Layer* (explained in details in Chapter 6) is in charge of maintaining a shared product space. The main abstractions supported by Product Layer are *shared product space*, *product objects* and *relations*. Shared product space is a virtual space where a group of cooperating users can make available and share their product objects. Product Layer provides services for inserting new product objects into the shared product space, for modifying the presence of the objects in different ways, and for creating arbitrary relations among the objects. Product Layer should not be mistaken for a repository. Product Layer is not meant for storing and processing product objects (as is normal in a CASE repository), but merely for *sharing* them. As such, it does not focus on services that are common in these repositories, such as version control and consistency preservation (Brown et al. 1992). Product Layer supports partial sharing of product objects. This means that the users decide how much and what aspects of a product object they want to share (normally those aspects that are necessary for the cooperation). The relations among the product objects are generic relations that can be specialized into product-specific relations (such as part-of or dependency relations) or to socially-defined “interest” relations. Product Layer actively provides its users with

awareness information about access to the shared product space. Product awareness provided by Product Layer is in a basic form (i.e. *units* of awareness information, or *product awareness events*). The basic awareness services of Product Layer are further customized for different contexts by the higher levels of the framework. Awareness events are provided based on the shared interaction model, and support both direct and mediated awareness (see Chapter 4). In addition, awareness events provided by Product Layer are connected to the developers who produce the events. In this way Product Layer supports opportunistic communication among awareness producers and consumers.

- *Cluster Layer* (explained in details in Chapter 7) is the intermediate level between a large shared product space and small groups interacting with this space. The main abstraction provided by Cluster Layer is *cluster*. Clusters supply centers of interaction with shared product information. Clusters are user-defined collections of product objects from the shared product space (Product Layer) that are considered by a group of users to be important for performing a task. Cluster Layer allows its users to create clusters, and to customize the clusters' contents (i.e. which objects that are part of the cluster) and form (i.e how the objects should be represented). In addition, Cluster Layer allows a group of users to share a cluster and its content, and to have access to product awareness that is generated by Product Layer. Clusters are created in a way to provide both focus (by selecting only a subset of existing product objects from the shared product space and hiding the other objects) and overview (by allowing the users to monitor product objects external to a cluster).

- *Workspace Layer* (explained in details in Chapter 8) provides a medium for informal cooperation in small groups of users working on a focused task. The main abstraction provided is a *shared workspace*. A shared workspace provides the medium for a center of interaction. Each shared workspace in IGLOO may consist of any number of clusters, *informal objects*, and *inhabitants*. Informal objects support the informal interaction among a group of inhabitants within the workspace. Informal objects might be notes, documents, tools, etc. An IGLOO shared workspace is provided as a basic construct, and may be specialized to support one or several types of group processes such as brain-storming, decision making, editing, discussions, etc., or be used as a flexible interaction medium (e.g. as a room). The same IGLOO shared workspace might be viewed in different forms, e.g. through a Web page or in an interactive graphical editor. This is due to the conceptual representation of the workspace in IGLOO that is independent of any interaction techniques. A shared workspace has in addition access to the shared product space through the clusters within the workspace. In this way the underlying shared product space is used as a unifying component among all the shared workspaces. Each shared workspace can decide to visualize the clusters and the objects from the shared product space in different forms. For instance, a graphical editor will visualize a cluster in form of a graphical diagram, while a browser tool will use a simpler outline. In addition, each shared workspace might provide tools for modifying the objects from the shared product space, and tools for supporting generic group processes such as communication through text, video, audio, etc.

As part of the specification of the layers, the services necessary for each layer are identified and specified in details. This modular approach with well-defined interfaces makes it easy to customize the functionality of the framework for different types of IGLOO client. For instance, one client may use only the services provided by Product Layer, while another may in addition

use the services of Workspace Layer. The framework allows for such configurations to co-exist and cooperate with each other.

Moreover, the definition of the services in form of service layers is independent of any design or programming languages. Each layer can be implemented in different ways, but must adhere to service definitions and provide the services to its clients. During the definition of the framework, we have implemented a generic implementation of each service layer. The design of these implementations, together with the API (Application Programmer Interface) provided by each of them, are described later in the thesis.

5.3 An Example IGLOO Client: MultiCASE

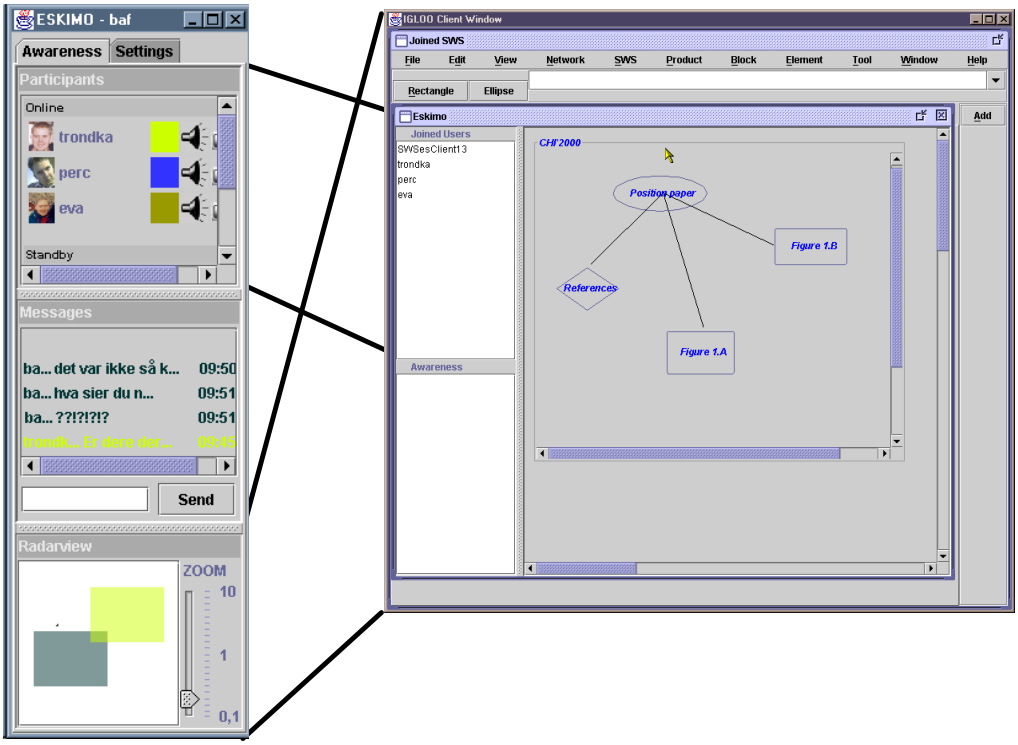
In this section we will describe a graphical multi-user editor called *MultiCASE*. MultiCASE is an IGLOO client as shown in the lower part of Figure 5.1. It uses the services provided by all the three layers of IGLOO. MultiCASE was developed for testing the usability of the IGLOO framework. It demonstrates the types of clients that can be developed using the services provided by IGLOO framework. MultiCASE is a simple graphical editor that allows a group of developers to cooperatively develop a product. The developed product is a software architecture consisting of software modules and dependency relations among them. The shared product space consists thus of software modules (as product objects) and dependency relations (as relations). Using MultiCASE, a group of geographically distributed developers can cooperatively develop a possibly large product consisting of interdependent software modules. MultiCASE allows the developers to create shared workspaces and to meet and cooperate with other developers in these workspaces regardless of existing geographical distances. The shared workspaces provide virtual places for cooperation where the developers can create and modify parts of the product being developed. A shared product space provides a high degree of sharing, meaning that all the changes to the product are immediately available to all the interested developers. In this way MultiCASE increases the visibility of work despite geographical distances, and provides centers of interaction for groups of geographically distributed developers.

5.3.1 Meeting in a shared workspace

After connecting to a MultiCASE server, the user will enter into a default shared workspace (i.e. the one he was working in last time). The user can choose to view this workspace in two different ways, with the possibility for easily switching from one view to the other. These two views are shown in Figure 5.3. The window to the left is the *monitor window*. Monitor window is used when the user is not actively using MultiCASE. It can be used as a background window that is always "on" (e.g. activated upon logging into the computer). It can be minimized or stay in the background, and will constantly monitor the shared workspace for various activities. It generates sound signals when something happens, e.g. when another user enters the workspace or when the contents of the workspace are accessed by other inhabitants[1]. In this way the monitor window will provide continuous awareness to the users without taking too much attention away from what they are currently doing.

The monitor window can be used for informal communication with the other inhabitants of the workspace. The window contains information about the inhabitants, including an image and

[1]The term *inhabitant* is used to denote all other users currently in the same workspace as this user.



A) High participant awareness. Low product awareness

B) Low participant awareness. High product awareness

Figure 5.3: Choosing between background monitoring mode and focused task mode in Multi-CASE.

an indication of what communication tools each inhabitant is equipped with (upper part of the window in Figure 5.3.A). A speaker icon means that the user can communicate using audio. This information facilitates opportunistic communication by providing easy access to available users. As the inhabitants of the workspace change, monitor window generates sound signals to notify the user. The middle part of the window is a text area used for chatting and for receiving system messages. The inhabitants can use this area for sending instant messages to each other. Each type of message (e.g. chat or system message) can be given its own priority, in this way adjusting the level to which the message should interrupt the inhabitants. The lower part of the window shows a radar view of the shared workspace, with the location of each of the inhabitants in the workspace. The locations are shown in form of rectangles colored with each inhabitant's characteristic color.

The window in Figure 5.3.B is the *editor window* and is the second type of view into the same shared workspace. This window can be activated by double-clicking on the radar view in monitor window. Editor window is used for focused cooperation in the workspace, and is where all the development activities happen. It provides functionality for modifying the product (the software

architecture and its modules) and for managing shared workspaces. There are three main areas in editor window. The middle part of the window (the large gray area) is the *active* shared workspace. A user can be in several workspaces simultaneously. These are called *joined* workspaces of the user. The active workspace is the one joined workspace that the user is actually working in. Other *non-active* joined workspaces are hidden behind the active workspace and can be brought to front (activated) by clicking on their tab. In addition to the joined workspaces of a user, there are a number of other workspaces in the system that are not visible to the user (because he is not currently interested in them) but can be joined at any time. This is explained later.

Inside the active shared workspace there is a graphical diagram showing a part of the software architecture the user is currently working on. The graphical shapes and lines within the diagram denote software modules and dependency relations among them. The gray area outside the diagram is the *informal part* of the shared workspace. This area is used for informal cooperation among the inhabitants, e.g. for leaving short messages and notes, and exchanging informal documents and other artifacts that are not part of the product (these artifacts are called *informal objects*[2]). The informal area is larger than the *current view* of the user, and can be navigated using the scroll bars of the shared workspace. The views of the clients are not connected to each other, so each user might independently from other users navigate to a different part of the shared workspace[3]. The small window on the top-left corner of editor window labeled "Joined Users" shows the names of all the inhabitants of this shared workspace. In Figure 5.3, in addition to the current user, three other users called "trondka," "perc" and "eva" are present in the workspace.

The contents of the editor window, i.e. the diagrams and their contents, can be modified through direct manipulation. As a user changes the contents by moving them around or modifying them otherwise, all the inhabitants of the workspace are notified through real-time synchronization of their screens. This continuous flow of visual clues is a part of what Gutwin et al. (1996) call *shared workspace awareness*. Besides the shared workspace awareness, MultiCASE also provides *monitoring* facilities for allowing the users to monitor the activities of other users in other workspaces as far as those activities change the shared product. The window on the lower-left corner in editor window labeled "Awareness" shows awareness of the product-related activities outside current shared workspace (i.e. *mediated product awareness*, see Chapter 4). This area is used for receiving mediated awareness events about parts of the software architecture that are external to this workspace.

By allowing easy switching between the two views (monitor and editor windows) we gain the advantage of continuously keeping the users within a shared context without requiring their full attention when they are busy with other tasks. MultiCASE can become an integrated part of the desktop of a user. The user can have the monitor window in the background while he is working with other things, and switch to the editor window when he wants to work with the product and cooperate with the other users. In this way, the user gets continuous background awareness about what the others are doing, and is all the time available for being contacted by them. This is similar to working in the same room with other people. Sometimes you work on your own, while other times you switch to cooperative mode unconsciously. This form of continuous background awareness has proven to be important for supporting long term cooperative learning (Mackay 1999).

[2] Informal objects are not implemented in this prototype but are part of IGLOO framework. See Chapter 8.

[3] This is called relaxed WYSIWIS (What You See Is What I See) as opposed to strict WYSIWIS where the screens and the views are fully synchronized (Lauwers and Lantz 1990).

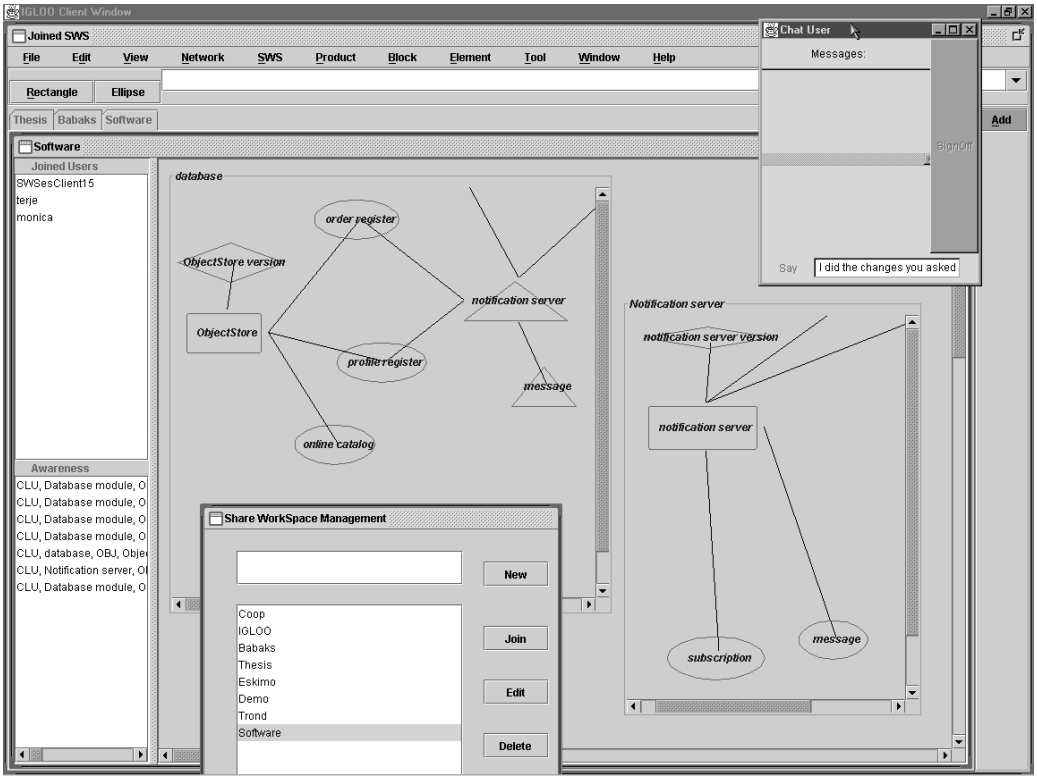


Figure 5.4: The user interface of the editor window in MultiCASE.

Besides the joined workspaces of each user, there might be a number of other workspaces in the system, possibly created by other groups of developers. Once shared workspaces are created they are stored in a *shared workspace database* and are accessible to all the users who want to enter them. The *shared workspace management tool* gives access to all the existing shared workspaces, and allows the users to create, delete, join other workspaces. Figure 5.4 shows a scenario of a user browsing the existing workspaces using the shared workspace management tool (the small window in lower-front). Figure 5.4 also shows how several joined workspaces are organized in form of layered windows. In this figure there are three joined workspaces called “Software” (the active one in the front) “Thesis,” and “Babaks.” Only the active workspace is visible. However, if something happens in a non-active shared workspace while the user is working in the active one, the tab belonging to the non-active workspace changes its color to indicate the activity.

Communication among the inhabitants is supported by a chat tool (the window in the upper-right corner in Figure 5.4). In addition, sound communication is supported when all the users are in the monitor window.

5.3.2 Editing the product

Editor window is used not only for meeting and communicating with other users, but also for editing the shared product. This is done by importing parts of the product into a workspace in form of graphical diagrams. Inside the "Software" shared workspace in Figure 5.4 there are two graphical diagrams called "database" and "Notification server." These diagrams contain different shapes and lines. The shapes represent software modules from the shared product space, and the lines are dependency relations. All diagrams are stored in a database and are accessible to all the users. When creating a diagram, the user can choose among already existing diagrams or create a new one. These choices are available from a window menu, or through a pop-up menu by right-clicking on the informal area of a shared workspace. Several diagrams can reside within the same shared workspace, and any diagram can be modified by any inhabitant of the workspace. All the modifications to a diagram are visible in real time to all the other inhabitants. Diagrams in MultiCASE are in this way quite different from conventional diagrams in CASE tools: MultiCASE diagrams can be shared (in real time) among a group of users.

Shapes in a diagram refer to software modules constituting a software architecture (remember that the shared product in MultiCASE is a software architecture, and the product objects are the software modules making up the architecture). The modules can be modified through direct manipulation of the shapes in a diagram. When creating a new shape in a diagram, the shape can be put to refer to an existing module from the shared product, or to refer to a newly created module. New modules are immediately created and stored as part of the shared product and can be accessed by others. In Figure 5.4 the diagram called "database" has a number of shapes in it. Each shape in "database" refers to a software module in the shared product. The modules "notification server" and "message" (shown as triangles in diagram "database") are also used in another diagram called "Notification server." It is important to note that even if the shapes represent the same modules from the same shared product, their use as shapes in the two diagrams can be quite different. As software modules, some of their properties are global. For instance, changing the name of one shape (e.g. "notification server" in diagram "database") will change the name of all the other shapes referring to the same module (including "notification server" in diagram "Notification server"). On the other hand, changing the line style of one shape will not affect the other shapes referring to the same modules. For instance, software module "notification server" is shown as triangle and rectangle in the two diagrams in Figure 5.4. This allows for some amount of customization within each diagram, while sharing the changes to the global properties. Those attributes of shapes that are local to a diagram are called *shallow properties* (e.g. geometric shape, position in the diagram, color, etc.). Those attributes that are global to the shared product, i.e. belong to the actual software modules, are called *deep properties* (such as name, version number, module type, etc.).

A diagram and its contents can be concurrently modified by all the inhabitants of the workspace. Shapes in a diagram can be moved around using the mouse, and their attributes such as their name, owner, contents, etc. can be changed through direct manipulation and through dialogue boxes. Figure 5.5 shows a properties dialogue box for a shape called "subscription." The properties of the shape include deep properties such as name, owner, type, and version (the left part of the property window in front lower-left corner in Figure 5.5), and shallow properties such as shape (the left part of the properties window in Figure 5.5). In addition, for each software module it is possible to attach a file that contains the contents of the module (e.g. a source code file). Content files belong to deep attributes, i.e. they are stored in the shared product space. Figure 5.5 shows how a file

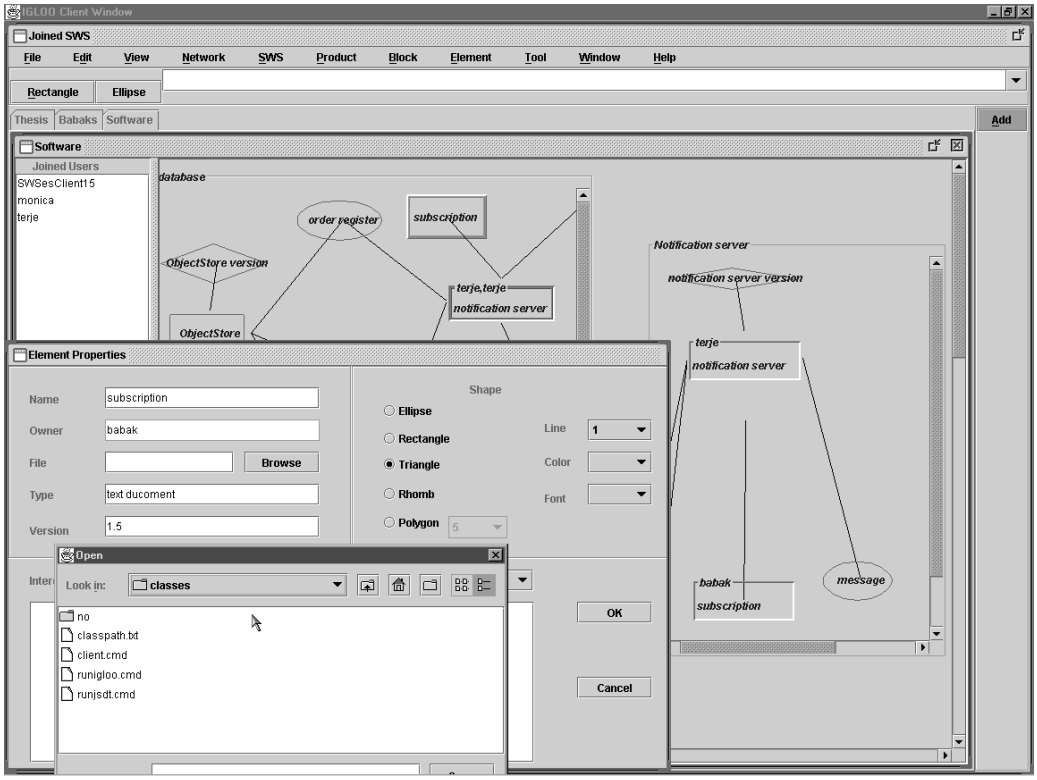


Figure 5.5: Changing the properties of a product object in MultiCASE.

from the local file system can be uploaded into the shared product using a file open dialogue box.

Concurrent access to diagrams and their contents requires concurrency control for preventing illegal sequences of user actions. Concurrency control is implemented in MultiCASE using *identified locks* (Mariani and Prinz 1993). When a shape is selected (i.e. when a user clicks on it), it acquires a *shallow lock* that allows the user to modify its shallow properties. Shallow locks are diagram-wide (i.e. one lock per shape in a diagram) and are identified, meaning that other inhabitants of the workspace can see that the shape is locked, and can see the name of the lock holder on the shape. In Figure 5.6 the shape called “message” inside the diagram called “database” has a shallow lock labeled “monica.” It is a shallow lock because another shape referring to the same module (i.e. “message” in diagram “Notification server”) is not locked. This identified lock allows the user to see that user “monica” is modifying a shallow property of the shape, and the user is not allowed to change the shallow properties of this shape as long as the lock is held by user “monica.” However, the other shape called “message” in diagram “Notification server” can be modified freely.

Unlike shallow locks, which are acquired implicitly when a shape is selected by clicking on it, *deep locks* are acquired explicitly by selecting a shape and choosing a menu entry (through a pop-up or window menu). Deep locks have to be acquired before modifying any deep property of

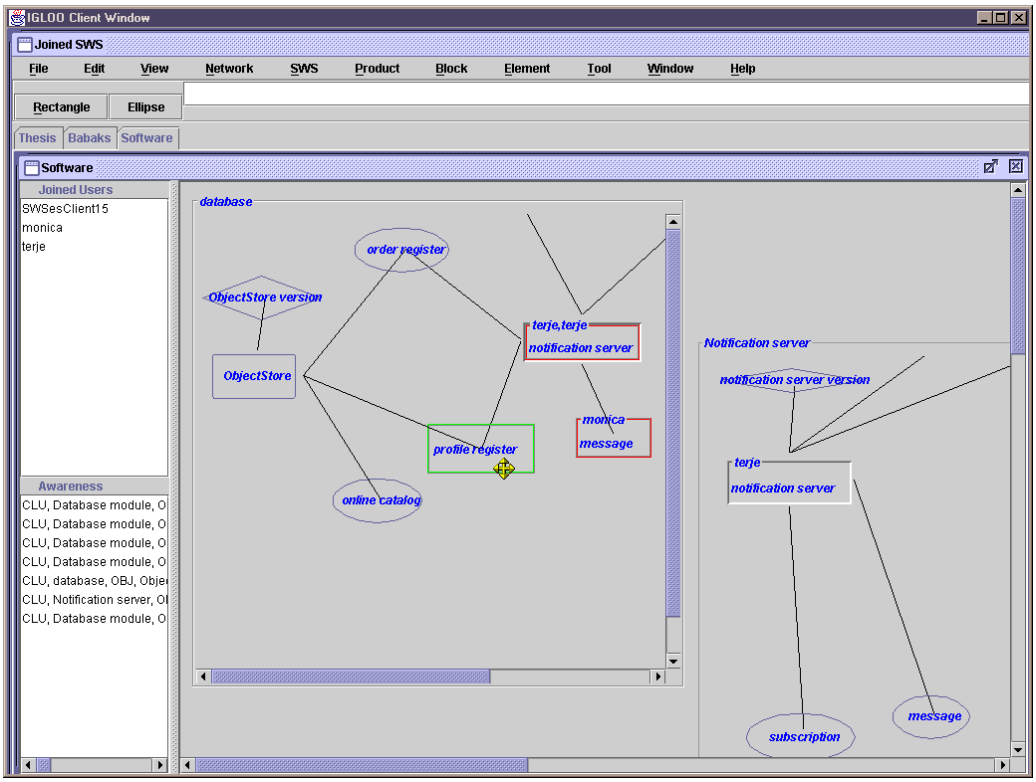


Figure 5.6: Element-based locking in MultiCASE allows high degree of flexibility.

a shape (i.e. for changing the underlying software module), and are unique throughout the shared product space (i.e. only one lock per software module). Deep locks are also identified, with one important difference: they are visible across diagrams. This means that if the same software module is represented by several shapes in different diagrams, once a user in one shared workspace acquires a deep lock all the diagrams containing a shape that refers to that software module will show that the module gets locked and by whom. Once a deep property (such as the name of a module) is changed, the changes are immediately visible in all the diagrams. In Figure 5.6 one can see that user “terje” has acquired a deep and a shallow lock on shape “notification server” in diagram “database” (the name “terje” appears twice on this shape) and that the deep lock has also locked the shape “notification server” in diagram “Notification server.” (One can see it is a deep lock because the shape is shown three-dimensional, in contrast to the shallow lock on shape “message” in diagram “database.”) However, the shape in diagram “Notification server” is not shallow-locked, and its shallow attributes can be changed by other users.

This combination of shallow and deep properties and locks has the advantage of providing the users with a high degree of flexibility and local customization. Shallow properties can be changed without being visible outside the shared workspace, and can be used to represent the same software modules in different ways for different types of user. Flexible locking allows users

to work in parallel on different software modules in a diagram. For instance, in Figure 5.6, users "monica" and "terje" are each working with one software module while the current user is working on the third (called "profile register" with a cross icon on it, indicating that the module is being moved), all concurrently. In addition, other users in other shared workspaces might change other shallow properties of the same software modules at the same time without interfering with the inhabitant of this workspace.

5.3.3 Interacting with composite products

Software architectures that are created using MultiCASE typically grow large and eventually will not fit within one shared workspace. MultiCASE allows its users to create multiple shared workspaces, and to work with different parts of the software architecture in different workspaces. Using MultiCASE it is possible for a group of developers to focus on one small part of the architecture, and at the same time to monitor the other parts of the architecture. This is a strong point of MultiCASE compared to other shared workspace applications that we have seen in Chapter 3, where the users are completely isolated from the activities outside their own workspace. Monitoring the product in MultiCASE is done using dependency relations.

MultiCASE allows its users to create dependency relations among software modules. A dependency relation is created explicitly by specifying its source and destination software modules. All dependency relations in MultiCASE are created as part of the shared product, and are visible to all the developers working with the same product. For creating relations, at least the destination software module has to exits in form of a shape in a diagram in the active shared workspace of the user. When both source and destination software modules are present as shapes in a diagram, relations can be created by direct manipulation, i.e. by clicking on the destination shape, choosing an item from a pop-up menu, and dragging a line to the source shape (Figure 5.7). In cases when only the destination module is present, a dependency relation can be created by selecting the destination, and choosing a source from a list of available software modules. Existing relations are visualized directly in the surface of a diagram as lines connecting the shapes. Lines leading to outside a diagram are dependency relations connected to software modules that are not currently represented in the diagram in form of shapes. For instance shape "notification server" in diagram "database" in Figure 5.7 has a line leading to the outside of the diagram. This is because the software module represented by the shape has a relation to another software module "notification server version" that is not currently inside the "database" diagram (the relation between "notification server" and "notification server version" can however be seen in diagram "Notification server" in Figure 5.6). Moving the mouse pointer on to this line displays the name of the external software module. Once a new shape is added to a diagram, the lines are updated to show the current configuration of relations. In this way the users can constantly see the dependencies among the parts of the product.

All modifications to the source software module of a dependency relation will generate a notification that will be sent to its destination software module. This is how the users can monitor changes to the remote parts of the software architecture. External notifications that are relevant for a diagram are displayed in the "Awareness" window in editor window (the window in the lower-left corner in Figure 5.7). These external notifications are useful if the user does not want his workspace to be overloaded by a large number of shapes and diagrams.

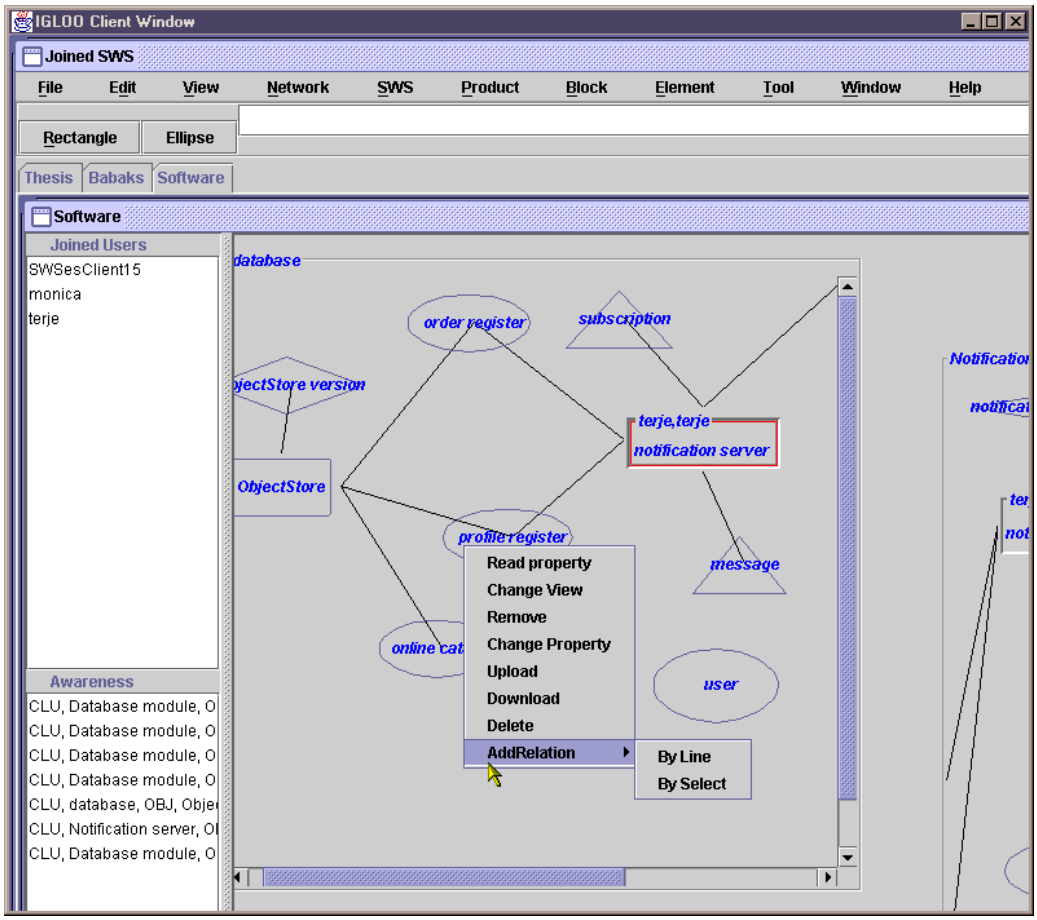


Figure 5.7: A pop-up menu used for creating a relation from cluster object “profile register” to “user”.

5.3.4 IGLOO functionality in MultiCASE

MultiCASE is an IGLOO client (see Figure 5.1). A large part of the functionality of MultiCASE is already implemented by IGLOO framework in form of services that are available to MultiCASE or any other IGLOO client. The part of the functionality of MultiCASE that is implemented by IGLOO framework includes:

- *A shared product*: IGLOO’s Product Layer allows MultiCASE users to share software modules and dependency relations among them. MultiCASE itself does not implement any sharing mechanisms. Sharing is done by using the services provided by Product Layer. The product, i.e. the software architecture in MultiCASE is placed in a shared product space provided by Product Layer. Anything that is placed in the shared product space is automatically

shared through the awareness services of Product Layer. Moreover, Product Layer allows *any* type of product object and relation to be shared. This allows the clients to share quite complex products with many conceptual object and relation types. The simple software architecture in MultiCASE is only one example. In addition, the propagation of awareness through the relations in a product is supported by Product Layer. This functionality is used by MultiCASE to implement product monitoring in a shared workspace. Product Layer provides a well-defined service protocol towards its clients. This protocol allows widely different types of clients to cooperate with each other. For instance, a Web-based client can be developed to view or modify the software architecture created using MultiCASE (an example of this is shown in Chapter 10.4).

- *Shared diagrams*: MultiCASE diagrams and their contents are implemented using Cluster Layer's cluster abstraction. Clusters implement real-time sharing. This means that MultiCASE does not need to implement sharing mechanisms within its diagrams. Seen from a MultiCASE client, a diagram is created as a normal (single-user) data structure. It is Cluster Layer that allows the diagram to be shared by many MultiCASE clients. All the users of the joined clients automatically see an updated view of the diagram as it is changed by other users. In addition, Cluster Layer allows any combination of shallow and deep properties for the objects in its clusters. Cluster Layer keeps all these attributes synchronized across all the joined clients. Cluster Layer, similar to Product Layer, implements a well-defined network protocol. This means that MultiCASE diagrams, or any cluster, can be used in any application. For instance, a MultiCASE diagram can be used in a single-user graphical editor. The single-user editor's user will see (in real time) all the modifications that are done by MultiCASE users, and vice versa.

- Shared workspaces: MultiCASE shared workspaces are implemented using Workspace Layer's shared workspace abstraction. Workspace Layer provides a conceptual data structure that allows its users to define any type of workspace with any type of content. The type of workspace used in MultiCASE is very simple and contains users, diagrams, and a chat tool. Whatever contents for a workspace is defined, Workspace Layer is responsible for keeping the contents synchronized across all the joined clients. For instance, when a user enters a workspace, Workspace Layer notifies all the MultiCASE clients. More advanced workspaces may include telepointers, radar views, video communication tools, etc. In addition, Workspace Layer provides a service interface for managing workspaces. For instance, the workspace management tool used by MultiCASE is implementing using this interface.

MultiCASE as an IGLOO client needs to implement only two things. First, it implements *visualization mechanisms* that are necessary for visualizing the software architecture, the diagrams, and the workspaces. Second, it implements *interaction mechanisms* that are needed for modifying the software architecture, the diagrams and their contents, and the shared workspaces and their contents. In fact, MultiCASE is implemented as a single-user application. Everything that has to do with sharing is implemented by IGLOO framework. Joining and leaving workspaces, managing existing workspaces, being aware of other users and being able to communicate with them, creating and modifying multi-user diagrams, and keeping all users informed about the changes to a shared product are made possible through the services provided by the three layers of IGLOO framework.

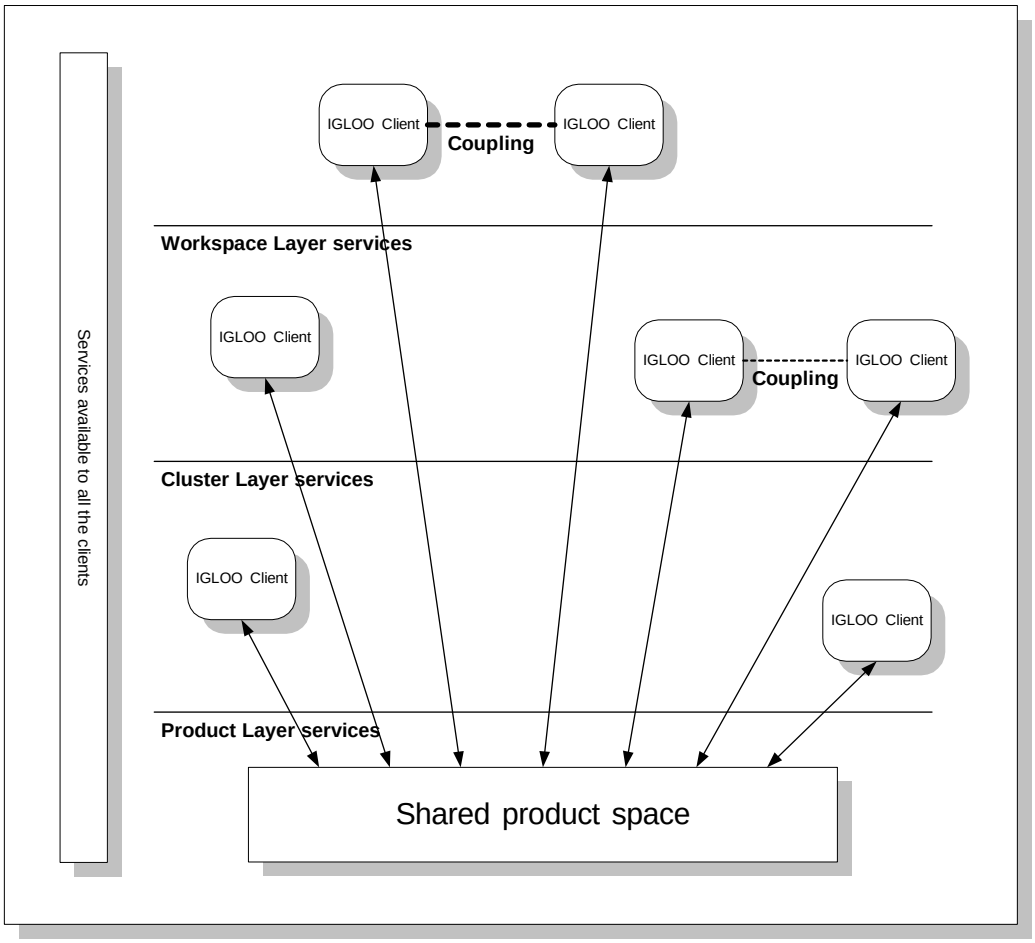


Figure 5.8: An IGLOO network.

5.4 Creating an IGLOO Network

MultiCASE is an example of an IGLOO client. Later in this thesis we will demonstrate a number of other IGLOO clients. What is common to all these clients is that they can communicate through IGLOO framework. They can create an *IGLOO network*. An IGLOO network is a collection of cooperating IGLOO clients. Figure 5.8 shows an IGLOO network. IGLOO clients in an IGLOO network may be viewers, editors, shared workspace applications, automatic agents, event monitors, single-user CASE tools, etc. IGLOO clients make use of IGLOO services provided by one or several service layers in the framework.

Not all IGLOO clients need to make use of all the three service layers of the framework. It might be the case that an IGLOO client will only need to add its product objects into the shared product space, and does not need advanced cluster and workspace functionality. Another client,

such as an already existing graphical editor, might want to use a cluster within its own workspace. An IGLOO network allows all these clients to cooperate with each other.

The minimum requirement for becoming a member of an IGLOO network is to use Product Layer services. This means that any application used for modifying or viewing a shared product space may become an IGLOO client, and become *integrated* into an IGLOO network. Integration in an IGLOO network happens through deciding which service layer contains the desired services, and to integrate the application in that level. For instance, a simple viewer application that is used to view the product objects in a shared product space will only need the services of Product Layer and can be integrated through that layer, while a multi-user graphical editor for real time modification of graphical diagrams will require services from higher layers. In general, a client application that does not require tight interaction among a group of developers will not need the services provided by the higher layers of the framework. However, if a client application has to support tight interaction in a small group, and/or support the sharing of a large amount of local context among the members of a group (i.e. requires high *coupling* about its users), it will need the services from Cluster Layer and Workspace Layer. Communication between two different IGLOO clients does not require them to integrate through the same layer, be of the same application type, or even be similar applications. But the framework requires them to share a product space as a minimum common denominator. For instance, a graphical document outline viewer and a text-based document editor can communicate by sharing their "product" (e.g. a document) in an IGLOO network using IGLOO services.

IGLOO networks are created through a *deployment process* that allows a project group to create a network in an incremental way. IGLOO clients can be added any time during the deployment process, and the IGLOO network can be improved all the time. Old clients can communicate with newly added ones. More details on the IGLOO deployment process will be given in Chapter 9.

5.5 Summary

In this chapter we gave an overview of IGLOO framework. We also introduced MultiCASE, a graphical editor for creating large software architectures. We gave a short introduction to IGLOO networks, and showed how they may support cooperation among different types of applications. IGLOO framework is based on the product-based shared interaction model developed in Chapter 4. IGLOO framework's primary task is to provide a shared product space that can support shared interaction in a product development project. This shared product space is implemented by Product Layer using awareness support mechanisms. Centers of interaction are implemented by Cluster Layer and Workspace Layer.

The rest of this thesis will describe IGLOO framework in details. The next three chapters will describe the three service layers of the framework. Each chapter will start by sketching the detailed services defined for the a layer, and will describe a generic implementation of the layer. Chapter 9 describes how IGLOO deployment process can be used for creating different types of cooperative environments, i.e. IGLOO networks, for supporting a variety of settings and projects.

Chapter 6

Product Layer

6.1 Introduction

As discussed in Chapter 2, the product being developed by a team of developers plays a central role in supporting cooperation among developers. Product objects are used by developers to externalize shared knowledge about the product. Product is also a resource for coordinating the day-to-day work of the project team. We also saw that geographical distribution has the effect of breaking down these processes of knowledge creation and coordination. Distributed groups such as ALPHA do not have access to a *shared product space* that can guarantee continuous access to information about the product being developed. This situation is shown in Figure 6.1.A, where each developer only has access to his own private space. The result is that not only access to shared data is hindered (with its own connected problems) but also opportunities for communication and cooperation are removed as a consequence of the lack of awareness of what others developers are doing. Figure 6.1.B shows a situation where all the developers interact with a shared space. In this case the developers have the possibility for making available the information they regard as important for the project (such as information that other developers depend on), and they can obtain a higher degree of awareness of what the others are doing to the shared space. This awareness can in turn facilitate learning (through continuous observation of changes) and opportunistic communication (by being aware of occasions for communication).

Product Layer is a detailed specification of services that can be used by a group of developers to build a shared product space, and to use it for supporting their cooperation. An overview of these services is shown in Figure 6.2 (a detailed description of the services is provided later in this chapter). A large part of the services are concerned with setting up and maintaining a shared product space (left-most branch in Figure 6.2). The shared product space that is set up using these services consists of *product objects* and *relations*, in accordance with the shared interaction model developed in Chapter 4. Product Layer facilitates access to a flexible space where developers can share a variety of product objects and relations among them. A second group of services is concerned with allowing the developers to exchange *product awareness information* among themselves (the middle branch in Figure 6.2). This is information about who is accessing the

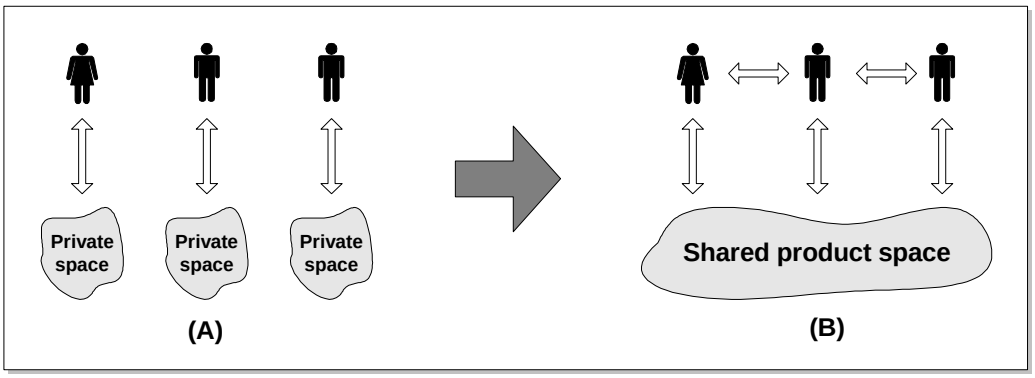


Figure 6.1: A shared space can increase awareness and support opportunistic communication.

contents of the shared product space and how (e.g. if they are reading, changing, browsing, etc.), and is produced automatically as a result of these accesses. This means that Product Layer users do not need to worry about explicitly sending awareness information to each other. However, each user has to set up an *awareness subscription*, telling Product Layer what product awareness information is of interest for that user.

Having access to a shared product space and being continuously informed about other developers' access to this space creates the basis for shared interaction. It also creates opportunities for communication among developers because now the developers can see *who* is doing *what*. The third set of services provided by Product Layer is therefore concerned with promoting opportunistic communication among a community of users (the right-most branch in Figure 6.2). These are services for *participant awareness* (i.e. who else is using Product Layer) and *instant message* (i.e. sending text messages to other users of Product Layer). Each piece of awareness information that is generated by Product Layer can be tracked down to the developer who caused it. This facilitates communication when necessary. For instance, if a developer A changes something in the shared product space and that change affects another developer B, Product Layer will inform B about the changes and will help B to contact A for possible clarifications.

The services are provided through a service interface (see Figure 6.3) that is independent of any specific design or implementation. The interface is meant to be a programming interface. This means that the actual users will not have direct access to the interface, but will access the services through their IGLOO clients (see Chapter 5 for a definition of IGLOO clients). IGLOO clients can create a shared product space by registering their product objects and relations in Product Layer. This is shown in Figure 6.3. In this figure each of the clients A to D have registered a subset of their product objects (the black objects) in the shared product space (the grey objects) with relations among them. For instance, MultiCASE (explained in Chapter 5) uses Product Layer to create a shared product space consisting of software modules and dependency relations among them. Each MultiCASE client can register new product objects and create relations among the existing objects using Product Layer services[1]. Access and modifications to the shared product

[1] In MultiCASE this is actually done through Cluster Layer. Cluster Layer acts as a mediator between Product Layer and Workspace Layer. This will be explained in more details in Chapter 7.

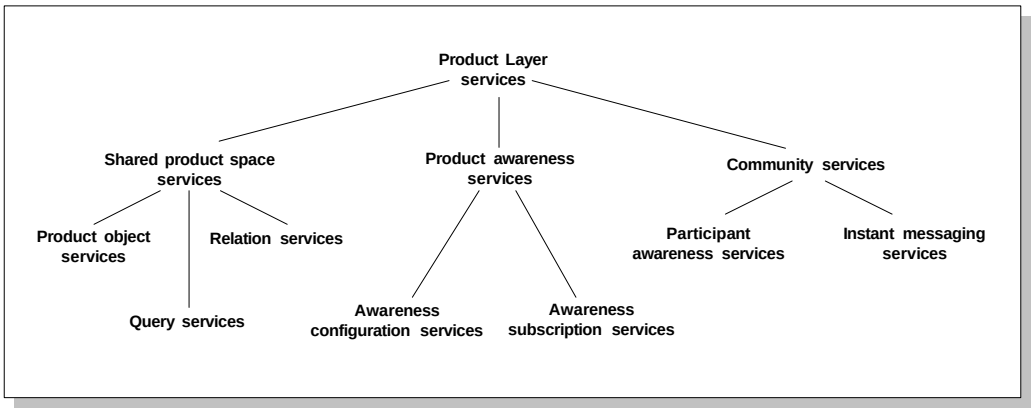


Figure 6.2: Overview of the services provided by the Product Layer.

space result in awareness information that is sent to each client based on the client's information needs. In MultiCASE, when a developer changes an attribute (for instance the name) of a software module, a unit of awareness information (a notification) is sent to all the other developers who use that software module. In this way, the developers can be engaged in continuous shared interaction by adjusting their views and having up-to-the-minute information about the shared product space and changes to it.

It is important to note that Product Layer is not a repository. Product Layer is a place to *share* information that is of importance for the cooperative work. Product Layer's primary task is not to store or process the product. The important requirement is that Product Layer has to be flexible regarding the types of product objects it can contain, and has to provide easy access to the shared product space and the awareness information that is generated (See Chapter 2 for a list of requirements for a shared product space.) The range of possible product objects and relations that can be registered in a shared product space is therefore not predefined by Product Layer. Objects and relations are registered in form of attribute-value pairs, which allows the clients to register virtually any type of objects and any type of relations among these objects.

6.2 Services of Product Layer

Product Layer is mainly defined in form of services. The services are grouped as shown in Figure 6.2 and will be described in details in this section. Product Layer services can be used in the following and similar situations:

- When a user wishes to share some information (documents, drawings, sketches, notes, or other information) with other users of Product Layer. In this case shared product space services can be used. These services allow a client to register product objects of arbitrary type consisting of an arbitrary set of attribute-value pairs. The client can register the necessary objects, create arbitrary relations (also consisting of attribute-value pairs) among the objects, and modify these objects and relations through shared product space services.

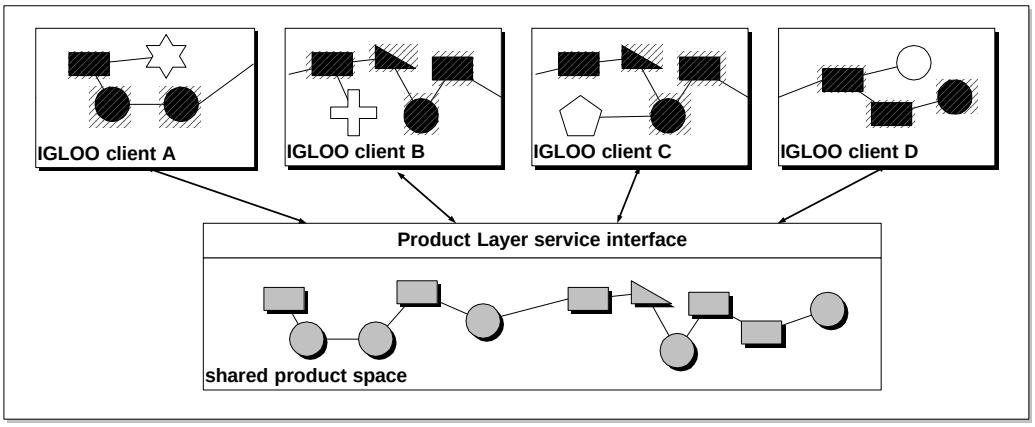


Figure 6.3: Clients can use the services provided by Product Layer in order to register product objects and relations among them in a shared product space.

- When a user wants to know what other users are doing. A user might want to know who has been reading the files that he/she registered in the shared product space some time ago. In this case the client application of the user will use the product awareness services of Product Layer to exchange awareness information with other clients.
- When a user wants to communicate with another user. All product awareness information in Product Layer is connected to specific users who created them as a result of their access to the shared product space. This allows users to know who has been reading or changing a product object in the shared product space. Community services help users to directly contact other users when situations for communication arise.

These three groups of services are explained in details in the following sections.

6.2.1 Shared product space services

As we saw in Chapter 3, the majority of integrated software engineering environments provide a shared repository where the product under development is *stored* and processed by different tools. The focus of such repositories is often to provide inter-operability among the various tools used throughout a development project, and to preserve the consistency of the product during modifications done by developers. Product Layer, on the other hand, is more concerned with *sharing* of product information among the developers. This information is held within the shared product space, and can be accessed and updated through shared product space services.

Product objects and relations are the main elements in a shared product space. A product object represents a part of a product. The part is registered in the shared product space in form of a product object when a user wishes to share it with other users of Product Layer. Each product object is distinguished by an identifier that is unique within the shared product space. Each product object is represented in Product Layer in form of a set of *attribute-value* pairs. This set is not predefined by Product Layer. This means that virtually any type of product object can be defined

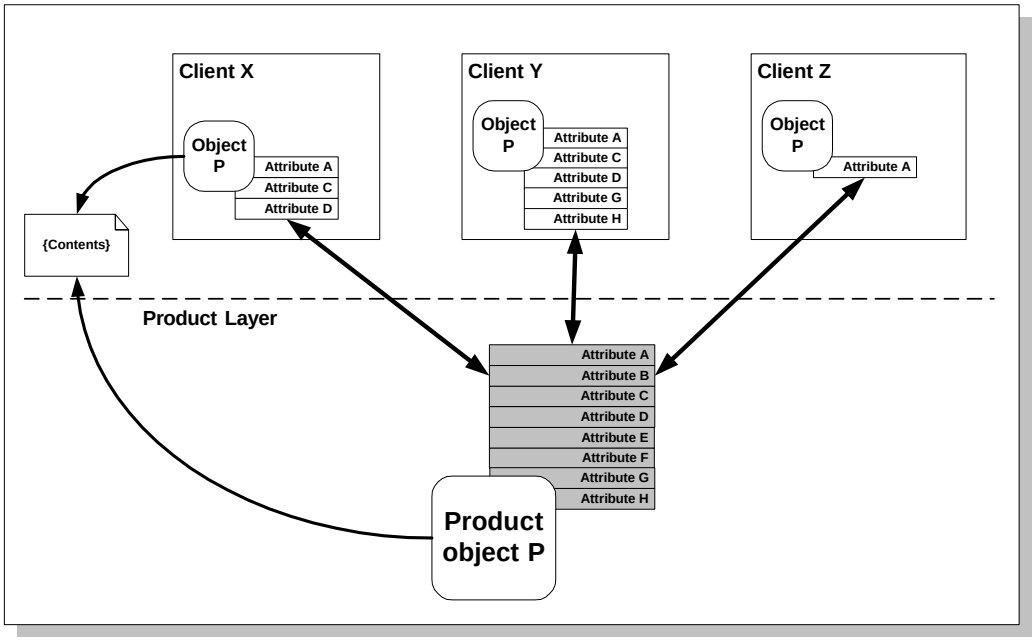


Figure 6.4: A product object can be viewed in differently by each client.

by just using the right combination of attributes. Attributes are distinguished by their name, and it is the responsibility of the users to decide and share a consistent set of attributes for each object type[2].

Figure 6.4 shows a product object P consisting of attributes A through H. This set of attributes is defined by the group of (heterogeneous) clients X, Y and Z, and reflects the information they want to share about P. The set of attributes is dynamic, meaning that at any time any client can decide to add new attributes, such as annotations[3]. In addition, not all types of client may need to use all of the existing attributes. In the figure, client Y is making use of five of the attributes, while X and Z use only three and one attribute respectively. This opens the possibility for different types of clients to cooperate with each other.

Consider a scenario where MultiCASE (the graphical editor described in Section 5.3) is only one of several types of IGLOO clients using Product Layer. In Figure 6.4, Z could be a MultiCASE client, showing only P's title in a graphical diagram. Y could be another IGLOO client, a web-based viewer, showing detailed information about P on a Web page. X can then be thought of as a third type of IGLOO client, a browser, showing the name of the object together with its owner and the date it was last changed. In this scenario the three clients will share attribute A, the name of the object. Any changes to P's name will be noticed by all the clients. In addition, X and Y

[2]Deciding a set of agreed-upon object types is crucial for cooperation among the users. This is part of the process of deploying IGLOO framework, and is explained in details in Chapter 9.

[3]This feature has to be supported by each client, i.e. the clients should allow the users to add arbitrary attributes to the object in runtime.

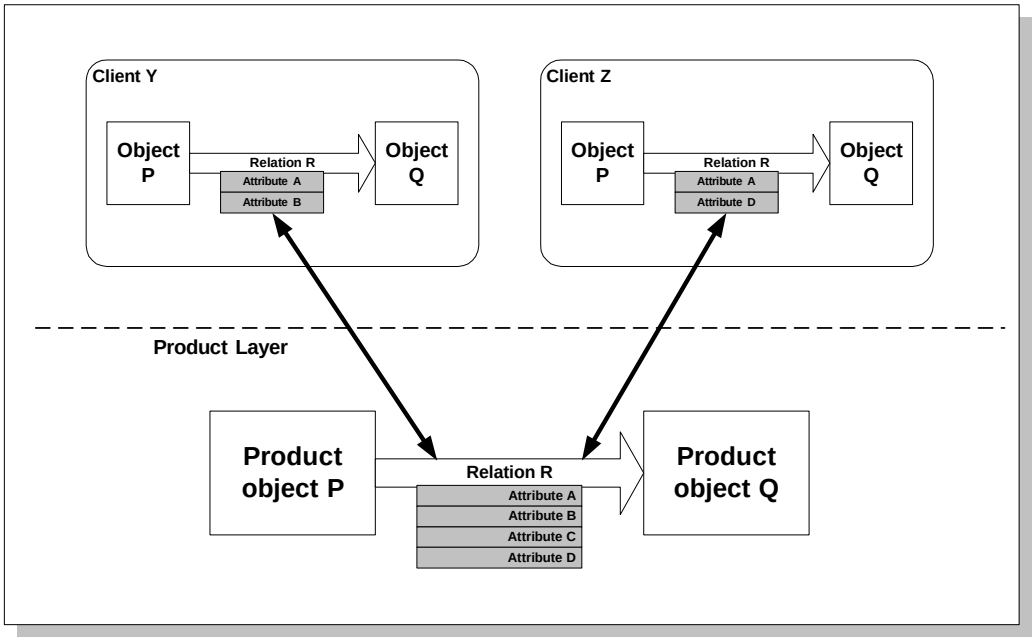


Figure 6.5: Relations in Product Layer are represented as attribute-value pairs with source and destination product objects.

will have a tighter cooperation through sharing additional attributes C and D. Still other types of clients can make use of other combinations of these attributes. A client might also want to store "private" attributes not known to any other client[4].

In addition to the set of attribute-value pairs, each product object has a *content* (shown in Figure 6.4 as a round rectangle for object P). Clients are allowed to register and modify one shared content for each product object in the shared product space. This functionality can be used for registering already existing documents, diagrams, photos, e-mail messages, etc. A content is treated as a file, and does not need to reside within the shared product space together with product object attributes. The product object may instead contain a pointer to its own content (for instance an Internet address). In Figure 6.4 product object P has a content file that is set by the user of client X (the content file shown as a sheet of paper to the left of the figure). The product object in Product Layer contains a link to this content, allowing clients X and Z to access it. Contents can be useful when a client already has data stored in a repository. Information about the data in the repository can be registered in Product Layer together with a pointer to the real contents. The pointer may also specify specific access methods to allow other clients access the contents (for instance if the content is a web page, the pointer may tell the clients to use the HTTP protocol in order to access it).

[4] In Chapter 5 we introduced the notion of shallow and deep properties for software modules in MultiCASE. In IGLOO framework all the attributes belonging to product objects in the shared product space are deep properties. Shallow properties are first created as local attributes for user-defined cluster objects. See Chapter 7 for more details on this.

Similar to product object services, Product Layer also provides a set of services for modifying relations among product objects. Relations are represented in form of attribute-value pairs with source and destination product objects. This is shown in Figure 6.5. In this figure, relation R from object P to object Q has four attributes. Again each client may be concerned with only a subset of the attributes that exist for each relation depending on the type of the client. Relations are also used for generating mediated awareness according to the shared interaction model. This is described in Section 6.2.2.

The services provided by Product Layer for setting up and maintaining a shared product space are shown in Table 6.1. These services also include a set of simple and flexible querying services for allowing the clients investigate and retrieve the contents of the shared product space.

The services defined here do not cover all the possible operations that the clients might want to perform on their products. In particular "repository-like" operations are missing. It would be impossible to predict all such operations. Product Layer defines a minimum set of required services. An implementation of Product Layer might choose to support more. For instance, our implementation of Product Layer called Gossip (described in Section 6.3) allows the clients to simulate *any* operation on the shared product space besides the ones defined here. These simulations are then mediated to all the clients by Gossip, even if the actual simulations do not modify the state of the shared product space. For instance, if the clients support version control on product objects they may issue commands such as "create new version" just to inform other clients that they are creating new versions.

Table 6.1: Product Layer's shared product space services

Service	**Semantics**
Product object services:	
Register a new product object	Register a new product object in the shared product space. The service also allows the client to specify a list of attribute-value pairs in order to initialize the newly created object. Each attribute's name must be a distinct string. Each attribute's value must be a string. A new product object identifier is created and returned to the client upon success.
Remove an existing product object	Remove an already existing product object from the shared product space. The client has to provide a valid product object identifier. All attribute-value pairs belonging to the object are also removed. In addition, all relations that this product object is source or destination for are removed.
Update an existing product object	Update the values of one or several attributes for an existing product object. The client has to provide a valid product object identifier, and a set of attribute-value pairs. For each attribute-value pair that is provided, Product Layer will: a) if the attribute already exists, update its value using the provided value, b) if the attribute does not exist, create the attribute and set its value to the provided value.

Continued on next page

Continued from previous page

Service	Semantics
Remove attributes from a product object	Remove one or several attributes from an existing product object. The client has to provide a valid product object identifier and a set of attribute names.
Update the contents of an existing product object	Set a new file or a URL as the new content for an existing product object. The client has to provide a valid product object identifier, a file or URL, and an optional (MIME) type for the new contents.
Relation services:	
Register a new relation between two product objects	Register a new relation in the shared product space. Valid source and destination object identifiers must be provided. The service also allows the client to specify a list of attribute-value pairs in order to initialize the newly created relation. Each attribute's name must be a distinct string. Each attribute's value must be a string. A new relation identifier is created and returned to the client upon success.
Remove an existing relation	Remove an already existing relation from the shared product space. The client has to provide a valid relation identifier. All attribute-value pairs belonging to the relation are also removed.
Update source for a relation	Update the source product object of a relation. The client has to provide a valid relation identifier and a valid product object identifier.
Update destination for a relation	Update the destination product object of a relation. The client has to provide a valid relation identifier and a valid product object identifier.
Update attributes of a relation	Update the values of one or several attributes for an existing relation. The client has to provide a valid relation identifier, and a set of attribute-value pairs. For each attribute-value pair that is provided, Product Layer will: a) if the attribute already exists, update its value using the provided value, b) if the attribute does not exist, create the attribute and set its value to the provided value.
Remove attributes from a relation	Remove one or several attributes from an existing relation. The client has to provide a valid relation identifier and a set of attribute names.
Shared product space query services:	
Get attribute values for a product object	Return the current values of one or several attributes for a given product object. The client has to provide a valid product object identifier and a set of one or several attribute names.
Get all attribute-value pairs for a product object	Return the set of all attribute-value pairs for a given product object. The client has to provide a valid product object identifier.

Continued on next page

Continued from previous page

Service	Semantics
Search for product objects with a specific attribute-value	Return product object identifiers for all the existing product objects that have a specific value for a specific attribute. The client has to provide an attribute name and a value for that attribute.
Search for product objects with a specific attribute	Return product object identifiers of all the existing product objects that have a specific attribute. The client has to provide an attribute name in form of a string.
Get all existing product objects	Return a list of the identifiers of all the existing product objects.
Get the contents of an existing product object	Return the current contents of an existing product object along with its (MIME) type. The content may be a file or a URL. The client has to provide a valid product object identifier.
Get attribute values for a relation	Return the current values of one or several attributes for a given relation. The client has to provide a valid relation identifier and a set of one or several attribute names.
Get all attribute-value pairs for a relation	Return the set of all attribute-value pairs for a given relation. The client has to provide a valid relation identifier.
Search for relations with a specific source	Return identifiers for all the existing relations that have a specific product object as source. The client has to provide a valid product object identifier.
Search for relations with a specific destination	Return identifiers for all the existing relations that have a specific product object as source. The client has to provide a valid product object identifier.
Search for relations with a specific attribute-value	Return identifiers for all the existing relations that have a specific value for a specific attribute. The client has to provide an attribute name and a value for that attribute.
Search for relations with a specific attribute	Return identifiers of all the existing relations that have a specific attribute. The client has to provide an attribute name in form of a string.
Get all existing relations	Return a list of the identifiers of all the existing relations.

6.2.2 Product awareness services

As discussed in Chapter 2, geographically distributed groups often lack access to continuous flow of awareness information. Having access to the right information, which is guaranteed when using shared repositories with global access, is often not enough by its own. The problem is merely that users are not hinted about changes in the shared product space simply because they are not physically co-located with those making the changes. Providing active support (i.e. explicitly informing users about others' activities) can strengthen awareness processes that are so important for cooperation. Product awareness services allow Product Layer clients to receive *product awareness events* about access to the shared product space. A product awareness event is a message generated automatically by Product Layer as a result of a client accessing the shared product space. Product awareness events are generated as the services shown in Table 6.1 are requested by a client and

performed by Product Layer. Each event is delivered to the other clients in order to inform them about the access. Such an event contains information about the kind of access (i.e. read, update, add, delete, etc.), what objects or relations were involved, which client accessed the shared product space, and some additional information depending on the type of the event. Through generating product awareness events, Product Layer becomes an active provider of awareness information.

There are three groups of services for allowing the clients to make use of product awareness in Product Layer (see Figure 6.2). First, awareness configuration services allow the clients define the overall type and amount of product awareness events to be generated by Product Layer. It might be that a group of users do not want Product Layer generate events for all types of accesses to the shared product space simply because the amount of product awareness information generated in this way might be seen as too much. The users can configure the number of generated events as long as it does not affect the internal state of the shared product space and the consistency of the clients.

When product awareness events are generated, they have to be delivered to the clients. The needs of the clients regarding product awareness information will be different from time to time depending on what part of the shared product space they are working with. The second group of subscription services are used by each client to inform Product Layer about the actual awareness information needs of that client. Each client is required to create an *awareness subscription* for itself in order to let Product Layer know what product awareness events have to be sent to that client. The third group of services is concerned with the automatic delivery of events to clients in accordance with each client's awareness subscription.

An overview of awareness support in Product Layer is shown in Figure 6.6. In this figure, each client has already set up an awareness subscription that consists of product objects and relations (called a *watch list*, more on this and other subscription options later). Client W's subscription consists of product object A, client X's subscription consists of product objects A and C, etc. Subscriptions tell Product Layer what objects and relations each client is interested in. The contents of a subscription are normally those objects that the client's user is currently working with. In MultiCASE, for instance, the subscription of each client contains all the product objects that are in all the joined workspaces of a user.

Product awareness information is generated according to the shared interaction model developed in Chapter 4. This means that product awareness is divided into *direct* and *mediated* awareness. In Figure 6.6 a scenario containing both direct and mediated awareness is initiated as client W issues a service request. This might be a request for any of the services listed in Table 6.1. Such a request will result in access to the shared product space. After the service is performed, for instance an attribute of product object A is updated, all the clients currently having product object A as part of their subscription will receive a *direct* product awareness event about the performed service. This includes client X. On the other hand, client Y that is using product object B is also informed about W's access to A because there is an awareness relation R from A to B. This mediated awareness is provided through a *mediated* product awareness event [5].

In the rest of this section we will take a closer look at each of the three groups of product awareness services.

[5] An implementation of Product Layer should have algorithms for detecting that client X, for instance, does not need a mediated event even if it subscribes to C. This is because the user of X is already informed about access to A through a direct event.

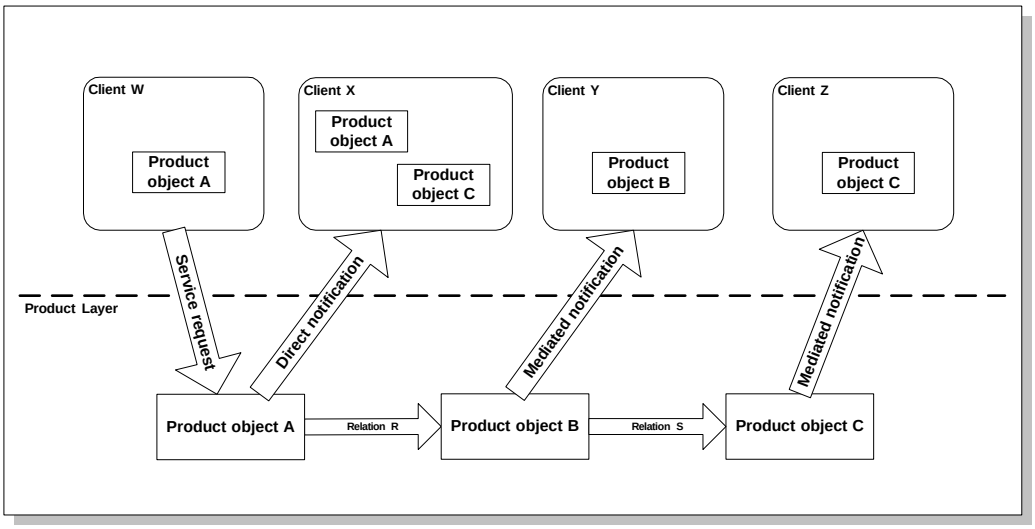


Figure 6.6: Product awareness mechanisms in Product Layer.

Awareness configuration services

The set of product awareness events that is generated for each service request and the information contents of each event are designed to guarantee that all the clients receive correct information about what the other clients are doing. Awareness needs of groups are highly diverse and one cannot assume that the same configuration of Product Layer will work in all situations and for all project groups. For instance in a co-located project, where the members have easy access to each other, lots of product awareness information may be exchanged simply as a result of being co-located. In this case it might not be necessary for Product Layer to generate product awareness events for every possible access to the shared product space. In fact most of the events will be of no use and the effect might turn out to be negative. On the other hand, a geographically distributed group might benefit from a higher amount of events in order to compensate for the lacking awareness of co-workers. Awareness configuration services are used to control the amount of product awareness events that are generated by Product Layer. It is important to configure Product Layer correctly because all clients will eventually receive a subset of the events that are generated by Product Layer in the first place. This means for instance that if Product Layer is configured not to generate product awareness events for query operations the clients will never get such events even if they would like to. Before we describe the awareness configuration services it is important to understand the mechanisms used by Product Layer for generating product awareness events.

All the services of Product Layer that access the shared product space (i.e. the services in Table 6.1) will potentially result in one or several *direct* product awareness events. These events are used to make public the exact nature of the activities of the clients. Assume that client X in Figure 6.4 updates attribute D on product object P. As a result, Product Layer will generate two product awareness events. The first event will be of type “Update product object” and the second will be of type “Update product object attribute.” The first event can be used by both clients Y

and Z in order to inform their respective users that object P was accessed (by blinking the object on the screen, for instance). The second event will not be useful for Z because attribute D is not used by this client. However, Y can use this event and update its copy of product object P in order to inform its user which attribute was updated. Note that product awareness events are not limited to "updates." Such events are also generated as a result of product objects being subject for queries. It might for instance be interesting for client X in Figure 6.4 to know if Y and Z are reading product object P, or that some other client just searched for product objects similar to P.

In addition to direct product awareness events, another type of product awareness events is generated by Product Layer that is called *mediated* product awareness event. Mediated events are distributed following the existing relations among product objects, and support the concept of mediated awareness as discussed in Chapter 4. Relations can be created among any two product objects as described above. A relation not only connects two product objects according to some criteria (for instance dependency or "sub_part"), but can also be set (explicitly by the clients) to *mediate* product awareness events. In Figure 6.6 there are two relations, relation R from product object A to B, and relation S from B to C. Once client W performs an operation on product object A, product object B may be informed about the operation through relation R. B will then generate a mediated product awareness event. This event is sent to the clients using B, in this case Y. This process is called *awareness mediation* (see Chapter 4). Awareness mediation can happen recursively all through a *mediation path* among product objects. For instance, in Figure 6.6, the mediated product awareness event from product object B also causes product object C to generate a mediated awareness event, which is sent to client Z. The mediated event received by Z originates from C, but the object identifiers of all the product objects in its mediation path (i.e A and B) are appended to the event in order to let the user of Z investigate the cause of the product awareness[6].

In addition to the set of attribute-value pairs that each relation may have, a set of *operation-strength* pairs can be defined for each relation. This set functions as a filter for selecting the product awareness events that can be mediated by the relation. The *operation* field in each such pair decides what kind of interaction types the relation will mediate, and the *strength* for that operation type decides how many product objects each mediated product awareness event can pass before it is stopped (i.e. the strength factor of the awareness relation, see Chapter 4. For instance, if a relation has an "Update product object attribute" field with a strength 2, the relation will mediate all the events that are of type "Update product object attribute," and that have not already been mediated twice.

Product Layer can be configured to generate different combinations of both direct and mediated product awareness events. For direct events, the users can decide if Product Layer should generate events for all kinds of access to the shared product space (including read access) or if Product Layer should generate events only for modifications to the shared product space. For mediated product awareness events, users can define the behavior of each relation in the shared product space regarding awareness mediation. Relations are by default not mediators of awareness. Each relation has to be "switched on" explicitly, and after it is switched on it will mediate to its destination all user access to its source. Using operation-strength values for each relation one can regulate the number of mediated events that are generated in the presence of relations. For instance, awareness of read operations on an attribute from a product object that is "three objects away" is normally considered not so important, while it would be interesting to get information

[6]Pointers to all the product objects in the mediation path are appended to each mediated event also in order to detect and avoid loops by stopping the awareness mediation when the same relation has mediated the same event once before.

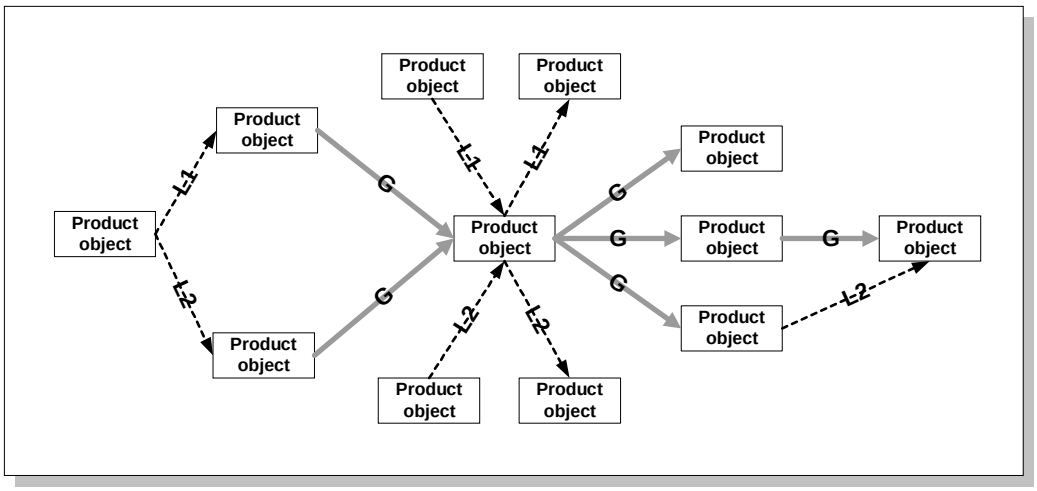


Figure 6.7: Each user can subscribe to one or more awareness schemes, or create his own.

about who is reading the objects in one's own workspace.

Both kinds of configurations mentioned above (i.e. controlling the generation of direct and mediated product awareness events) have global effects. This means that all users might become affected by changes to configuration. For instance, when Product Layer is set to generate only events as a result of modifications to the shared product space, no client will no longer receive read events. However, the situation is different for mediated events. Although all the relations exist in the shared product space, and basically every mediated product awareness event is accessible by any user, each user can decide what kind of mediated event he will actually receive. This is done through subscribing to the proper *awareness schemes*. Awareness schemes are described below, and their use in awareness subscriptions is described in the next section.

Each relation, after it is turned on and is ready to mediate awareness, will belong to an awareness scheme. Users can decide to subscribe to already existing schemes, or create their own (see next section for awareness subscriptions). Figure 6.7 shows a shared product space with three different awareness schemes. The thick gray relations labeled G belong to scheme G (G for global). All users might be expected (by the project manager) to subscribe to this scheme in order to get a shared view of the activities in the project. Such an awareness scheme will most probably not satisfy the awareness needs of all the users. Additional awareness schemes might be created by each user or group. In Figure 6.7 two such schemes exist, represented by relations labeled L1 and L2 (L for local). A user who subscribes to G and L1 will receive mediated product awareness events resulting from all relations labeled G and L2, but not by those labeled L2[7]. Awareness schemes may be used for supporting local group awareness: A group of users might create their own local scheme that allows them to receive the same awareness information about a part of the shared product space. Note that awareness schemes are not stored explicitly but exist implicitly.

[7]In recursive mediation (See Chapter 4), it is the last relation in the mediation path that is considered the originator of the mediated product awareness event. This means that if a mediated event passes several relations, it will eventually belong to the scheme of the last relation that mediated it.

This means that an awareness scheme exists if there is at least one relation in the shared product space that belongs to that scheme (i.e. a relation that is labeled with that scheme's name).

Following the discussion above, awareness configuration services fall into three sub-groups. The first group controls the generation of direct product awareness events, i.e. if Product Layer should generate direct product awareness events for all operations or only when the shared product space is modified. The second group controls the behavior of the relations in the shared product space by making them awareness mediators and configuring their filter settings. The third group is concerned with the management of awareness schemes by allowing the users decide which scheme each relation should belong to. These services are shown in Table 6.2.

Table 6.2: The services provided by Product Layer for configuring product awareness.

Service	**Semantics**
Generate product awareness events for queries	Tell Product Layer to also generate direct product awareness events for any read operation. This will result in generating an event for any access to the shared product space.
Don't generate product awareness events for queries	Tell Product Layer to stop generating direct product awareness events for read access. Direct events will still be generated for updates to the shared product space.
Make a relation an awareness mediator	Switch on an existing relation so that it can start mediating awareness from its source to its destination. A valid relation identifier and an optional list of operation-strength is provided by the client. If the list of operation-strength pairs is empty, the relation will propagate all the events from its source to its destination. The client may also provide an awareness scheme identifier this relation will belong to.
Update the operation-strength pairs of a relation	Update one or several operation-strength pairs for a given relation. The client has to provide a valid identifier for the relation, and a set of operation-strength pairs. Product Layer will, for each operation-strength pair, a) if the pair already exists, update its strength with the provided strength value, b) if the pair does not already exist, create the pair with the provided values.
Remove operation fields from an existing relation	Remove one or several operation fields from an existing relation. The client has to provide a valid relation identifier, and a set of one or several operation names.
Get operation strengths for a relation	Return the current strengths of one or several operation fields for a given relation. The client has to provide a valid relation identifier and a set of one or several operation names.
Get all operation-strength pairs for a relation	Return the set of all operation-strength pairs for a given awareness relation. The client has to provide a valid relation identifier.
Update awareness scheme for a relation	Update the awareness scheme that a relation belongs to. The client has to provide a valid relation identifier and a scheme label.

Continued on next page

Continued from previous page

Service	Semantics
Search for relations belonging to a specific awareness scheme	Return relation identifiers of all the existing relations that belong to an existing awareness scheme. The client has to provide a scheme identifier.
Get a list of awareness schemes	Return a relation of all the awareness schemes that currently exist in the shared product space.

Awareness subscription services

The awareness configuration services discussed above can be used to adjust the behavior of Product Layer regarding the generation of product awareness events. Once these events are generated they have to be delivered to the clients. It is highly probable that each client will have a different and varying interest in the shared product space. Therefore it is the responsibility of the clients to tell Product Layer what product awareness events they want to receive. The clients can do this by creating *awareness subscriptions* using Product Layer's awareness subscription services. An awareness subscription is a description of a client's awareness needs. Awareness subscriptions add a new level of customization, and give each client full control over its awareness needs[8].

A client's awareness subscription consists of three parts as shown in Figure 6.8. The first part (to the left in the figure) is the *watch list*. Each client will typically be interested in only a sub-set of product objects and relations from the shared product space. For instance a MultiCASE client uses only the product objects in the user's joined workspaces, and will have a watch list containing only those objects. A client can tell Product Layer what objects and relations it is currently using by adding those objects and relations to its own watch list. Figure 6.8 shows the awareness subscription of client X in Figure 6.6. X's watch list contains product objects A and C and no relations. When a watch list is defined by a client, only those direct and mediated product awareness events that originate from the objects and relations in the watch list are delivered to the client. In addition, for both objects and relations, the client can decide the type of operations it is interested in. For instance, in Figure 6.8 X has specified that it is interested in delete, update attribute, and read attribute operations on product objects, but only delete and update attribute operations on relations.

The second part of an awareness subscription contains the list of awareness schemes that the client subscribes to. In Figure 6.8 X has subscribed to schemes called G, L1, L2, and "Private scheme 1." The subscribed schemes decide which mediated product awareness events are delivered to the client. A mediated event that does not belong to one of the subscribed schemes of a client will not be delivered to that client.

The first two parts of an awareness subscription are concerned with product awareness (see Figure 6.8). The third part of an awareness subscription describes a client's interest on any other events that Product Layer may generate. This part includes subscriptions to events related to other clients' activities, e.g. when other client join or leave Product Layer ("Participant awareness" part). A client's user can also specify if he is interested in receiving instant text messages from

[8] An additional benefit of explicit subscriptions is that an implementation of Product Layer will be more efficient because it will not need to send all product awareness events to all clients. This would have caused too much network traffic in case of a network-based implementation. More details on this in Section 6.3.

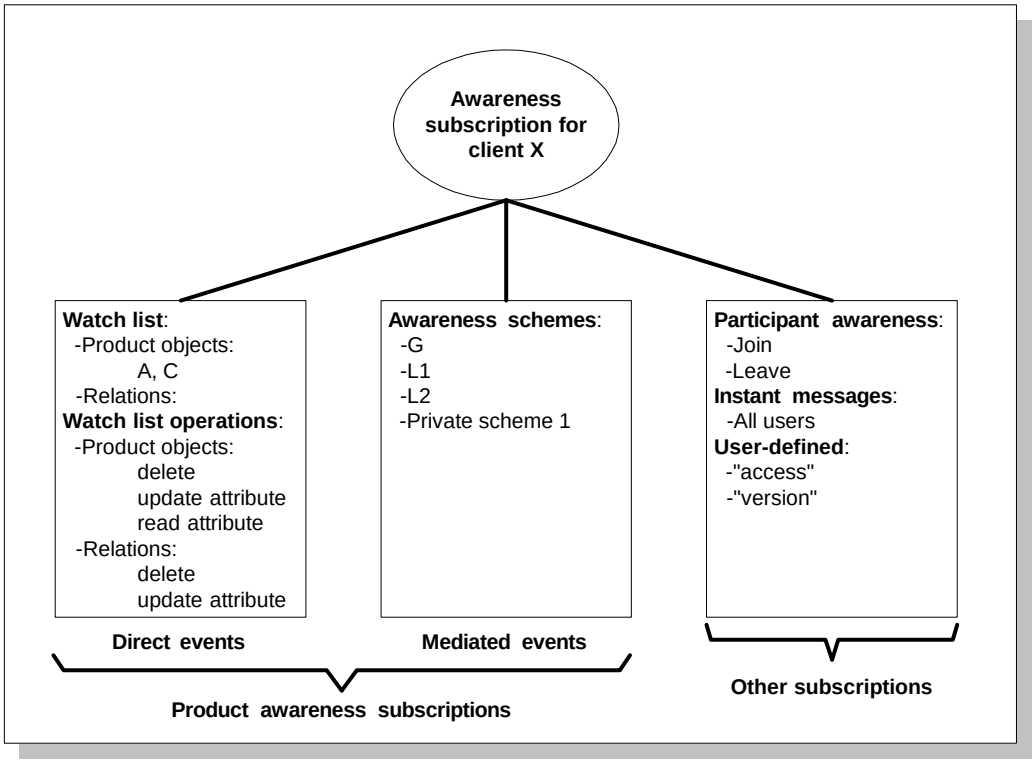


Figure 6.8: The different parts of an awareness subscription.

other users (“Instant messages” part). Finally, users can add user-defined strings that are checked for by Product Layer when user-defined operations are performed. These strings are stored in the “User-defined” part of a subscription. User-defined strings are used when awareness events about user-defined operation types (i.e. those not supported by Product Layer) are to be forwarded to the client. For instance, if Product Layer is used for sending version control information to other clients, a string “version” can be added to the subscription of a client. Product Layer will then know which clients want to receive events containing the string “version.”

A client might change its awareness subscription dynamically using awareness subscription services of Product Layer shown in Table 6.3. A client will start consuming awareness by joining the Product Layer and setting up a watch list in its awareness subscription, and will stop consuming awareness by leaving Product Layer.

Table 6.3: The services provided by Product Layer for creating and updating awareness subscriptions.

Service	Semantics
Add a product object to the watch list	Add an existing product object to the watch list of the client. The client has to provide a valid product object identifier.
Remove a product object from the watch list	Remove a product object from the watch list of the client. The client has to provide a valid product object identifier.
Add a relation to the watch list	Add an existing relation to the watch list of the client. The client has to provide a valid relation identifier.
Remove a relation from the watch list	Remove a relation from the watch list of the client. The client has to provide a valid relation identifier.
Get all product objects in the watch list	Return a list of the identifiers of all product objects in the watch list of the client.
Get all relations in the watch list	Return a list of the identifiers of all relations in the watch list of the client.
Update operation list for product objects	Update the list of subscribed operation types for product objects. The client provides a list of operation types. The new list replaces the old one.
Update operation list for relations	Update the list of subscribed operation types for relations. The client provides a list of operation types. The new list replaces the old one.
Subscribe to an awareness scheme	Add an awareness scheme to the list of schemes the client subscribes to. The client has to provide a label for the scheme.
Remove an awareness scheme	Remove an awareness scheme from the list of schemes the client subscribes to. The client has to provide a label for the scheme.
Get all awareness schemes	Get a list of all awareness schemes the client subscribes to.
Update participant awareness list	Update the list of subscribed participant awareness events. The client has to provide a list of valid event types. The old list is replaced with the new one.
Get participant awareness list	Return the list of subscribed participant awareness events.
Update instant message list	Update the list of users the client will receive instant messages from. The client has to provide a list of valid user identifiers. The old list will be replaced with the new one.
Get instant message list	Get the list of users the client receives instant messages from.
Update user-defined string list	Update the list of user-defined strings the client subscribes to.
Get user-defined string list	Get the list of user-defined strings the client subscribes to.

6.2.3 Community services

Product awareness results in increased up-to-the-minute knowledge about who is doing what in the shared product space. This knowledge creates opportunities for interpersonal communication that would have not existed without product awareness. Product Layer supports a few simple services in order to allow direct communication among its users. These are called community services and include services for joining and leaving Product Layer, finding out about other clients and their activity level, and for sending instant text messages to other joined clients. These services are shown in Table 6.4. Each of these services result in one or several *participant awareness events*, that are a special type of awareness events. The range of participant awareness events sent to each client is defined in the client's awareness subscription. In addition, *instant messages*, that are short text messages delivered to the users in real time, can be exchanged among the users.

Table 6.4: Different community services supported by Product Layer.

Service	**Semantics**
Join Product Layer	A client joins Product Layer. The client has to provide a valid client identifier.
Leave Product Layer	A client leaves Product Layer.
Get a list of all joined clients	The Product Layer will provide a list of the client identifiers of all the clients currently connected.
Get activity level for a client	Return an indicator of how active a client has been recently. The indicator can be high, medium, low, or no activity.
Get a list of clients watching a product object	Return the client identifiers of all clients that have a specific product object in their watch list. The client has to provide a valid product object identifier.
Get a list of clients watching a relation	Return the client identifiers of all clients that have a specific relation in their watch list. The client has to provide a valid relation identifier.
Send instant message to a client	The client can compose a text message and send it to another currently connected client. The sender has to provide a valid client identifier for the recipient.
Send instant message to all clients	The client can composed a text message and send it to all currently connected clients.
Send instant message to users of a product object	The client can composed a text message and send it to all the currently connected clients that have a specific product object in their watch list.
Send instant message to users of a relation	The client can composed a text message and send it to all the currently connected clients that have a specific relation in their watch list.

Community services connect product awareness to the users. Community services can be used to create chat-like tools such as the one used in MultiCASE, where users can engage in discussions. The services can also be used for creating e-mail solutions by allowing each user to have an inbox where instant messages can be stored. There are also services for sending instant messages to the

users of a specific product object or relation. These services can be used to initiate discussions related to these objects and relations, and can be used as basis for design rationale support.

6.3 The Implementation of Product Layer: Gossip

In this section an example implementation of Product Layer called *Gossip* is described[9]. The implementation is in form of an *awareness engine*, i.e. a specialized notification server that is in charge of supplying product awareness information to the clients in an IGLOO network. The basic mechanisms behind using notification servers for supporting awareness are shown in Figure 6.9. In this figure, the user of client A performs an action. The action results in client A sending a service request to the notification server. If the request is performed, an acknowledgement is sent to client A, and other clients receive a notification. The effect of the action is shown to client A's user as a *feedback*, while other users will observe the action performed by client A's user in form of a *feedthrough*. In this way, clients B, C and D can simulate for their own users the action performed by client A's user.

Notification servers have proved to be efficient means for supporting awareness in collaboration support systems (Ramduny, Dix and Rodden 1998). Notification servers provide generic awareness support, as opposed to specific solutions such as shared window systems. This means that clients of different kinds can use a notification server for exchanging awareness information as long as a uniform protocol is used for communication with the server. A notification server often employs an information push policy, where notifications are sent to clients instead of clients pulling the server for information[10]. This has the technical advantage of reducing network traffic to only a number of necessary notifications being exchanged between the server and the clients. Push policy is in particular useful for synchronous applications, where performance and feedthrough speed are critical (Day 1997).

The difference between a notification server and an awareness engine is fluid. In this thesis we will use the following distinction: An awareness engine is a notification server with an internal logic that guides the distribution of notifications to the clients. This internal logic is often based on a specific interaction model. For instance, NSTP (Day, Patterson, Kucan and Chee 1996) is an awareness engine that uses a room-based model of interaction. As a result, notifications of changes to the contents of a room are sent only to the inhabitant of that room. According to our definition all awareness engines are also notification servers.

Gossip is a notification server based on information push policy. Gossip implements a uniform network protocol where all service request adhere to a standard format. This protocol allows any IGLOO client to connect and perform the Product Layer services described in the previous section. Gossip will then distributed product awareness events to its clients in form of *notifications*. In addition, Gossip is an awareness engine because it also implements a specific awareness model based on the shared interaction model developed in Chapter 4. The shared interaction model guides the distribution of notifications, and in this way provides a customized flow of awareness information to the clients.

[9] A number of the properties of Gossip as described in this section are common for the implementations of the other layers of IGLOO framework, i.e. CoClust described in Section 7.4 and SWAL described in Section 8.4. These common parts, including the network protocol format and notification mechanisms, will not be repeated for CoClust and SWAL.

[10] Though a pull policy is also a perfectly possible solution in many cases. See (Ramduny et al. 1998) for a discussion.

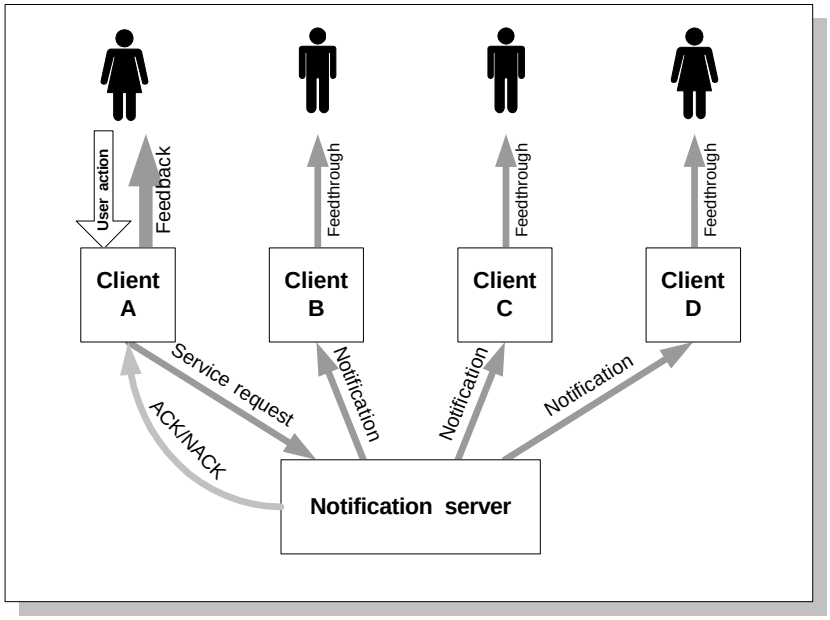


Figure 6.9: Using notification servers to support awareness.

6.3.1 An overall view of Gossip

Figure 6.10 shows the main components of Gossip's architecture, and their relationships. Boxes denote functional components, while gray boxes denote persistent object stores. Black thin arrows denote *Gossip event* communication, and white thick arrows denote other data communication. There are two types of Gossip clients. emphProducer clients can send requests to Gossip for performing a *Gossip operation*. They are the producers of awareness information. *Consumer clients* are those connected clients who have an awareness subscription in Gossip. These are the consumers of awareness information. A client can choose to be a producer, a consumer, or both[11]. An application becomes a client by explicitly connecting to the proper network channel. Consumer-only clients are not permitted to access any internal data but they might change their awareness subscriptions, so they will typically be event monitors. Shared product space browsers are for instance producer clients because they query the shared product space and may therefore produce product awareness.

The contents of the shared product space are stored in persistent registers. Attribute-value pairs for product objects are stored in *Product Object Register*, and relations are stored in *Relation Register*. Gossip supports the storage of one content file for each product object, but content files are logically external to Gossip (i.e. Gossip only stores a pointer to them internally). A content file can be stored in the *Content Database* (the local file system) with a local pointer

[11]There is also a third type of client that has the privilege of registering other clients as producer or consumer clients. CoClust, which is an implementation of Cluster Layer (see Chapter 7) is such a privileged client. Privileged clients are not a part of the public Gossip network protocol, and are used only internally by IGLOO framework.

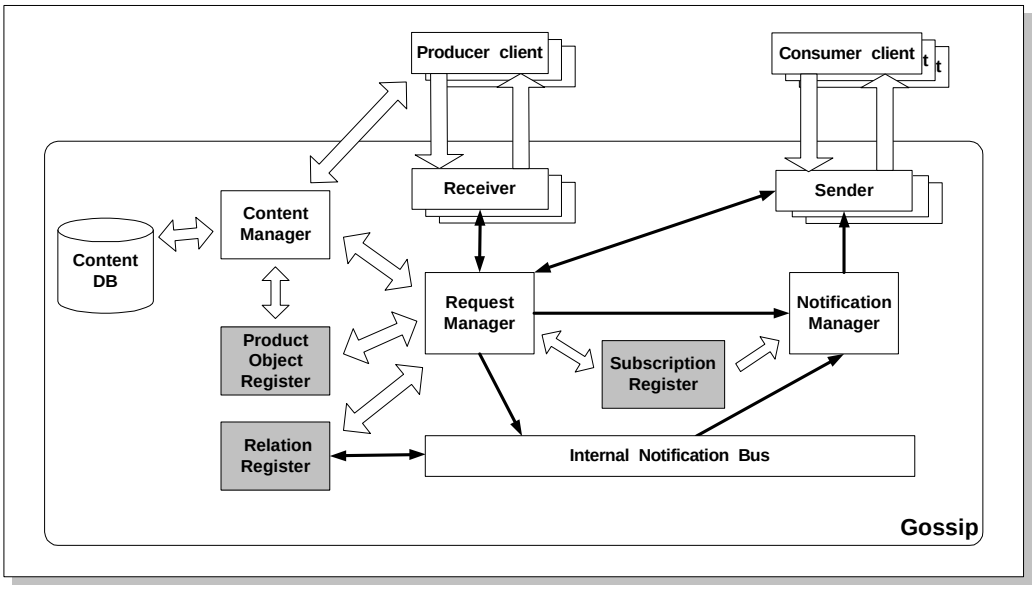


Figure 6.10: The internal architecture of Gossip.

from the corresponding product object to it. Contents can also be available on the Internet, in which case Gossip stores only a pointer to their location. Awareness subscriptions are also stored in a persistent register called *Subscription Register*. The contents of watch lists in awareness subscriptions are not stored persistently since they must be specified each time a client connects to Gossip.

Any Product Layer service can be asked for by a client sending a *request* to Gossip. Each request will initiate an operation and will result in one or several notifications, as defined by *Gossip network protocol* (see Section 6.3.2). All requests, notifications and other types of messages are *Gossip event* and follow a standard format. Gossip event is extendable, meaning that for instance additional request types may be defined by the clients and in future versions of Gossip. Requests of client-defined types are not processed internally by Gossip, and are simply forwarded to other clients interested in them (based on each client's subscription for user-defined strings).

All clients have to communicate their requests to a *Receiver*. A Receiver is a translator between a specific network protocol and Gossip's internal representation, i.e. Gossip event. There are several Receiver objects in Gossip, one for each supported network protocol (currently only one). When a request arrives through a network channel to a Receiver, the Receiver creates a Gossip event object of type *REQUEST* (see Table 6.5 for Gossip event types), and initializes it based on the contents of the received request. The Gossip event is then delivered to *Request Manager* that is responsible for the processing of requests. After the corresponding operation is performed, the client will get back an event of type *ACKNOWLEDGEMENT* or *NEGATIVE_ACKNOWLEDGEMENT* depending on whether the request was performed successfully. If the operation was successful a new Gossip event object of type *NOTIFICATION_DIRECT* or *NOTIFICATION_COMMUNITY* (depending on whether the original request was for a shared product

space service or a community service) is created by Request Manager based on the initial request. The new event is sent directly to *Notification Manager*. In cases where the event was of type *NOTIFICATION_DIRECT*, it is also sent to *Internal Notification Bus*. Request Manager can be configured (through awareness configuration services) to generate direct product awareness events for only a subset of Gossip operations.

Each awareness relation is implemented in Gossip in form of an *awareness agent* with related source and destination product objects. Awareness agents are responsible for generating mediated product awareness events on behalf of their destination product objects, and therefore listen to Internal Notification Bus in order to monitor accesses to product objects[12]. Each Awareness agent will check the bus for events that have originated from the agent's source product object. For each such event, the agent will generate a new Gossip event of type *NOTIFICATION_MEDIATED* on behalf of its destination product object. These events are again sent to Internal Notification Bus, and other awareness agents may in turn generate new mediated events based on them.

Notification Manager is in charge of sending proper notifications or instant messages to each client. Upon arrival of a new Gossip event (from Request Manager or Internal Notification Bus) Notification Manager consults the awareness subscription of each consumer client in order to see if the event should be sent to it. For each client that has a subscription for an event, the event is delivered to the client's preferred *Sender* object. Each client has a preferred network protocol, and Senders are in charge of translating the internal Gossip event to this preferred network protocol understandable by the Consumer client.

Awareness subscription requests are sent by consumer clients. This means that a client in order to change its awareness subscription has to connect as a consumer client. This is done in order to allow consumer-only clients have the possibility for changing their subscriptions without becoming producer clients. A consumer client cannot request any service that will change the shared product space or the awareness configuration of the server. This makes it easy to separate monitor clients from other kinds of clients. Sender is accordingly capable of receiving requests for awareness subscription services. These requests are sent to Request Manager as usual, and acknowledgements are sent back to the consumer client through Sender.

Awareness configuration and subscription requests are handled differently in that they do not produce notifications. In this case Request Manager will receive the request (for instance for changing the watch list of a client), will process the request, and send an acknowledgement back to the client without creating any notifications.

6.3.2 Gossip network protocol

The services of Product Layer are implemented in form of a logical network protocol to be used for communication with Gossip. All the messages sent using the protocol, both from a client to Gossip and from Gossip to a client, are *Gossip events* as shown in Figure 6.11. The fields may have different meanings according to originator and type of event. *clientID* is a unique identifier for the client that sent the initial event to Gossip. The same clientID is copied to all the events sent from Gossip to other clients as a result of the original event. For instance, if a producer client sends a *REQUEST* to update a product object, all the resulting events of type *NOTIFICATION_DIRECT*

[12] Conceptually mediated events are generated by the affected product objects (see Figure 6.6). However, from an implementation point-of-view it has been more convenient to let relations take the responsibility of generating mediated events. For the clients of Gossip these events are still received from the actual product object.

will have the clientID of the original request. This allows other clients see which client sent the initial request. *eventType* can be one of the supported types shown in Table 6.5, or other user-defined event type. An event from a client to Gossip is always of type *REQUEST*. Events sent from Gossip to the clients may be of any of the types shown in Table 6.5 except the *REQUEST* type.

operationType has different meanings according to the type of the event. In an event of type *REQUEST* sent from a client to Gossip, operationType indicates the requested Product Layer service, for instance *UPDATE_PRODUCT_OBJECT*. In an event of any other type sent from Gossip to the clients, operationType indicates the operation that resulted in the event being sent. For instance, if a client sends a *REQUEST* of type *UPDATE_PRODUCT_OBJECT* to Gossip, Gossip will send the following events:

- An event of type *ACKNOWLEDGEMENT* or *NEGATIVE_ACKNOWLEDGEMENT* (depending on if the request succeeded or not) with operationType *UPDATE_PRODUCT_OBJECT* to the client that sent the original request.

- In case the request was successful all other clients will get an event of type *NOTIFICATION_DIRECT* with operationType *UPDATE_PRODUCT_OBJECT* and additional information about the performed operation.

- If the updated object is source for any relations, Gossip will send one or more events of type *NOTIFICATION_MEDIATED* with operationType *UPDATE_PRODUCT_OBJECT*.

operationType of an event sent to the clients may sometimes be different from operationType of the originating request. For instance, a request of type *UPDATE_PRODUCT_OBJECT* will produce one event (of type *NOTIFICATION_DIRECT* as shown above) indicating an *UPDATE_PRODUCT_OBJECT* operation, and one or more additional events (of type *NOTIFICATION_DIRECT*) indicating *UPDATE_PRODUCT_OBJECT_ATTRIBUTE* operations for each updated attribute of the product object.

schemeID is an awareness scheme label, and has a meaning only for events of type *NOTIFICATION_MEDIATED*. It indicates which scheme the mediated product awareness event belongs to. *eventID* is a unique identifier derived from a time stamp, and can be used by the clients to synchronize the reception of events. *authenticationInfo* is used in combination with clientID to authenticate the connected clients. The body of the event contains additional information about the event. For instance an event of type *NOTIFICATION_DIRECT* and operationType *UPDATE_PRODUCT_OBJECT_ATTRIBUTE* will include the identifier of the updated product object, and the name and the new value of the updated attribute. This data is different from event to event, and a *bodySize* indicates its size.

The protocol provides a basis that can be expanded both in new versions of Gossip, and in the current version by the clients. Clients can define new event types and new operation types according to their needs. This can be useful in cases where the set of services provide by Product Layer does not include frequently used local activities of the developers. For user-defined event and operation types, Gossip will work as a pure notification server and only forward the events to those clients interested in them. A part of each client's awareness subscription is reserved for subscribing to such user-defined messages (the part called "User-defined" in Figure 6.8).

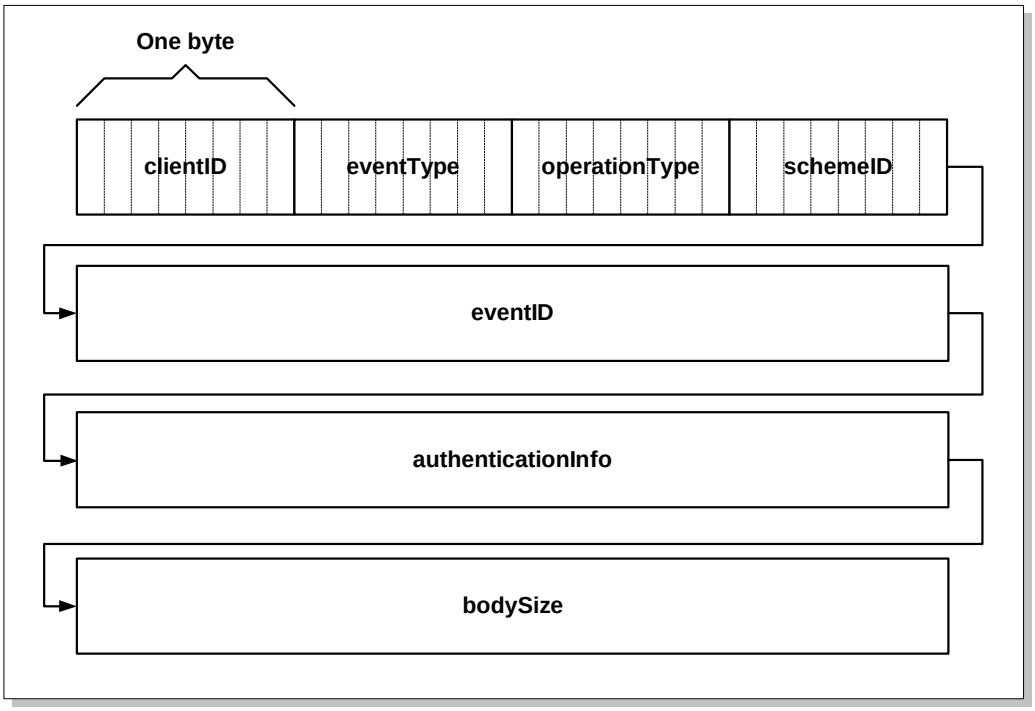


Figure 6.11: The format of a Gossip event. This format is used uniformly for communication with Gossip

6.3.3 Gossip client extension

As part of the Gossip server, a small library of classes called *Gossip client extension* is developed. The purpose of this library is to hide most of the details of Gossip network protocol from clients. Keeping the programming interface to Gossip as simple as possible has been one of our main goals in order to motivate developers to develop Gossip-enabled clients, or extend existing clients with Gossip-related functionality. Using Gossip client extension, clients can easily send and receive events from Gossip.

The client extension contains classes called *Gossip Server*, *Gossip Producer Channel*, *Gossip Consumer Channel*, *Gossip Awareness Subscription*, and *Gossip Event*. An application can initiate an instance of the Gossip Consumer Channel class, which will automatically set up a network channel towards a Gossip server, and register the application as a producer client of that server. After this, the application can make requests to the server using instances of the Gossip Event class. In the same way, creating an instance of a Gossip Consumer Channel will automatically register the application as a consumer client of the Gossip server. Upon the creation of a Gossip Consumer Channel, the application is requested to provide a call-back function. Events that arrive from the server will force the execution of this function. The application can then implement the call-back function in proper way in order to handle the received events. Gossip Awareness

Field	Semantics
REQUEST	A request event sent by a client for performing a service. All events from clients to Gossip are requests.
NOTIFICATION_DIRECT	A direct product awareness event sent as a result of access to the shared product space.
NOTIFICATION_MEDIATED	A mediated product awareness event sent as a result of awareness mediation.
NOTIFICATION_COMMUNITY	An event sent to a client as a result of a community activity, such as a client joining or leaving Product Layer.
NOTIFICATION_SYSTEM	An event sent to a client as a result of a system activity.
MESSAGE_COMMUNITY	An instant message sent to a client by another client.
MESSAGE_SYSTEM	An internal system message sent to a client.
ACKNOWLEDGEMENT	An event sent back to a client as a result of a successful request.
NEGATIVE_ACKNOWLEDGEMENT	An event sent back to a client as a result of a failed request. The event will contain an error code explaining the reason for failure.

Table 6.5: Different Gossip event types supported by current Gossip version. New event types can be defined by the clients

Subscription allows a client application to change its awareness subscription on the server. These classes support all the functionality provided by Gossip through an easy-to-use object-oriented interface.

6.3.4 Gossip's internal consistency

Gossip, as any other notification server (Ramduny et al. 1998), may be used in two quite different ways. It can be used in a tool to provide informal awareness about a shared field of work. A notification server, in particular one with an internal state (Gossip with its shared product space belongs to this category) may also be used in a more formal way for synchronizing a group of clients. In the first case, the consistency of the internal state is not so important as long as the notification server is able to send a more or less correct flow of notifications to the clients. For instance, an event monitor that shows a graphical representation of the level of activity within a shared state will not become inconsistent if the underlying notification server does not send all activity notifications, or sends some out of order.

In the second case, the notification server has to guarantee correct delivery of notifications, and if the clients rely on the internal state of the server, it also has to be able to preserve the

consistency of this internal state. For instance, if a notification server has to synchronize editors used by a group of developers, the server has to guarantee that all the developers see and change the same data, and that changes by one or several developers does not leave the internal state of the server inconsistent. The internal state in these cases is also of crucial importance for late comers. Whether Gossip will be used in the first or the second sense depends on its users. But since Gossip offers services that have to do with creating and maintaining product objects and relations which containing shared data, it has to be able to guarantee the consistency of this data. Other pure notification servers, such as Elvin (Segall and Arnold 1997), normally do not need to take this issue into consideration because they do not have any internal state.

Whenever consistency of shared data is of concern, a concurrency control mechanism has to be employed in order to control concurrent accesses to shared data. Concurrency control algorithms are used for this purpose. The choice of such an algorithm is influenced by several factors, maybe the most important one being performance. A central control mechanism may become a bottleneck in a distributed environment because all control has to go through a central controller. Another important issue is whether the algorithm makes extensive use of coarse-grained locking. If the algorithm has to lock all the internal state of the server in order to allow one client change one product object it is not useful because it will prevent others from working with other product objects.

Concurrency control mechanisms may be divided into two groups, pessimistic and optimistic ones (Prakash 1999). Pessimistic algorithms try always to stay in the safe side; no editing of the shared state is allowed before permission is granted by a controller. Optimistic algorithms allow unsafe editing of shared data, and use some recovery method (such as undo-skip-redo of operations) to recover from any inconsistent state that may occur.

The most common pessimistic algorithm is simple locking. For accessing a piece of shared data a lock has to be obtained from a lock server. Locks are also needed for simple viewing of data because the data might change while it is viewed by someone. Simple locking has shown to be unsuited for real-time editing of shared data because retrieving locks over network is time-consuming. A refined form of locking, that is also used by Gossip, is token-based locking. Each piece of shared data has a token that has to be obtained by anybody wanting to change that data. The advantage of this method is that a token does not need to be retrieved every time the data is edited. Once a client obtains a token, it can safely perform the editing and free the token. The free token stays with the client until some other client asks for it. In this way an optimization is achieved because each piece of data is often edited by only one person (Prakash 1999). This mechanism provides sufficient performance, and guarantees the consistency of the shared data. In Gossip, each product object and each relation have their own tokens. A finer grained implementations would allow to have one token per attribute in these objects, allowing concurrent editing of the same object or relation.

6.3.5 The implementation of Gossip

The current version of Gossip is implemented in the Java programming language. Network communication in Gossip, both towards clients and internally in the notification bus, is based on JSDT (Java Shared Data Toolkit[13]). JSDT is a flexible toolkit provided by Sun as an extension to the Java Development Toolkit. This toolkit implements useful groupware abstractions such as

[13] http://java.sun.com/products/java-media/jsdt/

sessions and channels. The connection between the clients and Gossip is through a JSDT session. Inside this session, two channels are used for communication between Gossip and the producer and consumer clients.

The internal notification bus is implemented in form of a network channel that is accessible only by other Gossip servers. Several Gossip servers can share the same notification bus, in this way creating a Gossip network. This is useful for scalability and performance reasons. Our experience with using Gossip shows that inside a high-speed local network notifications are distributed in real time. We have in fact used Gossip in a synchronous graphical group editor where the notifications are used to synchronize the screens of the clients. The notifications are distributed in a much slower speed on the Internet. Having one local server for each local group can help to build a more optimized Gossip infrastructure.

6.4 Summary

In this section we have defined Product Layer of IGLOO framework in form of a set of basic services that can be used to set up and maintain a shared product space, and increase product and participant awareness in a distributed project team. The underlying concept is that of a shared product space that can be accessed by a group of developers. Product Layer increases the visibility of these accesses, and allows developers be aware of each others' activities.

Product Layer fulfills the requirements we posed to a shared product space in Chapter 2. The shared product space offered by Product Layer is *open* in that access to it is supported by a well-defined set of services, and implemented in form of a network protocol that can be used by any application. The shared product space is also *accessible* in that information is represented in a uniform format and can be accessed in a uniform way. Updates to the contents can be done using a few services. Product Layer supports *unrestricted object types* by not predefining any object types at all, and allowing users define their own types. Objects and relations can be registered and modified in an *incremental way*, i.e. users can start with simple objects and refine them by adding new attributes. Each product object or relation can be viewed in a number of ways by using different combinations of attributes, in this way supporting the concept of *boundary objects* to some degree. The shared product space is *active* in that it actively provides product and other awareness information to its users. The delivery of information is *user-defined* and can be reconfigured in a number of ways. Product Layer supports *relations* of different types. Relations can be created among product objects according to the inherent structure of the product being developed. This is facilitated by allowing any type of relation with any attribute-value pair to be defined by the users. In addition, relations of a more social type can be defined based on social relations among users. This is facilitated by allowing awareness information be propagated through relations.

Product Layer is developed in form of a basic service for an IGLOO network. All clients within an IGLOO network will directly or indirectly rely on Product Layer for receiving product awareness. In addition, Product Layer takes the first step in integrating the product in a social context. First, each piece of awareness information is augmented with a reference to the originating user, in this way making it easy for the users to initiate opportunistic communication. Second, the community services allow the users be aware of who is currently interacting with the shared product space. In Chapter 9 we will see examples of how Product Layer can be used in combination with the other layers of IGLOO in order to create IGLOO networks for supporting cooperation.

Chapter 7

Cluster Layer

7.1 Introduction

In this chapter we will look at the second component of IGLOO framework, i.e. *Cluster Layer*. Cluster Layer is concerned with supporting focused cooperation in small groups of developers. Product Layer (see Chapter 6) allows a project team to create and maintain a shared product space, while Cluster Layer allows smaller groups to interact with this space. The discussion in Chapter 2 revealed the need for *centers of interaction*. A center of interaction provides a local context for cooperation. It has the property of being *bounded*, i.e. distinguishing what is within it from what is outside it. At the same time, the boundary *is not rigid*, i.e. people and artifacts within the center can easily move in and out. Based on this analysis, the product-based shared interaction model developed in Chapter 4 includes centers of interaction as a central part. In this model, a center of interaction consists of two main parts, a view and a medium. Cluster Layer is concerned with the view part (see also Figure 5.1 on page 106).

In Product Layer, the shared product space is created as a global information space for integrating all the work involving the shared product. Because of this focus on large-scale cooperation, Product Layer has little support for creating centers of interaction. This is illustrated in Figure 7.1.A. In this figure, users A and B are using the same product object. While Product Layer supports communication between these two users (through product awareness, participant awareness and instant messages, see Chapter 6), Cluster Layer allows them to create a local context for their cooperation (Figure 7.1.B). Local context is created by providing a *cooperative customizable view* into the shared product space. This view contains product objects and relations that are the subject of cooperation in the center of interaction. The view, together with the medium of interaction (provided by Workspace Layer, see Chapter 8) allows the members of the center of interaction to get involved in focused cooperation.

The main abstraction provided by Cluster Layer is *cluster*. A cluster helps developers to select a group of product objects and relations from the shared product space, and to use these objects and relations in a center of interaction. A cluster tailors the functionality of the shared product space to small group usage. The developers using a cluster still have access to the shared product space,

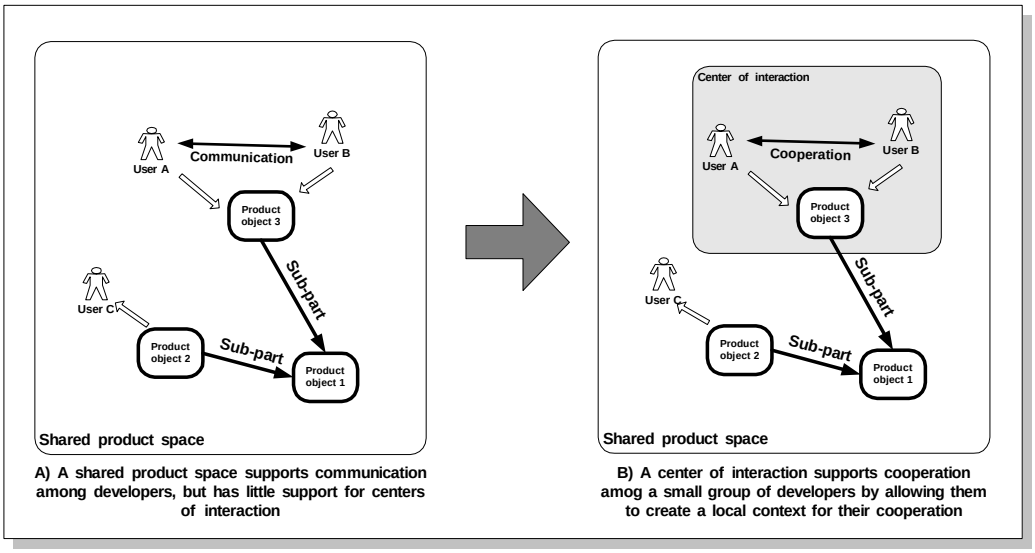


Figure 7.1: A center of interaction provides a context for groups of developers in order to support cooperation among them.

and their interaction with the cluster is fully integrated with the shared product space. In particular, all awareness information generated by Product Layer is available to the users of a cluster. This is quite different from conventional shared workspace applications, where interactions within a shared workspace is often isolated from those outside the workspace.

As an example, MultiCASE (see Section 5.3) uses clusters to allow its users create graphical diagrams. The graphical diagrams in MultiCASE contain software modules and dependency relations in a software architecture. All the objects inside a MultiCASE diagram are fully integrated into the underlying shared product space, which contains the whole software architecture. Through these graphical diagrams, MultiCASE users can change the software architecture, and at the same time be aware of the changes to the shared product space while being involved in cooperation within a shared space.

In addition to providing a shared view into the shared product space, another important property of a cluster is that it allows its users to customize the shared product space to their local needs. Different clusters can provide different views to the shared product space. In this way, clusters support local understandings of their users, who might have varying backgrounds and interests in the shared product. As a simple example, in MultiCASE, the same module can be represented in each cluster using a different shape.

Similar to Product Layer, Cluster Layer is defined in form of a set of services. These services can be used by the clients of Cluster Layer in different forms and combinations in order to provide desired functionality to the users. There are three main groups of Cluster Layer services. An overview of these services is shown in Figure 7.2. A large part of these services is concerned with managing and customizing clusters and their contents (the right-most branch in Figure 7.2). These services allow the clients to create new clusters and manage existing clusters in different

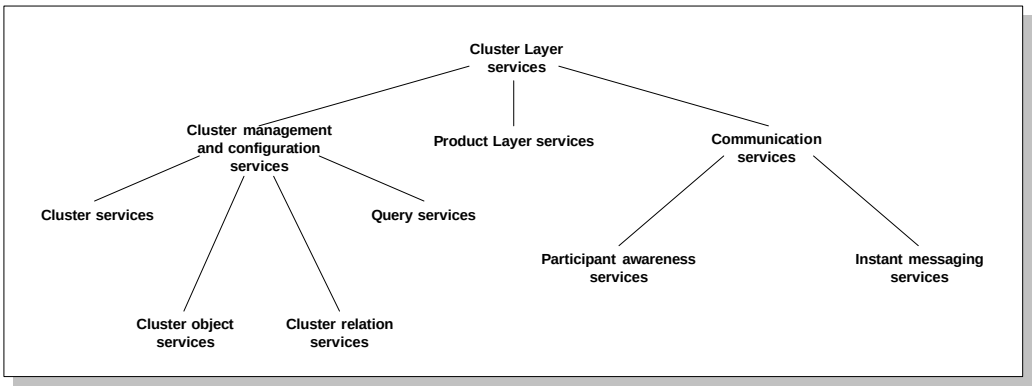


Figure 7.2: Overview of the services provided by Cluster Layer.

ways (e.g. delete, search, update, merge, etc.). They also allow the clients to modify the contents of the clusters by adding and removing objects and relations. A second important group of services allow the clients to communicate with the other clients (the left-most branch in Figure 7.2). Clients can join and leave different centers of interaction, and communicate with other clients sending and receiving instant messages.

Cluster Layer communicates with Product Layer in order to give the users access to the shared product space. A third group of services provide therefore a mapping from some of Product Layer services (the middle branch in Figure 7.2). These services include awareness configuration and shared product space query services of Product Layer. Through this mapping, the users of Cluster Layer do not need to communicate directly with Product Layer; Cluster Layer provides full access to the underlying Product Layer.

Before we describe these services in more details, in the next section we will give an explanation of the core concepts used in Cluster Layer.

7.2 Clusters, Cluster Objects and Cluster Relations

Cluster Layer provides the *cluster* abstraction as the main means of supporting cooperation in a small group. Clusters can be seen as user-defined shared views into the shared product space, used to access, arrange, edit, contextualize, customize, and possibly visualize a sub-set of product objects and relations in a locally defined way. Figure 7.3 shows a shared product space consisting of a number of product objects and relations among them (for the details of the shared product space see Chapter 6). In this figure, clusters A and B are defined on the top of the shared product space, each containing only a sub-part of this space. The users of A and B view the shared product space, and interact with it through these clusters.

Views are not uncommon in computer systems. Databases often contain large amounts of data, while individual users often need access to a small part of this data. Different users accessing the same database are therefore provided with different views. For instance, as a fundamental functionality, SQL provides support for views into relational databases, where “a view may be

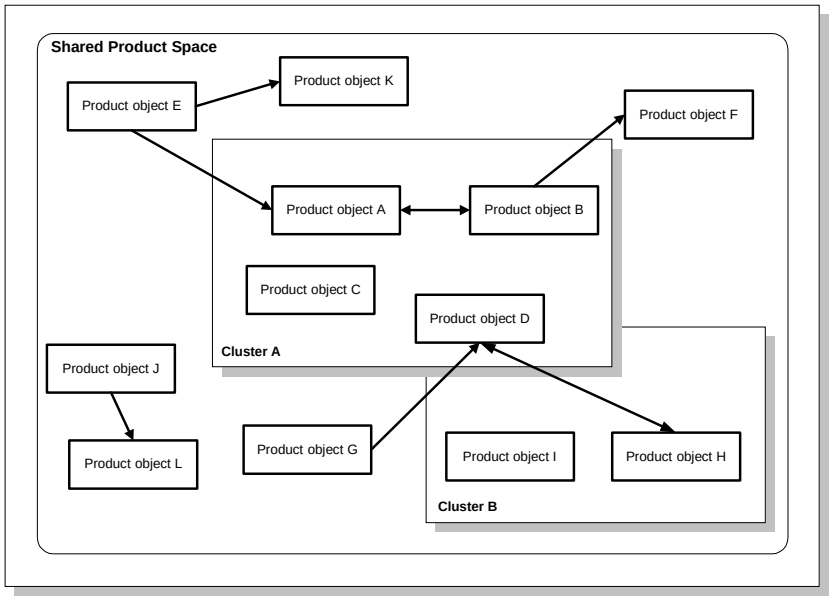


Figure 7.3: Clusters are user-defined views into the shared product space.

a subset of the database or it may contain virtual data that is derived from the database files but is not explicitly stored"(Elmasri and Navathe 1989, p.9). It is also quite common in distributed computing to provide views into shared resources, e.g. shared information spaces. All the product development tools we reviewed in Chapter 3 provide views into a shared repository containing the product. As another example, Bentley, Rodden, Sawyer and Sommerville (1992) describe an architecture for multi-user applications where each user owns a view into a shared information space. However, these views are often *individual*, while cooperation in a center of interaction needs *shared* views. Tables in a SQL database, and graphical diagrams in CASE tools, are individual views. Bentley et al. (1992) and other groupware researchers have solved this problem by having mechanisms for synchronizing the views of several users. In Cluster Layer, a cluster is an *inherently shared* view. Users can *join* a cluster, and once joined they will share the cluster and all its contents with other joined users. This implies that the activities of each joined user is visible to all the other joined users.

An additional property of clusters is that they are *cooperative*. As an example consider clusters A and B in Figure 7.3. If A and B were simply shared views, a group of users would be working with A while another group would be working with B. This means that having only shared views will allow the members of each group to cooperate internally, but the members of the two groups will not be able to cooperate across group boundaries. Clusters solve this problem by cooperating with each other. For instance, users of cluster A will be informed about the activities of the users of cluster B because there are relations connecting the contents of the two clusters. This is made possible through integrating clusters into a shared product space, which allows the users of the clusters to exchange mediated awareness information and in this way cooperate across centers of

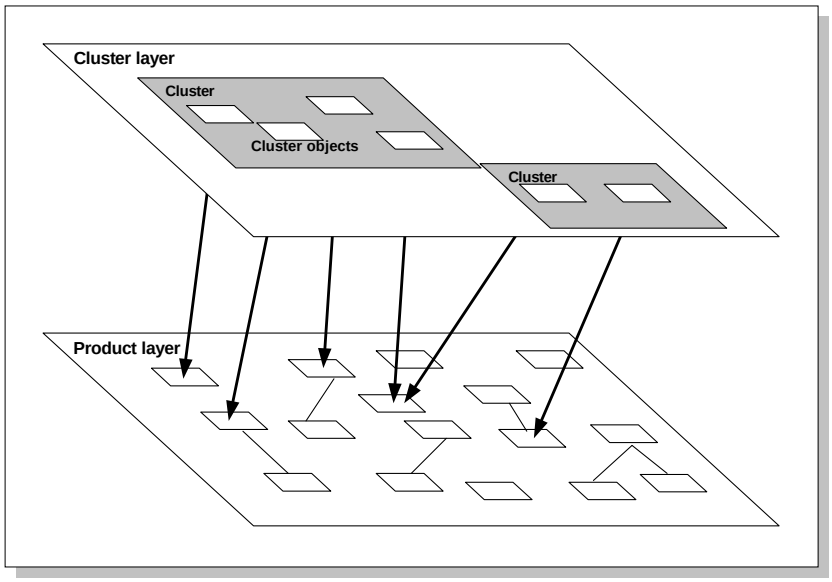


Figure 7.4: Cluster Layer uses cluster objects and cluster relations to represent product objects and conceptual relations from the shared product space.

interaction.

A third property of clusters is that they are *customizable*. Clusters, in addition to providing a view of the shared product space, allow their users to enrich this view with local information. Arbitrary information in form of local properties can be added to each object and relation in a cluster. This customization is user-defined, and is local to each cluster. Customizing a cluster supports the local activities and understandings of the users of the cluster. In addition, it allows the clients to specialize each cluster differently based on the type of activity that is to be supported.

Product objects and relations are represented in a cluster in form of *cluster objects* and *cluster relations*, respectively. Each cluster object in a cluster represents a product object in the shared product space, and each cluster relation represents a conceptual relation in the shared product space. These relations are shown in Figure 7.4. Awareness relations are not represented in Cluster Layer, but the resulting mediated awareness is made available to the users of the clusters. There is a one-to-many relation from product objects and conceptual relations in Product Layer, to cluster objects and cluster relations in Cluster Layer. Each cluster object can represent only one product object, but several cluster objects can represent the same product object. The same is true for cluster relations. Changes to a cluster object or relation in Cluster Layer may change the corresponding product object or conceptual relation in the underlying shared product space. Also, changes to product objects and conceptual relations in the shared product space will change all the cluster objects and cluster relations representing them in the Cluster Layer.

This way of separating cluster objects and cluster relations from their corresponding product objects and conceptual relations has several advantages, both conceptually and architecturally. First, it allows IGLOO clients to use Product Layer services without any knowledge of Cluster

Layer, and vice versa (see Chapter 5 on IGLOO networks). Second, separating clusters from the underlying shared product space makes it easier to customize each cluster separately, without affecting the shared product space. Third, from an architectural perspective it allows us to implement Product Layer independently from Cluster Layer. This allows for a distributed architecture with high amounts of local processing, which leads to higher performance[1].

Figure 7.5 shows in details the relations between cluster objects and cluster relations in Cluster Layer, and product objects and conceptual relations in Product Layer. Cluster objects and relations are represented in Cluster Layer in form of attribute-value pairs, while cluster relations in addition have a source and a destination cluster object (these correspond to the conceptual relation's source and destination product objects). In this figure, cluster M consists of two cluster objects X and Z, and one cluster relation Y. Cluster objects X and Y represent product objects P and R, respectively. Cluster relation Y represents conceptual relation Q in Product Layer. Note that representing a product object or a conceptual relation does not mean that all the attributes of these will be available to the users of the cluster. A cluster object may contain only a subset of the properties of the underlying product object. Cluster objects in cluster M, for instance, include only three of the six attributes that are defined for the product objects they refer to (the same is true for cluster relation Y with respect to conceptual relation Q). Those attributes of cluster objects and cluster relations that represent attributes from the underlying product objects and conceptual relations are called *deep* attributes (these are colored gray in Figure 7.5). Deep attributes are shared across different clusters since they are attributes from objects residing in the shared product space. However, each cluster may use different combinations of deep attributes available in the underlying product objects and conceptual relations.

Each cluster object, beside representing a product object from the shared product space, has a set of additional attributes that are accessible only to the users of the parent cluster. In the case of cluster M in Figure 7.5, cluster object attributes labeled M and N (colored in white) are local to the cluster. These are called *shallow* attributes. Shallow attributes are defined by the users of the cluster and may be any additional attributes.

The two types of attributes can be used to customize a cluster's contents. This is demonstrated in Figure 7.6. In this figure, we can see two clusters called M and N, each with four and three cluster objects respectively. Cluster objects X and Y representing a product object P in Product Layer are shown in more details. Attributes A, B and C in cluster object X, and attributes A, B and E in cluster object Y are deep attributes. In addition, each cluster object has its own shallow attributes. Attributes K and L in cluster object X, and attributes G, H, I and J in cluster object Y are shallow attributes. The users of a cluster can in this way decide what aspects of a product object (or a conceptual relation) should be visible in a cluster, and what additional local attributes should be added.

As an example consider the graphical diagrams used in MultiCASE (see Section 5.3). The objects in these diagrams have both deep and shallow attributes. Deep attributes include the name of the objects, their version number, the name of their owner, etc. For instance, changing the name of an object in one diagram will change the name of the object in all the other diagrams containing that object. Shallow attributes include the shape and the spatial position of each object in the diagram. Moving an object around in one diagram or changing its shape will not affect the

[1] In fact, our implementation of Cluster Layer, called CoClust (see Section 7.4) can also work without a Gossip server. This means that users can still work with clusters and their contents locally even if a shared product space is not available. This has benefits for example for mobile users. More on this later.

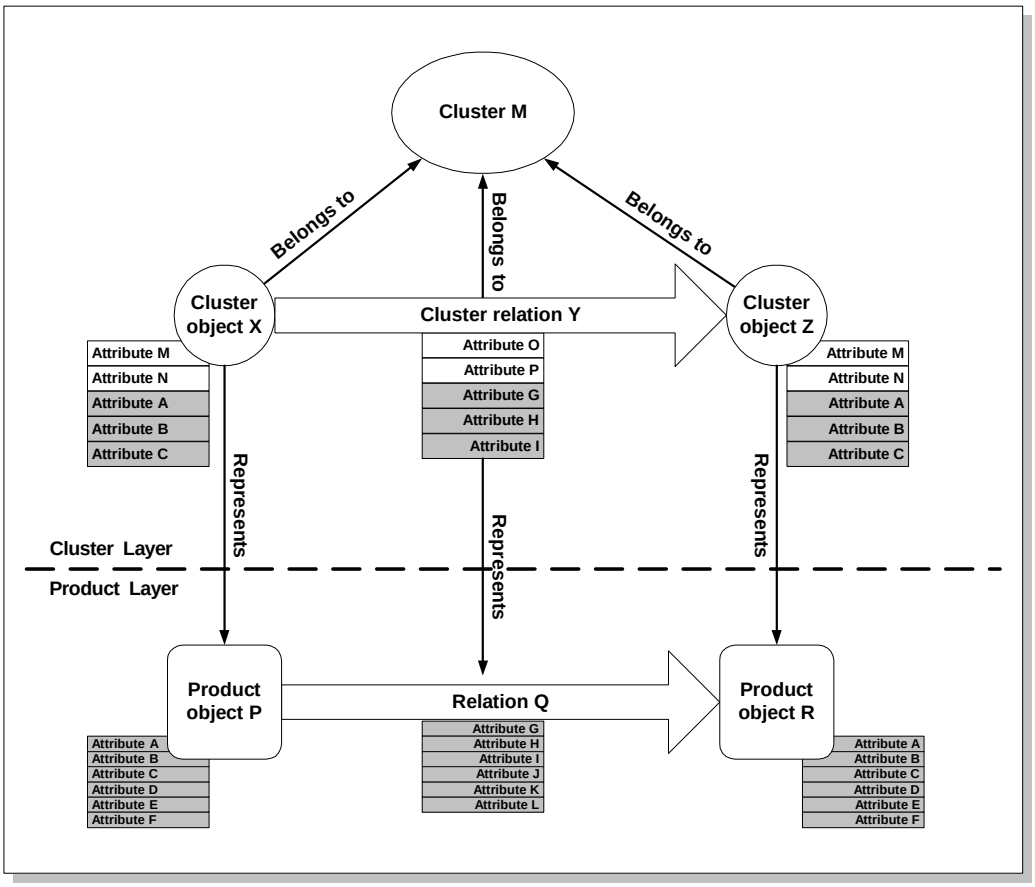


Figure 7.5: The relation between product objects and conceptual relations in Product Layer, and cluster objects and relations in Cluster Layer.

object in other diagrams.

Cluster Layer does not restrict the possible set of deep or shallow attributes for cluster objects and relations. The range of possible deep attributes is limited to the set of available attributes in the underlying product objects and conceptual relations, while shallow attributes can be defined based on the needs of the users using the cluster.

7.3 Services of Cluster Layer

The services of Cluster Layer are divided into three groups: 1) services for managing and customizing clusters, 2) services for communication among the users of Cluster Layer, and 3) services for interacting with Product Layer (see also Figure 7.2). The following sections describe each group in details.

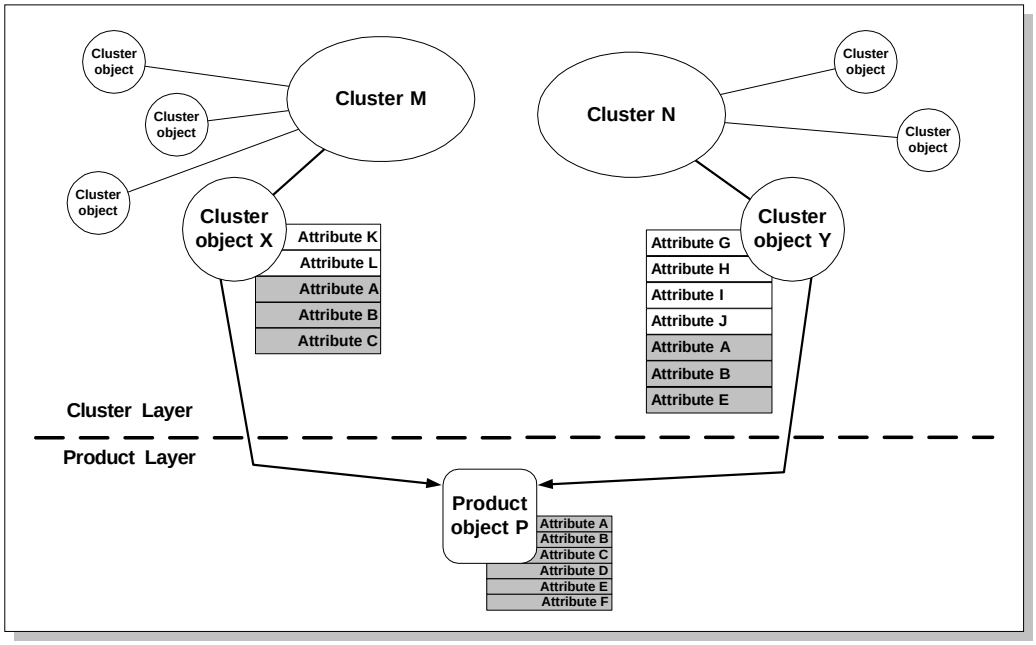


Figure 7.6: Customizing clusters using different combinations of deep and shallow attributes.

7.3.1 Cluster management and customization services

A large group of Cluster Layer services is related to the management and customization of clusters and their contents. For convenience, these services are further divided into sub-groups that relate to clusters, cluster objects, and cluster relations. In addition, services related to querying Cluster Layer for information about clusters and their contents are grouped together as a fourth sub-group.

Cluster management services are shown in Table 7.1. These services support modifying a cluster in different ways. A cluster, in addition to its contents, has a set of attribute-value pairs used for sharing information about the cluster itself. The services allow a client to create clusters, to copy clusters, to merge two clusters, to modify the set of attribute-value pairs for a cluster, and to delete clusters.

"Copy a cluster" and "Merge two clusters" need further explanation. "Copy a cluster" creates a completely new cluster containing copies of each of the cluster objects and relations in the old cluster. This means that the new cluster will have new cluster objects and relations in it, but these objects and relations will point to the same objects and relations in the shared product space as those of the original cluster.

"Merge two clusters" results in a new cluster that contains copies of all cluster objects and relations in the two original clusters. A new copy of those cluster objects and relations that exist in only one of the original clusters will be included in the new cluster without any changes. However, if two cluster objects (or two cluster relations) in the two original clusters refer to the same product object (or the same conceptual relation) in the shared product space, they have to be merged

themselves. This is because a cluster cannot have more than one cluster object (or cluster relation) pointing to the same product object (or conceptual relation). The new cluster will contain only one cluster object (or cluster relation), including all the shallow and deep attributes from both original cluster objects (or cluster relations).

Table 7.1: Cluster Layer's cluster management services

Service	**Semantics**
Create a cluster	Create a empty cluster. The client may provide a set of attribute-value pairs for initializing the cluster. A cluster identifier is returned to the client.
Delete an existing cluster	Delete an existing cluster. The client has to provide a valid cluster identifier. All cluster objects and relations within the cluster are also deleted, but product objects and conceptual relations they refer to are not changed.
Update an existing cluster	Update one or several attribute-value pairs in a cluster. The user has to provide a valid cluster identifier and a set of attribute-value pairs. For each attribute-value pair that is provided, Cluster Layer will: a) if the attribute already exists, update its value with the provided value, b) if the attribute does not exist, create the attribute and set its value to the provided value.
Remove attributes from a cluster	Remove one or several attributes from an existing cluster. The client has to provide a valid cluster identifier and a set of attribute names.
Copy a cluster	Make a new copy of a cluster, including all its cluster objects and relations. The client has to provide a valid cluster identifier for the original cluster. The identifier of the new cluster is returned to the client. The contents of the new copy point to the same product objects and conceptual relations as those of the original cluster.
Merge two clusters	Create a new cluster containing copies of all cluster objects and relations in two existing clusters. The client has to provide two valid cluster identifiers. The identifier of the new cluster is returned to the client.

Cluster object services of Cluster Layer are shown in Table 7.2. These are services for creating, deleting, and modifying cluster objects. The services distinguish between deep and shallow attributes, and allow clients to modify them separately. These services can also be used to customize clusters in a number of ways. Both deep and shallow attribute sets for cluster objects and cluster relations can be defined by the clients using these services. Note that some of these services make use of the services provided by Product Layer for modifying product objects (see Table 6.1 on page 129). However, clients of Cluster Layer will not need to know how Cluster Layer uses those services internally. Some of cluster objects services need further explanation.

For creating a cluster object, clients can choose one of two services: "Shallow-create" and "Deep-create." The first one creates a cluster object based on an already-existing product ob-

ject, while the second one also creates a new product object in the shared product space. Using "Shallow-create" will normally require browsing the contents of the shared product space in order to allow the clients to choose a product object. This access is provided by Product Layer service mapping as described in Section 7.3.3. Both services allow the clients to initialize the new cluster object with shallow and deep attributes.

Deleting a cluster object is similar. There are two services for deleting cluster objects: "Shallow-delete" and "Deep-delete." The first one deletes only the cluster object and its shallow attributes, while the second one also removes the underlying product object from the shared product space.

There are two services that allow a client change a shallow attribute to deep, and vice versa. These services are important in order to allow local modifications become public gradually. Clients may create cluster objects that are "empty," i.e. that contain a minimum set of attributes. These objects can then be refined as the users gradually get a clearer idea about what attributes the objects should contain. As we saw in Chapter 2, gradual refinement of ideas is characteristic for product development. Proper use of these two services by a group of clients can make this gradual refinement highly user friendly[2]. Also note that clients cannot directly add or remove deep attributes from a cluster object. A deep attribute has to be added as a shallow attribute first, and then transferred into a deep attribute. In the same way, a deep attribute has to be transferred into a shallow attribute before it can be deleted[3].

Table 7.2: Cluster Layer's cluster object services

Service	**Semantics**
Shallow-create cluster object	Create a new cluster object based on an existing product object from the shared product space. The client has to provide a valid cluster identifier for indicating which cluster this cluster object should belong to. In addition, the client has to provide a valid product object identifier. The new cluster object will be set to point to the product object with the provided identifier. An optional list of attribute-value pairs can be passed for initializing the shallow attributes of the cluster object. An optional list of attribute names can be passed to tell Cluster Layer which deep attributes (from the attributes of the underlying product object) should be included in this cluster object.

Continued on next page

[2] An example of a system supporting this type of transition from informal to formal is provided by Pendergast, Aytes and Lee (1999). In this system, users can gradually create conceptual models by attaching textual annotations to concepts. These annotations are later refined into attributes for those concepts.

[3] This is also what happens when a client calls "Deep-delete" service on a cluster object. Cluster Layer transfers all the deep attributes of the object into shallow attribute before deleting it. This way of accessing the shared product space supports future implementations of checking out/in of information to the shared product space.

Continued from previous page

Service	Semantics
Deep-create cluster object	Create a new cluster object. Also create a product object in the shared product space and set this cluster object to point to it. The client has to provide a valid cluster identifier for indicating which cluster this cluster object should belong to. Two lists of attribute-value pairs can be passed for initializing both shallow and deep attributes of the cluster object. The underlying product object will be initialized with the attribute-values in the deep attributes list. The identifier of the new object will be returned to the client.
Shallow-delete cluster object	Delete an existing cluster object. The client has to provide a valid cluster identifier and a valid cluster object identifier. This service does not delete the underlying product object that the cluster object points to.
Deep-delete cluster object	Delete an existing cluster object. The client has to provide a valid cluster identifier and a valid cluster object identifier. This service also deletes the underlying product object that this cluster object points to.
Update shallow attributes of an existing cluster object	Update one or several shallow attribute-value pairs of a cluster object. The client has to provide a valid cluster identifier, a valid cluster object identifier, and a set of one or more attribute-value pairs. For each provided attribute-value pair that is provided, Cluster Layer will: a) if the shallow attribute already exists, update its value with the provided value, b) if the shallow attribute does not exist, create the shallow attribute and set its value to the provided value.
Update deep attributes of an existing cluster object	Update one or several deep attribute-value pairs of a cluster object. The client has to provide a valid cluster identifier, a valid cluster object identifier, and a set of one or more attribute-value pairs. All the provided attributes have to be valid deep attributes of the cluster object. For each attribute-value pair, Cluster Layer will update the value of the attribute (note that the corresponding attribute of the underlying product object is also updated).
Remove shallow attributes of an existing cluster object	Remove one or several shallow attributes from an existing cluster object. The client has to provide a valid cluster identifier, a valid cluster object identifier, and a set of attribute names.
Make a shallow attribute deep	Remove a shallow attribute from a cluster object and add the attribute to the cluster object as a deep attribute. Note that the attribute is also added to the underlying product object. The client has to provide a valid cluster identifier, a valid cluster object identifier, and the name of the shallow attribute.

Continued on next page

Continued from previous page

Service	Semantics
Make a deep attribute shallow	Remove a deep attribute from a cluster object and add the attribute to the cluster object as a shallow attribute. Note that the attribute is also removed from the underlying product object. The client has to provide a valid cluster identifier, a valid cluster object identifier and the name of the deep attribute.

A set of services similar to those for cluster object manipulation is provided for cluster relations. These services are shown in Table 7.3. Note that source and destination of a cluster relation is always the same as those of the underlying conceptual relation. It is up to the clients to make proper use of this information. As an example consider cluster M in Figure 7.5 on page 157. Here, the client using M may wish to visualize cluster relation Y as an arrow from Y's source to its destination. For doing this, the client has to find out whether there are any cluster objects in M that point to Y's source or destination product objects (in this case, cluster objects X and Z).

Table 7.3: Cluster Layer's cluster relation services

Service	Semantics
Shallow-create cluster relation	Create a new cluster relation based on an existing conceptual relation from the shared product space. The client has to provide a valid cluster identifier for indicating which cluster this cluster relation should belong to. In addition, the client has to provide a valid conceptual relation identifier. The new cluster relation will be set to point to the conceptual relation with the provided identifier. An optional list of attribute-value pairs can be passed for initializing the shallow attributes of the cluster relation. A list of attribute names can be passed to tell Cluster Layer which deep attributes (from the attributes of the underlying conceptual relation) should be included in this cluster relation.
Deep-create cluster relation	Create a new cluster relation. Also create a conceptual relation in the shared product space and set this cluster relation to point to it. The client has to provide a valid cluster identifier for indicating which cluster this cluster relation should belong to. Two lists of attribute-value pairs can be passed for initializing both shallow and deep attributes of the cluster relations. The underlying conceptual relation will be initialized with the attribute-values in the deep attributes list. The identifier of the new relation will be returned to the client.
Shallow-delete an existing cluster relation	Delete an existing cluster relation. The client has to provide a valid cluster identifier and a valid cluster relation identifier. This service does not delete the underlying conceptual relation that the cluster relation points to.

Continued on next page

Continued from previous page

Service	Semantics
Deep-delete an existing cluster relation	Delete an existing cluster relation. The client has to provide a valid cluster identifier and a valid cluster relation identifier. This service also deletes the underlying conceptual relation that this cluster relation points to.
Update the source of a cluster relation	Update the source of an existing cluster relation. This update is done in both the cluster relation and the underlying conceptual relation.
Update the destination of a cluster relation	Update the destination of an existing cluster relation. This update is done in both the cluster relation and the underlying conceptual relation.
Update shallow attributes of an existing cluster relation	Update one or several shallow attribute-value pairs of a cluster relation. The client has to provide a valid cluster identifier, a valid cluster relation identifier, and a set of one or more attribute-value pairs. For each provided attribute-value pair that is provided, Cluster Layer will: a) if the shallow attribute already exists, update its value with the provided value, b) if the shallow attribute does not exist, create the shallow attribute and set its value to the provided value.
Update deep attributes of an existing cluster relation	Update one or several deep attribute-value pairs of a cluster relation. The client has to provide a valid cluster identifier, a valid cluster relation identifier, and a set of one or more attribute-value pairs. All the provided attributes have to be valid deep attributes of the cluster relation. For each attribute-value pair, Cluster Layer will update the value of the attribute (note that the corresponding attribute of the underlying conceptual relation is also updated).
Remove shallow attributes from a cluster relation	Remove one or several shallow attributes from an existing cluster relation. The client has to provide a valid cluster identifier, a valid cluster relation identifier, and a set of attribute names.
Make a shallow attribute deep	Remove a shallow attribute from a cluster relation and add the attribute to the cluster relation as a deep attribute. Note that the attribute is also added to the underlying conceptual relation. The client has to provide a valid cluster identifier, a valid cluster relation identifier, and the name of the shallow attribute.
Make a deep attribute shallow	Remove a deep attribute from a cluster relation and add the attribute to the cluster relation as a shallow attribute. Note that the attribute is also removed from the underlying conceptual relation. The client has to provide a valid cluster identifier, a valid cluster relation identifier and the name of the deep attribute.

The last group of cluster management services is concerned with querying the contents of Cluster Layer. These services are shown in Table 7.4. Clients can query Cluster Layer for information about clusters, cluster objects, and cluster relations. There are also several search services

for searching for objects or relations with specific attributes. Note that search services operate on single clusters. For instance, searching for a cluster object with a specific attribute can be done only within a single cluster. This is done deliberately in order to emphasize the privacy of the users. The contents of a cluster are meant to be accessed by the users of that cluster. Being able to search all the contents of Cluster Layer for such local information is not desirable. Product Layer services for querying the shared product space (shown in Table 6.1 on page 129) are supported by Cluster Layer and can be used for global search (see also Section 7.3.3).

In addition, many of these services require the clients to to join a cluster (see Section 7.3.2 for joining a cluster) before being able to query Cluster Layer for information about the cluster. Joining a cluster will allow other users of the cluster to get notified about the new user. The users will then know that the new user might access the information that is represented by the cluster.

Table 7.4: Cluster Layer's query services

Service	**Semantics**
Get attribute values for a cluster	Return the current values of one or several attributes for a given cluster. The client has to provide a valid cluster identifier and a set of one or several attribute names.
Get all attribute-value pairs for a cluster	Return the set of all attribute-value pairs for a given cluster. The client has to provide a valid cluster identifier.
Search for clusters with a specific attribute-value	Return cluster identifiers of all the existing clusters with a specific value for a specific attribute. The client has to provide an attribute name and a value for that attribute.
Search for clusters with a specific attribute	Return cluster identifiers of all the existing clusters that have a specific attribute. The client has to provide an attribute name in form of a string.
Get all existing clusters	Return a list of the identifiers of all the existing clusters.
Get shallow attribute values for a cluster object	Return the current values of one or several shallow attributes for a given cluster object. The client has to provide a valid cluster identifier, a valid cluster object identifier, and a set of one or several attribute names.
Get deep attribute values for a product object	Same as above for deep attributes.
Get all shallow attribute-value pairs for a cluster object	Return the set of all shallow attribute-value pairs for a given cluster object. The client has to provide a valid cluster identifier and a valid cluster object identifier.
Get all deep attribute-value pairs for a cluster object	Same as above for deep attributes.
Get all attribute-value pairs for a cluster object's underlying product object	Return all attribute-value pairs of a cluster object's underlying product object. The client has to provide a valid cluster identifier and a valid cluster object identifier.

Continued on next page

Continued from previous page

Service	Semantics
Search a cluster for cluster objects with a specific shallow attribute-value	Return cluster object identifiers of all the existing cluster objects in a cluster with a specific value for a specific shallow attribute. The client has to provide a valid cluster identifier, a shallow attribute name, and a value for that attribute.
Search a cluster for cluster objects with a specific deep attribute-value	Same as above for deep attributes.
Search a cluster for cluster objects with a specific shallow attribute	Return cluster object identifiers of all the existing cluster objects in a cluster that have a specific shallow attribute. The client has to provide a valid cluster identifier and an attribute name.
Search a cluster for cluster objects with a specific deep attribute	Same as above for deep attributes.
Get all existing cluster objects in a cluster	Return a list of the identifiers of all the existing cluster objects in a cluster. The client has to provide a valid cluster identifier.
Get shallow attribute values for a cluster relation	Return the current values of one or several shallow attributes for a given cluster relation. The client has to provide a valid cluster identifier, a valid cluster relation identifier, and a set of one or several attribute names.
Get deep attribute values for a product relation	Same as above for deep attributes.
Get all shallow attribute-value pairs for a cluster relation	Return the set of all shallow attribute-value pairs for a given cluster relation. The client has to provide a valid cluster identifier and a valid cluster relation identifier.
Get all deep attribute-value pairs for a cluster relation	Same as above for deep attributes.
Get all attribute-value pairs for a cluster relation's underlying conceptual relation	Return all attribute-value pairs of a cluster relation's underlying conceltual relation. The client has to provide a valid cluster identifier and a valid cluster relation identifier.
Get the source for a cluster relations	Return the identifier of the source product object for a cluster relation. The client has to provide a valid cluster identifier and a valid cluster relation identifier.
Get the destination for a cluster relations	Return the identifier of the destination product object for a cluster relation. The client has to provide a valid cluster identifier and a valid cluster relation identifier.
Search a cluster for cluster relations with a specific shallow attribute-value	Return cluster relation identifiers of all the existing cluster relations in a cluster with a specific value for a specific shallow attribute. The client has to provide a valid cluster identifier, a shallow attribute name, and a value for that attribute.
Search a cluster for cluster relations with a specific deep attribute-value	Same as above for deep attributes.

Continued on next page

Continued from previous page

Service	Semantics
Search a cluster for cluster relations with a specific shallow attribute	Return cluster relation identifiers of all the existing cluster relations in a cluster that have a specific shallow attribute. The client has to provide a valid cluster identifier and an attribute name.
Search a cluster for cluster relations with a specific deep attribute	Same as above for deep attributes.
Get all existing cluster relations in a cluster	Return a list of the identifiers of all the existing cluster relations in a cluster. The client has to provide a valid cluster identifier.

All cluster management and configuration services produce awareness information as they are performed. This awareness information is in form of *awareness events* that are distributed to the proper clients. In this sense, production and distribution of awareness information is similar to that of Product Layer (see Section 6.2). However, as opposed to Product Layer, where each client has to precisely define an awareness subscription, distribution of awareness information to the clients of Cluster Layer is mainly based on what clusters each client is using. A client will receive awareness information related only to its joined clusters (see Section 7.3.2 for joining a cluster). This makes unnecessary the need for awareness subscription services similar to those of Product Layer; once a client joins a cluster, Cluster Layer provides for necessary awareness information.

Cluster management and configuration services may produce awareness information that belongs to one of two types: *product awareness* and *cluster awareness*. Product awareness is generated as a result of access to the shared product space; as Cluster Layer's clients access deep attributes in cluster objects and cluster relations, Cluster Layer communicates these accesses to Product Layer, which generates the necessary product awareness information as described in Chapter 6. Cluster Layer in turn receives product awareness information produced by Product Layer and re-distributes this information to its own clients.

Cluster awareness information, on the other hand, is generated by Cluster Layer itself. As Cluster Layer's clients access shallow attributes of cluster objects and cluster relations in a cluster, cluster awareness information is generated. This information is sent only to the other users of the same cluster. In this way, product and cluster awareness support the cooperation within a cluster, while product awareness is used to support the cooperation across clusters.

7.3.2 Communication services

Communication services support communication among the users of Cluster Layer. These services are meant as a basic set of enabling services that can be used for building more advanced support for communication. A group of services are used for supporting *participant awareness*, i.e. to let users know who is using Cluster Layer and its clusters. Joining/Leaving Cluster Layer and Joining/Leaving a cluster results in participant awareness events sent to the other joined clients. Each client can see at any time which other clients are using Cluster Layer, and which other clients are using a particular cluster. In addition, a set of services allow clients to send instant text messages to other clients.

Table 7.5: Cluster Layer's communication support services

Service	Semantics
Join Cluster Layer	Join Cluster Layer. The client has to provide a valid client identifier.
Leave Cluster Layer	Leave Cluster Layer.
Get a list of all joined clients	Get a list of the identifiers of all the clients that have currently joined Cluster Layer.
Join a cluster	Join a cluster and starts sharing it with other joined clients. The client has to provide a valid cluster identifier.
Leave a cluster	Leaves an already joined cluster. The client has to provide a valid cluster identifier.
Get a list of the clients that have joined a cluster	Get a list of the identifiers of all the clients that have currently joined a cluster. The client has to provide a valid cluster identifier.
Get activity level for a client	Return an indicator of how active a client has been recently. The indicator can be high, medium, low, or no activity.
Send instant message to all the joined clients of a cluster	Send an instant text message to all the clients that have joined a cluster. The client has to provide a valid cluster identifier. Requires being already a joined client of the cluster.
Send instant message to a joined client of a cluster	Send an instant text message to a client that has joined a cluster. The client has to provide a valid cluster identifier and a valid client identifier. Requires being already a joined client of the cluster.
Send instant message to clients that have joined Cluster Layer	Send an instant text message to a client that has joined a cluster. The client has to provide a valid cluster identifier and a valid client identifier. Requires being already a joined client of the cluster.

7.3.3 Product Layer services

The set of services provided by Cluster Layer has to be complete enough in order to allow an application to integrate into an IGLOO network through Cluster Layer without having any knowledge of Product Layer (see Figure 5.8 on page 120 and the discussion on IGLOO networks in Section 5.4). This means that Cluster Layer, in addition to providing its own advanced services, must provide a mapping onto the services provided by Product Layer. For this reason, many of the services that are provided by Product Layer are also supported by Cluster Layer. Cluster Layer simply requests these services from Product Layer on behalf of its clients, and forwards the results of the services to the clients that requested the services.

Accessing product objects and conceptual relations in the shared product space is possible through Cluster Layer's cluster management and customization services discussed in Section 7.3.1. In addition, Cluster Layer supports shared product space query services of Product Layer (these services are shown in Table 6.1). These services allow the clients of Cluster Layer to

investigate the contents of the shared product space. This might be necessary for instance when inserting objects and relations into a cluster.

We have already explained how product awareness information that is generated by Product Layer is made available to the clients of Cluster Layer. In addition to providing product awareness information to its clients, Cluster Layer gives access to the awareness configuration services of Product Layer shown in Table 6.2 on page 136.

Regarding subscription services of Product Layer (see Table 6.3 on page 139), a watch list is maintained automatically by Cluster Layer for each cluster. Cluster Layer's clients only need to defined a cluster's contents. Cluster Layer automatically creates a watch list for the users of the cluster (the watch list is obviously the same for all the users of the cluster). However, the clients may update the operation lists and the subscribed awareness schemes for each cluster.

7.4 The Implementation of Cluster Layer: CoClust

CoClust is an implementation of Cluster Layer in the Java programming language[4]. CoClust implements a sub-set of Cluster Layer services discussed in Section 7.3. CoClust consists of a stand-alone network *server* that provides access to Cluster Layer services through a well-defined *client extension*. CoClust client extension not only makes it easier to use the server (by providing a set of Java classes with easy-to-use interfaces), but also increases the performance of the clients by allowing for increased local processing. Good performance is crucial for Cluster Layer because cooperation in a center of interaction often has a high pace and require high feedback and feedthrough speed. In this section we will first look at the overall architecture of CoClust, and will then describe CoClust client extension and how it can be used by Cluster Layer clients.

7.4.1 An overall view of CoClust

Figure 7.7 shows the overall architecture of CoClust. *CoClust server* is where all the existing clusters and their contents are stored. CoClust server is also the component of CoClust that is responsible for communication with an underlying Gossip server in order to provide access to a shared product space[5].

There are two types of cluster in CoClust server. *Active* clusters are those that have at least one joined client, while *inactive* clusters are not being used by anyone. Inactive clusters are stored in *Cluster Database*. Clients can ask CoClust server to join them to a cluster. Once a client joins an inactive cluster, the cluster is moved from the Cluster database to *active cluster space*. Active clusters can be changed by both a number of clients and an underlying Gossip server. For instance, clients might want to change shallow and deep attributes of a cluster's contents, while Gossip will need to change a cluster if other Gossip clients have made changes that affect that cluster's deep attributes. Therefore, active cluster space enforces consistency preservation policies on active clusters. Each active cluster is subscribed to proper network channels, and updating the clusters happens according to network traffic in these channels. This will be explained bellow.

[4]The initial version of CoClust is implemented by Lie-Nielsen (2000), a diploma student at NTNU who was supervised by this author, Terje Brasethvik, and Monica Divitini.

[5]CoClust will also work without a Gossip server, but will then function as a "cluster server," i.e. the clients will only have access to shallow properties of cluster objects and relations.

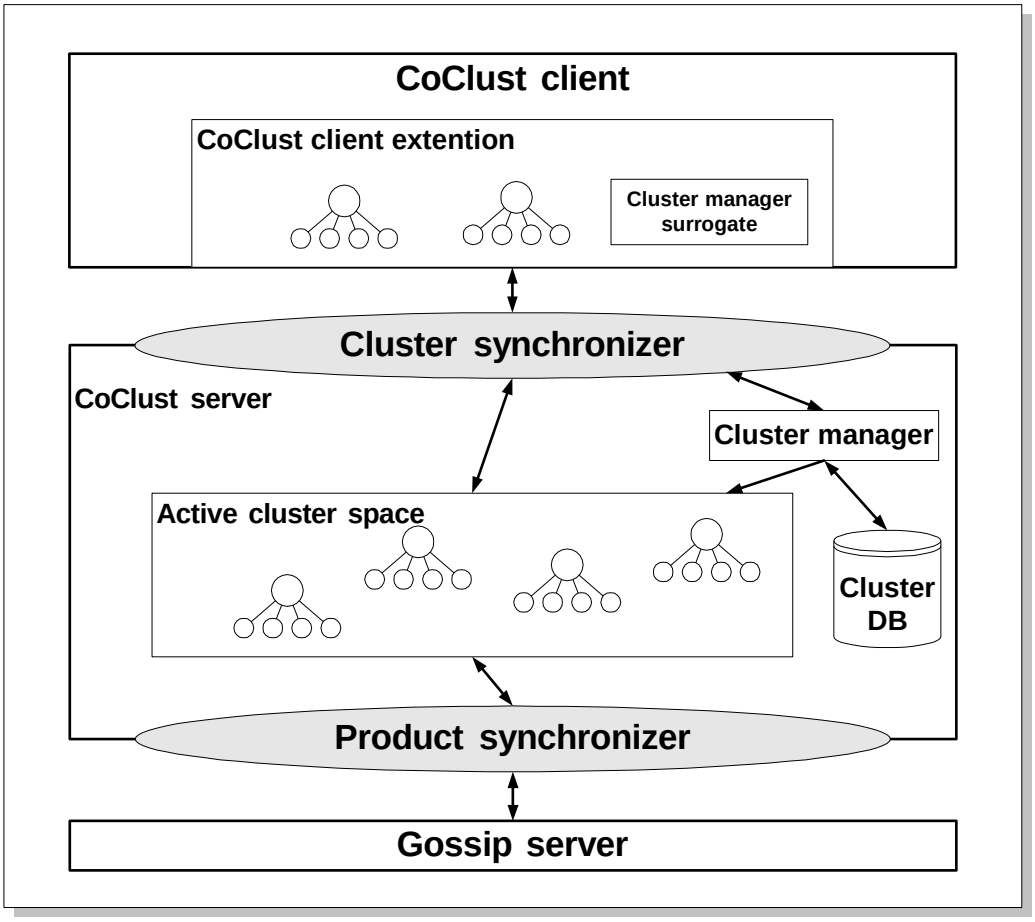


Figure 7.7: The overall architecture of CoClust.

One important responsibility of CoClust is that of keeping the different *replicas* of an active cluster identical to each other. For each active cluster, in addition to a *master replica* on the server, a *local replica* exists at each of the joined clients (see Figure 7.7). CoClust server contains the master replica of all the active clusters. A master replica is where clients retrieve cluster information when they want to join a cluster[6]. A master replica also functions as a "multiplexer" between Cluster Layer clients and the underlying Gossip server. This means that all changes to the deep attributes of a cluster's contents (changes that are possibly done by several clients using the cluster) are reported to Gossip by the master copy. In addition, changes in the shared product space are first received by the master copy and then distributed to the clients of Cluster Layer.

[6]In this way CoClust does not implement a pure peer-to-peer architecture as for instance TeamWave (see Chapter 3 for TeamWave). In TeamWave, a new client can ask any of the other clients for state information, while in CoClust this information can only be retrieved from CoClust server that has the master replica.

Local replicas are implemented by CoClust client extension. CoClust allows the clients to manipulate their local replicas as any other local data structure they might have, while synchronizing the replicas is the responsibility of CoClust. Having a local replica means increased ease of use (because clients do not need to operate "over the network" all the time) and increased local processing (because the data is stored locally at each client's network node). The replicas are automatically synchronized with each other and with the master replica as the clients or the underlying Gossip server make modifications to them. Note that synchronizing the replicas of a cluster is the main *awareness support* mechanism in CoClust. Once a local replica of a cluster is changed by a client's user, the change is replicated to all the other users sharing that cluster. In this way all the users can see how other users change a cluster. In addition, each cluster also provides its joined clients with mediated notifications received from Gossip. More on how this is done later.

Cluster synchronizer is a network channel that is used for synchronizing local replicas of an active cluster with each other and with the master replica. For each active cluster there is one cluster synchronizer. For each active cluster, the master replica (on CoClust server) and each of the local replicas (on each joined client) subscribe to this channel. When a local replica or the master replica is changed, notifications about the change are broadcasted on this channel. These notifications contain enough information to allow all the replicas update their state according to the changes. Cluster synchronizer allows thus the real-time sharing of a cluster by a group of clients.

When clients access a part of their local replica that involves the shared product space (i.e. when they access deep attributes of cluster objects and relations), the access has to be propagated to the underlying Gossip server. Updates can go downwards (in cases when CoClust has to change the shared product space) and upwards (in cases when Gossip informs CoClust about changes to the shared product space). These updates are synchronized through a network channel called *product synchronizer*. There is one product synchronizer for each active cluster. Only the master replica of each active cluster subscribes to the product synchronizer. When a notification from Gossip arrives through the product synchronizer, the master replica updates itself and broadcasts the update onto the cluster's cluster synchronizer, which allows each local replica to update itself. When a local replica accesses a deep attribute of a cluster object, the master replica for that cluster detects the modification and informs Gossip about it. The reason we have chosen to have a separate network channel for updates related to the shared product space is that local updates can be performed without worrying about the consistency of the underlying shared product space. Having a separate channel for local updates makes it easier to optimize the performance of these updates.

Cluster manager (see Figure 7.7) is in charge of cluster management services. These services include creating new clusters, deleting existing clusters, browsing existing clusters, merging clusters, glancing into a cluster to see who is sharing it, and joining or leaving a cluster. Cluster manager is also responsible for making an inactive cluster active and vice versa.

7.4.2 CoClust client extension

As we have seen, there are two types of cluster in CoClust, active and inactive. When a cluster is active, i.e. is joined by one or several clients, it has two or more replicas, one master replica at CoClust server and one local replica at each of the joined clients. This is shown in Figure 7.8. In this figure, cluster M is joined by clients A and B, and cluster N is joined by clients C and D. Sharing a cluster in this way supports tight coupling among the members of a small group,

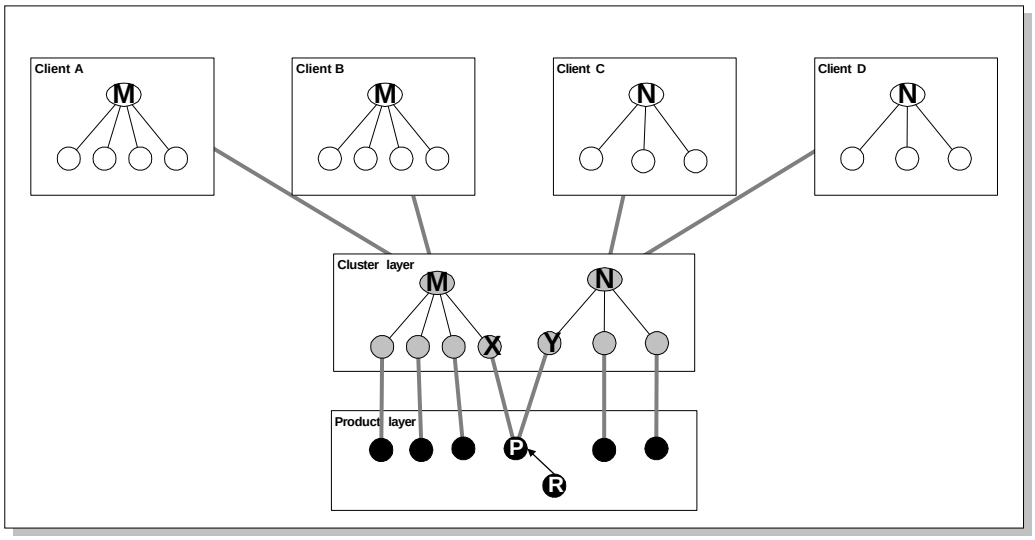


Figure 7.8: Overview of cluster sharing and product awareness in Cluster Layer.

meaning that any changes to cluster objects and relations in the cluster will be visible to all the users of the cluster. *CoClust client extension* is in charge of implementing the local replicas, and providing an easy-to-use interface to CoClust. The client extension implements a set of java classes that provide access to Cluster Layer services while hiding the complexities of network connections and synchronization algorithms used in CoClust.

The main classes provided by the client extension are *Cluster*, *Cluster Object*, *Cluster Relation*, and *Cluster Manager*. A client connects to a CoClust server by initiating a Cluster Manager. Cluster Manager will automatically set up a network connection to a server based on the information provided by the client (i.e. server address, port number, user name, password, etc.). Once connected to a CoClust server, Cluster Manager functions as a local surrogate for the server (see Figure 7.7). The client will have access to all the services of the server by simply calling methods on Cluster Manager. In particular, the client can ask Cluster Manager to create local replicas of clusters by making new instances of the Cluster class. A new cluster can be created through instantiating such a Cluster class by providing initialization information. An existing cluster in the CoClust server can be accessed by instantiating a Cluster class using the identifier of the cluster. Upon instantiation, the new local replica will set up all necessary network channels and start communicating with its master replica and possibly other local replicas.

Once a Cluster is instantiated, it can be accessed by the client in the same way as a local data structure, e.g. by calling methods on the Cluster and its Cluster Objects. The only difference is that the methods can be executed only if the necessary *token* for the affected object can be retrieved from the cluster server (see Section 7.4.3 for details). Once a method is called and the local cluster is updated, all its replicas on the cluster server and other clients sharing the cluster will be updated.

Once a local replica of a client is changed as a result of changes done by other clients or

Gossip, the changes have to be communicated to the client so that it can take the necessary action (for instance animate other user's actions for its own user). For making this possible, CoClust requires its clients to provide a number of *call-back functions* for different types of notifications that may arrive at the local replicas. One such function is called upon updates to a Cluster or its contents (direct notification call-back). Another function is called when mediated notifications arrive from Gossip (mediated notification call-back). Other functions are required for communication and system notifications. Each client can decide to implement these callback functions in different ways.

As opposed to Gossip, CoClust does not have a public network protocol. Although the services we have defined for Cluster Layer could have been implemented in form of a network protocol[7], we have decided to provide a client extension because of several reasons. First, clusters are higher level abstractions. A linear network protocol would have been too complicated and difficult to use. An object-oriented client extension is more intuitive than a network protocol with many parameters. Second, the client extension provides an API that can be implemented in different ways. In particular it is highly probable that we will improve the concurrency control mechanisms used in the current version of CoClust. These improvements will require changes in the protocol, while the client extension can hide the changes. Third, and maybe most important, is that CoClust client extension allows the clients to operate on local data structures (local replicas). This, combined with an optimistic concurrency control mechanisms, can provide high performance. With a network protocol the clients would not have access to local replicas, or would have had to implement local replicas in an ad hoc manner.

7.4.3 CoClust's internal consistency

Clients operate on their local replica when they want to update a shared cluster. Once a client modifies its local replica, the master replica and other local replicas have to be updated. Since a cluster can be updated by many clients and possibly by a Gossip server, concurrency control is needed in order to prevent inconsistencies in the shared state of the cluster. This problem is similar to concurrently editing product objects in Gossip. We explained in Section 6.3.4 how tokens can be used for controlling access to product objects. For CoClust we have an additional requirement that was not as important for Gossip, e.g high performance. Since clusters are used in a small, tightly coupled group, the users will make rapid and frequent changes to the clusters. The feedthrough of changes has to be fast in order to allow group members constantly see the last updated picture. This is in particular true for shallow attributes since they are the "working memory" of the group and are changed more rapidly than deep attributes. As we implemented the first version of MultiCASE (see Section 5.3 on page 110), we used a token-based mechanism combined with *identified locks* (Mariani and Prinz 1993) for concurrent updating of clusters. Such a mechanism is useful in that the user interfaces of all the clients show all the time which objects are locked and by whom. This provides essential awareness information that may somehow outweigh the pessimistic nature of locking because users are hinted about what is happening inside. However, users wishing to modify a cluster still have to wait for the proper token to be retrieved through the network. In this section we describe how this token-based mechanism works in CoClust. The implementation of an optimistic algorithm is explained in (Lie-Nielsen 2000) and will be integrated into future

[7] Of course the services are implemented as a network protocol, but this protocol is not meant to be public. See (Lie-Nielsen 2000) for the details of this protocol.

versions of CoClust.

Consider a scenario where client A's user (see Figure 7.8) modifies a shallow attribute in cluster object X as a result of user interaction. This type of modification is common when a small group is working with the local information in a cluster. For this to happen, the local replica of M at A has to acquire a token for the affected cluster object. When this is done, the local replica can be changed safely. The change is broadcasted to all other replicas (including the master replica). In this case client B (belonging to another member of the small group) contains a local replica of M. This replica will receive a notification indicating what was changed by which client. The local replica of M at B will update itself accordingly, and will then call the call-back function of B. B can then update its screen image, possibly animating the action of the user of A.

Another scenario is when client A's user changes a deep attribute of cluster object X. This is a normal case when members of a small group wish to modify the shared product space. In this case the cluster needs to acquire a token for product object P. Once this is done, updates to X are propagated as above while product object P is also changed in the process. This change is propagated to Gossip, and is further broadcasted by Gossip as a direct notification to all the users of P, which also includes cluster N with cluster object Y. Y is updated accordingly, and the update is communicated to clients C and D (members of another small group using cluster N). The call-back function for direct notifications is called at C and D. C and D can then decide on how to react to the notification.

A third scenario is when a product object not represented in any of the clusters M or N is modified, and there exists an awareness relation from this object to an object in one of the clusters. This is a typical case of peripheral awareness, where a small group monitors a part of the shared product space for changes. In Figure 7.8, if product object R is modified by a fifth client E (not shown in the figure), product object P will be notified and a mediated notification will be sent (by Gossip) to all the users of P. In this case this includes clusters M and N. This mediated notification is communicated to all the local replicas of M and N, and each replica will call the mediated notification call-back function of its client. Note that mediated notification does not change the cluster's shared state, and contains only peripheral information for the users of the cluster. As a consequence there is no need for acquiring a token.

In none of these three scenarios we have made any assumptions about the clients except that they should have a number of call-back functions. This was one of the initial intentions of having the client extension, i.e. requiring minimum changes to the clients.

7.5 Summary

In this section we have described Cluster Layer, a component of IGLOO framework that is in charge of supporting small groups of developers interacting with a large shared product space. Cluster Layer works in close cooperation with Product Layer in order to allow groups of developers create their local view into the shared product space, share this view, and customize the contents of the view according to their local needs. A set of services for Cluster Layer are defined. Cluster Layer's implementation in form of a stand-along network-based application called CoClust is described.

Cluster Layer satisfies a part of the requirements we posed to a center of interaction in Chapter 2. Cluster Layer allows developers to create generic focus points that can be used for interacting with a large shared product space. The cluster abstraction of Cluster Layer allows the users

to define such focus points in a flexible way. Cluster Layer does not make any assumptions about what kind of information will exist in a cluster. Clusters may be created as the need for having them emerge, or they may be created in a predefined way as steps in a formal workflow definition. Clusters have boundaries in that only a sub-part of the shared product space is within a cluster. At the same time, these boundaries are fluid. First, objects and relations can be added or removed easily. Second, a cluster is not totally isolated from what is outside in that it provides peripheral awareness of those parts of the shared product space that are not included within its boundaries.

Clusters also have fluid boundaries with respect to the users, i.e. users can easily join and leave clusters and see who is working on different clusters. The contents of a cluster can be customized in a variety of ways, and clusters can be specialized for supporting different domains (more on this in Chapter 9). Through sharing of clusters, users can get rich cluster awareness that will help them in coordinating the work within a center of interaction.

Cluster Layer can be used by any application that needs to share data with other applications. A cluster may contain any data. In MultiCASE we used clusters to share software modules. A text editor may use clusters to share the contents of a document with other (possibly different) text editors. Cluster Layer requires some re-implementation before an application can use it. However, this re-implementation is minimum and can be limited to implementing a number of call-back functions.

Cluster Layer implements only a part of a center of interaction. A center of interaction has a view and a medium, while Cluster Layer implements only the view part. The medium of a center of interaction is implemented in Workspace Layer of IGLOO framework, which is the topic of the next chapter.

Chapter 8

Workspace Layer

8.1 Introduction

In the definition of a center of interaction in Chapter 4 we distinguished between a center's *view* and its *medium*. A cluster, as defined in Chapter 7, provides a center of interaction with a shared customizable view into the shared product space. However, beside access to the product, there are other elements in a center of interaction that are equally important for supporting the cooperation within the center. Workspace Layer of IGLOO framework is concerned with some of these other elements. In particular, Workspace Layer provides services for creating the medium through which the developers within a center of interaction can cooperate with each other. Workspace Layer combines the clusters provided by Cluster Layer with additional artifacts, awareness information, and interaction mechanisms, in order to create a richer picture of cooperation within a center of interaction. This is done in Workspace Layer by providing the *shared workspace* abstraction as the medium.

Figure 8.1 shows a scenario of using shared workspaces. In this figure, all the clients share a cluster that allows them to cooperatively modify a part of the underlying shared product space. For clients A, B and C there are also other contextual elements that are shared in addition to the cluster. The gray part at each of these clients is the additional shared context. It is the shared workspace. The shared workspace includes people, their tools, documents and manuals they use in their daily work, notes, social events, etc. The objects in a shared workspace are called *informal objects*, i.e. objects that are not part of the shared product but are used as resources for supporting cooperation in a center of interaction. The people in a shared workspace are the *inhabitants*, i.e. the developers that are involved in an activity in a center of interaction. By allowing the sharing of informal objects, these clients provide to their users a rather rich picture of shared interaction as it happens in the center of interaction. In comparison, client D is another client application that also has access to the same shared cluster, but is not sharing the shared workspace. D's picture of cooperation is limited to the information provided through the cluster it is sharing. As a consequence, D's user will see all the changes the others make to the shared cluster, but he or she will not see *how* these changes are done, e.g. what caused the changes, what other intermediate

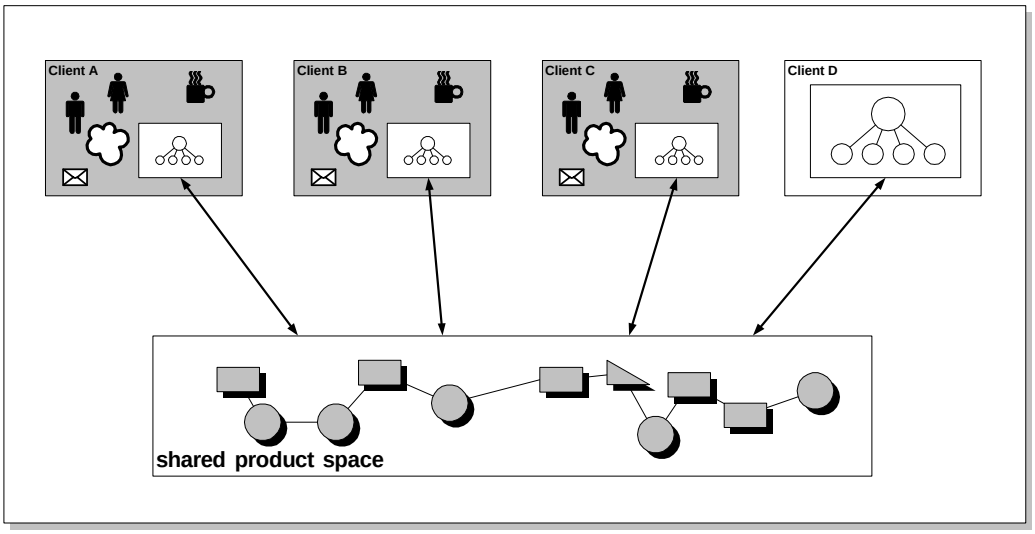


Figure 8.1: A shared workspace may contain much more than a cluster. The contents of the workspace are used as resources for cooperation in the center of interaction.

activities (such as informal brainstorming, discussions, social activities, etc.) were performed before the changes were done, etc. The goal of Workspace Layer is to allow the developers who are involved in a center of interaction to have access to a shared context that is not necessarily related to the shared product, but may contain other, completely informal elements.

Workspace Layer supports in this way two important aspects of product development. First, as we saw in Chapter 2, a product is developed through highly flexible interaction among developers in a number of centers of interaction. Cluster Layer, through shallow attributes and customization services, supports a flexible "interface" to the shared product space. Cluster Layer allows the users of a cluster to gradually refine the shared product. Workspace Layer extends this flexibility by allowing the developers make use of other artifacts than those that are a direct part of the shared product. It is not always clear what makes up a product. Product objects and conceptual relations are initially made as drafts and sketches, and discussed in small groups before they are "published" as part of the product. This informal aspect of refining the product is often neglected in existing development tools. For instance, the interfaces to conventional CASE tools are often highly formalized and allow data entry only through predefined formal semantics (Jarzabek and Huang 1998). Proper configuration of a shared workspace will allow the client applications to provide a higher degree of flexibility to the users. In this way, IGLOO can support virtual rooms used for a variety of product development activities.

Second, as we also saw in Chapter 2, increasing the visibility of work does not only imply making visible the shared product and the changes to it. There are many other aspects of cooperative product development that are invisible when product development teams are geographically distributed. In particular, working in a center of interaction requires access to fine-grained awareness information about the activities within the center. This is due to the highly interactive nature

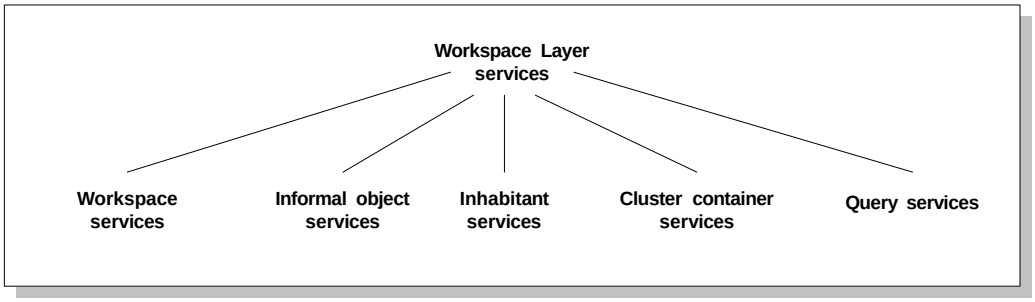


Figure 8.2: An overview of services provided by Workspace Layer.

of the focused cooperation that happens in a center of interaction, and the rich context information that is necessary for letting this cooperation happen. Workspace Layer can provide this awareness information in form of *shared workspace awareness* (Gutwin et al. 1996). Through services for sharing this context information, Workspace Layer can be used to provide different degrees of shared workspace awareness to developers involved in centers of interaction.

Workspace Layer is defined in terms of services for creating and managing shared workspaces and their contents. These services are divided into five groups. An overview of Workspace Layer services is shown in Figure 8.2. There are service groups for managing workspaces, informal objects within the workspaces, the inhabitants of the workspace (i.e. the developers that are within each workspace), and the clusters within each workspace. Cluster services are mainly a mapping of the services provided by Cluster Layer, and are used for inserting clusters in a shared workspace. There is also a group of services for querying Workspace Layer for information about workspaces and their contents.

The structure of this chapter is as follows. We will first explain the conceptual model of a shared workspace in Workspace Layer. We will then describe in details the services provided by Workspace Layer. The last part of this chapter outlines a detailed design for a shared workspace server called SWAL.

8.2 Shared Workspaces and Their Contents

Workspace Layer uses a conceptual model of shared workspace as shown in Figure 8.3. This model is based on the *room metaphor*, in that a shared workspace is seen as a room containing different artifacts and people. A room in Workspace Layer is persistent. This means that a room, once created, will continue to exist until a user explicitly deletes it. (Many groupware applications are *session-based*, and a session exists as long as there are people in the session.) A shared workspace can have different human *inhabitants*, i.e. developers that are currently in the workspace. In addition, a shared workspace may contain a number of *informal objects*. An informal object is any artifact that is not currently a part of the product. These can be early sketches of ideas, external documents and manuals, notes left for others, etc. Human inhabitants and informal objects are represented in form of attribute-value pairs. This means that any type of informal object can be shared by the users, and for each human inhabitant the users can decide which properties of the

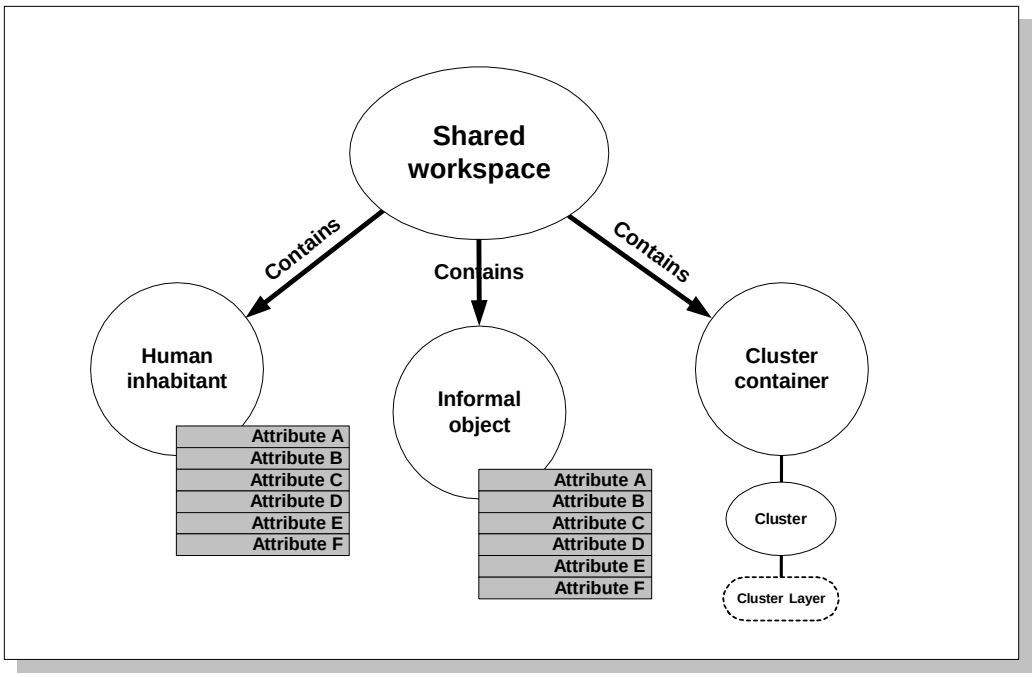


Figure 8.3: The conceptual model of a shared workspace and its contents, as defined by Workspace Layer.

inhabitant to share in a shared workspace. In addition, a shared workspace may have one or several *cluster containers*. A cluster container is used to insert clusters in a shared workspace, and in this way connects the workspace to a shared product space.

One advantage of using a room metaphor is that that it supports a seamless transition from single- to multi-user and from synchronous to asynchronous cooperation (Greenberg and Roseman 1998). The same room can be used by one or several people. When used by one person, a room is a private workspace, while when used by many it becomes a shared workspace. Moreover, two or more people can use a room to cooperate synchronously, i.e. by being in the same room at the same time, or use the room for asynchronous cooperation by leaving work material for others who may arrive later. Support for these two transitions is a powerful means for creating flexible shared workspaces.

There are two other aspects of the room metaphor that are important for our purposes. First, the room metaphor provides a suitable tool for realizing shared workspaces that make use of spatial relations among objects. Although spatial relations are not central in structuring a shared product space (see Chapter 4), having a common spatial frame is very useful for developers working in a center of interaction. This is because developers make extensive use of gestures and spatial relations among artifacts for conveying ideas and for communicating with each other. The model shown in Figure 8.3 can be easily made spatial by defining a spatial frame (i.e. a coordinate system) for the shared workspace, and enriching the contents (i.e. inhabitants, informal

objects and cluster containers) with spatial information (coordinates). A client application can then use these spatial measures to simulate a physical room. This can be used for instance to create shared workspaces that are simulation of shared physical desktops, similar to rooms in Teamwave[1](Roseman and Greenberg 1996).

Second, a room can be used as a resource for different kinds of activities. Physical rooms are familiar concepts to all of us. We are familiar with using rooms for different purposes, and we can in fact use the same room for completely different activities. As an example, a meeting room in a department can be used for having a variety of meetings, ranging from the most formal to the most informal. The same room can even host several activities at the same time. De Michelis et al. (2000) observed how the same room in a design studio was containing several focus areas, one for each design project. These areas, or what we have called centers of interaction, were distinguished mainly by being in different corners of the room, and by conventions among the designers about how to use each corner. So rooms can be used efficiently for supporting and organizing the developers' activities.

However, the same properties of a room that make it such a strong resource for cooperation can also limit its value when developing cooperation support technology. Physical rooms have rigid boundaries that isolate the inhabitants from the happenings outside the room. When the properties of physical rooms are copied (without changes) to room-based collaboration technologies, the same isolation problems continue to exist. In fact, these technologies may even amplify the problem. As stated by Kaplan et al.: "... room-based systems provide rigid rooms with strong boundaries. This makes it very difficult to work on 'several things at the same time' or keep one's eye on activities in one situation while concentrating on another" (1997, pp.546). In TeamWave, for instance, even though two rooms can be connected using doors, the participants in each room do not have any idea of what is happening in a neighboring room unless explicitly entering that room (Herlea 1998). This limitation of space is also noticed by De Michelis et al. (2000) in the case of the design studio, by observing that when the same designers moved into a new building and each got their own offices, the rich cooperation that was going on in the old design studio disappeared.

For product developers this shortcoming is serious because of two reasons. First, product development will typically involved a large number of developers. It is therefore not realistic to put everybody in the same room. One will need a number of rooms for allowing everybody to focus on their own tasks and get the work done. The problem of isolation can then occur as it did for a large group of developers using TeamWave for requirements gathering in Herlea and Greenberg's (1998) study. Second, even though the work of developers within a center of interaction is highly spatial, the overall structure of their work is not so spatial. As we saw in Chapter 2, organization of work in large development teams is likely to reflect the structure of the product being developed. Software is characterized by "conceptual distances" than by physical ones. So, even though rooms can be used as resources for supporting cooperation in a center of interaction, cooperation among centers of interaction has to depend on other factors than spatial relations.

Our approach for solving this problem in Workspace Layer has been to augment the shared workspace with cluster containers. A cluster container in a shared workspace can be seen as a

[1]TeamWave has inspired the work reported in this chapter. TeamWave support a host of other group-oriented functionality, such as advanced shared workspace awareness and navigation tools, that we have not yet included in the design of Workspace Layer (see Chapter 3 for an overview of TeamWave).

window into the shared product space. A cluster allows the inhabitants of the different workspaces not only to focus on different parts of the shared product space, but also to monitor the shared product space for other relevant product awareness (see Chapter 7 for details on cluster). In this way, a room can be regarded as a center of interaction with fluid boundaries, and the isolation problem mentioned above can be eliminated to some degree.

8.3 Services of Workspace Layer

In this section we look closer at the services provided by Workspace Layer. These services are defined to allow Workspace Layer's clients manipulate workspaces and their contents in an easy and flexible way. There are service groups for managing workspaces, informal objects, inhabitants, and cluster containers.

8.3.1 Shared workspace services

Shared workspace services are concerned with creating and maintaining entire shared workspaces. These services are shown in Table 8.1. The users can create new workspaces or delete the existing ones. Each workspace has a set of attribute-value pairs that can be used for sharing information about the workspace. These attributes are user-defined and can be set and removed using workspace services.

Table 8.1: The services defined by Workspace Layer for creating and maintaining shared workspaces.

Service	**Semantics**
Create a shared workspace	Create a new shared workspace. The client may provide a set of attribute-value pairs for initializing the workspace. A workspace identifier is returned to the client.
Delete an existing shared workspace	Delete an existing shared workspace. The client has to provide a valid workspace identifier. This service will also delete all the informal objects and all the inhabitants in the workspace, but will not delete the clusters in the cluster holders.
Update an existing shared workspace	Update one or several attribute-value pairs for an existing shared workspace. The client has to provide a valid workspace identifier and a set of attribute-value pairs. Workspace Layer will: a) if the attribute already exists, update its value with the provided value, b) if the attribute does not exist, create the attribute and set its value to the provided value.
Remove attributes from an existing shared workspace	Remove one or several attributes from an existing shared workspace. The client has to provide a valid workspace identifier and one or several valid attribute names.

This set of services supports a number of typical workspace functionality found in conventional groupware applications. For instance, an attribute of a workspace can be defined to be a progress indicator. The indicator can be a discrete value (e.g. just started, in progress, finished, deferred, etc.) or a continuous value (e.g. the percentage of finished work). It can be useful for both project managers and developers who would like to know the status of a specific task. An IGLOO client can combine this indicator with simple notifications in order to automatically inform specific workspaces about the status of work progress in other workspaces.

The services in Table 8.1, together with query services discussed later, can be used to implement a user-friendly "open door" session management policy similar to that in TeamWave and CBE (Greenberg and Roseman 1999). These tools show a list of all existing workspace. The users can browse this list and get an overview of what cooperation sessions are going on at each time. Defining an attribute for each workspace to indicate a "door status" can tell the developers if they can enter a workspace. For instance, a half-closed door could mean that one can glance into the workspace but cannot enter and participate in the session (Mackay 1999). The status of the door can also decide if the system should show a list of participants or not.

8.3.2 Informal object services

Informal objects implement objects that can be used to support informal cooperation within a shared workspace. An informal object is similar to a product object, with the difference that it is invisible to all but the inhabitants of the shared workspace containing the object. Services for managing informal objects are shown in Table 8.2.

Table 8.2: The services defined by Workspace Layer for creating and maintaining informal objects.

Service	Semantics
Create an informal object	Create a new informal object in a shared workspace. The client must provide a valid workspace identifier. The client may also provide a set of attribute-value pairs for initializing the object. The identifier of the object will be sent back to the client.
Delete an existing informal object	Delete an existing informal object from a shared workspace. The client has to provide a valid workspace identifier and a valid informal object identifier.
Copy an existing informal object	Make a new copy of an existing informal object. The client has to provide a valid informal object identifier. The client may also provide an optional workspace identifier where the new informal object will be placed. If no such identifier is provided the copy will be placed in the same workspace as the original. The identifier of the new object is returned to the client.

Continued on next page

Continued from previous page

Service	Semantics
Update attributes of an existing informal object	Update one or several attribute-value pairs for an informal object. The client has to provide a valid workspace identifier, a valid informal object identifier, and a set of one or several attribute-value pairs. For each provided attribute-value pair, Workspace Layer will: a) if the attribute already exists, update its value with the provided value, b) if the attribute does not exist, create the attribute for the informal object and set its value to the provided value.
Remove attributes from an existing object	Remove one or several attributes from an existing informal object. The client has to provide a valid workspace identifier and an informal object identifier, and a set of one or several attribute names. All the attributes will be removed from the given informal object.
Copy an informal object into a cluster	Make an informal object public by copying it into a cluster. The client has to provide valid identifiers for the workspace, informal object, and the cluster. The informal object and the cluster have to be in the same workspace. The service creates a new cluster object (using deep-create service of Cluster Layer) and copies the contents of the informal object into the new cluster object as shallow attributes. The identifier of the new cluster object is returned to the client.
Copy a cluster object into a new informal object	Copy the contents of a cluster object into a newly created informal object. The client has to provide identifiers for the workspace, the cluster, and the cluster object. The service will create a new informal object in the same workspace, and copy all deep and shallow attributes of the cluster object into the informal object. The identifier of the informal object is returned to the client.

Most of informal object services are similar to product and cluster object services (see Tables 6.1 and 7.2). There are additional services for allowing the transfer of an informal object into and out of a cluster. These two services can be used by a client to implement user-friendly mechanisms for manipulating a cluster in a shared workspace. For instance, a client may allow the user to drag and drop an informal object "into the product."

There are no explicit communication mechanisms in IGLOO's model of shared workspace. Explicit communication among the inhabitants of a shared workspace is an important part of the cooperative activity. Informal objects can be used to implement simple communication mechanisms. For instance,a chat tool can be implemented in form of an informal object. Instead of sending message to each other, the inhabitants can append messages to an informal object. For each message, the attribute name can be the time stamp of the message, and the value can be the name of the sender and the contents of the message. This can be useful because latecomers can have access to the history of the communication that has been going on in the room. In fact, more complex communication can be supported by allowing messages to have different types.

An example can be an informal object that implements an IBIS-like conversation, in this way implementing support for design rationale[2] (Moran and Carroll 1996).

8.3.3 Inhabitant services

The services provided by Workspace Layer for managing inhabitants are shown in Table 8.3. Each inhabitant is represented as a user-defined set of attribute-value pairs. An inhabitant is created and deleted from a shared workspace as users enter and leave the workspace. Note that each user can be in several shared workspaces at the same time. In this case an inhabitant will be created for the user in each shared workspace.

Table 8.3: The services defined by Workspace Layer for creating and maintaining inhabitants.

Service	Semantics
Enter a shared workspace	Allow a client enter a shared workspace. A new inhabitant object is created in the shared workspace. A valid workspace identifier has to be provided. The identifier of the inhabitant object is sent to the client. A client can inhabit several shared workspace at the same time.
Leave a shared workspace	Allow a client to leave a shared workspace. This service will delete the inhabitant object from the shared workspace. A valid workspace identifier and a valid inhabitant identifier have to be provided.
Update attributes for an existing inhabitant	Update one or several attribute-value pairs for an inhabitant object. The client has to provide valid identifiers for shared workspace and inhabitant object. For each provided attribute-value pair that is provided, Product Layer will: a) if the attribute already exists, update its value with the provided value, b) if the attribute does not exist, create the attribute and set its value to the provided value.
Remove attributes from an existing inhabitant	Remove one or several attributes from an existing inhabitant object. The client has to provide a valid inhabitant identifier, a valid workspace identifer, and a set of attribute names.

Inhabitants are used to represent users in shared workspaces. This is important in order to allow those already in the shared workspace to be aware of who else is there, but also to identify who does what changes to which objects in the shared workspace. An inhabitant can be represented in different forms by defining different combinations of attributes. The simplest way is to register each inhabitant with a global identifier. A more advanced representation may include biographical and contact information, a photo, location within the shared workspace, activity level, a telepointer, etc. In this way a richer picture of the users can be provided, contributing to increased awareness of what others are doing in the shared workspace.

[2] Audio communication within a shared workspace has been implemented in MultiCASE (Christensen and Karlsen 1999). See Chapter 5

Although arbitrary information about an inhabitant can be represented in a shared workspace, in practice this information will be limited because of two reasons. First, most of this information is highly dynamic and requires fast updating. This will require a high level of performance in any implementation of Workspace Layer. Workspace Layer must guarantee that the inhabitant is able to "catch up" with the real user's activities. For instance, in a slow network it might not be possible to provide a telepointer for each inhabitant because telepointer coordinates change rapidly. Second, advanced devices and sensors might be needed to register user activities that exceed simple keyboard and mouse input (Prinz 1999). For instance, activity level of an inhabitant is often not easy to measure because a user might be actively involved in a task without ever touching the input devices of his computer.

By conceptually separating inhabitants from informal objects we allow for a more specialized implementation of inhabitant objects in a future implementation of Product Layer. Such an implementation may choose to prioritize inhabitant objects during network communication because of the high pace of changes in these objects as opposed to informal objects.

8.3.4 Cluster services

A cluster is an important part of a shared workspace in IGLOO framework. Clusters are inserted into shared workspaces in order to provide access to the shared product space that underlies the workspaces(see Figure 8.1). The inclusion of clusters in shared workspaces distinguishes Workspace Layer from other room-based shared workspace applications because it removes the rigidity of the room boundaries.

A cluster is inserted into a shared workspace by first creating a cluster container. A cluster container gives access to the services of Cluster Layer for manipulating clusters (these services are discussed in Chapter 7). Once a cluster container is created in a shared workspace, the clients can request Cluster Layer services from the container. In this way Workspace Layer services do not need have to also be Cluster Layer services. Cluster container services are shown in Table 8.4.

Table 8.4: The services defined by Workspace Layer for creating and maintaining cluster containers.

Service	Semantics
Create a cluster container in a shared workspace	Create a new cluster container in a shared workspace. The client has to provide a valid workspace identifier. The client can also provide an optional cluster identifier. If a cluster identifier is provided, the container will be filled with the cluster. The identifier of the container is sent back to the client.
Delete an existing cluster container from a shared workspace	Delete an existing cluster container from a shared workspace. The client has to provide valid workspace and container identifiers. Note that this service does not delete the cluster that is in the container.
Set cluster	Set the cluster that is in the cluster container. The client has to provide valid cluster, cluster container and workspace identifiers. If the container already contains a cluster, it will change its cluster with the new one.

Continued on next page

Continued from previous page

Service	Semantics
Get cluster	Get the identifier of the cluster that is currently in a container. The client has to provide valid container and workspace identifiers.
Copy a cluster container	Copy a cluster container. The client has to provide a valid workspace and container identifier. The client can also provide an optional workspace identifier where the new copy should reside. Otherwise the container is copied into the same workspace. The new copy will contain the same cluster as the original container. The identifier of the new container is returned to the client.

A cluster container is in fact a small "applet" that is used by the inhabitants of a shared workspace to create and maintain a cluster. The advantage of this solution is that a cluster container can be specialized to provide tools for manipulating its cluster. For instance, in MultiCASE a cluster (a graphical diagram) can be modified using easy direct manipulation operation that are available in a context-sensitive menu upon clicking on its cluster container. In the same way, different containers can be specialized for providing different visualization techniques.

8.3.5 Query services

Workspace Layer contains all the information regarding shared workspaces and their contents. The services shown in Table 8.5 can be used to query Workspace Layer for such information.

Table 8.5: The services defined by Workspace Layer for creating and maintaining shared workspaces.

Service	Semantics
Get all shared workspaces with a specific attribute-value	Return workspace identifiers for all shared workspaces that have a specific attribute with a specific value. The client has to provide an attribute name and a value.
Get all shared workspaces with a specific attribute	Return workspace identifiers for all shared workspaces that have a specific attribute. The client has to provide an attribute name.
Get all existing shared workspaces	Return workspace identifiers for all existing shared workspaces in an installation of Workspace Layer.
Get attribute values for an informal object	Return the current values of one or several attributes for a given informal object. The client has to provide valid workspace and informal object identifiers, and a set of one or several attribute names.
Get all attribute-value pairs for an informal object	Return the set of all attribute-value pairs for a given informal object. The client has to provide valid workspace and informal object identifiers.

Continued on next page

Continued from previous page

Service	Semantics
Search a shared workspace for informal objects with a specific attribute-value	Return informal object identifiers of all the existing informal objects in a shared workspace with a specific value for a specific attribute. The client has to provide the identifier of the shared workspace, and the attribute-value pair.
Search a shared workspace for informal objects with a specific attribute	Return informal object identifiers of all the existing informal objects in a shared workspace that have a specific attribute. The client has to provide a valid workspace identifier and an attribute name.
Get all existing informal objects in a shared workspace	Return a list of the identifiers of all the existing informal objects in a shared workspace. The client has to provide a valid workspace identifier.
Get attribute values for an inhabitant object	Return the current values of one or several attributes for a given inhabitant object. The client has to provide a valid workspace and inhabitant identifiers, and a set of attribute names.
Get all attribute-value pairs for an inhabitant object	Return the set of all attribute-value pairs for a given inhabitant object. The client has to provide valid workspace and inhabitant identifiers.
Search a shared workspace for inhabitant objects with a specific attribute-value	Return object identifiers of all the existing inhabitant objects in a shared workspace with a specific value for a specific attribute. The client has to provide a valid workspace identifier, and an attribute-value pair.
Search a shared workspace for inhabitants with a specific attribute	Return object identifiers of all the existing inhabitant objects in a shared workspace that have a specific attribute. The client has to provide a workspace identifier and an attribute name.
Get all existing inhabitants in a shared workspace	Return a list of the identifiers of all the existing inhabitant objects in a shared workspace. The client has to provide a valid workspace identifier.

8.4 The Implementation of Workspace Layer: SWAL

Workspace Layer is implemented in form of a shared workspace server called SWAL[3]. SWAL is a generic shared workspace server that can be used by its own. In a stand-alone mode, SWAL can be used by a group of clients to create shared workspaces based on a room metaphor, and to populate these workspaces with inhabitants and informal objects. Used in combination with a CoClust or Gossip server, SWAL also provides its clients with functionality for creating cluster containers and accessing a shared product space. This is shown in Figure 8.4. The gray parts in the figure are the parts that are implemented by SWAL, while the white parts are implemented by CoClust. This architecture is useful because it makes SWAL's clients highly independent of the

[3] A first version of SWAL is implemented by Rømma and Skjønhaug (2000) in their diploma work. This author has been supervising their work.

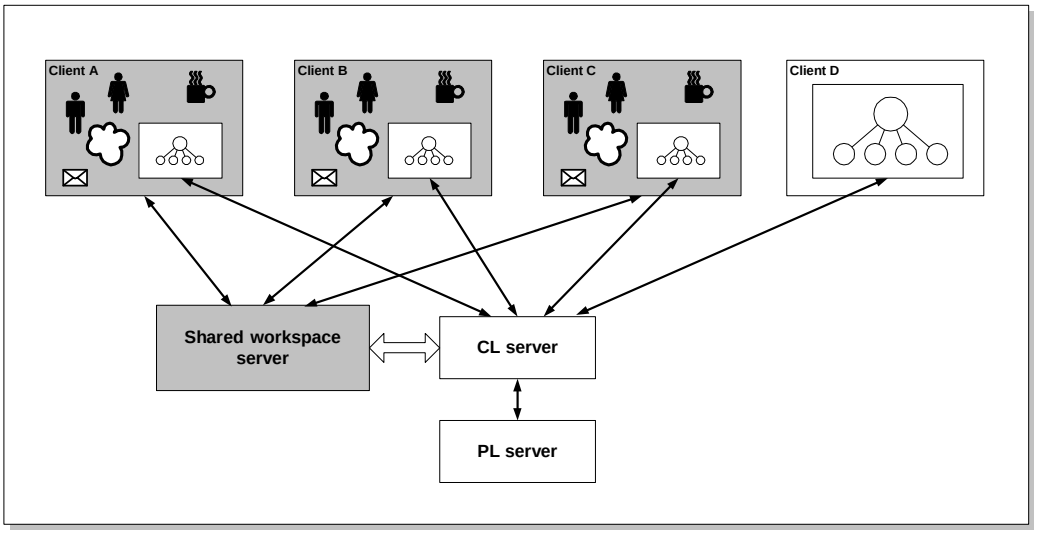


Figure 8.4: SWAL can be used as a generic shared workspace server or in combination with CoClust.

rest of IGLOO framework. A SWAL client does not need to know about the rest of IGLOO client if it does not need to work with clusters.

This section describes the design of and implementation of SWAL. Technically, SWAL is quite similar to CoClust. While CoClust's main responsibility is to keep multiple replicas of clusters synchronized as users change them, SWAL is in charge of providing access to synchronized shared workspaces and their contents. In fact, SWAL is much simpler than CoClust because it does not need to synchronize the shared workspaces with an underlying data structure.

8.4.1 An overall view of SWAL

Figure 8.5 shows the overall architecture of SWAL. Similar to CoClust, SWAL consists of a *server* and a *client extension*. SWAL server contains the master copies of all the existing workspaces and their contents in form of informal objects and cluster containers. SWAL server also implements the services of Workspace Layer. SWAL client extension is an implementation of a set of classes that give easy access to the services of Workspace Layer without worrying about network communication and technical details of SWAL server.

An important part of SWAL server is the *Workspace Manager* that is the access point for all the clients. Workspace Manager allows the clients to create and modify workspaces, to browse through existing workspaces based on different criteria (for instance door status, work progress status, inhabitants, etc.), to glance into workspaces, etc. Workspace Manager also allows users to easily enter into different workspaces.

There are two types of shared workspaces in SWAL. If a workspace has at least one inhabitant it is an *active* workspace and resides in the *active workspace space*. Otherwise it is an *inactive* workspace and is stored in a *workspace database*. Each active workspace has a *master replica* on

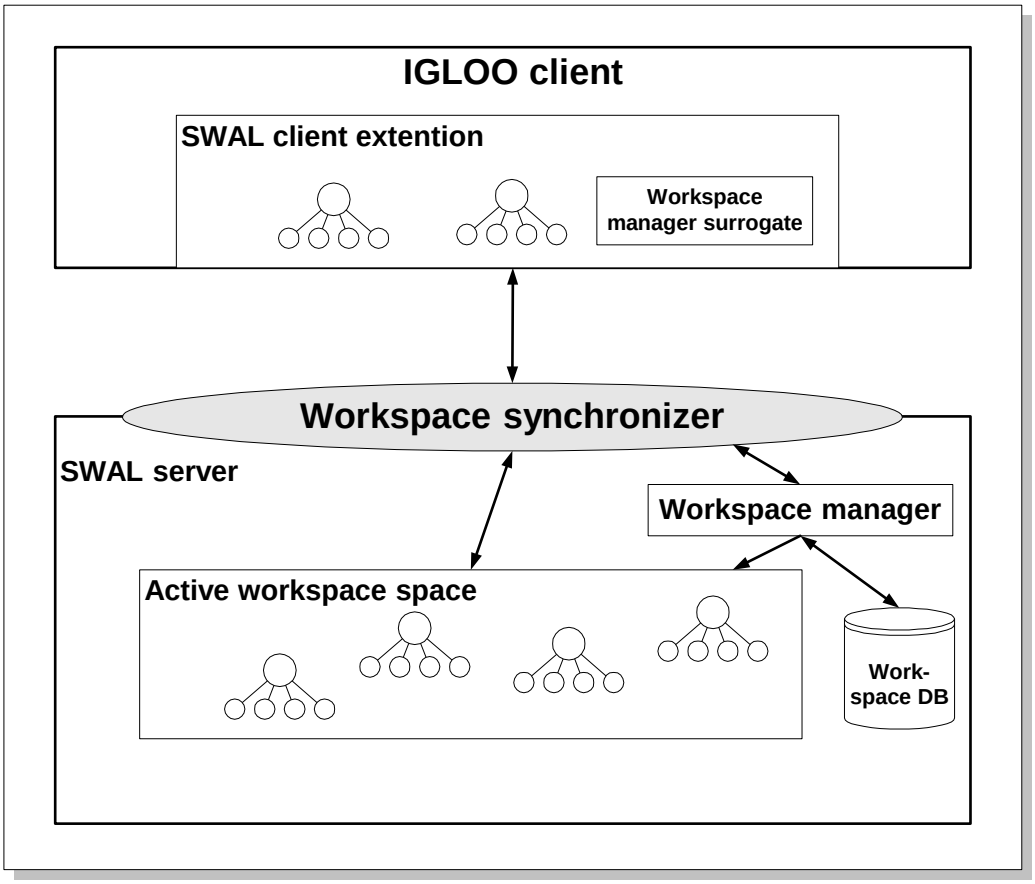


Figure 8.5: The overall architecture of the SWS server.

the server and one or several *local* replicas residing at the clients whose users are the inhabitants of the workspace. A *workspace synchronizer* exists for each active workspace, and is used to synchronize all the replicas, including the master copy of the workspace. The synchronization mechanisms are similar to those of CoClust (see Section 7.4)

It is important to note that clusters are not stored on SWAL server. Instead, cluster containers are stored together with the identifier of the cluster they contain. When the first user enters a workspace that contains a cluster container, the container is in charge of contacting CoClust and setting up proper connections for initializing and synchronizing the cluster. In this way the responsibility of synchronizing clusters is fully delegated to CoClust, while creating and deleting cluster containers is the responsibility of SWAL.

8.4.2 SWAL client extension

In order to allow user-friendly access to the SWAL server, a set of classes are implemented that can be used by the clients. These classes are implemented in Java and Visual Basic, and enable clients of a widely different range to be implemented for SWAL. SWAL client extension includes a *Workspace Manager* class, a *Workspace* class, an *Informal Object* class, an *Inhabitant* class, and a *Cluster Container* class. The methods that are defined for these classes provide access to all the services implemented by SWAL. A client creates a local replica of a workspace by creating a new instance of a Workspace class. Once the Workspace class is instantiated, the client can use the methods of the Workspace class to modify the new workspace. The workspace class is responsible for synchronizing its state data with that if its master replica on the server and with its other replicas. The mechanisms for doing this are similar to those of CoClust client extension described in Section 7.4.2. More details on SWAL can be found in (Rømma and Skjønhaug 2000).

8.5 Summary

In this chapter we have described Workspace Layer of IGLOO framework. Workspace Layer is the part of the framework that is mostly concerned with cooperation in small groups. It is defined in a way to allow informal cooperation among a few developers within a shared workspace. Informal objects are used to share information within a workspace, while cluster containers provide access to an underlying shared product space. An implementation of Workspace Layer called SWAL is described. SWAL is implemented in Java programming language, and includes client extensions for both Java and Visual Basic programming languages.

Workspace Layer is the last layer of IGLOO framework. The three layers of the framework are defined in a way that they can function by their own, but are more powerful when combined. The framework can be used to set up different types of IGLOO networks. This is explained in the next chapter.

Chapter 9

Deploying IGLOO Framework

9.1 Introduction

In the last three chapters we have developed IGLOO framework as a generic platform for supporting shared interaction, and we have described in details the different components of the framework. In this chapter we will discuss the issue of deploying these components for building a specific environment for cooperation, i.e. an *IGLOO network*. The concept of an IGLOO network was introduced in Chapter 5 (see Section 5.4 on page 120) as a group of *IGLOO clients* that use IGLOO framework to communicate with each other, and support cooperation among their users. IGLOO framework is developed as a generic platform. An IGLOO network uses an *instance* of IGLOO framework to support a specific project group with specific needs. The process of *instantiating* IGLOO framework, and building an IGLOO network based on the resulting instance, is called the *deployment process* and is the topic of this chapter.

The structure of this chapter is as follows. First, in this introduction we will revisit some of the principles underlying IGLOO framework. The deployment process takes these principles as its starting point, and specializes them into context-specific principles to underlie a particular IGLOO network. The contents of an instance of IGLOO framework are described in Section 9.2. In Section 9.3 we will give an overview of the deployment process, and we will describe the necessary activities within this process. Section 9.4 outlines some trade-offs in deploying the framework, and describes how a deployment process can be done incrementally. In Section 9.5 we will review some of the architectural features of IGLOO framework, and will describe how these features can influence the deployment process by allowing the creation of an architecturally high-quality and efficient IGLOO network.

Product development involves large groups of people and a high amount of cooperation. As we have seen in Chapter 2, cooperation among developers is highly unpredictable and situated. Due to the vague nature of the product it is difficult to predict how a group of developers will organize their daily product development activities, what kind of information they will need, who they will talk to, etc. Developers' needs in form of information they access and people they talk to continuously change in the course of the same project and from one project to the other. In

addition, different groups working with different parts of the product or with different types of tasks often have radically different needs. The value of the product as a resource for cooperation was pointed out in Chapter 2 because it provides a flexible means for externalizing knowledge, negotiating meaning, and coordinating action. In Chapter 4 we developed the product-based shared interaction model. This model emphasizes the importance of product as a resource for cooperation. The model is based on a few principles that we have developed based on empirical data from our analysis of ALPHA in Chapter 2. These principles are:

- A large part of developers' everyday activities involves different types of product objects. Product objects are not only information containers, but are also used for supporting cooperation. Product objects embody the shared knowledge about the product. Product objects are also used for coordinating the work in that they are used as resource for supporting interactions among developers and for preventing breakdowns in coordination. The does not presume any categorization of the possible range of product objects, and does not treat differently the different product objects that are within a shared product space.

- A product often consists of a large number of product objects, and developers at any given time only use a subset of these objects. This subset changes dynamically during the project. Developers might like to have a customized view of the part of the product they work with. The model does not assume any predefined patterns of usage or courses of interaction when using product objects or groups of product objects.

- Developers might work in groups of different size, or might work individually. An important part of the activities of individual or groups of developers will be to modify the product objects. The model does not assume any specific configuration of groups, or any predefined set of refinement or modification operations.

- Work within small groups is different from work within the large project team in that work within small groups is more focused, more dynamic, more interactive, and makes use of richer interaction media and more fine-grained context information. The model does not try to predict the process of interaction within small groups, and provide only the basis for an interaction medium (as part of a center of interaction).

- Product objects are related to each other, and as a result people working with them dependent on each other in their work. The model does not assume what kind of relations and dependencies these are, or which specific product objects will be dependent on which other product objects.

- Developers do not wish to make public all the details of their work. Making public any information about the work has to have some benefit for the publishers, for instance to help them articulate the work they are doing. The model does not assume which details will be available within the shared product space, but allows for different degrees of detail.

IGLOO framework was developed in Chapters 5–8 in order to implement this shared interaction model. IGLOO framework is a formalization of these principles in form of a *generic implementable framework*. Gossip, CoClust and SWAL are *generic implementations* of IGLOO framework. They are generic because they use the above general principles without specializing them in any way. It is assumed that different project groups will share the need for the services

provided by IGLOO framework and its generic implementations, but will have additional needs that also have to be supported. IGLOO framework therefore allows each project group to create an *instance* of the framework that provides additional active support for that specific project group. Each instance of IGLOO framework is still based on the initial principles above, but makes new assumptions about how its particular users will work. An instance of IGLOO framework can then be used to set up an IGLOO network that is better suited to the needs of the specific project group. The process of creating such an instance and using it to build an IGLOO network is the deployment process.

One obvious advantage of this approach is *reuse* of code. The generic implementations and their API (Application Programming Interface) can be reused and improved further. Another more crucial advantage is that of *interoperability*. New instances can co-exist with old ones, and new tools can communicate with old tools. In this way, building an IGLOO network becomes an iterative process of *incremental integration*. This is in particular important because of the diverse and changing needs of project groups. An IGLOO network, once created, is not a static construct and can be tailored to the ever-changing needs of its users.

9.2 The Instance

The main goal of the IGLOO deployment process is to create an *instance* of the framework that is suited to the needs of a specific project. An instance is a set of definitions of what objects and relations will be shared in a project. An instance also defines the functionality of the clients that have to be implemented or recoded. The different parts of an instance are shown in Figure 9.1. An instance consists of a number of specialized *vocabularies*, specialized *awareness policies*, and specialized *clients*. The term vocabulary normally means an agreed-upon definition of the meaning of information that is shared by a group (Chen 1994). Here we use the term in a more restricted form to denote a definition of the different object and relation types used by IGLOO. The specialized vocabularies define the type of information (in term of objects and relations) that will *populate* the shared product space, the clusters and the workspaces in an IGLOO network. The specialized clients are used to *manipulate* the resulting shared product space, clusters and workspaces. The awareness policies govern awareness production and distribution in the resulting IGLOO network.

There are three types of vocabularies in an instance. An *organizational vocabulary* is a definition of product-related information that is shared globally among a project's members (in terms of product objects and conceptual relations), while *local vocabularies* are definitions of product-related information that is shared by small groups that are part of a project (in terms of cluster objects and cluster relations). *Workspace vocabularies* define the kind of information that will be available within the different workspaces (in terms of workspaces, informal objects, and inhabitants). An instance can have only one organizational vocabulary, but may have several local and workspace vocabularies.

Another part of an instance is specialized *awareness policies*. These policies govern the production and distribution of awareness information in an IGLOO network. Awareness policies consist of:

- *A set of rules for how awareness relations will be created*: One such rule can for instance be that all the conceptual relations of a specific type (e.g. "dependency") will also be awareness

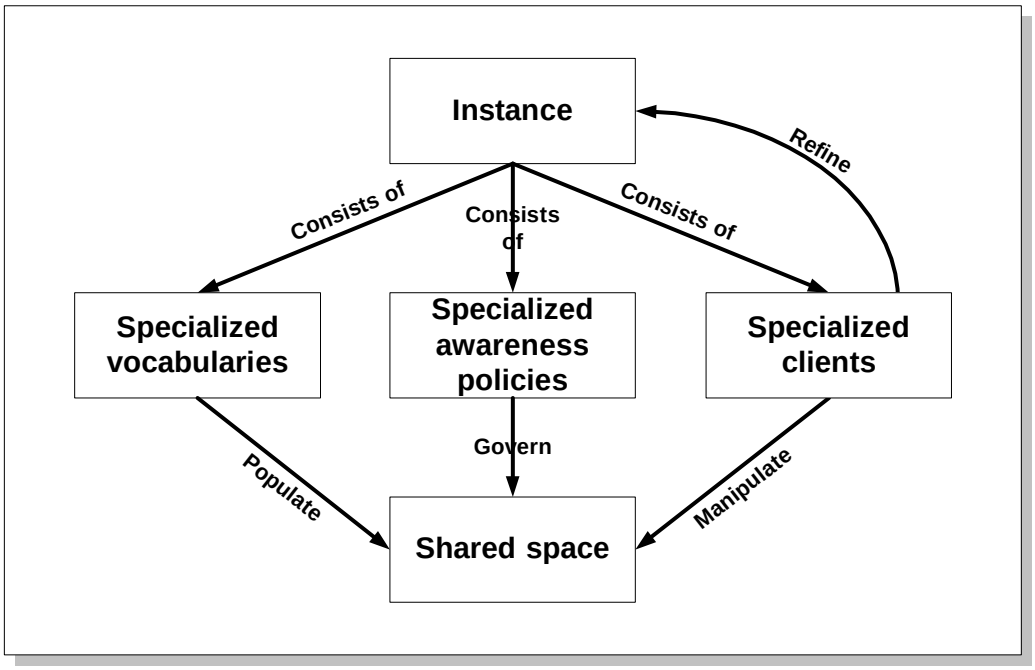


Figure 9.1: An instance of IGLOO framework.

relations. These rules will normally be programmed into the tools of the developers. E.g. in a graphical editor creating a "dependency" relation might automatically create an awareness relations.

- *A set of global awareness schemes and rules for how these schemes will be maintained*: An awareness policy might decide what global awareness schemes will exist, who will be responsible for creating and maintaining them, and who will have to subscribe to them. E.g. it might be decided that the project manager will maintain a global awareness scheme, and that all the developers have to subscribe to this scheme.

- *A decision about what user operations should generate product awareness events*: Product Layer may be configured to produce awareness events for only a subset of user operations (see Chapter 6 for awareness configuration services). An awareness policy may decide what these operations are. E.g. it might be decided that events will only be generated for modifications to the shared product space. In this case developers will not be notified if someone reads a product object.

The third part of an instance is the specialized *clients*. Specialized clients are either already-existing applications that are integrated into IGLOO, or are newly developed IGLOO clients. They are used for manipulating the shared information in an IGLOO network. Specialized clients may be developed to manipulate product objects and conceptual relations, cluster objects and cluster

relations, and the contents of the shared workspaces. They may in addition provide communication tools (such as video and audio communication tools), workflow functionality, advanced editing and other types of services. IGLOO framework already implements much of product-based shared interaction support that will be needed by the specialized clients. This means that developing specialized clients does not need to be a complex and time consuming process. Specialized clients may in many cases be the same single-user applications that the developers have been using before the deployment of IGLOO framework. For instance, the cluster abstraction described in Chapter 7 already implements real-time sharing and collaboration functionality. A single-user graphical editor can be modified to make use of such clusters for sharing its graphical diagrams. In this way, the editor can be made "multi-user" through minimum effort.

9.3 Activities in the Deployment Process

There is a set of activities that have to be performed during a deployment process. These are activities that are necessary for creating the three parts of an instance. Performing these activities systematically assure that the necessary information is available, the necessary decisions are taken, and the necessary results are produced. The activities may also be used as guidelines for estimating the cost of deploying IGLOO framework for a specific setting. The activities are the following:

- *Define the specialized vocabularies*: In order to be able to use a large shared space for cooperation, there must be an agreement about the meaning of its contents. For instance, the clients should know what product objects or conceptual relations might exist in the shared product space, and what attributes they should have. In addition, local and workspace vocabularies might be developed for supporting cooperation in different specialist groups.

- *Define the specialized awareness policies*: It might be desirable to predefine how awareness relations should be created among the product objects in the shared product space, what awareness schemes will exist, what types of product awareness events should be generated, etc. One might also decide to allow awareness configurations to be created in an ad hoc manner, independently from the structure of the shared product space.

- *Develop specialized clients and workspaces*: In this step one might identify and integrate already-existing tools into an IGLOO network, or develop specialized clients for manipulating the shared space. One might also develop specialized shared workspaces for supporting specific group processes such as voting, brainstorming, etc. or specific tasks such as co-authoring, graphical modeling, etc.

It is assumed that these activities are performed in conjunction with a parallel activity of identifying the cooperation needs of the specific project team. One should know about the type of cooperative activities that exist, the cooperation needs of the developers, the kinds of development tools and methods that are already in use and that may be used in the future, the maximum cost allowed for the deployment process, etc.

It is important to note that the ordering of these activities is not predefined. One may also start with one activity and incrementally perform the others later. In addition, one will normally need a number of iterations through these activities in a typical deployment process. The rest of this section describes these three activities in details.

Defining the different vocabularies– IGLOO framework provides a flexible set of services that can be used to share any type of objects and relations with any combination of attributes. This provides the developers with a high degree of flexibility but at the same time requires that the developers define the type of objects and relations they will need to share. There are three groups of objects and relations whose format and purpose have to be defined during this step:

- *Organizational vocabulary*: An instance of the framework has to define the different types of product objects and conceptual relations, their purpose, what attributes they will have, and how they will be distinguished from each other. This is the organizational vocabulary for the specific instance. An organizational vocabulary may for instance be based on a specific modeling language (such as UML), or it may be more informal and developed in an ad hoc manner[1].

- *Local vocabularies*: An instance of the framework has to define the different cluster, cluster object, and cluster relation types that will be used by the different groups in the project. These definitions include the different types, their purpose, their attributes, which attributes are deep (and consequently based on the defined organizational vocabulary) and which are shallow (i.e. are local to the clusters). These definitions will constitute the local vocabularies for the instance. Note that local vocabularies are strictly based on the organizational vocabulary. In particular each cluster object type has to correspond to a specific product object type and each cluster relation type has to correspond to a conceptual relation type. There might be several local vocabularies in each instance, e.g. one for each specialized group.

- *Workspace vocabularies*: An instance of the framework has to define the different workspace, informal object, and inhabitant types to be used by the project. An instance will have to define what kind of workspaces will exist in the IGLOO network and what attributes they will have, what kinds of informal objects can be created in the workspaces and what attributes they will have, and what information about the inhabitants should be shared in the workspaces. This is the workspace vocabulary of the instance. A workspace vocabulary has to be defined for each type of workspace that will exist in the IGLOO network.

In addition to the vocabularies, it might be necessary to define a set of user-defined operations that need to be simulated by Product Layer. As we have seen in Chapter 6, Product Layer allows the clients to define a set of operations on their product objects and conceptual relations. Although Product Layer does not understand the meaning of these operations, it will forward awareness events about them as reported by the clients (i.e. Product Layer will simulate these user-defined operations). The users can decide what operations will be needed, and what parameters the resulting awareness events should contain. Although these definitions are not part of the vocabularies, it is a good ideas to define them in conjunction with the vocabularies. E.g. user-defined operations can be defined for each object and relation defined in the vocabularies.

Not all of the vocabularies need to be defined if one plans to use only a subset of the services in IGLOO framework. For instance, if only Product Layer is used, local and workspace vocabularies are not needed. It is also important to note that the definitions of product objects, conceptual

[1]Note that each of the developers or groups of developers can still add their own product object and conceptual relations to the shared product space, or augment the existing ones with additional attributes. However, only the definitions in the organizational vocabulary will be recognized by all project members.

relations, cluster objects, and cluster relations need not reflect all the details of these objects and relations. The shared product space should be seen as a place to share information that is necessary for cooperation, not as a repository. A lot of the information locally available to each developer, or stored in a local repository, will consist of details that may not be necessary for cooperation. As an example, a programming group may not need to share all the contents of their source code files. It might be enough to share information such as file name, owner, last changed, version number, and interface definitions.

Also note that these definitions do not require any coding activities. The result of this activity is a set of definitions as described above. These definitions are then used to develop (i.e. code or re-code) the specialized clients.

Defining awareness policies– IGLOO framework is responsible for generating awareness information about access to the contents of the shared product space, the clusters and their contents, and the workspaces and their contents. For each IGLOO network specific awareness policies have to be defined for how awareness information will be produced and distributed to the developers in the network. A part of the deployment process is therefore to define these policies. Awareness information production can be configured for the shared product space only (see Chapter 6[2]). The policies are therefore based on the awareness configuration services of Product Layer:

- *Rules for creation of awareness relations*: Awareness relations overlap conceptual relations in the shared product space. However, awareness relations are not automatically created for each conceptual relation. It must be decided in advance whether creating a certain type of conceptual relation should also create an awareness relation between the source and destination product objects. This can be necessary in cases where the structure of the product enforces strong dependencies among the developers working on the different parts of the product (e.g. import relations among source code files). It can also be decided that awareness relations are going to be created in an ad hoc manner. In this case, the developers should decide who will be responsible for creating the awareness relations. Mandatory awareness relations might be created automatically by the specialized clients.

- *Rules for creating and subscribing to awareness schemes*: Awareness schemes define what mediated awareness each developer will receive. The developers might want to define what standard schemes will exist, what global schemes will exist, who will be responsible for creating them, and who will be expected to subscribe to them. If there are to be global awareness schemes (e.g. one for all project members, one for all the programmers, etc.) the name of these schemes have to be decided. It can also be decided to allow the developers to create private or group schemes.

- *Rules for what product awareness events will be generated*: Awareness configuration services of Product Layer allow for different amounts of direct product awareness events. An awareness policy might decide if Product Layer should produce direct events for all kinds of access to the shared product space, or only for updates. This decision can affect the amount of produced awareness information, and should also be considered with respect to privacy.

[2]Future versions of IGLOO framework may also allow the users to customize the awareness information that is produced by Cluster Layer and Workspace Layer.

Again the definitions here do not require any coding activities. The different choices that are made are all supported by the generic implementations of IGLOO framework. These decisions however will have to be implemented by the specialized clients that are developed for the specific IGLOO network.

Developing specialized clients and workspaces– This step is concerned with developing (i.e. coding or recoding) the specialized clients that will be used by the developers for cooperation and interaction with the shared product. These clients can be seen as tools that are used to manipulate the contents of the shared space (see Figure 9.1). Their behavior is partly predefined by the vocabularies and awareness policies of the particular instance. This is a coding activity. Clients can be developed using one of two approaches: creating new clients or integrating existing applications.

New clients should be capable of communicating with an IGLOO network through the service interfaces provided by the generic implementations of IGLOO framework. Clients may use the services directly, and communicate with the IGLOO network through the network protocols provided by Gossip, CoClust and SWAL. Clients can also be developed by specializing the client extensions provided by the generic implementations. These client extensions can be specialized in an object-oriented manner. For instance, the Cluster class provided by CoClust can be specialized to become an editor for specific kinds of clusters. The specialized cluster can provide visualization and user interaction mechanisms, and can automate some of the manipulation activities (e.g. automatically creating awareness relations when a specific type of conceptual relation is created by the user). The advantage of using the client extensions for creating specialized clients is that the need to program complex network communication algorithms is eliminated, and one can instead focus on implementing the functionality of the client.

The second approach to creating new clients, i.e. integrating existing applications, might be necessary in cases where applications such as graphical editors and CASE tools are already in widespread use among the developers. In most cases integration will happen through recoding the existing application, and will therefore requires access to the application's source code. Integration might happen through different layers of the framework. Generally, integrating through Product Layer will require little recoding (and in some cases no recoding at all), while integration through Cluster Layer and Workspace Layer will require a considerable amount of recoding. Figure 9.2 shows an example of integrating a single-user CASE tool into an IGLOO network. In this example the integration is through Product Layer, i.e. through Gossip network protocol. The CASE tool has to be modified in such a way that it can access the shared product space in Gossip, and receive and use awareness information produced by Gossip. Integration in this case can be done in three ways. These are shown in Figure 9.2 with corresponding numbers:

1. The editor of the CASE tool is changed into an IGLOO client. The editor can be recoded so that all user interaction is analyzed for possible accesses to the shared product space. If a user interaction requires access to the shared product space (e.g. if the user creates a product object that is to be shared), the editor will not only do its normal duties (e.g. update the local repository) but will in addition issue a service request to Gossip. Also the awareness information arriving from Gossip (as a result of other users' access to the shared product space) has to be translated into local simulations of other users' actions. (Note that these local simulations may require changes to the contents of the local repository.)

2. An *IGLOO agent* (a special type of IGLOO client) is used. The agent is located between the

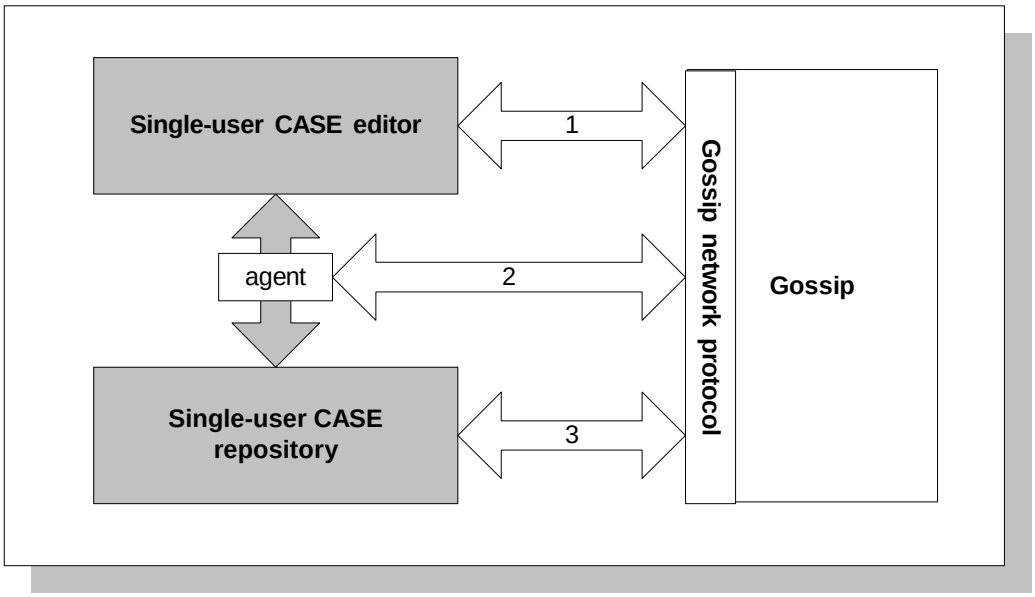


Figure 9.2: An example of integrating single-user tools into an IGLOO network.

editor and the repository. This agent works as a repository seen by the editor, and as an editor seen by the repository. All communication between the editor and the repository passes through the agent. The agent issues service requests as the user of the CASE tool accesses the shared product space. In addition, awareness information from Gossip is translated into local actions on both the repository and the editor. This solution might be feasible without any recoding if the communication between the editor and its local repository is through a well-defined network protocol.

3. The repository of the CASE tool is changed into an IGLOO client. In this case the repository is changed so that it can analyze all input from the editor and issue services requests to Gossip if the editor accesses the shared product space. In addition, the repository will receive awareness events from Gossip and change its own state and the state of the editor accordingly. The editor will not need any modification if the communication between the editor and the local repository is through a well-defined network protocol.

So far we have discussed IGLOO clients as tools for modifying shared information (i.e. different objects and relations) in an IGLOO network. Specialized clients may also be developed in order to allow the developers to refine the instance that is used in the IGLOO network. This is shown in Figure 9.1 in form of a “Refine” link from specialized clients to the instance. This “reflective” way of operation can be interesting because an IGLOO network can now be turned into a “specialization environment.” Developers can now build their own instance of IGLOO framework while they use the IGLOO network. An IGLOO network can be defined and refined in an incremental way, which leads us to the discussion of incremental deployment.

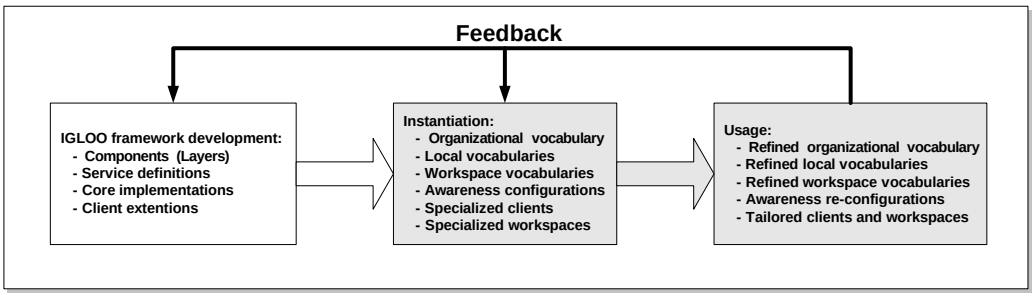


Figure 9.3: An example of IGLOO framework deployment. Rectangles are typical phases with their typical end-products. Gray rectangles denote phases that are a direct part of the deployment process.

9.4 Incremental Deployment

Activities in a deployment process can be performed in different phases of a product development project. Figure 9.3 shows a possible organization of the deployment process and its relation to the development of IGLOO framework. This example illustrates a common way of deploying a framework: developing the framework, specializing it to a domain, and using the specialization (Codenie, De Hondt, Steyaert and Vercammen 1997). In this example, the activities in the deployment process happen in different phases (shown as rectangles in the figure). Each phase produces a set of results. Typical results of each phase are shown inside the phase's rectangle. Thick arrows denote flow of results from one phase to the other, while thin arrows denote evaluation and feedback information. The *instantiation* and *usage* phases are directly concerned with the deployment process, while the *IGLOO framework development* phase is the process of developing the core parts of IGLOO framework and is common to all the deployment processes.

The phase IGLOO framework development is described in Chapters 5–8 of this thesis. The outcome of this phase includes the definitions and the generic implementations of the three layers of the framework, plus the client extensions that are provided as a means of facilitating the integration of external applications. The effort here is open-ended (as illustrated in Figure 9.3 by feedback arrows from the other phases), meaning that the framework itself will be continuously refined as more empirical data are collected from different deployment processes. (In some cases developing the core framework might be seen as an integral part of the deployment process, e.g. in open source development.)

The instantiation phase results in an initial instance of IGLOO framework. This instance is used as the basis for building the IGLOO network that is used in the usage phase. The instance itself might change during the usage phase. Refining the instance will change the properties of the IGLOO network in order to adapt it to the emerging needs of the developers.

The deployment process can be done *incrementally*. This can be necessary in order to adapt an IGLOO network to its environment. Incremental deployment can be done in one of two ways: Incremental deployment of framework layers, and/or incremental definition of the instance. These two approaches are called *vertically incremental deployment* and *horizontally incremental deployment* respectively.

Not all the layers in IGLOO framework need to be deployed in an IGLOO network. The higher layers of the framework provide more advanced support for focused cooperation in a center of interaction, but also require a more costly deployment process. One can build an IGLOO network with only Product Layer. Deploying only Product Layer will allow a project to create a shared product space, will provide a flexible interface for modifying this space, and will supply the developers with awareness information about changes to the product. With only Product Layer the costs related to the deployment process can be minimized since some of the deployment activities can be delayed or eliminated totally. For instance, local and workspace vocabularies are not required when using only Product Layer. Also developing specialized clients can become a less costly process because it is easy to integrate existing tools into an IGLOO network through Product Layer service interface. Such a minimal IGLOO network involves less risk in terms of cost and radical changes to the existing development environment. Once the minimal network is in place, is used for some time, and an organizational vocabulary is established and supported by a host of tools, the network can be "upgraded." Adding Cluster Layer and Workspace Layer to a minimal network does not require abandoning the existing network. The existing tools and vocabularies can still be used, and more importantly, cooperate with the new ones. IGLOO clients that are integrated through a higher layer of the framework can communicate with those integrated through a lower layer. For instance, clients that use Cluster Layer services can communicate with those using Product Layer services because both of them share the same organizational vocabulary and awareness policies.

The second approach, i.e. horizontally incremental deployment, is based on relaxing the distinction between instantiation and usage phases. Activities in the deployment process can be performed either in the instantiation phase or in the usage phase. In a deployment process the decision of which activity to perform in which phase will have great effect on the flexibility and adaptability[3] of the resulting IGLOO network. The less that is fixed in the instantiation phase, the more the users of the instance will be able to adapt the resulting IGLOO network during the usage phase. However, there are some trade-offs to consider. Some of these trade-offs are shown in Table 9.1 on page 206.

In one extreme, all the components of the instance are defined and developed in the instantiation phase. This means that during the instantiation phase all the vocabularies will be defined, and all the policies and rules for awareness information production and distribution will be decided. In addition, a set of specialized clients based on these definitions will be developed. Though this is possible and in some cases advantageous (e.g. in the case of very large project groups using formal methods and tools), it will result in minimum flexibility in the usage phase. The developers will be forced to use specific types of product objects and conceptual relations to express their ideas, and the awareness information that they will receive may result in information overload because it is not tailored to their real needs.

The other extreme will leave all or most of the activities to the usage phase. Developers will have to create all the vocabularies during the usage phase, they will have to define their own awareness policies, and they will have to develop their own clients. This situation can be desirable in cases where the developers have a high self-discipline and are capable of developing their own specialized clients. It can also be useful in cases where there is no central decision-making instance that can enforce vocabularies and tools.

It is obvious that a hybrid approach will be more desirable in the majority of cases. One such

[3] An adaptable system here is defined as a system that allows its users to adapt it to their needs.

hybrid approach can be the following:

- *Define a core organizational vocabulary in the instantiation phase*: This vocabulary will define a minimum set of product object and conceptual relation types, and a minimum set of attributes for each type.
- Define a global awareness policy in the instantiation phase: This policy can define a set of awareness schemes that have to be subscribed to by all the developers, and a handful of rules for creating mandatory awareness relations.
- *Develop a set of flexible clients in the instantiation phase*: This set of clients can be developed to allow the users define local and workspace vocabularies, and to add additional attributes to the objects and relations in the organizational vocabulary.
- *Define local and workspace vocabularies in the usage phase*: The clients can support users in defining new types of cluster objects and relations, and new types of informal objects.
- *Allow for development of local awareness policies in the usage phase*: In addition to the global awareness policy defined in the instantiation phase (and enforced by the specialized clients) each developer or group of developers can define their own awareness needs in the usage phase.

The two approaches to incremental deployment are not mutually exclusive. For instance, horizontally incremental deployment can also be desirable if only Product Layer is used.

9.4.1 The role of specialized clients in incremental deployment

As shown in Figure 9.1 on page 194, specialized clients are mainly responsible for manipulating the shared space in an IGLOO network. However, it is also possible to develop clients that are capable of *refining* the instance of the network in the usage phase (also shown in Figure 9.1 as a "Refine" link from specialized clients to instance). A proper suite of such clients can facilitate the horizontally incremental deployment process in that the instance itself can be defined incrementally in the usage phase. A similar approach is for instance used in Information Lens (Malone, Grant, Lai, Rao and Rosenblitt 1989). Information Lens is a structured message system where the users can incrementally develop new semi-structured message types. Information Lens provides end-user tools for constructing message templates for new types of semi-structured messages. In addition, the users have access to tools for creating rules that will process the different messages. Information Lens in this way allows for an incremental deployment in that the users define message and rule types in use. OVAL (Malone et al. 1995) followed the same approach in allowing the users to create any object type using similar editors.

In the same way, it is possible to develop specialized IGLOO clients that allow the users not only to manipulate the shared space in an IGLOO network, but also to refine the instance that the network is based on, i.e. to refine the vocabularies, the awareness policies, and even the clients themselves. However, we have to be aware of one important difference: Information Lens and OVAL are not heavily based on shared spaces as IGLOO is. Changing any vocabulary or awareness policy in an instance of IGLOO will affect all the users of the IGLOO network that is based on that instance. In Information Lens, changes to a message template or rule will not necessarily affect all the users. The same is true for OVAL. An example will illustrate this

difference. A rule in Information Lens is created by a user, and is also used by that user initially. A rule becomes shared only if the user who created it wishes to share it with others, and only if the other users want to use that rule. It is the same for message templates; the shared space will not automatically become more structured if an Information Lens user creates a new message template because all templates have local effect *initially*. In IGLOO, changes to the shared space will affect, to different degrees, all the users of the IGLOO network.

Examples of clients that allow the developers change the instance of an IGLOO network are:

- *Generic shared product space managers*: These are mainly browsers that allow developers to create and view any type of product object and conceptual relation. They allow the developers to browse the shared product space based on arbitrary attributes, to sort objects and relations differently, or to filter objects and relations based on arbitrary attributes. These managers may also allow the users to change the objects and relations by adding or removing arbitrary attributes, or to define completely new product or relation types. All this functionality is provided by the services of Product Layer, and a generic shared product space manager only needs to perform a mapping from these services into its own user interface.

- *Generic cluster managers*: In the same way as for shared product space managers, generic tools may be developed to allow the users to browse the existing clusters in an IGLOO network based on arbitrary attributes, to change the contents of the clusters, to create arbitrary cluster objects and relations, and to annotate the clusters by addition arbitrary shallow attributes. The functionality of generic cluster managers will be directly based on the services of Cluster Layer.

- *Generic workspace managers*: These are tools that allow users to browse existing workspaces, to browse the contents of the workspaces based on arbitrary attributes, to create workspaces with arbitrary informal objects, etc. The functionality of such clients will be based on the services provided by Workspace Layer.

- *Generic event monitors*: These are applications that can be configured to monitor an arbitrary part of the shared product space in an IGLOO network. Event monitors may also allow their users to decide what part of the shared product space or which clusters and shared workspaces to monitor. They may also allow for local configuration of awareness, e.g. by allowing their users to decide what awareness information should be shown.

This set of generic tools may be used by a group of developers in early phases of requirements and analysis in a product development project. For instance, being able to create any product object and conceptual relations can facilitate idea generation activities, where the ideas can be grouped into clusters, etc. Development of generic clients is an ongoing process (Garli and Lund 2000, Bhatnagar 1999), and a suite of such clients is planned to be a part of the generic implementations of IGLOO framework.

9.5 Architectural Features

Besides the conceptual issues involved in deploying IGLOO framework, architectural issues may play an equally important role in the usability of the resulting IGLOO network. Project groups

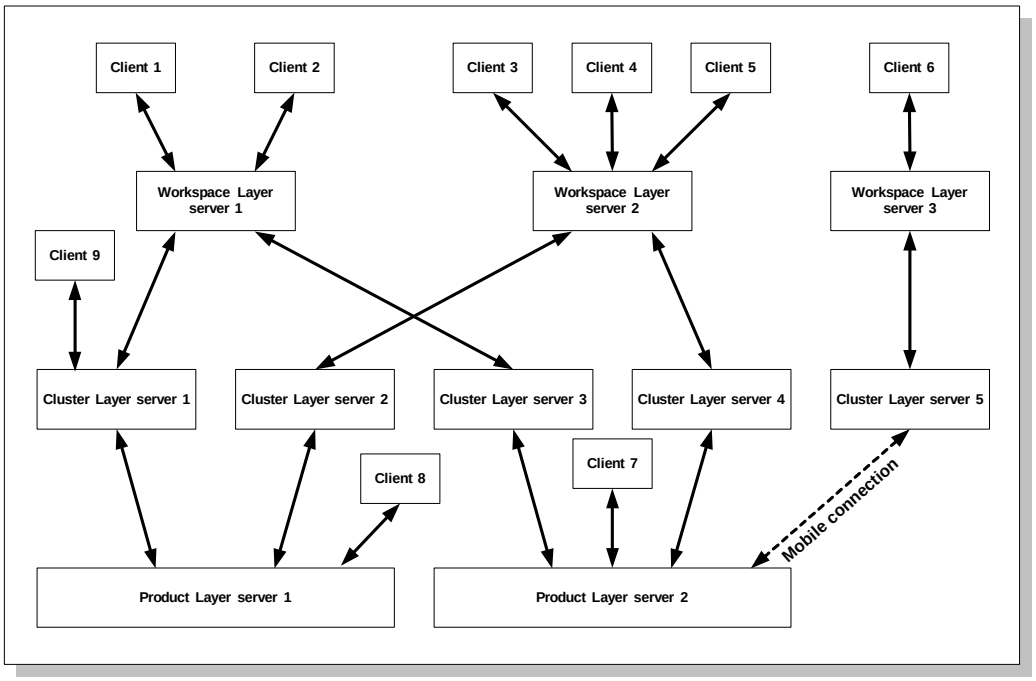


Figure 9.4: A configuration of layer servers.

will vary not only in their needs for information and cooperation support, but also in the way they are distributed geographically. Each IGLOO network will need a specific configuration of a distributed architecture in order to guarantee acceptable performance to its users. This brings us to the architectural flexibility of IGLOO framework that opens for different physical configurations of an IGLOO network.

IGLOO framework is defined in form of layers. Each layer provides a standard service interface to the IGLOO clients above it. This makes it possible (though not necessary) to implement each layer as an stand-alone network server, what we term as a *layer server*. As we have seen in Chapters 6–8 each layer has an example implementation that demonstrates this possibility. Having one stand-alone server for each layer has advantages in terms of performance and reliability. Layer servers encourage large local computation at each layer, making it possible to minimize the communication among the layers. For instance, when using CoClust, all the real time changes to the shallow part of the clusters can be processed locally by CoClust, while changes to deep attributes are communicated to an underlying Gossip server. This will result in increased performance by for instance having Cluster Layer server located on a network node closer to the group using it mostly. In this way we can achieve an architecture that is distributed, concurrent, and replicated (Dewan 1995).

In addition, the minimized communication among the layers makes it possible to use single layer servers in isolation, without loosing all the functionality of an IGLOO network. For instance, CoClust can be used in the absence of a Gossip server, and still allow its users to modify

the shallow parts of the clusters. (Of course changes to the shared product space will not be possible unless the underlying Gossip server is connected.) This is useful in cases where a network node running a layer server is not available due to network problems, or in cases where network connections are so slow that one would prefer to loose the Product Layer services in favor of better performance. A third important case is that of mobile computing, where network connections are not available all the time.

Designing the framework as three layers also makes it possible to scale up the network. Though we have not done much experimentation in this area, it is feasible to think of an IGLOO network consisting of different combinations of layer servers. For instance, the same Gossip server can serve several CoClust servers, each server located close to the group using Cluster Layer services related to their local clusters. Figure 9.4 shows a rather complex IGLOO network consisting of two Product Layer servers, five Cluster Layer servers, and three Workspace Layer servers. In particular, each Workspace Layer server may access more than one shared product space. Cluster Layer server 5 in the figure is connected to the shared product space through a mobile connection. Support for mobility is not existing in the current implementation of the framework, but can be implemented through caching mechanisms similar to the ones proposed in (Signer, Erni and Norrie 2000).

Though not strictly related to the deployment process, deciding what combination of layer servers to use can help to build an IGLOO network that is tailored to the specific needs of the project team using the network.

9.6 Summary

In this chapter we have outlined a deployment process for IGLOO framework. This process is a necessary step in specializing the generic services provided by IGLOO framework, and assures that the resulting IGLOO network will fit to the needs of any product development project. There are trade-offs in deploying IGLOO. At the same time there is also enough room for tailoring the process. Incremental deployment guarantees that the cost of deploying IGLOO framework will correspond to the gained cooperation support and technical quality. It is not necessary to deploy the complete framework if only a subset of the services are needed. It is also possible to upgrade a minimal IGLOO network to a more advanced network when this is needed.

The next chapter will described a detailed example of a deployment process. This example will illustrate the different activities in a deployment process , and will demonstrate how horizontally and vertically incremental deployment approaches can be combined in the same process.

		Deployment process phase			
		Instantiation		Usage	
		Pro	**Con**	**Pro**	**Con**
Deployment activities	**Identifying the settings**	Can function as a requirement phase before starting the whole process.	Real requirements may not be revealed before the usage phase.	Real requirements can be collected directly and used as the basis for the process.	The overall picture of the project and what the IGLOO network will do can be lost.
	Defining the vocabularies	Can make it easier to understand each other in a large project group.	Can impose rigidity by pre-defining the types of the objects and relations.	Can provide the developers with more flexibility and freedom to express their ideas.	Will make it difficult to communicate. Can introduce a lot of ad hoc "translation" work.
	Defining awareness distribution policies	Can assure that all the developers get the same awareness information.	Can make individual learning difficult. Can result in information overload.	Can allow each developer or group of developers tailor their own awareness information access. Allow for individual or local learning.	Can result in overhead work needed for defining local policies. Can result in divergence in awareness among the developers.
	Developing specialized clients and workspaces	Can secure the consistency of shared data. Can enforce a uniform interface to shared data.	Can result in rigid tools with little tailoring possibilities.	Can allow the developers create the tools they really need, or tailor their tools.	Requires that all the developers can develop their own clients and tools, or that client and tool builders are available all the time.

Table 9.1: The trade-offs when choosing to perform each deployment activity in a specific phase.

Chapter 10

Evaluating IGLOO Framework

10.1 Introduction

This chapter evaluates IGLOO framework. First, in Sections 10.2 and 10.3 we test the applicability of IGLOO framework through a detailed example of a deployment process. ALPHA is used as the settings for this example. The example will demonstrate how the various activities of the deployment process are performed in order to create an operational IGLOO network. Second, in Section 10.4.1 we evaluate the resulting IGLOO network against the requirements of Chapter 2 in order to see which of the requirements are satisfied and which are not. (These requirements are listed in Table 2.5 on page 43.) As part of the evaluation an estimate of the costs connected to deploying IGLOO framework is given. We also compare the functionality of IGLOO to other tools and environments.

The example used in this chapter is based on our understanding of ALPHA and similar projects, and on the functionality provided by IGLOO framework. The example demonstrates the applicability of the generic implementations, and IGLOO framework in general. ALPHA, which was used as the initial scenario for this thesis, provides a rather complex and realistic scenarios. Using ALPHA as an example will also make it easier to evaluate the intended functionality of IGLOO as stated in the requirements of Chapter 2. We have to emphasize that the described deployment process is not performed in a real-world situation. A real world test has not been possible because of the time frame connected to this thesis, but also because of the lack of access to a project similar to ALPHA.

The example assumes that the generic implementations of IGLOO framework, i.e. Gossip, CoClust and SWAL (described in Sections 6.3, 7.4 and 8.4)), are used. During the chapter we will demonstrate a number of IGLOO clients. Prototypes of most of these clients have been developed in the course of this research, with identical or similar functionality. We also describe some IGLOO clients that are not implemented yet or are in the process of being implemented. This is necessary in order to demonstrate the full potential of IGLOO framework.

Figure 10.1 shows an overview of the deployment process for ALPHA, which is a mixture of vertically and horizontally incremental deployment processes (see Chapter 9 for an overview of

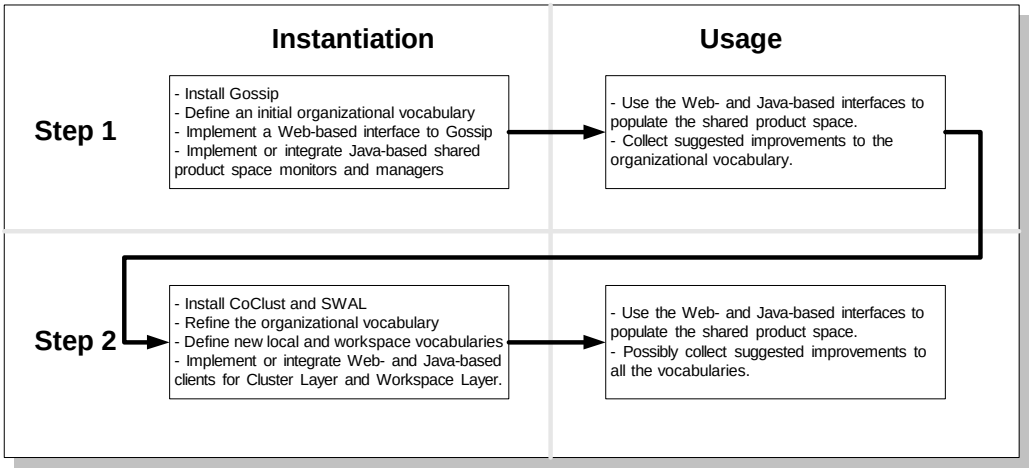


Figure 10.1: An overview of the deployment process for ALPHA.

incremental deployment). The process starts by deploying Product Layer in step 1 (described in Section 10.2), and upgrading to Cluster Layer and Workspace Layer later in step 2 (described in Section 10.3). An organizational vocabulary is defined in step 1. This vocabulary is refined in step 2 through a process of collecting suggestions for refinement during step 1. Step 2 also includes the definition of a number of local and workspace vocabularies. As we saw in Chapter 2, ALPHA consisted of developers with highly diverse backgrounds. Some of them had very low level of knowledge about computers and Internet. The advantage of starting only with Product Layer in step 1 is that we guarantee a smooth transition from basic to advanced cooperation support. This helps the developers with little knowledge about computers to gradually learn about IGLOO's services and the functionality of IGLOO clients. In addition, the possibility to gradually refine the vocabularies allows ALPHA members to adapt the IGLOO network to their changing needs and to the evolving product they are developing.

10.2 Step 1: Initial Deployment

In step 1 we create an initial instance of IGLOO framework for ALPHA. This step is concerned with providing a simple cooperation infrastructure for the developers in ALPHA. We want to allow the developers to set up an initial shared product space without much effort, and to start creating and sharing a product. We make use of Product Layer (in form of a Gossip server) and a set of IGLOO clients that will help the developers to set up and populate a shared product space. Most of the clients are Web-based. This will facilitate the initial introduction of the clients because most of the developers are already familiar with Web-based applications. In addition, a number of simple Java-based clients will be developed to allow for more interactivity. We will also describe how a number of familiar desktop tools can be integrated into the resulting IGLOO network.

As we saw in Chapter 2, two major problems ALPHA was facing were the lack of continuous and flexible access to the product. The assumption here is that the shared product space and

the associated IGLOO clients that are created during step 1 will allow the developers to have continuous and flexible access to the product. We do not consider centers of interaction in step 1, and will return to this issue again in step 2.

10.2.1 Defining the organizational vocabulary

IGLOO framework allows the developers to share any type of product object and create any type of conceptual relation among the objects. Therefore it is important to agree on a set of object and relation types (i.e. the organizational vocabulary) so that it is known what will be shared and what clients need to be developed to populate and maintain the shared product space. We start by defining an initial organizational vocabulary for ALPHA. As explained in Chapter 9, an organizational vocabulary consists of the definitions of product object and conceptual relation types, including what attributes each type will have. We then allow the developers to use this vocabulary, and to suggest improvements to it. Based on these suggestions, the initial vocabulary will be refined in the beginning of step 2. This is a suitable approach for ALPHA because of the informal nature of the product. We saw in Chapter 2 that the product in ALPHA was mainly constituted by highly informal product objects, and it was not always clear what objects and artifacts belonged to the product. By allowing a gradual refinement of the organizational vocabulary the developers will be able to "discover" in situ what objects they want to share as part of the product. This approach takes advantage of IGLOO's flexibility and can result in a vocabulary that fits to the needs of the project. In other cases it might be feasible to have a completely predefined organizational vocabulary. This can be the case for projects using standard methods and formalisms. In such projects, the organizational vocabulary used to define the product will be a standard predefined language.

The initial organizational vocabulary for ALPHA is shown in Table 10.1. The first six product object types are the ones that were already shared to some extent by ALPHA developers through WWW and mailing lists. The last three product object types (i.e. virtual napkin, whiteboard and discussion) are object types that represent some of the artifacts that were often used locally in face-to-face group meetings but were hardly shared across geographical sites. It is also possible to define user-defined operations to be simulated by Product Layer, but we omit this part here for briefness.

Two conceptual relation types are defined in the organizational vocabulary. A "Dependency" relation will be used for relating any two product objects that are somehow dependent on each other. Most "Dependency" relations will be created manually by the developers. This is necessary because in the beginning of the project a large part of the product is in form of informal product objects (e.g. natural language documents). For stronger dependencies among product objects we define another conceptual relation type called "Import." This type of relation is used whenever the structure of the product strictly enforces dependencies, e.g. dependencies among source code files because they import each other. "Import" relations may be created automatically by the tools that the developers are using. E.g. many compilers can detect import relations among source code files.

Table 10.1: An initial organizational vocabulary for ALPHA

Object/relation type	**Purpose**	**Attributes**
Natural language documents (product object)	These are various kinds of documents that are written by the developers. They can be notes, deliverables, design documents, etc.	Product object type, name, owner, document type, last changed, version, URL, title, abstract, format
Visual documents (product object)	These include diagrams that are used to visualize some aspects of the product	Product object type, name, owner, last changed, version, URL, title, abstract, format
Notes (product object)	Thes are short text messages that are left in the shared product space for others to read	Product object type, name, owner, last changed, URL, title
Prototypes (product object)	These are executable prototypes of the different parts of the end-product	Product object type, name, owner, last changed, version, URL, operative system
User-interface mock-ups (product object)	These are animations that show the functionality of those parts of the end-product that do not yet exist in form of executable prototypes	Product object type, name, owner, last changed, version, URL, helper program, helper program URL
Source code files (product object)	Files including source code of various kinds	Product object type, name, owner, last changed, version, URL, programming language
Virtual napkin (product object)	An image of an artifact that is used in a local meeting. This can be a scan of a piece of paper (or a napkin) with sketches or drawings, a photo of a situation or scenario that was used to explain an issue, a photo of a blackboard with writings on it, etc.	Product object type, name, owner, last changed, version, URL, title, comment, format
Whiteboard (product object)	This is a bitmap file that everybody can draw on. It is in a format that can be modified by common image editors found in Windows and other operating systems (e.g. the Paint program in Windows can be used for editing). The object will be used for freehand drawing and annotations	Product object type, name, owner, last changed, version, URL, title, comment, format

Continued on next page

Continued from previous page

Object/relation type	Purpose	Attributes
Discussion (product object)	A text file that the developers can use for discussing different issues. Each developer can append a paragraph with his own signature to the end of the file	Product object type, name, owner, last changed, version, URL, title
Dependency (conceptual relation)	This is a relation that is created from object A to B if B depends on A in some way. It indicates a need for informing B if A is modified	Conceptual relation type, name
Import (relation)	This is a strong dependency relation from an object to another. It is used for representing dependencies that can be derived based on the structure of the product, e.g. import relations among source code files	Conceptual relation type, name

This list of object and relation types can be extended. IGLOO framework allows new types to be created as the project proceeds and the need for new types arises. For ALPHA we choose to have a "semi-controlled" process for refining the organizational vocabulary, where the developers will not be able to add new types directly. However, we allow them to make suggestions for new types or modifications to the existing types. These suggestions will be collected automatically by the clients, and will be used in step 2 for refining the vocabulary. Note that this specific process is not imposed by IGLOO. IGLOO also supports decentralized and ad hoc refinements, i.e. each developer adding new types and attributes without any control. We choose deliberately to have a more controlled process for ALPHA in order to prevent inconsistencies among the sites, and to simplify the implementation of future clients.

The organizational vocabulary defined here is a convention that ALPHA agrees to follow. It is not "hard-coded" in any form. This means that all the developers agree to (initially) share object and relation types of the ones defined in Table 10.1, and to develop their IGLOO clients in a way that they will respect the definitions.

Since step 1 does not make use of Cluster Layer or Workspace Layer, there is no need for defining any local or workspace vocabularies.

10.2.2 Defining the awareness policies

Once an initial organizational vocabulary is in place, the next step in creating the instance is to define its awareness policies. Awareness policies consist of rules for choosing the scope of direct awareness, for creating awareness relations, and for creating and subscribing to awareness schemes.

We can configure Product Layer to generate direct awareness events for any access to the

shared product space (including read access) or only for accesses that result in modifications to the shared product space (see Chapter 6 for details). The latter configuration will result in far less awareness information because most access types (e.g. read, search, view) will not result in any notifications. Note that choosing one option does not exclude the other. We can change the configuration any time.

We start by letting Product Layer generate awareness events for all kinds of access to the shared product space. This can be useful in the beginning in order to allow the developers to become familiar with the different types of events that are generated. This does not need to result in information overload because the developers will use the Web-based interface initially (Web pages are not affected by real-time notifications). For those developers using Java-based clients this configuration may result in too many awareness events. However, the Java-based clients use most of the events for implicit synchronization of screens, and do not impose information overload problems. In addition, these clients can be developed in such a way that the developers can configure them locally for filtering out certain types of event.

For the two conceptual relation types "Dependency" and "Import" we define two different rules. We decide that conceptual relations of type "Dependency" will not by default have awareness mediation property, e.g. they will not be awareness relations by default. A developer has to explicitly "switch on" each relation if he wants it to mediate awareness. For conceptual relations of type "Import" we decide that they will be awareness relations by default. We make this distinction because relations of type "Import" are stronger and more subjective (e.g. imposed by the structure of the product), while relations of type "Dependency" are weaker and are made because an individual developer found it necessary.

We also define one global awareness scheme for ALPHA called "project," and require all developers to subscribe to it. We allow the daily management team be responsible for maintaining this awareness scheme. This awareness scheme can for instance be used by the daily management team to make sure that the right awareness information is reaching the right groups. For instance, dependency relations belonging to the "project" scheme can be made from some central product objects (e.g. a requirements document) to local product objects at each site in order to assure that all the local sites are informed when these central objects are modified.

10.2.3 Developing specialized clients

Once an organizational vocabulary is defined, it is necessary to develop a set of clients in order to allow the developers to populate the shared product space with objects and relations based on the vocabulary. We start by describing a set of Web-based clients for ALPHA. This interface is easy to implement, is low-cost, and will give access to most of the functionality of Product Layer. In addition, Web-based interfaces are intuitive and do not need any installation of local software. Therefore they are natural "first choice" clients for ALPHA. Later we will describe a set of more advanced Java-based direct manipulation clients.

The Web-based clients

All the Web-based clients explained here implement a mapping of the services of Product Layer on to HTML pages. They provide access to most of the functionality of Product Layer, in particular the shared product space services shown in Table 6.1 on page 129.

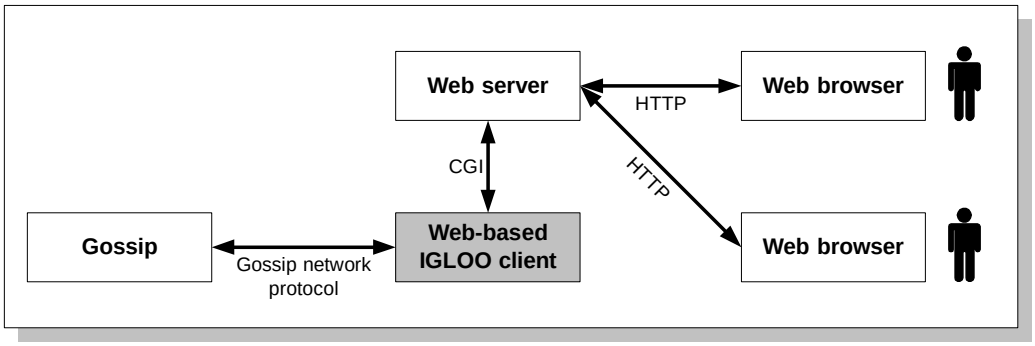


Figure 10.2: The architecture of Web-based IGLOO clients.

Web-based clients are implemented as simple Web server extensions. Communication with a Gossip server is performed through the architecture shown in Figure 10.2. A Web-based IGLOO client is invoked by a user requesting the URL of the client from the project Web server. Communication between the user's browser and the Web server happens through standard WWW protocol HTTP. This means that the user does not need to install any software in addition to his standard Web browser. Upon the arrival of the request, the Web server invokes the IGLOO client through the standard CGI protocol. CGI communication means that any standard Web server can be used. The IGLOO client uses Gossip network protocol to issue a service request to Gossip. The result of the request is received by the client and is communicated to the Web server in form of an HTML Web page This Web page is in turn sent to the user's browser. Examples of such Web-based clients were implemented for interfacing with ICE (see Appendix A, and Farshchian and Divitini 1997). In (Bhatnagar 1999) other examples based on Java servlets are proposed that are also capable of communicating with Gossip and a Java-based Web server.

The Web-based interface for ALPHA can be implemented in form of a collection of Web-based IGLOO clients[1]. All these Web-based clients are generic. Once implemented, they can be used in other settings and by other projects. In fact, they can be developed as part of the future version of IGLOO's generic implementations:

- *Generic product space manager*: Reads the contents of any shared product space and displays it in form of a list in an HTML page. Allows the user to invoke other Web-based clients for creating, viewing, modifying, or deleting product objects. Gives also access to other connected users, and allows the user to invoke an instant messenger for sending text messages to these users.

- *Generic object creator*: Allows the user to create a new object of a specific type. It provides fields for filling in the values of the object's attributes, and a file holder for uploading a content file for the object.

- *Generic object modifier*: Displays information about an arbitrary product object on a Web

[1]These clients can also be implemented in form of one large client. However, our experience shows that having a collection of smaller CGI programs provides higher reliability and performance than having one large CGI program.

page, including the object's attribute-value pairs. Allows the user to modify the object's attributes, or upload and download its content file.

- *Generic object viewer*: Allows the user to view the attribute-value pairs of an arbitrary product object, and to download the object's content file.

- *Generic instant messenger*: Allows the user to send a text message to a single user or a group of users. (This is not a pure HTML-based client, but makes use of a Java applet inside a HTML page.)

- *New product object type application*: Allows the user to describe a new product object type by specifying a set of attributes and a purpose for the new type. It then sends an email to the administrator of the system, informing about the object application. The application is saved for step 2, where it is possibly used for refining the organizational vocabulary.

- *New conceptual relation type application*: Similar to the above, but for conceptual relations.

The Web-based interface does not support the creation of relations because WWW and HTML pages are tedious for performing such interactive tasks. Relations are created using the Java-based direct manipulation clients explained later.

All the Web-based clients provide links to product object and conceptual relation applications. This is important for enabling an efficient process for refining the organizational vocabulary. Making it easy to specify what objects and relations are used by the different developers will make it easier to provide the possibility for sharing those objects and relations in the future versions of the vocabularies and clients.

A screen shot of a Web-based generic product space manager is shown in Figure 10.3. Through this IGLOO client a developer in ALPHA can view the shared product space, and he can invoke the other Web-based clients for creating, viewing, and modifying various product objects. The contents of the shared product space is shown in form of a table in the lower part of the screen, while the upper part provides a range of functionalities for modifying the contents of the space. A drop-down list of defined product object types is available (above-left corner in Figure 10.3). The developer can choose one of these object types as the basis for creating a new product object. To the left of this list there is another drop-down list showing currently connected ALPHA developers. The developer can choose the name of another developer from the list, and send an instant text message to that developer (or a message to all currently connected developers). The above-right corner shows two "Apply" buttons that allow the developer to apply for new product object or conceptual relation types.

The view of the shared product space (the lower part of the screen in Figure 10.3) shows a list of currently existing product objects in the shared product space. By pushing "View" or "Modify" buttons for each object the developer can invoke an object viewer or modifier on that object. By marking an object using a check mark and pushing the "Delete" button the developer can delete any object from the space. The developer can also filter the view of the shared product space by limiting the view to objects with a specific attribute-value pair. In order to allow the developer to see at a glance which objects are changed recently, the left-most column in the table gives an indication of when the object was changed last time. A "New" object is created newly, while a "Changed" object has been changed recently, and an "Old" object has not been changed for the

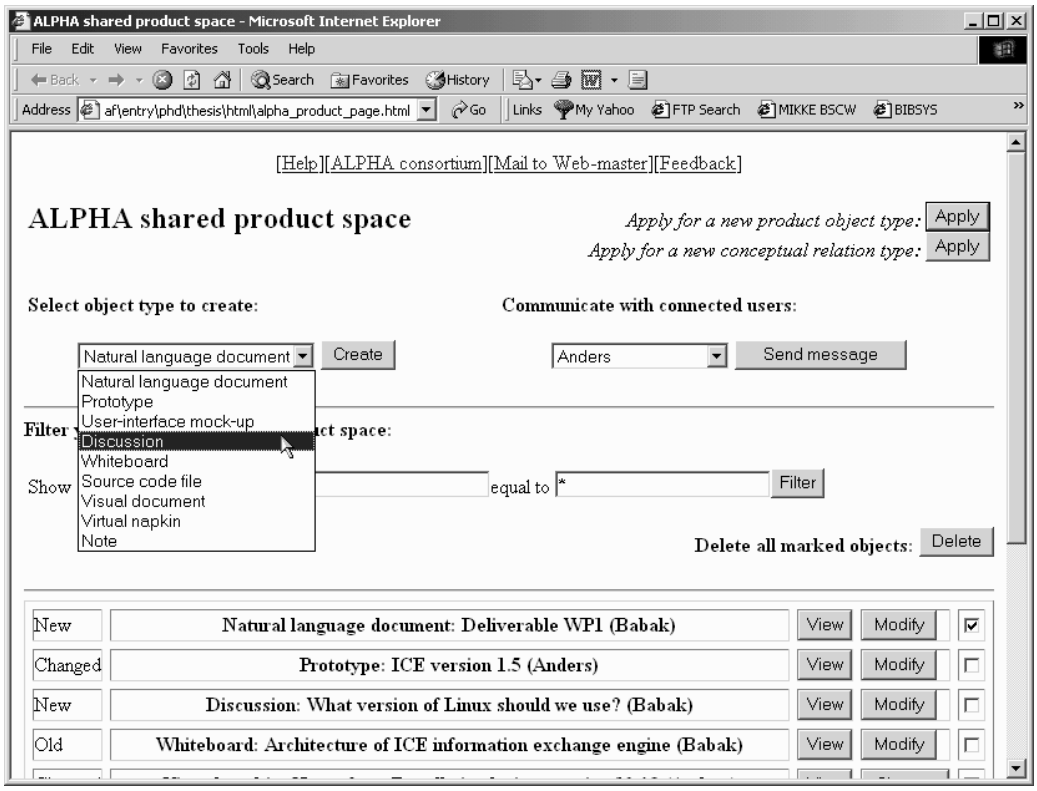


Figure 10.3: ALPHA's Web-based generic product space manager.

last N days[2].

By clicking on the "Modify" button of a product object in Figure 10.3 the user will invoke an object modifier. Figure 10.4 shows a Web-based generic object modifier. This client gives access to the attribute-value pairs and the content file of the product object. The developer can modify any attribute, upload a new content file, or add/delete attributes. Note that we decided to have a semi-controlled process for refining the organizational vocabulary. This means that adding or deleting an attribute will not change the organizational vocabulary directly. Adding or deleting attributes through the object modifier of Figure 10.4 will not actually add or delete the attribute, but will result in a message (a suggestion) being sent to the administrator. The suggestion will be used in step 2 for refining the organizational vocabulary.

The generic object modifier of Figure 10.4 can be used for modifying any product object. However, for some object types it will be necessary to provide a more tailored modifier. An exam-

[2]A more advanced awareness mechanism based on Java applets could have been used here instead of discrete values such as "New," "Changed" or "Old." Such a mechanism is implemented as an extension to ICE (Knudsen and Solheim 1999). The idea is to allow the developers see, in real time, all accesses to product objects shown in a Web page. This is done by having a small Java applet (a "product object monitor") attached to each product object in the Web page. The attached Java applet changes color when other developers access the product object.

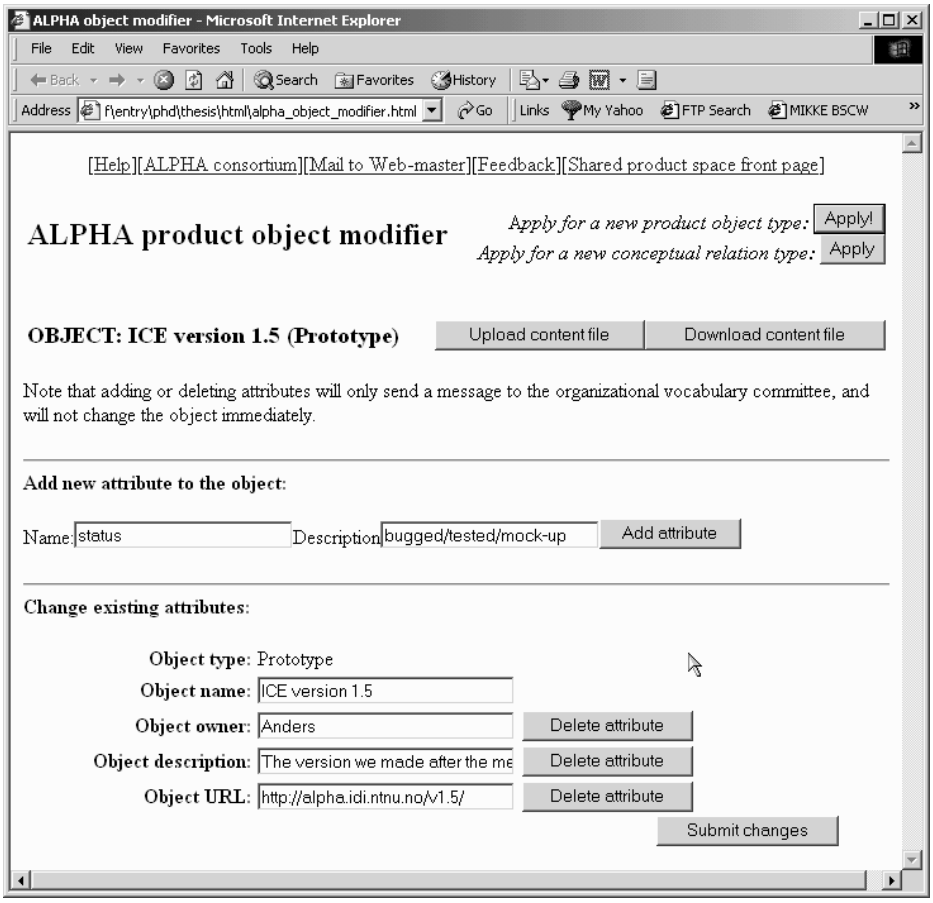


Figure 10.4: ALPHA's Web-based product object modifier.

ple of a tailored modifier for the "Discussion" product object type is shown in Figure 10.5. This modifier provides an easy interface to the developers to append messages to a discussion object and to read the messages sent to a discussion. The above part has as usual buttons for applying for new product object and conceptual relation types. The middle part allows the developer to create a new message to be sent to the discussion by specifying a subject, a body, and a message type (according to an IBIS-like categorization of messages). The lower part of the screen contains a list of already posted messages.

The above collection of Web-based clients provides a flexible interface to an initial shared product space for ALPHA. All developers can share their product objects through a few simple steps using an intuitive interface. The interface is flexible and allows the developers to modify the contents of the shared product space in a decentralized manner. New object types can be added as needed, and the attribute set of each object type can be changed through a refinement process. Explicit communication is supported through instant messages (for opportunistic communication) and discussion objects (for more structured and focused discussions).

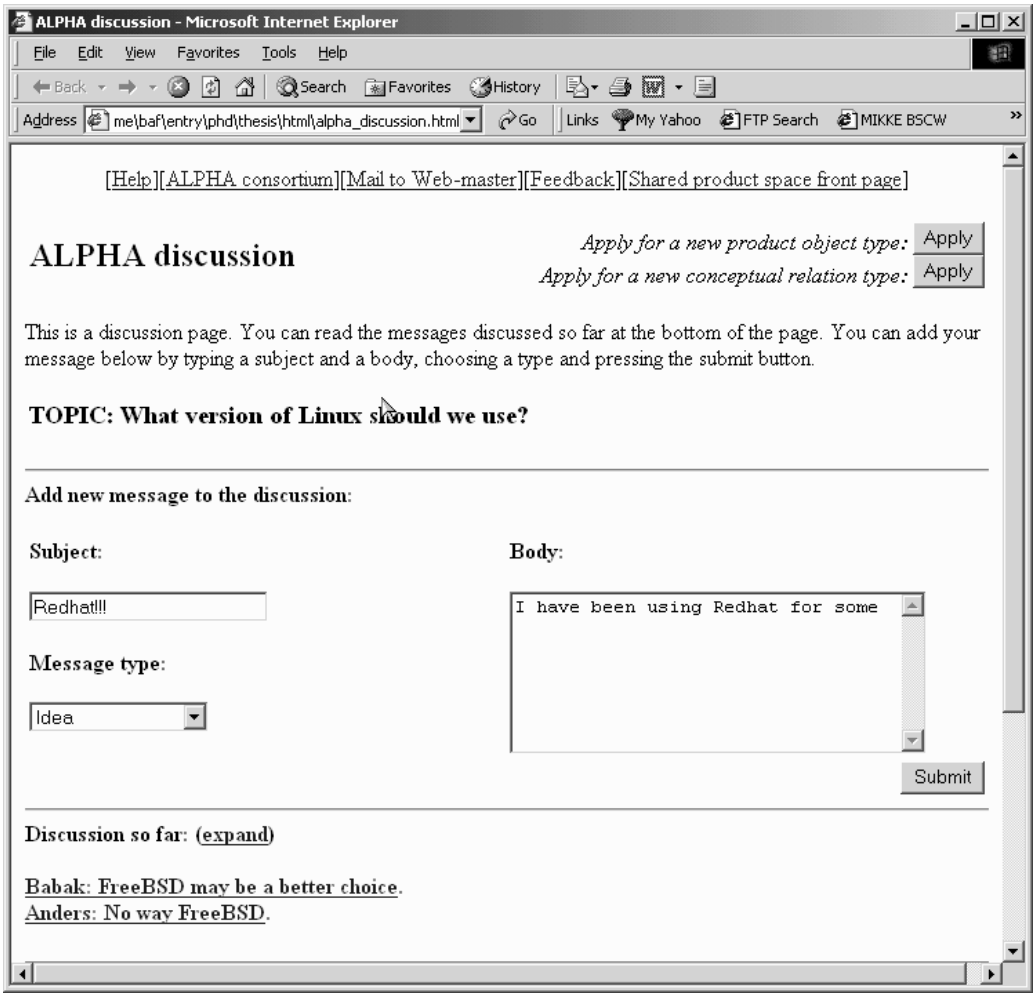


Figure 10.5: A Web-based client for interacting with a discussion object.

Awareness information related to the activities of the developers is available to some extent through this Web-based interface. In particular, a list of connected developers is available all the time, and can be used for knowing who is available. Changes to product objects are also visible through a "last-changed." However, more advanced awareness will only be available through the Java-based interface to the shared product space.

Java-based clients

In addition to the Web-based interface, a set of Java-based clients can be useful for ALPHA[3]. The first essential Java-based client is a shared product space monitor (an example of such a monitor, called Eskimo, is implemented as part of MultiCASE, see Figure 5.3 on page 111 and Christensen and Karlsen 1999). This client can monitor the shared product space for any access type. It also allows the developers to specify the part of the shared product space they want to monitor. It is started every time a developer logs on to his computer. It remains in the background, and generates sound signals when something happens in the shared product space. As we saw in the previous section, the Web-based clients provide limited (delayed) awareness support regarding changes to the product objects. The Java-based shared product space monitor can solve this problem by providing real-time awareness of accesses to the product objects.

Another useful Java-based client is a generic shared product space manager shown in Figure 10.6. This client is very similar to the familiar file managers used in conventional operative systems. The large white area shows a view of the product objects in the shared product space. A developer can browse the contents of the shared product space, and can delete or modify any product object. He can add any file from his own local hard drive to the shared product space as product object. Adding product objects can be done either through a menu, or by dragging and dropping the local file into the view area of the client (in which case a dialog box will open and asks for object type and attribute values). Conceptual and awareness relations among any two product objects can be created by highlighting the source object (by a mouse click), highlighting the destination object, and selecting a menu item. Relations are not shown in the view area. Deleting and modifying existing relations happens through a dialogue box.

The view area can be configured (filtered) to show all the product objects in the shared product space, or only the product objects belonging to a specific type. The view is filtered by specifying a product object type in the drop-down list with the title “Show object type:”. (It may also be possible to select multiple object types. This functionality will require an additional dialogue box containing a check box for each object type.) All visible objects are visualized and sorted by one specific attribute-value (specified in the text fields “Filter” and “equal to” in the lower-left corner). The titles of the product objects in the view area show the values of the specified attribute for each object. The other attributes of the objects can be viewed or modified by double-clicking on each object.

Other developers’ accesses to the contents of the shared product space are visualized by product objects changing color from white to gray when they are accessed. For instance, in Figure 10.6 product object “Note on DB connection” has a gray color indicating that some other user is accessing it. Moving the mouse pointer on to the object will show a text describing which user is accessing the object and what kind of access it is. Additional awareness support can be to use sound icons for different events in the shared product space.

The continuous awareness of who is accessing the product objects (i.e. being able to see the objects change color) provides occasions for opportunistic communication. The client supports instant message exchange among connected users. A user can send instant messages to other connected users, and receive messages from them. A message is sent by choosing the name of the receiver from a drop-down list of all connected users (or choosing to send a message to all

[3]Note that all these Java-based clients can in principle be used as applets inside HTML-based Web-pages. We have found it however more convenient to focus on developing them as stand-alone applications. This is mainly because most Internet browsers do not yet support Java’s full functionality (or are incompatible with other browsers).

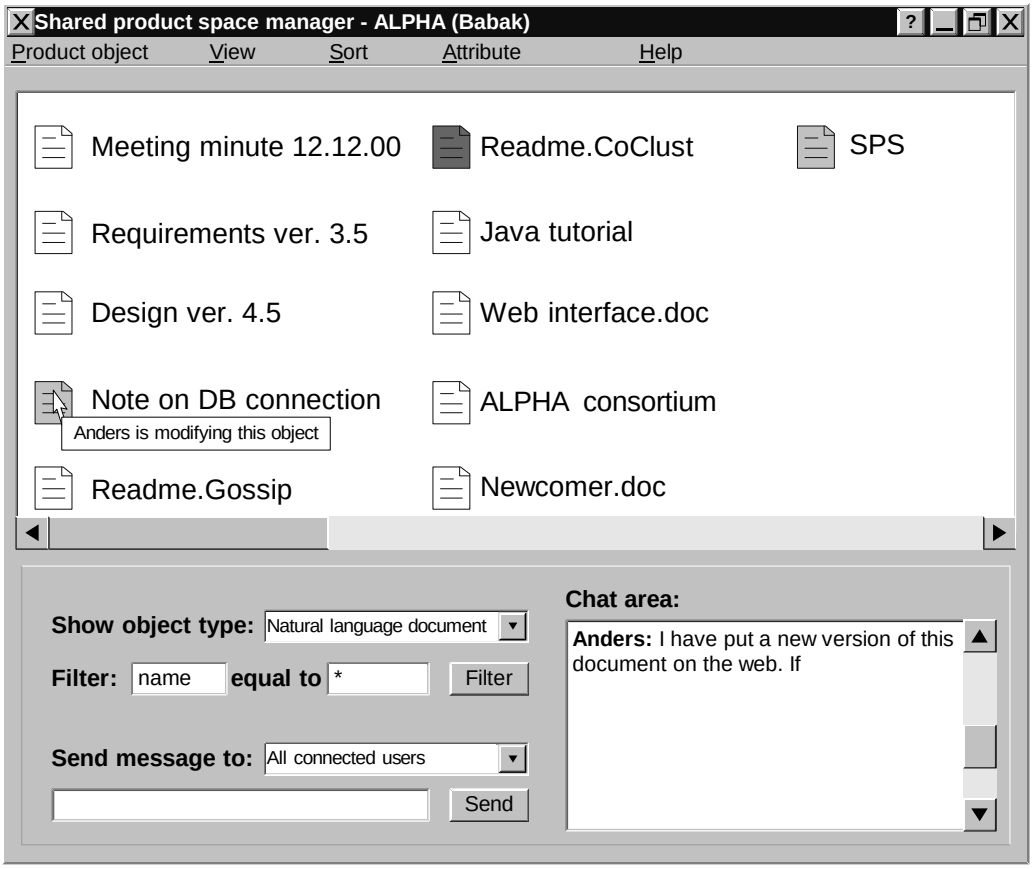


Figure 10.6: A direct manipulation product space browser with a Windows graphical user interface.

connected users). An initial version of this shared product space manager is described and partly implemented by Garli and Lund (2000).

Additional clients can be implemented in order to make the interaction with the product even easier. Gossip network protocol can be used directly to integrate the conventional tools of the developers into ALPHA's IGLOO network. An example can be to implement a Visual Basic extension for Microsoft Word. The extension can be made available as a "share" menu item that allows the developers to upload documents into the shared product space directly from Microsoft Word. A similar extension in Elisp programming language can be implemented for emacs.

10.3 Step 2: Enhancing the Cooperation Support

Using Product Layer services with a set of generic Web- and Java-based clients allows the developers in ALPHA to create and maintain a shared product space. Product Layer and the IGLOO clients developed in step 1 allow ALPHA developers to interact with the shared product space in a highly flexible and active way, and continuously keep updated about changes to this space using the Java-based clients. The solution also supports collecting feedback on the organizational vocabulary so that the real needs of the developers regarding their interaction with the product are registered.

When the solution implemented in phase 1 is used for a while we can expect a number of things to happen. First, the ease of interacting with the shared product space (in particular using the Web-based clients), and the decentralized control over its contents, will most probably result in a growth in the size of the shared product space. This expectation is based on our observation of ALPHA using ICE. ICE provided a similar interface for uploading and modifying objects, and was extensively used by a group of developers for sharing objects. (This is explained in Chapter 2.) As more product objects and conceptual relations are added to the shared product space, more advanced mechanisms for viewing the product in different ways will be needed. Second, groups of developers will start working in a more focused manner on different parts of the product. These two expectations require more support for centers of interaction.

In addition, the product will become more detailed. The more the developers know about each product object the more they will need to add details to it, and the more the developers know the product the easier it will be to create conceptual relations among the product objects. This will require a refinement to the organizational vocabulary in order to make it more detailed, possibly adding more conceptual relation and product object types.

The main goal of step 2 in ALPHA's deployment process is therefore to upgrade the already existing IGLOO network with functionality for creating centers of interactions. For this we will use Cluster Layer (for creating clusters) and Workspace Layer (for creating workspaces). In addition, an equally important task during step 2 will be to refine the organizational vocabulary based on the suggestions of the developers in step 1.

Note that these changes to the existing IGLOO network do not imply that the already existing shared product space and its contents will become superfluous. Step 2 performs an upgrade of ALPHA's IGLOO network. Existing product objects and conceptual relations will still be fully available in the upgraded network. In addition, the clients developed in step 1 will still work as they did, and will be able to communicate with the clients that are going to be developed during step 2.

10.3.1 Refining the vocabularies

This activity consists of two parts. First, the initial organizational vocabulary will be refined based on the suggestions provided during step 1. Second, local and workspace vocabularies will be defined for the new clusters and shared workspaces that are going to be created.

Applications for new product object and conceptual relation types, and suggestions for changes to the attributes of the existing object and relation types are collected during step 1. These applications and suggestions are used in step 2 to refine the organizational vocabulary. Some of the suggested additions to the original organizational vocabulary are shown in Table 10.1. The basis

for this table is that we assume more relations will be needed as the product is developed conceptually, and that additional support for programming some parts of the product will be in demand. A number of conceptual relations are added in order to allow the creation of more specific relations among product objects. Additional object types for facilitating Java programming are also added. All the new conceptual relation types will be created manually. The "Import" relation from step 1 is still the only relation that can be derived based on the structure of the product. However, "Import" relations can now be used also among product objects of type "Package."

Table 10.2: A refined organizational vocabulary for ALPHA. These definitions come in addition to those in Table 10.1.

Object/relation type	**Purpose**	**Attributes**
Java source code file (object)	This is a specialization of the already existing source code file object type. Each such object contains one Java class definition and implementation	Product object type, class name, owner, last changed, version, URL, programming language, comment
Package (object)	This is an object type for creating groups of objects. It allows the users to create collections of object, but does not enforce any name space	Product object type, package name, owner, last changed, version, comment
Generalization (relation)	A relation from A to B if B is a generalization of A. This is the same relation that is used in object-oriented analysis and design	Conceptual relation type, name
Association (relation)	A relation between A and B if there is an association of some type between the objects	Conceptual relation type, name, source role, destination role, source cardinality, destination cardinality, association type
Part of (relations)	This is a relation from a product object to another product object or to a package. It defines the contents of a package or parts of a product object. It is a special type of association	Conceptual relation type, name

The increase in the size of the shared product space necessitates the use of clusters. It is no longer feasible to only rely on shared product space managers such as the ones shown in Figures 10.3 and 10.6 to interact with the shared product space, in particular because they do not provide much support for centers of interaction. E.g. they do not support organizing the space

into groups of product objects, real-time sharing of views, annotating or customizing local views, or any shared workspace functionality. These shared product space managers can still be used for browsing the space, but additional tools for managing clusters and shared workspaces are needed. Clusters and shared workspaces are used in IGLOO for providing this type of functionality. For using clusters and shared workspaces we need to define local and workspace vocabularies. For ALPHA we will define two local vocabulary and one workspace vocabulary in order to illustrate the process.

We want to define a *basic cluster* type for allowing developer groups in ALPHA to share parts of the product using a graphical editor similar to MultiCASE (see Chapter 5 for MultiCASE). The local vocabulary for this basic cluster consists of one cluster object type and one cluster relation type, and is shown in Table 10.3. The vocabulary provides direct access to all product object and conceptual relation types in the shared product space (including all the attributes of each object and relation). The vocabulary defines an additional set of shallow attributes. These shallow attributed are used by the multi-user graphical editor to visualize the cluster objects and relations, and to support annotating objects and relation in a cluster. Two shallow attributes X and Y are defined to denote the position of a cluster object on the two-dimensional surface of a cluster. We also define an attribute for each cluster object to denote a graphical image or icon for visualizing the object. For cluster relations we define attributes for line style and thickness, and for arrow shape.

Table 10.3: A basic local vocabulary for creating simple clusters using a multi-user graphical editor.

Object/relation type	**Deep attributes**	**Shallow attributes**
All product object types from the organizational vocabulary	All attributes defined for the underlying product object type	Cluster object type, X and Y attributes, graphical icon, font type, font size, annotation
All relations from the organizational vocabulary	All attributes defined for the underlying conceptual relation type	Cluster relation type, line type, line thickness, arrow type, annotation

We define an additional local vocabulary for creating a more specialized type of cluster, i.e. a *UML cluster*. This vocabulary will be used by a modified version of MultiCASE, which we call *MultiUML*. MultiUML allows the developers to formalize parts of the shared product space in form of UML class diagrams (Booch, Rumbaugh and Jacobson 1999). Table 10.4 shows the UML cluster vocabulary for ALPHA. UML-specific object and relation types are based on already existing product object and conceptual relation types from the organizational vocabulary, but additional shallow attributes are defined. In particular, the cluster object type “Class” is based on product object type “Java source code file” from the organizational vocabulary, and cluster object type “Stereotype” is based on any product object type, while cluster object type “Package” is based on the product object type “Package.” (UML cluster supports a very small subset of UML language, and is used here solely as a demonstrator of how a typical conceptual modeling language can be used in combination with a shared product space in an IGLOO network.)

Table 10.4: A local vocabulary for making UML class diagrams in ALPHA.

Object/relation type	**Deep attributes**	**Shallow attributes**
Class (based on product object type "Java source code file")	All the attributes from the underlying product object type	Cluster object type, data, methods, X and Y attributes, graphical icon, font type, font size, comment
Stereotype (based on any of the objects in the shared product space)	All the attributes from the underlying product object type	Cluster object type, X and Y attributes, graphical icon, font type, font size, comment
Package (based on product object type "package")	All the attributes from the underlying product object type	Cluster object type, X and Y attributes, graphical icon, font type, font size, comment
Dependency (based on conceptual relation type "dependency")	name	Cluster relation type, line type, line thickness, arrow type, comment
Import (based on conceptual relation type "import")	name	Cluster relation type, line type, line thickness, arrow type, comment
Generalization (based on conceptual relation type "generalization")	name	Cluster relation type, line type, line thickness, arrow type, comment
Association (based on conceptual relation type "association")	name, source role, destination role, source cardinality, destination cardinality, association type	Cluster relation type, line type, line thickness, arrow type, comment
Part of (based on conceptual relation type "part of")	name	Cluster relation type, line type, line thickness, arrow type, comment

The clusters that are created based on the above local vocabularies can be used by specialized "cluster editors," or they can be used as elements in shared workspaces (see Figure 8.1 on page 176, and Chapter 8 in general for details on workspaces and how clusters are used in workspaces). We choose to create a basic workspace type for ALPHA. Shared workspaces are useful for ALPHA because they provide integrated support for creating centers of interaction. A shared workspace can contain both clusters and other informal objects. In addition, workspaces support explicit communication and cooperation within a center of interaction.

We define a shared workspace vocabulary for a *basic workspace*. This vocabulary will be used for creating shared workspaces similar to those in MultiCASE. A basic workspace will allow the users to create product objects within the workspace. The idea is to allow developers to create

any product object informally in their workspace without registering it as part of the product. For instance, a natural language document can be created in a shared workspace, and moved into the shared product space later. In addition, basic workspace will have a richer representation of its inhabitants than what is provided by the shared product space managers developed in step 1. The workspace vocabulary for basic workspace is shown in Table 10.5.

Table 10.5: A workspace vocabulary for basic shared workspaces in ALPHA.

Object type	Purpose	Attributes
All product object types from the organizational vocabulary	Allow the developers within a shared workspace to create product objects for local use, without having to insert them in the shared product space	Informal object type, X and Y attributes, graphical icon, font type, font size, annotation
Inhabitant	A representation of a workspace inhabitant	Inhabitant name, email address, image, location X and Y, color

10.3.2 Refining the awareness policies

In step 1 we decided to allow Product Layer generate awareness events for all accesses to the shared product space. As the size of this spaces grows, and as the objects and relations become more detailed, the number of awareness events produced by Product Layer will grow. The original configuration may now result in information overload, and we decide therefore to change the configuration so that Product Layer generates events only if the users make modifications to the shared product space.

The refinement of the organizational vocabulary in step 2 has resulted in a number of conceptual relation types. For each of these relation types we can define whether the relation type by default will mediate awareness, i.e. whether it will be both conceptual and awareness relation. For the new conceptual relation types that are defined in step 2, i.e. “Generalization,” “Association,” and “Part of” we decide not to require them to be awareness relations by default. For each of these three relation types, the users have to explicitly “switch on” the relation if they want it to mediate awareness. This of course requires that the clients provide the right dialogue boxes to the users for changing the behavior of each relation.

10.3.3 Developing specialized clients

This activity consists of two separate sub-activities. The first one is to refine the Product Layer clients developed in step 1, so that they will support the new organizational vocabulary. The second sub-activity is to develop new clients for Cluster Layer and Workspace Layer. At this step, both MultiCASE and ICE will be integrated into the existing IGLOO network. MultiCASE is an IGLOO client (a client of Workspace Layer) but ICE was originally developed as a stand-alone application and has to be integrated into IGLOO.

Modifying the existing Product Layer clients

This activity is optional, and depends on how the original Product Layer clients were implemented. It is fully possible to implement these clients so that they can use a set of product object and conceptual relation "templates" to configure themselves. For instance, the shared product space manager shown in Figure 10.3 can be implemented to initialize itself using a set of templates. In this case, adding new templates or changing the existing ones will be enough to upgrade the client. Otherwise the clients have to be recoded to include the refinements to the organizational vocabulary.

Web-based Cluster Layer clients

We implement a Web-based interface for Cluster Layer. This interface consists of a set of CGI-based IGLOO clients similar to those developed in step 1 (see Figure 10.2 on page 213). Here, the clients will communicate with CoClust instead of Gossip. The developed clients are:

- *Generic cluster manager*: Lists all the existing clusters. The user can view a list of all clusters, and can invoke other Web-based clients for creating, deleting, modifying or viewing each of the clusters. The user can filter the existing clusters based on different attributes, and see who else is using Cluster Layer.
- *Generic cluster creator*: Allow the user to create a new cluster. The user can create a cluster based on one of the defined cluster types.
- *Generic cluster modifier*: Allows the user to modify an arbitrary cluster. The client is used to add or remove cluster objects from the cluster, or to modify cluster attributes.
- *Generic cluster viewer*: Allows the user to view the contents of an arbitrary cluster.

Note that these clients provide only access to a small subset of Cluster Layer services due to the limitations in the interactivity of a Web-based interface. The Web-based interface gives easy access to some necessary services, such as creating and modifying clusters, and can be useful for some of the users who do not want to install new software. More interactivity is provided by the Java-based clients described later.

We describe here only the generic cluster manager. Figure 10.7 shows the interface of this client. The lower part of the window is a list of existing clusters. The upper part of the window is used for creating new clusters, and for communicating with Cluster Layer users. A developer can create a new cluster based on the existing cluster types. In addition, instant text messages can be sent to individual developers or all the currently connected developers. The client also allows the developers to apply for new types of product object, conceptual relation, and cluster. This can be useful for further refinements of the vocabularies.

The Web-based clients described above provide functionality that is similar to that of ICE. In ICE, users are provided a Web page that gives access to the existing workspaces (see Figure A.2 on page 249[4]) and additional Web pages for adding and modifying single objects. ICE's interface can be used for interacting with Cluster Layer. The work required for doing this is mainly related to creating the right communication procedures for allowing ICE to communicate with Cluster

[4]Workspaces in ICE are much more limited than IGLOO workspaces, and are simply collections of objects. They resemble IGLOO workspaces without clusters and inhabitants.

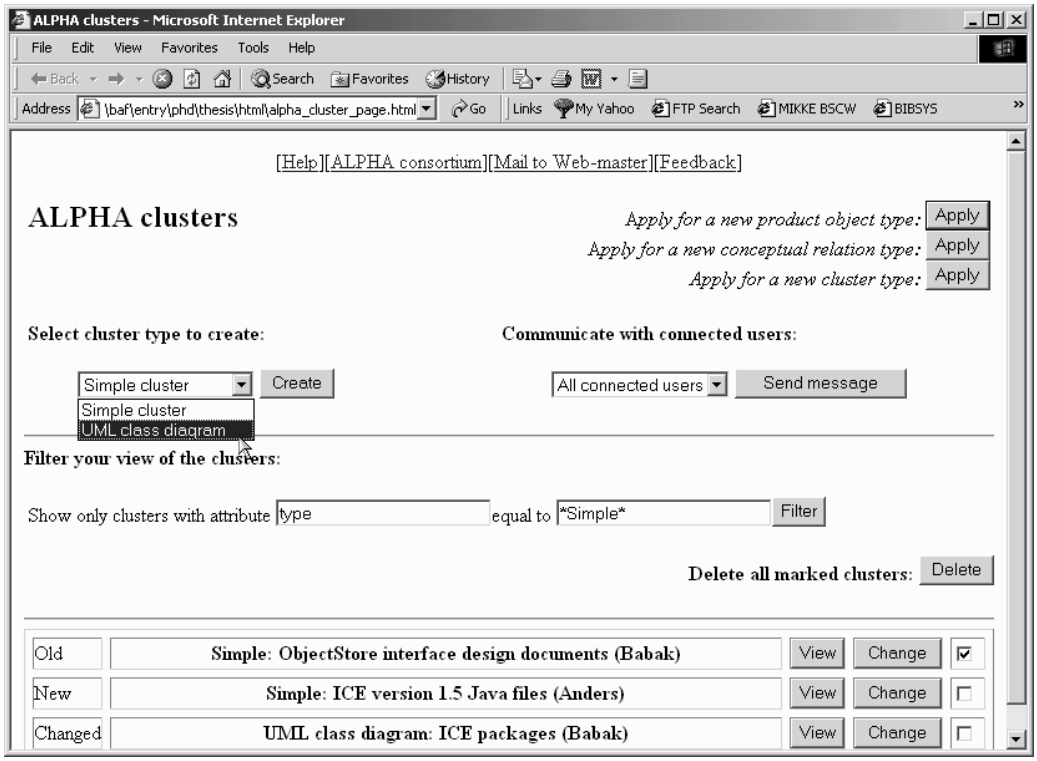


Figure 10.7: A Web-based IGLOO client for managing clusters in ALPHA.

Layer. ICE is written in Perl programming language, and cannot use the provided client extension of Cluster Layer (these extensions are currently written in Java). Therefore, the communication between ICE and Cluster Layer has to be done through Cluster Layer's network protocol. After ICE is integrated into IGLOO, most of its reported limitations (see Chapter 1) will be eliminated. In particular, the workspaces in ICE will no longer be isolated from each other because they now contain objects from a global shared product space.

The Java-based clients

Two groups of Java-based clients will be developed for ALPHA. A set of event monitors will be used to allow the developers to monitor clusters or shared workspaces. A typical monitor will stay in the background while the user is working with other things, and will generate sound signals or similar effects when something is changed in a cluster or a workspace. This kind of monitors were demonstrated in Chapter 5 as part of MultiCASE (see Figure 5.3 on page 111 and Christensen and Karlsen 1999).

The second kind of client are shared workspace clients similar to MultiCASE. These clients allow the developers to create and manage workspaces, to create clusters within their workspaces, to meet other developers and to cooperate with them, to create informal objects, etc. MultiCASE is

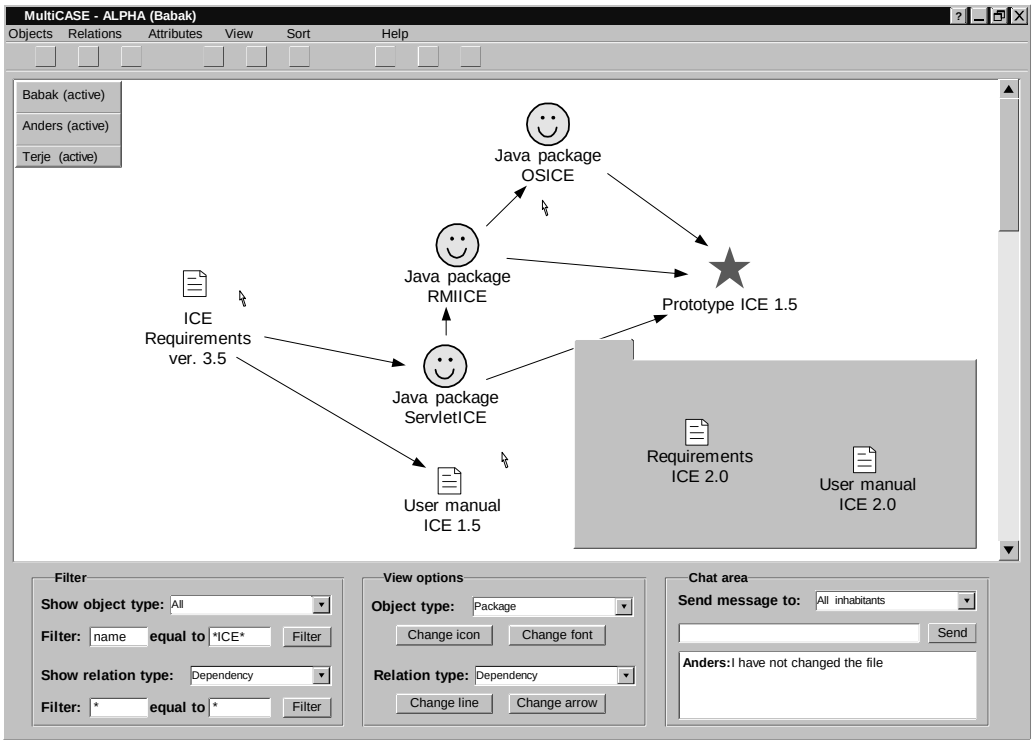


Figure 10.8: A shared workspace manager with functionality for creating clusters.

one such client. Figure 10.8 shows a modified version of MultiCASE. This is a client that is based on the basic cluster and workspace vocabularies defined in Section 10.3.1. The client is mainly a cluster editor, but support also simple shared workspace functionality. The large white area is the surface of a two-dimensional basic cluster. The objects and relations in this white area are cluster objects and cluster relations. The cluster shows a subset of the underlying shared product space. The location of the objects, and the graphical representation of the objects and relations are all shallow properties of the cluster and are visible only to the users of this cluster. The gray area on the cluster area is the informal area of the shared workspace. All the objects in this area are informal and are not part of the underlying shared product space. The informal area can be moved around and hidden.

The client provides simple participant awareness by showing a list of current inhabitants (the gray area with a list of names to the upper-left corner of the cluster area) and multiple telepointers showing where each inhabitant points at. The list of inhabitants shows whether each of the inhabitants has been active in the last period of time. The distinctive color of each inhabitant's telepointer allows the users to distinguish each inhabitant's telepointer. In addition, a chat tool supports text-based communication among the inhabitants.

The lower-left part of the window provides a set of services for modifying the cluster's visual appearance. The user can filter the objects and relations that are shown, change the graphical icon

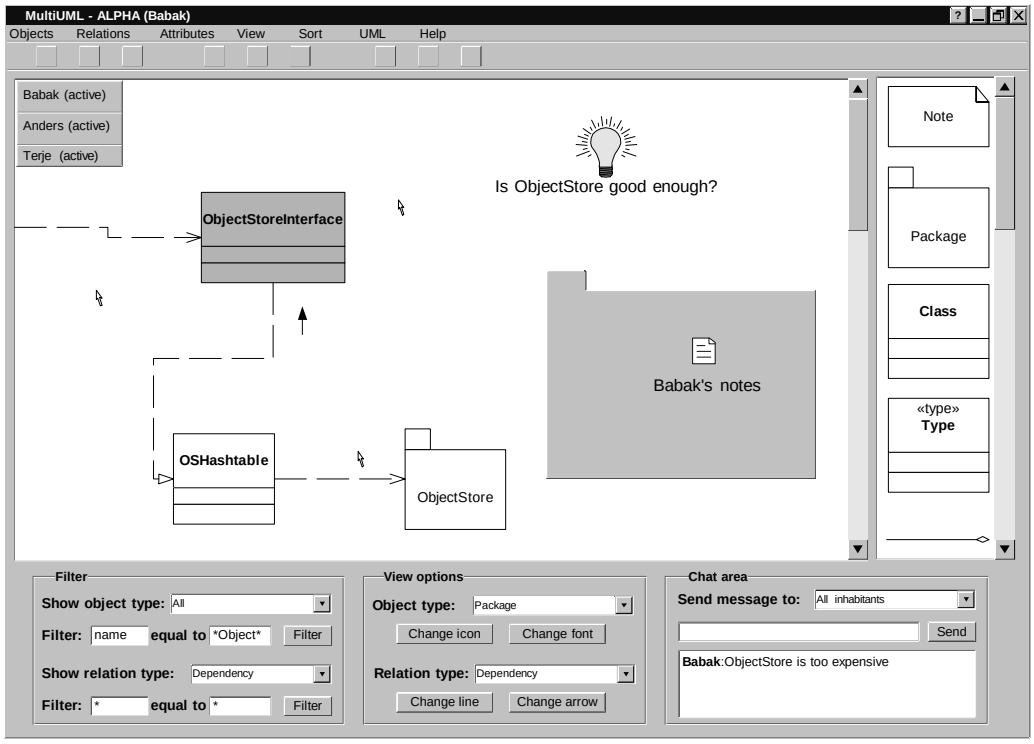


Figure 10.9: A simple UML editor that makes use of the services of Cluster Layer.

used for visualizing each cluster object type, or change the appearance of a specific cluster relation type.

Mediated awareness events, which were shown in MultiCASE in form of simple text messages in an awareness window, are integrated into the graphical representation of cluster objects in the client in Figure 10.8. Once mediated awareness events related to a specific cluster object arrive at the cluster, the specific cluster object starts blinking. In this way, the inhabitants will know where within the cluster the particular change to the product can be connected to. In this way, while within a shared workspace the inhabitants are also in the context of the larger product, and are aware of the changes that other developers do to it.

A second shared workspace client is shown in Figure 10.9. This client is used for creating UML class diagrams within a shared workspace. The functionality of the client is identical to the previous one, except that the cluster that is created by the inhabitants is a UML class diagram instead of a basic cluster.

10.4 Evaluation

In the preceding sections we have seen an example of deploying IGLOO framework in a distributed product development project. The deployment was incremental. The project started with an IGLOO network based on Product Layer only, and upgraded the network to include both Cluster Layer and Workspace Layer. An overview of the resulting IGLOO network is shown in Figure 10.10. In this figure, the gray parts are the clients. Some of the clients had to be developed specifically for ALPHA, while others were generic enough to be used by any similar project. The white part in Figure 10.10 shows the generic implementations of IGLOO framework. A large part of the functionality of the clients, i.e. all the functionality that is related to product-based shared interaction, is provided by these generic implementations. These generic implementations have provided a shared product space that is adaptable to the changing needs of ALPHA, and that can be accessed through virtually any type of interface. They have also provided ALPHA with shared customizable views into the shared product space, and with functionality for building shared workspaces.

In this section we will first see how the requirements of Chapter 2 (see Table 2.5 on page 43) are met by ALPHA's IGLOO network. We will then compare IGLOO framework to the tools and environments that were included in our state-of-the-art survey in Chapter 3. Finally we will give some estimates of the costs related to the deployment of IGLOO framework in a project.

10.4.1 Meeting the requirements

The example has revealed a number of the properties of IGLOO framework. What has been central is the creation of a shared product space (REQ.1 of Chapter 2, see Table 10.6). Cooperation in ALPHA has been based on this space since the start of the deployment process. The same shared product space and its contents were used throughout ALPHA, and the contents were gradually refined and detailed as more information about the product became available (REQ.5). Also, we have seen that new product object and conceptual relation types have been added to the vocabularies of the IGLOO network in different points of time during the process (REQ.3, REQ.4). The shared product space has been available through a Web-based and a Java-based interface. Due to IGLOO's well-defined network protocols, any other type of client could have been added. In particular, many conventional CASE and SE tools have predefined interfaces for interacting with external tools. These interfaces can be explored further in order to integrated these tools into the IGLOO network (REQ.15). In addition, the shared product space has provided a flexible interface where all the developers can add objects and relations and modify the space using the developed clients (REQ.2). Continuous awareness of modifications to the product has been delivered to the developers through various event monitors (REQ.7). The fact that these event monitors can continuously monitor user-defined areas of the shared product space also allows for opportunistic communication. This is because the developers are all the time exposed to communication opportunities by being available and being aware of these opportunities (REQ.17).

Step 2 of the deployment process introduced Cluster Layer and Workspace Layer into the existing IGLOO network. These layers and the associated clients allow the developers to create centers of interaction with the same ease that graphical diagrams are created and modified in conventional graphical CASE editors (REQ.10, REQ.11). These centers of interaction allow several developers to meet and perform different tasks. The developers can change the contents of the centers in an ad hoc manner by adding or removing product objects and relations (REQ.11). Inter-

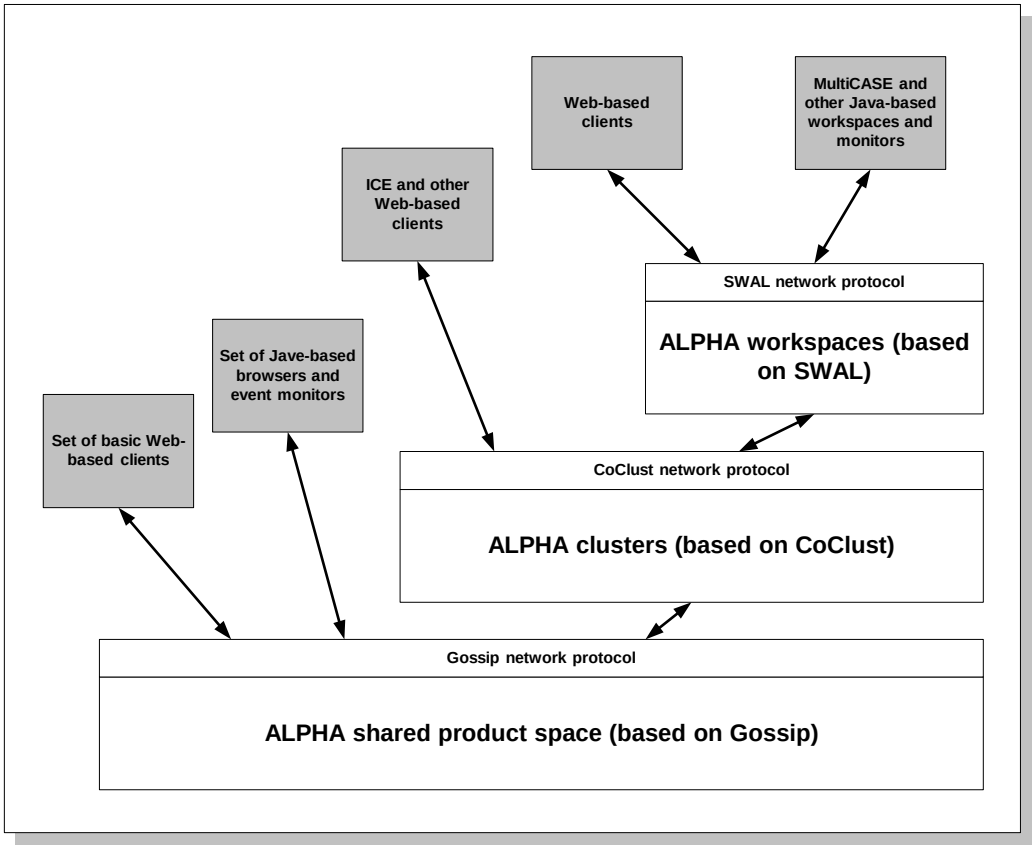


Figure 10.10: The resulting IGLOO network as deployed for ALPHA.

action within a center of interaction is supported to some degree by using chat tools, telepointers, and dynamic inhabitant information (REQ.13). More advanced cooperation tools within a shared workspace (such as video and audio communication) can be provided by more enhanced clients.

Customization of the contents through clusters and workspaces (REQ.14) and tailored functionality by choosing which clients to use (REQ.16) are central properties of the resulting IGLOO network. Another important property of IGLOO framework is that the different clients (regardless of which service layer they communicate with) can be used by different developers simultaneously for supporting their cooperation. A summary of the requirements that are met by ALPHA's IGLOO network is shown in Table 10.6.

Table 10.6: Meeting ALPHA's requirements by an IGLOO network.

Req. ID	Requirement description	How the requirement is met
REQ.1	**Shared product space**	High support. Central to IGLOO. All interaction with the product happens within the shared product space and is visible to all the developers.
REQ.2	**Flexible access to the product**	High support. The product can be changed easily by all the developers. The structure of the space can be changed dynamically. New object and relation types can be added dynamically.
REQ.3	**Unrestricted product object types**	High support. There is no restrictions on what objects are stored. The only restriction is imposed by the clients. Generic clients provide full freedom for having any product object types.
REQ.4	**Unrestricted relation types**	High support. The same as above.
REQ.5	**Incremental product refinement**	High support. New object and relation types can be added. New details to the existing objects and relations can be added. Objects and relations can be annotated.
REQ.6	**Support for boundary objects**	High support. An object can be viewed in many ways. Each community of practice can add their own attributes. Clusters provide additional local customization. Workspaces provide additional level of local customization for informal cooperation. At the same time the underlying product and its structure is shared among all the communities of practice.
REQ.7	**Active delivery of information**	High support. Basic awareness mechanisms are integrated into the resulting IGLOO network and can be utilized by all the clients. Active delivery in form of notifications is the basis for the IGLOO network.
REQ.8	**User-defined information delivery**	Medium support. The mechanisms are provided in form of awareness subscription services of Product Layer. The specific client has to utilize these services in order to give the user full control over his information needs.
REQ.9	**Representation of developers**	Medium support. The basic mechanism for connecting all the events to individual users. Again the specific client is responsible to use this basic information in order to provide a richer representation of the users.
REQ.10	**Centers of interaction**	High support. Clusters and shared workspaces are shared views. The basic mechanisms of sharing are provided by the framework. The clients do not need to put too much effort on implementing shared views.

Continued on next page

Continued from previous page

Req. ID	Requirement description	How the requirement is met
REQ.11	**Emergent creation of centers of interaction**	Medium support. This is highly determined by the clients. The services of Cluster Layer are flexible enough but have to be utilized by a client that is also flexible enough.
REQ.12	**Emergent boundaries for centers of interaction**	High support. The awareness mediation property of the shared product space eliminates rigid boundaries and allows the boundaries of centers of interaction change dynamically.
REQ.13	**Dynamic and rich interaction in centers of interaction**	Medium support. This is highly dependent on the kind of client. IGLOO does not provide services for rich communication (such as audio or video) or other kind of specific interaction techniques such as direct manipulation. The clients have to implement these mechanisms.
REQ.14	**Local customization of contents**	High support. Customization of contents can happen in many levels and freely.
REQ.15	**Multiple user interfaces**	High support. The well-defined network protocols allow any kind of interface to be connected to an IGLOO network as long as they adhere to the service definitions.
REQ.16	**Tailored functionality**	High support. The services are defined in a modular way, and not all the services need to be used by all the clients. In addition, the three layers of the framework provide another tailoring possibility by using different combination of layers.
REQ.17	**Support for opportunistic communication**	High support. Continuous awareness of changes to the shared product space, and the fact that the changes are connected to individual users. In addition, each layer has its own community services that allows for cheap and spontaneous communication among the users.

10.4.2 A comparison to other systems

Compared to the other systems reviewed in Chapter 3, IGLOO framework offers a number of improvements:

- The services of both product development tools and shared workspace applications are integrated by IGLOO framework within the same IGLOO network. The developers have access to both a shared product space (similar to central repositories in ClearCase, TDE and MetaEdit+) and centers of interaction (similar to shared workspaces in BSCW, CBE, TeamWave and Orbit).

- The shared product space provides higher support for shared interaction than what is provided by the repositories in ClearCase, TDE and MetaEdit+. All changes to the contents of the space are visible to others through active delivery of awareness information. In addition, awareness information is integrated with participant awareness in that all product

awareness is connected to individual developers. This opens for opportunistic cooperation among geographically distributed developers.

- The shared workspaces in an IGLOO network are an integral part of the underlying shared product space. In this way, they are connected to each other through the relations that exist among the different parts of the product. This is a strong improvement of all the reviewed shared workspace applications, where the single shared workspaces are almost totally disconnected from each other.
- Interaction with an IGLOO network can happen through different interfaces. All the clients, no matter which interface they use to interact with the IGLOO network, can communicate with each other. This is an improvement to the other tools we have reviewed in that an IGLOO network can be integrated into the desktop of a developer in various ways, ideally through the tools already used by the developer.
- The vocabularies used in an IGLOO network are defined according to the real needs of the supported project. In addition, the same organizational vocabulary can be the basis for a number of more specialized local vocabularies. Customization of the product to support different communities of practice is a distinct feature of IGLOO framework compared to the other tools we have seen.

On the other hand, ALPHA's IGLOO network does not provide a number of more specialized services provided by the other tools. For instant, the IGLOO network does not provide any of the advanced repository services provide by ClearCase and MetaEdit+, e.g. version control or access control. In addition, compared to advanced shared workspace applications such as TeamWave, ALPHA's IGLOO network has a more limited support for cooperation inside a shared workspace. Although more support for cooperation can be added inside an IGLOO shared workspace, such as more advanced awareness widgets similar to those used by TeamWave, this support will require more advanced IGLOO clients.

10.4.3 The cost of deploying IGLOO

The cost of having the level of flexibility provided by IGLOO framework is that one has to implement a deployment process for each project that will use IGLOO framework. There are different costs related to the different activities in the deployment process, i.e. defining vocabularies, defining awareness policies, and implementing specialized clients. In this section we will give some estimates of what these costs will be. The estimates are based on our own experience developing the different parts of IGLOO framework.

Deploying IGLOO for the first time will most probably require a considerable amount of work related to developing clients. The Web-based clients are least costly. Our experience shows that one undergraduate student would spend 4 months full time for developing, testing, and debugging all the Web-based clients discussed in this chapter[5]. These clients, once implemented, can be made part of the core IGLOO framework. This is possible because most of these clients are generic clients and can handle any organizational or local vocabularies.

[5]This estimate is based on our development of ICE. The basic functionality of ICE was implemented by two undergraduate students, each spending 4 months of work. However, the task was not well-defined at that time, so a lot of time was spent in defining the functionality of ICE. In addition, a large part of the functionality of ICE is now implemented by the generic implementations of IGLOO framework.

The Java-based clients are normally more costly, but can be developed in form of generic clients and be reused in several projects. Our experience shows that the set of 5 Java-based clients discussed in this chapter (i.e. the generic product space manager shown in Figure 10.6, the modified MultiCASE shown in Figure 10.8, MultiUML shown in Figure 10.9, a generic product monitor, and a generic cluster monitor) can be developed within six work-months by a Java programmer[6]. Again all these clients are generic. In fact, MultiUML's functionality is largely supported by the enhanced version of MultiCASE because the shapes of the cluster objects in MultiCASE can be UML-specific shapes.

However, it is obvious that more advanced tools will be needed for any serious modeling work to be supported by IGLOO framework. At this end one should focus on integrating existing clients as much as possible. A set of possible tools to integrate would be Rational Rose UML-based CASE tools, Visual Cafe Java programming environment, Emacs, and Microsoft Word. All these tools have functionality that allows them to invoke external programs in an easy and user-friendly way[7].

Subsequent deployment processes can make use of generic clients developed during the first deployment, and can add their own clients. In this way, the mandatory cost of developing clients will be much lower than first time deployment. What will be a more or less constant cost is the cost of developing and refining the vocabularies. Here, standard formal vocabularies (such as a UML vocabulary) can be reused because they do not change much from one project to another. However, informal vocabularies have to be defined and refined for each project. The cost of defining informal vocabularies depends on how complex they are. The cost of refining vocabularies involves also the possible cost of refining the clients. If the clients are generic enough they will not need modifications even when the vocabularies are changed[8]

There are two relate factors that are important to consider when discussing the cost of deploying IGLOO framework. First, the issue of who will do which part of the deployment is a determinant factor for the distribution of the costs. The generic clients we have discussed in this chapter can in fact be developed as a part of the core IGLOO framework. Together with a set of tools and methods for managing the evolution of the vocabularies, these clients can reduce the costs for the end-users of an IGLOO network. This is because a larger part of the IGLOO network can be delivered in form of a software product, possibly implemented by specialist "IGLOO developers." Second, the desired level of flexibility can determine the costs. In general, more flexibility in defining the vocabularies will increase the cost of the deployment process both because the clients have to be made equally flexible, and because additional steps have to be taken in order to control the evolution of the vocabularies.

In short, first time deployment of IGLOO will have a rather large cost related to the development of clients. Subsequent deployment processes will have a much reduced cost related to client development, but may have a constant cost related to defining and refining informal vocabularies.

[6]This estimate is based on our experience implementing MultiCASE. MultiCASE's user interface was developed by one student (new to Java programming) within three months. This was done at the time when the generic implementations of IGLOO were not defined, so a lot of time was actually spent on negotiating an interface towards the lower layers in MultiCASE.

[7]Gossip currently supports a "plug-in" interface that allows the addition of new tools and interfaces. Garli and Lund (2000) describe how an ICQ plug-in is developed.

[8]One possible way of making clients generic enough for dynamic vocabularies is to base them on object and relation templates similar to Information Lens (Malone et al. 1989).

10.5 Summary

In this chapter we have described a detailed example of deploying IGLOO framework in ALPHA. We have seen that the resulting IGLOO network supports most of the requirements we posed to product development environments that will be used for supporting cooperation. The resulting IGLOO network also offers a number of additional features compared to the other tools we have reviewed in Chapter 3.

Whether the resulting IGLOO network will support ALPHA in a satisfactory way is an open question and will need empirical evidence. We believe that the IGLOO network described in this chapter will indeed solve some of the major problems that ALPHA was originally facing. The IGLOO network provides a flexible interface to a shared product space through a number of interfaces. Adding and modifying the product will be much easier than what was possible using traditional Web servers with central control. This flexibility will hopefully encourage the developers to share more product objects, as they did using ICE. In addition, the IGLOO network with associated event monitors will place the developers in a shared context where they will continuously be exposed to awareness information about each other's interactions with the product. This can enhance the coordinative effect of the product and result in improved learning processes on the long run.

The real value of an IGLOO network will be revealed using a suitable set of clients. Clients that are integrated into the daily activities of the developers will increase the possibility of a useful deployment of IGLOO framework. Web-based clients, light-weight event monitors, desktop tools integrated into IGLOO, and advanced shared workspaces are crucial for successful usage of IGLOO.

Chapter 11

Conclusions and Future Research Directions

11.1 Introduction

This chapter concludes the thesis and suggests directions for future research. We start by reviewing the research questions for the thesis. We will describe how each question is answered in the course of the reported research. We then review the major contributions of the thesis. A number of proposals for future research are given at the end.

11.2 Answering the Research Questions

The main research question as stated in Chapter 1:

> *How can we support, through computer-based tools and environments, cooperation among developers in geographically distributed product development projects?*

has been answered through an analysis of the problem, a conceptualization of the needs, several cycles of prototyping and evaluation, and a proposal for a computerized cooperation support framework. We have based our analysis of the problem on an empirical investigation of ALPHA (a real-world distributed product development project from which we had first hand experience) and several other empirical studies of product development projects published in the literature. Moreover, several cycles of prototyping and evaluation have been performed during this research. Two large prototyping efforts have resulted in a Web-based cooperation support system called ICE, and a Java-based cooperative product development tool called MultiCASE. Each of these prototypes have been used and evaluated. ICE has been used by the members of ALPHA, while MultiCASE has been used for a short time internally in our research group. In addition a number of smaller prototypes, both Web-based and Java-based, have been developed. Each cycle of prototyping and

evaluation has been analyzed and used for further prototyping and evaluation. IGLOO framework for product-based shared interaction, as presented in this thesis, is the final result of this bootstrapping process.

The related research questions as stated in Chapter 1 have been answered in the following way:

> *RQ1: What is the nature of cooperation in product development groups, i.e. what is the meaning of "cooperative product development"?*

This thesis has provided an understanding of the meaning of "cooperative product development" through an analysis of empirical investigations of product development teams published in the literature. Moreover, an investigation of existing computer support for product development has been done in order to gain an understanding of what aspects of cooperative product development are essential, and what aspects can be supported by available technological means. These analyses have revealed the importance of the product in supporting knowledge creation, cooperative learning, and coordination.

> *RQ2: What is the effect of geographical distribution on cooperative product development?*

An analysis of a real-world distributed product development project, i.e. ALPHA, is presented. This analysis is used to point out the specific problems that arise as a result of geographical distribution. The analysis has shown that the product plays a central role as a resource for cooperation in distributed projects. At the same time, it is shown that geographical distribution makes it difficult to utilize the product as a resource for cooperation. The majority of the problems that ALPHA was facing were related to the fact that the members in ALPHA were not able to use the product for externalizing their knowledge, for resolving misunderstandings, and for coordination their daily activities.

> *RQ3: What kind of computer-based tools and environments are needed for supporting cooperative product development?*

Based on the results from the empirical investigations related to **RQ1** and **RQ2**, and from our own prototyping activities, a set of requirements for computer-based support tools is proposed. A survey of relevant computer-based systems is presented. Based on the requirements and the state-of-the-art survey, a framework and associated tools are proposed for supporting shared interaction in distributed product development projects.

11.3 Major Contributions

The contributions of this thesis are the following:

- *An analysis of cooperative product development in geographically distributed projects*. The analysis has revealed the importance of the product being developed as a resource for cooperation. The results of this analysis confirm the importance of shared artifacts as cooperation support mechanisms. Although prior research has focused on the small scale use of shared artifacts (often within small shared workspaces) our results show the usefulness of large and composite artifacts (i.e. software products) for supporting cooperation in large groups.

- *A model of product-based shared interaction.* This model is a generalization of the empirical evidence. The model extends the notion of shared artifact to cover large composite artifacts. The model consists of a shared product space. This space is used for supporting shared interaction among developers working with a large product. The interaction with the shared product space happens through a number of centers of interaction, where groups of developers can meet and perform specific tasks.

- *A framework for product-based shared interaction*. The model of product-based shared interaction is further formalized into a framework for supporting shared interaction. A detailed description of the framework in form of three service layers is provided.

- *Generic implementations of the framework*. The defined framework is partly implemented in form of three interconnected network servers. The implementations together realize a network-based platform for product-based shared interaction. This platform can be used for creating a variety of product development environments with cooperation support functionality.

- *A deployment process for the framework*. A companion deployment process is outlined for the resulting framework. The process defines the different activities that are necessary for tailoring the proposed framework to a specific project.

- *Example clients*. A number of applications have been developed to demonstrate the functionality of the resulting framework. These applications are implemented in form of special client applications for the cooperation framework and its generic implementations.

11.4 Directions for Future Research

We suggest four interconnected directions for future research activities. First, a complete set of client applications has to be implemented to support the creation of an IGLOO network. Second, the generic implementations have to be improved. Third, resulting IGLOO networks have to be tested empirically in real-world or student projects. A fourth direction is related to the integration of the proposed framework with existing CASE tools.

11.4.1 Implementing a suite of IGLOO clients

IGLOO framework is designed and implemented in form of a set of services. For utilizing the functionality of IGLOO framework a number of IGLOO clients are needed. The clients that are developed during this research are not completely integrated with each other. The IGLOO network described in Chapter 10 shows a number of clients that should be implemented as part of future research. Suggestions for activities in this direction are:

- The implementation should be focused on developing generic clients. It is important that the developed clients can demonstrate the generality of the underlying IGLOO framework. Examples here are generic shared product space managers, cluster managers, and workspace managers.

- Clients supporting multiple interface (e.g. Java, WWW) should be implemented. This is important in order to test the interoperability of IGLOO framework with respect to different computing environments. Having multiple interface clients also facilitates the integration of IGLOO framework into the desktop of the developers using it.

- Clients should be developed for testing different mixtures of formal and informal development methods. In this way, IGLOO framework can be used for involving domain experts with limited knowledge of formal product development methods.

- Specialized clients for managing the structure of the product should be developed. Maintaining the structure of a large product can become a tedious task. Specialized clients can help automate parts of this process. Examples here are clients for visualizing the product in different forms, for searching and replacing objects, relations and attributes in the shared product space, etc.

11.4.2 Improving the generic implementations

The generic implementations of IGLOO framework, i.e. the layer servers that implement each service layer, can be improved in a number of ways. Suggestions for improvements or enhancements are:

- Support for repository-like services can be added to all the three layer servers, in particular to Product Layer and Cluster Layer. One such enhancement can be the addition of version control mechanisms together with corresponding services. Another enhancement can be to add access control mechanisms. Access control mechanisms can be valuable if they can be applied at the same granularity level as the existing product objects and conceptual relations. For instance, access restrictions can be put on some attributes of a product object and not the whole object.

- Support for name spaces and hierarchical structures can be added to Product Layer in order to control the visibility of the product. Combining name spaces with the existing awareness support mechanisms can result in enhanced awareness support and the production of more meaningful awareness information.

- Client extensions for each layer server have to be made more user-friendly, and client extensions for multiple programming languages have to be developed. Client extensions should be promoted as the main mechanism of interaction with an IGLOO network.

- Support for access logs can be implemented through some form for event logging. This can be useful for registering accesses to the shared product space or the clusters and shared workspaces. Event logging functionality can either be implemented in form of specialized clients, or be integrated into the layer servers.

11.4.3 Empirical testing of example IGLOO networks

IGLOO framework as presented in this thesis is the result of empirical investigations of one real-world project (i.e. ALPHA), investigation of other theoretical and empirical studies of product

development groups, analysis of existing product development tools and environments, and evaluation of a number of prototypes that were developed and used during this research. However, an empirical evaluation of the whole framework has not been performed due to the limitations in time.

An immediate future research activity should therefore be to evaluate IGLOO framework empirically in a number of student projects. In our institute there is already an established culture for organizing student projects, and a number of subjects that are currently run are based on project work. These subjects provide an ideal setting for testing IGLOO. A minimal IGLOO network should be adapted to the settings of one of these subjects, and several empirical studies should be performed. The goal of these studies should be to clarify how IGLOO framework will be integrated into the cooperative activities of the students, and how it will improve cooperation in these student projects. By giving credit to the students for their usage one can encourage valuable feedback from such studies.

A more ambitious goal in this direction would of course be to test the framework in a project similar to ALPHA, e.g. a geographically distributed product development project involving participants with diverse backgrounds and a high level of local autonomy.

11.4.4 Integration with existing CASE tools and methods

IGLOO framework has been developed with CASE integration in mind. The focus on the product as a resource for cooperation and the existence of a shared product space is already in line with how the majority of CASE tools are designed. From this perspective, IGLOO framework and its generic implementations are much more suited for being integrated in CASE environment than the other cooperation technologies we have investigated. Further activities in this direction can be envisioned in form of two sub-activities.

First, integration into specific CASE tools should be investigated in more details. The starting point for this is the already existing network protocols defined by all the three service layers of IGLOO framework. More work here is needed to take into account existing standards such as XML (Standard Markup Language) and specific CASE standards such as CDIF (CASE data interchange format). Second, integration with existing CASE methods should be investigated. The existing deployment process should be further developed in order to support conventional methods such as structured analysis and object-oriented analysis and design.

Appendix A

ICE: An Object–oriented Toolkit for Tailoring Collaborative Web Applications[1]

A.1 Introduction

It is widely recognized that user involvement is an important aspect of IS development (Keil and Carmel 1995, Greenbaum and Kyng 1991, Kendall and Kendall 1988). It is also a well–known fact that there are many different user groups in an IS organization, which makes the task of involving users in IS development intricate. The problem is partly due to the fact that the various involved user groups have conflicting mental models, expectations, and experiences (Wastell 1993). The process of IS development is to a large extent that of resolving these conflicts and reaching to an agreement about what the IS will be. The larger the number and the variety of the users affected by the IS, the more difficult it is to reach to this agreement.

User involvement can be realized in different ways in an IS organization. In its simplest form a few user groups, such as the management and the development team, will take most of the decisions related to IS development. This is however not realistic in many knowledge–intensive organizations, where the expertise is distributed among all the workers (Drucker 1988). Another approach is to consult the different user groups or their representatives, and to take their points of view into consideration when making IS–related decisions. This consultation is normally realized by interviews and surveys of the users' desires (Kendall and Kendall 1988). A more active user involvement is when all affected user groups, or their representatives, participate in the IS development. Prototypes are frequently used in the latter case to provide early feedback to the design of the IS. Prototypes are often in form of user interface mock–ups. They are used because they are easy to understand for most of the user groups, and because they uncover requirements that

[1] The material in this chapter was presented at IFIP WG 8.1 Working Conference on Information Systems in the WWW Environment, Beijing, China, July 1998. The conference proceedings are published in (Rolland, Chen and Fang 1998).

may not surface outside their related work context (Ehn and Kyng 1991). A step further is radical tailorability, where the users are in charge of building their own prototypes using simple building blocks (Malone et al. 1995).

We see a potential in using the Web as an environment for promoting user involvement. The Web as a multi–user interface to a large amount of information has become popular in the last years. This popularity is due to the underlying prerequisites of simplicity and accessibility that have been with us since the first days of the Web (Berners-Lee et al. 1994). Lately, the Web has also been used to develop more interactive applications with information processing capabilities. In this paper we show how the Web can also be used for encouraging user involvement. This is because of two reasons. First, the Web provides a user interface that is comprehendible for most of the user groups involved in a typical IS development process. Second, the underlying communication infrastructure is so well–established that collaboration among the various user groups in the development process becomes feasible.

In this paper we describe a system for supporting collaborative IS development using tailorability. The system is called ICE (Internet Collaboration Environment) and is based on Web technology. ICE consists of a set of building blocks that can be manipulated directly by the users with little or no programming knowledge. These building blocks are used for tailoring Web applications with different functionalities. ICE is in this way intended as a simple application development environment. The applications that are tailored using ICE are distributed Web–based groupware applications (Ellis et al. 1991) with focus on information sharing.

In addition to being used for building Web applications, ICE also supports the development process by providing support for collaboration among different user groups and capturing the design rationale. A group of users can use ICE to tailor their initial prototypes collaboratively. These prototypes can then be used as such, as input to the development process, or they can be refined for further use. Usability of prototypes built using ICE is dependent on the background knowledge of the users involved in the development process and the complexity of the intended application.

The rest of this paper is organized as follows. In section A.2 we will introduce the main ICE building blocks used for tailoring Web applications. In section A.3 we discuss how these building blocks can be used for both building collaborative Web applications and supporting the development process. An example is provided in section A.3.3. Section A.4 gives an overview of the ICE architecture. In section A.5 we present some related research. Our conclusions and future work are presented in section A.6.

A.2 ICE building blocks

The building blocks are the main elements in ICE. All ICE applications are built as a composition of these elements. During the design of the building blocks we have taken into consideration two important criteria. First, the building blocks should be user–friendly in such a way that users without much programming knowledge will be able to use them for tailoring their own applications. Second, the building blocks should have enough functionality so that their composition will result in the desired applications. The first criteria is met by the fact that the building blocks are Web–enabled and provide an easy user interface to the users. The second criteria is met by taking into consideration other experiences in the field of radical tailorability (Malone et al. 1995) and groupware modeling (Ellis and Wainer 1994b). The building blocks are *Information Objects*, *Col-*

laboration Objects, and *User Interface Objects*. Information objects are used to build data models for the applications. Collaboration objects provide basic support for collaboration and coordination. User interface objects provide user interaction mechanisms to other objects. Information and collaboration objects are accessed through the user interface objects, which are mainly implemented for Web–based access. In this section we will take a closer look at these building blocks. For more details on the building blocks see (Farshchian and Divitini 1997).

A.2.1 Information objects

It is highly recognized that information understandability is important for collaborative work settings where the members of the group are distributed in space or time (Schmidt and Bannon 1992). Understanding information is not only dependent on the information itself, but also on the context within which that information was created, such as the problem solving strategies of the creator, the decisions made while creating the information, and the identity of the creator. ICE attempts to support information understandability for the users of a distributed application. Our approach to increase understandability is based on providing support for what Bannon and Bødker (1997) call *closure* and *malleability* of information spaces. Closure refers to the way information is packaged for an audience. A closed piece of information provides, and even enforces, an interpretation to its user. Malleability, on the other hand, is concerned with changes, different understandings, argumentation, and knowledge creation. The Web with its support for packaging of information into "files" does not support the malleability of the information. In addition, even though the information on the Web is packaged to some degree, this packaging is often not suited for propagating the intended meaning of the information (Dix 1997).

The main building block for information sharing and supporting understandability in ICE is the *Information Object*. An information object is used to store information in an ICE application, and consists of an *Information Content File* (ICF) and its attached *meta–data*. ICF is the actual information content. It is uploaded by the user as a file and is stored in ICE in its native file format. The meta–data is added to the ICF by the user to provide ICE with extra information about the ICF. This extra information includes bibliographical data, keywords, different summaries of the content, access rights, and type–specific data. Each information object can also be subject to cooperation. For supporting cooperation connected to an object two mailing lists are attached to the object. These mailing lists are explained in detail in section A.2.2.

Information objects can be presented to an audience in different ways. The objects use their meta–data to provide several possible *views* of themselves, each view supporting some specific understanding for an audience. This property of "packaging" information for different audience corresponds to the closure property noticed by Bannon and Bødker (ibid.).

Information objects are also open for changes in the sense that once they are created and inserted into an ICE application, they can be changed by a group of users during a collaborative process. An information object can be accessed by authorized users who want to consume, edit, comment, or discuss the object. The contents of an information object can be accessed (and consumed) through its user interface object (see section A.2.3 for a description of user interface objects). Editing of the content is done by the user's desktop applications, such as a diagram editor or a word processor. Commenting is done by creating new fields in the meta–data schema for the object type. Discussions are realized by the mailing lists connected to each object. Figure A.1 shows an information object in use.

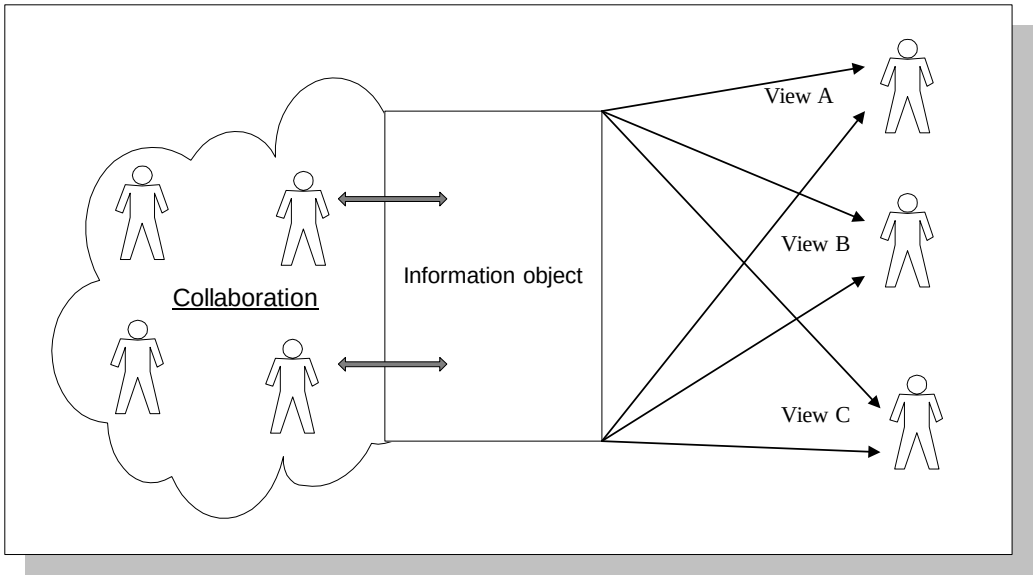


Figure A.1: Information objects in use. Each information object provides a user interface to a group of users (to the left). This interface supports collaboration among the group members with the aim of changing the information object. Collaboration here is supported using email lists. The closed part of the object (to the right) provides different views of the updated information content in order to support different understandings.

The basic information object can be specialized in an object–oriented fashion. This specialization will result in a set of information object types that will constitute the data model of the applications built using ICE.

A.2.2 Collaboration objects

For complicated work settings, involving several people and a large amount of information, there is a need for active collaboration support. We borrow a definition of active support from Bogia et al. (1996): "*active [support] means that the environment can have some knowledge of ongoing collaboration and can use that knowledge to assist users in various ways.*" Though the Web can handle information sharing in an acceptable way, it does not actively support collaboration (Dix 1997). For this purpose, we provide *Collaboration Objects* as building blocks for incorporating collaboration mechanisms in applications built using ICE. These objects are *users*, *groups*, *object awareness lists*, *object discussion lists*, and *shared workspaces*.

The basic collaboration objects are user and group objects. Each user of the system is registered with information about herself including biographical information, user preferences, group membership, and other temporal information such as where the user is at each time and when she was logged in last time. The group object is used in combination with the user object to connect several users in the context of a task or a practice.

Communication among the users is supported by object discussion lists. Each information object has a discussion list attached to it. Messages to this discussion list are sent by the users of the object through their email client, and are received by other users subscribing to the list in form of normal email messages.

Awareness is an important factor in increasing the performance of group members (Gutwin and Greenberg 1998). ICE supports awareness of changes done to information objects. Awareness in ICE is realized by attaching an awareness email list to each information object. The users of an information object can subscribe to its awareness list and receive various messages connected to that object. Messages to the awareness list are sent by the object itself. These messages are normally notifications of the changes done to the object, with information about the type of the changes, the user responsible for the changes, the time the changes were done, etc.

Another important step in providing active support for cooperation is to provide the users with a place to work. Shared workspaces are used by various groupware applications in order to provide a place for cooperation (Spellman, Moiser, Deus and Carlson 1997). In a shared workspace the users can coordinate their activities, be aware of what others are doing, and access relevant information easily. ICE shared workspace objects are used as interfaces to tasks, where the interface provides easy access to the information needed to fulfill the task, and access to the other users involved in the task. A shared workspace object can contain several user and group objects. A shared workspace provides in addition easy methods for inserting information objects and other workspaces, and methods for manipulating these objects in various ways. Workspaces can be used as building blocks in simple process models for ICE applications. However, more sophisticated coordination mechanisms for connecting shared workspaces are absent in the current version of ICE.

A.2.3 User interface objects

User interface objects are used by other ICE objects for visualization and user interaction purposes. Objects in an ICE application have one or more user interface objects. Each user interface object implements an interface towards a specific environment. In the case of the Web environment, user interface objects are responsible for generating HTML translations of other objects. These translations contain both visualizations of the object meta–data, and interaction mechanisms such as buttons and forms for changing the meta–data. User interface objects cannot exist as stand–alone objects, but they must be connected to already existing objects. In this way it is possible to have different user interfaces connected to the same ICE application.

User interface objects have one other important role, that of providing support for dialogue with the users. One main problem with Web–based application development is that the Web is stateless. This means that the Web server cannot know what the clients are doing, and therefore cannot control the dialogue with the users. ICE provides standard objects for constructing simple user interface dialogues for the applications built using ICE. These objects implement an HTML frame–based user interface where the different frames provide various standard functions to the users. This frame is shown in figure A.2. For details on dialogue handling in ICE see (Damskog 1997).

A.3 ICE functionality

In this section we will explain how ICE can be used for collaborative application development. The basic idea is that the users will use ICE building blocks in combination with concepts from their work domain to tailor the desired applications. In doing so, they will probably need to combine their expertise from different domains. The collaboration support provided in ICE will connect users with different expertise and help them build the application. ICE will also capture the design rationale and the history behind the development process. This history can be used as input to a further process of refining the application. At the end of this section we provide an example that illustrates this process.

A.3.1 Tailorability in ICE

The main user interface to ICE is towards the Web. Tailoring applications in ICE starts normally by a user creating a shared workspace. A newly created workspace is a blank Web page with a simple user interface including a frame with some default buttons for generic functions such as login, home, and search. The user then inserts some user objects into the workspace. This is done by giving these users access to the workspace. The group of users can then start building their Web–based application in this workspace. All the authorized users can insert building blocks into the workspace. The first building block they will insert is normally an information object. This object is used as a design document for the application (See section A.3.2). At the same time the users can subscribe to the discussion list of the object and use it as a communication medium.

Each application domain will have a set of specialized information object types. These object types are available to the users through a list box in every shared workspace. The users can create and manipulate instances of these object types using HTML–based forms. Information objects can also be put together using inter–object communication to build composite information objects (See section A.4.2). One such object type implemented in the current version of ICE is the COURSE object that consists of several MODULE objects, several EXERCISE objects, and one EXAM object (Sivesind and Grimstad 1997). Figure A.2 shows a shared workspace with its Web–based user interface.

Collaboration and information objects can be used in this way to build larger distributed applications. Workspaces can be used to build simple process models in form of (currently) disconnected workspaces, and information objects can be used to build the data model of the applications.

A.3.2 Support for development process

ICE promotes an iterative design methodology where the users and their applications are in constant contact, and where the users actually use the applications while they are building them. We can think of two modes in an ICE application; use mode and development mode. The use mode is when the users are using the application for its intended purpose. Development mode is when the users, confronted with a breakdown in their work, start to refine the application. Unlike what is common for most development environments, there is no visible distinction between these two modes in ICE. Instead, we provide support for cooperation and capturing design rationale. In this way not only the prototype itself, but also the history behind its development can be stored and analyzed in further development.

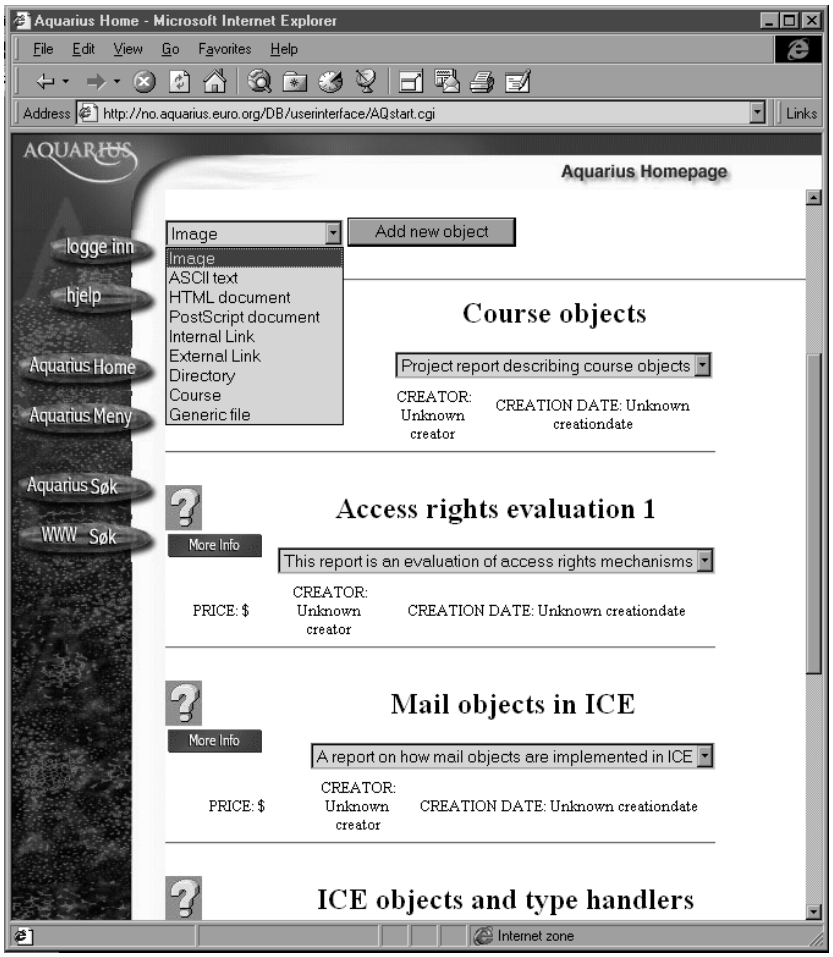


Figure A.2: A Web–based interface to a workspace. The white area in the middle is used to insert objects. The frame around the white area provides some generic functions that are available in all the workspaces. A list of available object types is accessible in every workspace.

We believe that ICE can be used to support such a process of iterative system development. Many studies point to the fact that system development is a collaborative and knowledge–intensive process, where the necessary knowledge has to be acquired from domain experts during a collaborative process (Walz et al. 1993, Marmolin, Sundblad and Pehrson 1991, Curtis et al. 1988). There are also numerous studies of the process of tailoring that point at the same direction (Trigg and Bødker 1994, Mackay 1990). ICE supports this process by providing a medium for different users to meet and cooperate, and by capturing the design rationale during this process.

The fact that ICE is based on email and WWW makes it more accessible to most of the users, which we believe is important for promoting user involvement. Cooperation and information–

sharing mechanisms in ICE can also be used to support the development process. As an example consider the object discussion lists. Several properties of these lists make them suited for the development process. First, it is an easy task to subscribe and unsubscribe from these lists. This makes it feasible for external domain experts to subscribe to a list, give their contribution, and unsubscribe. In this way the development group can have undisrupted access to experts of different kinds, provided that the experts are available by email and are willing to contribute.

Second, each object keeps an archive of the messages sent to its lists. This archive is always accessible through the object's Web interface. In this way, not only the discussions among the more permanent team members are stored, but also contributions from temporary expert members are registered and kept for later references. Overlooking past discussions in development teams is a problem that is pointed out by both Curtis et al. (ibid.), and Potts and Catledge (1996).

Third, all the messages are stored connected to their information object and the workspace the information object resides in. This provide a context for the discussions and makes it easier to understand the discussions in a later point of time. It also makes it possible to use the contributions to the discussion list in refining the information object into a design document for the application.

There are numerous other possibilities for using ICE to support the development process. But the main points are that ICE: 1) makes it easier to bring together experts from different domains, 2) provides the development process with input in form of a prototype and its design rationale, 3) imposes not a process but a set of tools for both developing applications and supporting the process of developing them.

A.3.3 An example of using ICE

We will now consider a simple scenario where two teachers develop a Web–based application for supporting the creation of new courses. The teachers start by creating a simple prototype of their course application. They start by creating a main workspace. This workspace will contain their final prototype and a record of the process of building the prototype. For recording, and later documenting the process, they create a new information object of type DOCUMENT and insert it in the main workspace. This document opens for email communication between the teachers. They continue to create new workspaces for the various tasks they believe they will perform when creating the course. For each workspace they also create a DOCUMENT that will register the discussions in that workspace. Workspaces and information objects are created by choosing the proper object from an HTML list, and filling the necessary information in an HTML form. They choose to have workspaces for creating syllabus, exercises, and exam. They decide to close the exam workspace. They do this by setting proper access rights for the exam workspace using an HTML form. They start then to create information objects in the syllabus workspace. During the process, they use the email lists in each workspace to discuss different aspect of the course application.

At one point, the teachers find out that they need some mechanism for registering and contacting the students. Since they don't know how to do this using ICE, they decide to find an expert to ask. They search among the user objects to find somebody from the ICE development team who is available at that time. They find a developer and invite her to join them in their workspace. The developer subscribes to the discussion list for the course application and starts receiving questions from the teachers. The developer suggests that they could make an information object with some information about the course, and then subscribe all the students to this object. In this way they can both provide updated course information to the students, and contact the students by sending

email messages to this object's discussion list. The teachers do as the developer suggests. When the teachers are finished with their application, they start using it for creating a new course. Using the application results in new changes that are again done by the teachers in collaboration with the developers, and possibly some students.

ICE provides in this way a true prototype of the application and the reasoning behind it. The application itself can be improved and used as the final application. The developers can also read and analyze the discussions between the two teachers, and investigate the changes done to the application during the development. One possibility for improvement that the developer in this scenario notices is the need for a COURSE object that the teachers can use to register students. The developer implements this object by specializing another more general object, and provides it for future use in the ICE environment. Following this process, the resulting application will hopefully not be a surprise to any of the users.

A.4 ICE architecture

ICE implementation details are explained in (Olsrød and Isaksen 1996). In this section we present the main features of the ICE objects and system architecture. All the building blocks in ICE are implemented as objects in an object–oriented manner. First we will describe how these objects are implemented and used as building blocks in ICE applications. Then we will explain inter–object communication that is the basic mechanism for building composite objects. At the end of this section we will explain briefly some generic features of the architecture, such as access control mechanisms and email services.

A.4.1 ICE objects

All the objects in ICE, except for the user interface objects, are implemented in a similar way, and are here called *ICE objects*. Each ICE object is stored as an *Object Description File* (ODF), together with a pointer to the location of an *Information Content File* (ICF). ODF contains all the semantic information about the ICE object, including meta–data and access rights. The ICF is a file uploaded by the user, and is stored in its native format. ICF can possibly be located outside the ICE database, but it has to be available to ICE through a standard protocol, such as HTTP.

ODF is a text file in form of an SGML/XML document type description (DTD) and provides a uniform representation of all ICE object types in the system. ODFs consist of two groups of attributes:

- Primary attributes: These are the meta–data for the ICE object type. Meta–data give additional information about the ICF stored in the database. In the current implementation meta–data consists of title, abstract, author name, location, and date of creation.

- Secondary attributes: These attributes are normally used by the system for manipulating the ICE objects. Attributes here are access right scheme, email notification and discussion subscribers, keywords, and language specifications.

ICE objects are created and manipulated directly by the users of the system. Assistance for this manipulation is given by *Object Manipulation Agents* (OMAs). Each OMA has full knowledge of

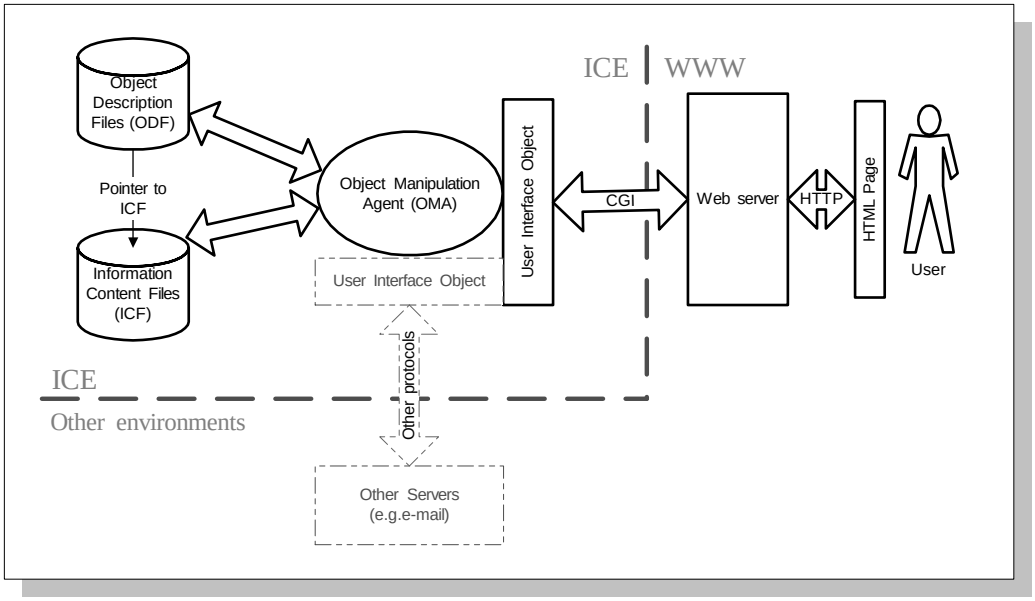


Figure A.3: Object Manipulation Agents (OMA) are used to interact with the objects in the database. User Interface Objects (UIO) are responsible for translating from network protocols to the internal language understood by the OMA. Various UIOs can be connected to an OMA to let the OMA communicate with the users via other interfaces than the Web.

an ICE object type and all the valid operations on that type. OMAs are implemented as object–oriented classes with inheritance. Different views to ICE objects are implemented as methods in OMAs. For each method in an OMA a user interface object (UIO) is implemented that handles the communication between the OMA and a specific protocol (See figure A.3). The only protocol implemented in the current version of ICE is CGI (Common Gateway Interface) for Web server communication.

HTTP requests from a Web server are sent to a proper OMA through its UIO. The job of an OMA is to receive requests from the Web server and return HTML code containing the result of the requests. An HTTP call to an OMA contains the name of an ODF, and a list of operations to be executed on the ODF and its related ICF. By delegating the interpretation and HTML generation from the Web server to UIOs and DMAs we open for flexibility and the possibility of implementing unlimited object types in the system without recompiling the server program for each new ICE object type.

Each view to an ICE object is implemented as a method. An HTTP request to the OMA contains a parameter to let the OMA know which method is to be executed. The generic methods implemented for OMAs are:

- Viewing methods: These methods are used for visualizing various parts of the ICE objects. Some of the methods implemented in the base OMA are: NORMAL (shows some important meta–data with an icon for downloading the ICF), THUMBNAIL (shows a small version of

an image), SUMMARY (shows brief information about the object, such as name, type and owner), and INFO (shows all the meta–data, plus discussion and awareness lists).

- Editing methods: An OMA can receive data from the user, and can use that data to update an existing ICE object. An editing method will present the user with an HTML form where the user can change the meta–data and submit the changes to the OMA, possibly with a newer version of the ICF. The OMA will then update the current ODF and ICF (ICE does not currently support version control).

Figure A.3 shows an overview of how ICE objects are manipulated by the users. The users will interact with the OMA through an HTML page that is generated by a UIO and sent to them by the Web server. This page provides a view of an ICE object and a set of HTML interaction elements in order to let the user manipulate the object. The Web server transmits all the user requests back to the UIO through a CGI interface. The gray area in figure A.3 indicates that new UIOs can be implemented in order to interact with other environments than the Web. An example that is implemented partly in the current version of ICE is an email interface where the users can interact with the objects through an email server. Object discussion lists use this interface for communicating with the users.

A.4.2 Inter–object communication

Figure A.3 shows the simplest case where each OMA is responsible for one ICE object in the database. For composite ICE objects, such as shared workspaces consisting of several information objects, the OMAs will have to communicate with each other in order to generate an integral presentation to the users. For this we have chosen a simple hierarchical structure, letting the parent OMA (such as the workspace OMA) interact with the child OMAs to generate the final output to the server. This is shown in figure A.4. In this figure, OMA1 can request an HTML presentation from OMA2–4. These presentations are then composed by OMA1 to a final HTML page and sent to the server.

The advantage of this communication scheme is its simplicity in that OMA1 does not need to know anything about the internal logics of the other OMAs. OMA1 receives one HTML file from each of the OMA2–4, and generates a result by appending these files without analyzing their content. A specialized inter–object communication language will obviously provide us with more flexibility. This is a planned extension to a new version of ICE.

A.4.3 Access control in ICE

Access control in ICE is at method level, meaning that the users can put access restrictions for different methods in an OMA. As an example, a user can decide to give some users access to the SUMMARY method of an object. This is how users can construct different views of their objects (Figure A.1). For instance, the users can decide to hide the details of a DOCUMENT object while they are writing the document by removing the NORMAL and INFO methods from the object's access list. They can still provide a SUMMARY method that shows the abstract, the deadline, and the status of the document.

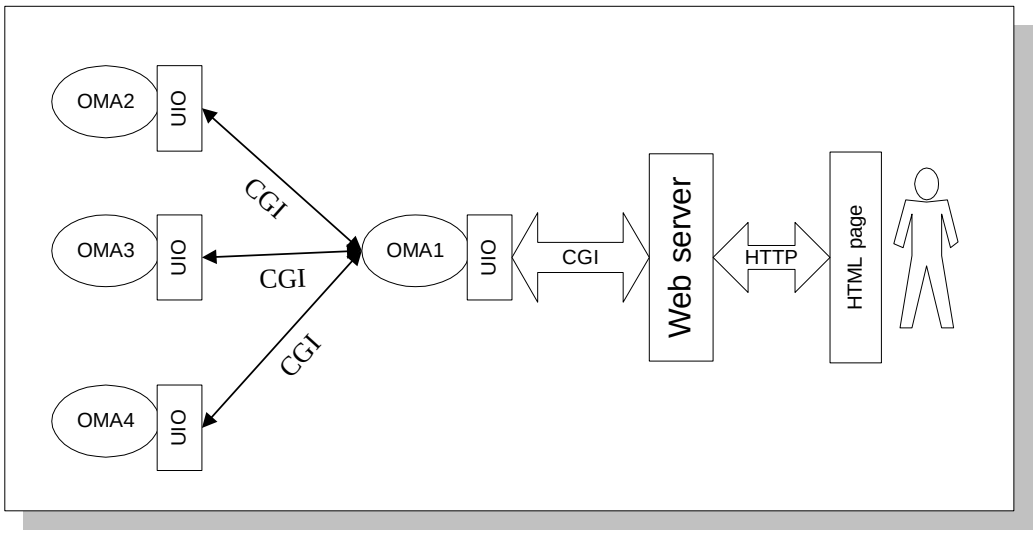


Figure A.4: Communication among objects is done through the user interface objects. The parent object plays the role of a server when communicating with child objects. The final output is sent to the user through the actual server.

A.4.4 Email interface to ICE

As mentioned before, the Web interface is the only interface fully implemented so far. There is however work in progress for connecting ICE to an email interface. The first result of this effort is the discussion lists (Asbjørnsen and Ellingsen 1997). The email interface is implemented in conjunction with a standard email list server software. Each object receives its own email account on an email server. A discussion list for the object is also created on the list server. The object subscribes to its own discussion list in order to receive the messages sent by the users. The *Reply–To* field in all the messages are set to the email address of the discussion list, so that a copy of the messages is always received by the object and archived for Web–based presentation and other use.

A.5 Related research

The building blocks in ICE correspond to the three complementary components of the groupware model discussed by Ellis and Wainer (1994b). These components are ontological model (corresponding to our information objects), coordination model (corresponding to our collaboration objects), and user interface model (corresponding to our user interface objects). According to Ellis and Wainer most of the available groupware applications consist more or less of these three sub–models. These sub–models are also visible in systems for radical tailorability such as Oval (Malone et al. 1995) and Information Lens (Malone et al. 1989). Information Lens is a system for tailoring message–based applications. OVAL is an extension to Information Lens, and consists of building blocks (called Objects, Views, Agents, and Links) similar to ICE, but with the exception

of agents. The most important difference between ICE and Oval is that Oval does not support the process of building applications. Applications in Oval are built off–line without much support from the system.

An example of a system supporting the collaborative process of customizing itself is mentioned by Wulf (1995). In this system, a meta–function for negotiation is provided in form of a voting mechanism. Whenever a user wishes to change the application in some way, other users are confronted with a voting tool where they can choose to accept or reject the change. The limitation of the system is that it only permits a predefined set of changes to an application (application preferences). However, it is evident that ICE could enjoy such a negotiation functionality for supporting the application development process.

ICE as a Web–based system is similar to BSCW (Bentley, Appelt, Busbach, Hinrichs, Kerr, Sikkel, Trevor and Woetzel 1997). BSCW is a shared workspace system that provides a small group of people with mechanisms for sharing and co–authoring of documents (Horstmann and Bentley 1997). Awareness events are an important part of this process. BSCW provides only a limited set of objects related to the co–authoring domain, which are documents and email messages. This makes BSCW best suited for co–authoring, but does not support system development processes.

A.6 Conclusions and further work

There are three characteristics of ICE that we believe are important to highlight. First, ICE provides the users with a continuous prototyping environment with support for the process of prototyping. This results in prototypes that are somehow self–contained, in that they also contain information about the process of prototyping. This information can be used as input to further development.

Second, ICE provides a medium where different users can cooperate during the development process. System development teams are highly dynamic, with new members arriving and old members leaving the project all the time. Easy, *light–weight* communication among the members is crucial.

A third point, and maybe the most important one, is that these prototypes are developed directly in the work context. The users build the prototypes while they are working with them. The prototypes include features from the actual work context that are found to be important. Later on, the users build improved prototypes that are again placed in the work context and used. This makes it possible to continuously refine the system while solving real world problems.

The relevance of these three points is confirmed by our experience in the development of ICE itself. ICE was implemented in the context of an EU project called AQUARIUS (AQUARIUS Consortium 1998). The system itself is the result of a collaborative effort among a large group of users distributed among three European countries. The process of developing ICE was two–folded. The developers were mostly specializing the existing building blocks. These building blocks were then used and tested by other users in their own work context. At the beginning, ICE was not used for supporting the development process. This need evolved later in the project when we became aware of the mediating effect of the prototypes, and we saw the need for storing an archive of the development process within the prototypes.

Extensions to ICE are foreseen in three directions. First, we want to improve the existing building blocks. We need to include more advanced coordination mechanisms in order to be able

to create applications with more sophisticated process models. We also need mechanisms for allowing the creation of new objects without the need for programming the system.

Second, it is also desirable to improve the existing support for development processes. Tailorability is not scalable. With larger applications we will have to incorporate other methods for user involvement, such as collaborative conceptual modeling and enterprise modeling methods. This will again require change management.

Third, we want to fully exploit the possibilities offered by collaboration technologies to bring different user groups together. This will result in extensions to the cooperation support currently implemented in ICE in order to include mechanisms for decision–making, synchronous cooperation, and video and audio support.

A.7 Acknowledgement

The first prototypes of ICE were implemented and tested in the context of the AQUARIUS project. I thank the project members for cooperatively developing the ICE prototype. Specially, I thank Stig Petter Olsrød and Trond Isaksen for coding the prototype, and Hendrik Klompmaker for testing it under difficult conditions. I also thank Monica Divitini for shuffling my ideas about ICE, and for constructive comments on earlier versions of this paper.

Bibliography

Ackerman, M. S. (ed.): 1996, *Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work, CSCW'96, Cambridge, Mass., USA*, ACM Press, New York.

Agostini, A., De Michelis, G., Grasso, M. A., Prinz, W. and Syri, A.: 1996, Contexts, Work Processes, and Workspaces, *Computer Supported Cooperative Work: The Journal of Collaborative Computing* **5**(2–3), 223–250.

Allen, L., Fernandez, G., Kane, K., Leblang, D., Minard, D. and Posner, J.: 1995, ClearCase MultiSite: Supporting Geographically–Distributed Software Development, *in* J. Estublier (ed.), *Software Configuration Management: ICSE SCM–4 and SCM–5 Workshops Selected Papers*, number 1005 in *LNCS*, Springer, Berlin, pp. 194–214.

Andersen, R.: 1994, *A Configuration Management Approach for Supporting Cooperative Information System Development*, PhD thesis, Norwegian Institute of Technology, IDT, Trondheim, Norway.

Andersen, R. and Sølvberg, A.: 1993, Conflict Management in Systems Development Groups, *in* N. Prakash, C. Rolland and B. Pernici (eds), *Proceedings of the IFIP WG8.1 Working Conference on Information System Development Process, Como, Italy*, North–Holland, pp. 207–227.

AQUARIUS Consortium: 1998, AQUARIUS Web site. http://aquarius.euro.org.

Asbjørnsen, K. E. and Ellingsen, B.: 1997, PAT – Mail Handling System In Aquarius, *Technical report*, Norwegian University of Science and Technology.

Babich, W. A.: 1986, *Software Configuration Management – Coordination for Team Productivity*, Addison–Wesley, Reading, Massachusetts.

Baecker, R. M. (ed.): 1993, *Readings in Groupware and Computer–Supported Cooperative Work– Assisting Human–Human Collaboration*, Morgan Kaufmann.

Bannon, L. J. and Bødker, S.: 1997, Constructing Common Information Spaces, *in* Hughes, Prinz, Rodden and Schmidt (1997), pp. 81–96.

Bannon, L. J., Robinson, M. and Schmidt, K. (eds): 1991, *Proceedings of the Second European Conference on Computer Supported Cooperative Work, ECSCW'91, Amsterdam, The Nederlands*, Kluwer Academic Publishers.

Beaudouin-Lafon, M. (ed.): 1999, *Computer Supported Co–operative Work*, Trends in Software, John Wiley & Sons, New York.

Benedikt, M. L.: 1992, Cyperspace: Some Proposals, *in* M. L. Benedikt (ed.), *Cyberspace: First Steps*, MIT Press, Cambridge, Massachusetts, pp. 273–302.

Benford, S., Bowers, J., Fahlén, L. E., Greenhalgh, C. and Snowdon, D.: 1995, User Embodiment in Collaborative Virtual Environments, *in* I. R. Katz, R. Mack, L. Marks, M. B. Rosson and J. Nielsen (eds), *Proceedings of the CHI'95 Conference, Denver, Colorado, USA*, ACM Press, pp. 242–249.

Benford, S., Bowers, J., Fahlén, L., Mariani, J. and Rodden, T.: 1994, Supporting Cooperative Work in Virtual Environments, *Computer* **37**(8), 653–668.

Benford, S. and Fahlén, L.: 1993, A Spatial Model of Interaction in Large Virtual Environments, *in* De Michelis, Simone and Schmidt (1993), pp. 109–124.

Bentley, R., Appelt, W., Busbach, U., Hinrichs, E., Kerr, D., Sikkel, K., Trevor, J. and Woetzel, G.: 1997, Basic Support for Cooperative Work on the World Wide Web, *International Journal of Human Computer Studies* **46**(6), 827–846.

Bentley, R., Busbach, U., Kerr, D. and Sikkel, K. (eds): 1997, *Groupware and the World Wide Web*, Kluwer Academic Publisher, Dordrecht. Reprint from the Computer Supported Cooperative Work: The Journal of Collaborative Computing, Volume 6, Nos. 2–3, 1997.

Bentley, R., Horstmann, T. and Trevor, J.: 1997, The World Wide Web as enabling technology for CSCW: The case of BSCW, *Computer Supported Cooperative Work: The Journal of Collaborative Computing* **6**(2–3), 111–134.

Bentley, R., Rodden, T., Sawyer, P. and Sommerville, I.: 1992, An architecture for tailoring cooperative multi–user displays, *in* Turner and Kraut (1992), pp. 187–194.

Berners-Lee, T., Cailliau, R., Luotonen, A., Nielsen, H. F. and Secret, A.: 1994, The World Wide Web, *Communications of the ACM* **37**(8), 76–82.

Bhatnagar, R.: 1999, *Support for XML in IGLOO (tentative title)*, Master's thesis, Norwegian University of Science and Technology.

Bin, Y., Farshchian, B. A., Li, T., Rao, J. and Su, X.: 1999, MultiCASE implementation report, *Technical report*, Department of Computer and Information Science, Norwegian University of Science and Technology, NTNU.

Bly, S. A.: 1988, A Use of Drawing Surfaces in Different Collaborative Settings, *in* CSCW'88 (1988), pp. 250–256.

Bogia, D., Tolone, W., Bignoli, C. and Kaplan, S.: 1996, Issues in the Design of Collaborative Systems: Lessons from ConversationBuilder, *in* Shapiro, Tauber and Traunmüller (1996), chapter 24, pp. 401–422.

Booch, G., Rumbaugh, J. and Jacobson, I.: 1999, *The Unified Modeling Language User Guide*, Addison–Wesley, Reading, Massachusetts.

Brooks Jr., F. P.: 1975, *The Mythical Man–Month – Essays on Software Engineering*, Addison–Wesley, Reading, MA.

Brown, A. W., Earl, A. N. and McDermid, J. A.: 1992, *Software Engineering Environments – Automated Support for Software Engineering*, McGraw-Hill, London.

Brown, J. S. and Duguid, P.: 1991, Organizational Learning and Communities–of–Practice: Towards a Unified View of Working, Learning, and Innovation, *Organization Science* **2**(1), 40–57.

Button, G. and Sharrock, W.: 1995, Practices in the work of ordering software development, *in* A. Firth (ed.), *The Discourse of Negotiation – Studies of Language in the Workplace*, Pergamon, chapter 7, pp. 159–180.

Button, G. and Sharrock, W.: 1996, Project Work: The Organisation of Collaborative Design and Development in Software Engineering, *Computer Supported Cooperative Work: The Journal of Collaborative Computing* **5**(4), 369–386.

Chen, H.: 1994, Collaborative Systems: Solving the Vocabulary Problem, *IEEE Computer* **27**(5), 58–66.

Christensen, P. O. and Karlsen, T. G.: 1999, *Bevissthet i delte virtuelle rom (Awareness in shared virtual spaces)*, Master's thesis, Norwegian University of Science and Technology, Trondheim, Norway.

Clark, H. H. and Brennan, S. E.: 1991, Grounding in communication, *in* L. B. Resnick, J. M. Levine and S. D. Teasley (eds), *Perspectives on Socially Shared Cognition*, American Psychological Association, Washington, DC, pp. 127–149. Reprinted in (Baecker 1993).

Codenie, W., De Hondt, K., Steyaert, P. and Vercammen, A.: 1997, From Custom Applications to Domain–Specific Frameworks, *Communications of the ACM* **40**(10), 71–77.

Conway, M. E.: 1968, How do committees invent?, *Datamation* **14**(4), 28–31.

Covi, L. M., Olson, J. S., Rocco, E., Miller, W. J. and Allie, P.: 1998, A Room of Your Own: What Do We Learn about Support of Teamwork from Assessing Teams in Dedicated Project Rooms?, *in* N. A. Streitz, S. Konomi and H.-J. Burkhardt (eds), *Proceedings of the First International Workshop on Cooperative Buildings, CoBuild'98, Darmstadt, Germany*, number 1370 in *LNCS*, Springer, Berlin, pp. 53–65.

CSCW'88 (ed.): 1988, *Proceedings of the Conference on Computer–Supported Cooperative Work, CSCW'88, Portland, OR, USA*, ACM.

CSCW'90 (ed.): 1990, *Proceedings of the Conference on Computer–Supported Cooperative Work, CSCW'90, Los Angeles, USA*, ACM.

Curtis, B., Krasner, H. and Iscoe, N.: 1988, A Field Study of the Software Design Process for Large Systems, *Communications of the ACM* **31**(11), 1268–1287.

Damskog, H. O.: 1997, AQUARIUS User Interface, *Technical report*, Norwegian University of Science and Technology.

Day, M.: 1997, What Synchronous Groupware Needs: Notification Services, *Proceedings of The Sixth Workshop on Hot Topics in Operating Systems, Cape Cod, Massachusetts*, IEEE Computer Society Press, Los Alamitos, California, pp. 118–122.

Day, M., Patterson, J., Kucan, J. and Chee, W. M.: 1996, Notification Service Transfer Protocol (NSTP) version 1.0, *Lotus Workgroup Technologies Technical Report 96-08*, Lotus Research.
URL: *ftp://ftp.lotus.com/pub/lotusweb/corp/research/nstp.ps*

De Michelis, G., De Paoli, F., Pluchinotta, C. and Susani, M.: 2000, Weakly Augmented Reality: observing and designing the work-place of creative designers, *Proceedings of the Conference on Designing Augmented Reality Environments, DARE 2000, Helsinore, Denmark*, pp. xx–yy.

De Michelis, G., Simone, C. and Schmidt, K. (eds): 1993, *Proceedings of the Third European Conference on Computer–Supported Cooperative Work, ECSCW'93, Milano, Italy*, Kluwer Academic Publishers, Dordrecht.

DeSanctis, G. and Gallupe, B. R.: 1987, A Foundation for the Study of Group Decision Support Systems, *Management Science* **33**(5), 589–609.

Dewan, P.: 1995, Multiuser architectures, *in* L. J. Bass and C. Unger (eds), *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human Computer Interaction, Yellowstone Park, U.S.A.*, Chapman & Hall, pp. 247–270.

Dijkstra, E. W.: 1968, Go to statement considered harmful, *Communications of the ACM* **11**(3), 147–148.

Dix, A.: 1997, Challenges for Cooperative Work on the Web: An Analytical Approach, *Computer Supported Cooperative Work: The Journal of Collaborative Computing* **6**(2–3), 135–156. Reprinted in (Bentley, Busbach, Kerr and Sikkel 1997).

Dourish, P. and Bellotti, V.: 1992, Awareness and Coordination in Shared Workspaces, *in* Turner and Kraut (1992), pp. 107–114.

Drucker, P. F.: 1988, The Coming of the New Organization, *Harvard Business Review* (January–February), 45–53.

Ehn, P. and Kyng, M.: 1991, Cardboard Computers: Mocking–it–up or Hands–on the Future, *in* Greenbaum and Kyng (1991), chapter 9, pp. 169–195.

Ellis, C. A. and Wainer, J.: 1994a, Goal–based models of collaboration, *Collaborative Computing* **1**(1), 61–86.

Ellis, C., Gibbs, S. and Rein, G.: 1991, Groupware – Some Issues and Experiences, *Communications of the ACM* **34**(1), 39–58.

Ellis, C. and Wainer, J.: 1994b, A Conceptual model of Groupware, *in* Furuta and Neuwirth (1994), pp. 79–88.

Elmasri, R. and Navathe, S. B.: 1989, *Fundamentals of Database Systems*, Addison–Wesley, Redwood City, California.

Endsley, M. R.: 1995, Toward a Theory of Situation Awareness in Dynamic Systems, *Human Factors* **37**(1), 32–64.

Farshchian, B. A.: 1999, Shared workspace applications for collaboration in the large: A product–centric approach, *in* T. Kindberg (ed.), *Proceedings of Changing Places – A one–day workshop on workspace models for collaboration, London, U.K.*, Department of Computer Science, Queen Mary & Westfield College, University of London, pp. 1–7.

Farshchian, B. A.: 2000a, Gossip: An awareness engine for increasing product awareness in distributed development projects, *in* Wangler and Bergman (2000), pp. 264–278.

Farshchian, B. A.: 2000b, IGLOO: A framework for developing product-oriented shared workspace applications, *in* R. Dieng, A. Giboin, L. Karsenty and G. De Michelis (eds), *Proceedings of the 5th International Conference on the Design of Cooperative Systems, COOP'2000, Sophia Antipolis, France*, IOS Press, Amsterdam, pp. 337–350.

Farshchian, B. A. and Divitini, M.: 1997, ICE: A Highly Tailorable System for Building Collaboration Spaces on the WWW, *in* A. Mørch, O. Stiemerling and V. Wulf (eds), *Proceedings of the Workshop on Tailorable Groupware: Issues, Methods, and Architectures, Phoenix, Arizona, U.S.A.*
URL: *http://www.ifi.uib.no/staff/anders/research/group97/*

Faucheux, C.: 1997, How Virtual Organizing Is Transforming Management Science, *Communications of the ACM* **40**(9), 50–55.

Fitzpatrick, G., Mansfield, T. and Kaplan, S. M.: 1996, Locales Framework: Exploring foundations for collaboration support, *in* J. Grundy and M. Apperley (eds), *Proceedings of the Sixth Australian Conference on Computer–Human Interaction, Hamilton, New Zealand*, IEEE Computer Society Press, Los Alamitos, California, pp. 34–41.

Fitzpatrick, G., Tolone, W. J. and Kaplan, S. M.: 1995, Work, Locales and Distributed Social Worlds, *in* H. Marmolin, Y. Sundblad and K. Schmidt (eds), *Proceedings of the Fourth European Conference on Computer–Supported Cooperative Work, ECSCW'95, Stockholm, Sweden*, Kluwer Academic Publisher, pp. 1–16.

Fogel, K.: 1999, *Open Source Development with CVS*, CoriolisOpen Press, Scottsdale, Arizona.

Forte, G. and Norman, R. J.: 1992, A Self–Assessment by the Software Engineering Community, *Communications of the ACM* **35**(4), 28–32.

Fuggetta, A.: 1993, A Classification of CASE Technology, *IEEE Computer* **x**(x), 25–38.

Furuta, R. and Neuwirth, C. (eds): 1994, *Proceedings of the Conference on Computer–Supported Cooperative Work, CSCW'94, Chapel Hill, North Carolina, USA*, ACM Press.

Garli, N.-H. and Lund, A.: 2000, *Gossip: An Integrated Shared Product Space and Awareness Engine for the IGLOO Framework*, Master's thesis, Norwegian University of Science and Technology, Trondheim, Norway.

Gray, J. P. and Ryan, B.: 1997, Applying the CDIF standard in the construction of CASE design tools, *in* P. Bailes (ed.), *Proceedings of the Australian Software Engineering Conference*, IEEE, pp. 88–97.

Greenbaum, J. and Kyng, M. (eds): 1991, *Design at Work: Cooperative Design of Computer Systems*, Lawrence Erlbaum Associates.

Greenberg, S. and Roseman, M.: 1998, Using a Room Metaphor to Ease Transitions in Groupware, *Technical Report 98/611/02*, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada.

Greenberg, S. and Roseman, M.: 1999, Groupware Toolkits for Synchronous Work, *in* Beaudouin-Lafon (1999), chapter 6, pp. 135–168.

Grinter, R. E.: 1995, Using a Configuration Management Tool to Coordination Software Development, *in* N. Comstock and C. Ellis (eds), *Proceedings of the Conference on Organizational Computing Systems, COOCS'95, Milpitas, California, USA*, ACM Press, pp. 168–177.

Grinter, R. E.: 1996, Supporting Articulation Work Using Software Configuration Management Systems, *Computer Supported Cooperative Work: The Journal of Collaborative Computing* **5**(4), 447–465.

Grinter, R. E.: 2000, Workflow Systems: Occasions for Success and Failure, *Computer Supported Cooperative Work* **9**(1?), 189–214.

Grinter, R. E., Herbsleb, J. D. and Perry, D. E.: 1999, The Geography of Coordination: Dealing with Distance in R & D Work, *in* S. C. Hayne (ed.), *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work, Group'99, Phoenix, Arizona*, ACM Press, pp. 306–315.

Grudin, J.: 1994a, Computer–Supported Cooperative Work: History and Focus, *IEEE Computer* **27**(5), 19–26.

Grudin, J.: 1994b, Groupware and Social Dynamics: Eight Challenges for Developers, *Communications of the ACM* **37**(1), 92–105.

Gutwin, C. and Greenberg, S.: 1998, Effects of Awareness Support on Groupware Usability, *in* CHI'98 (ed.), *Proceedings of the Conference on Human Factors in Computing Systems, CHI'98, Los Angeles, CA, USA*, ACM Press, New York, pp. 511–518.

Gutwin, C. and Greenberg, S.: 1999, A Framework of Awareness for Small Groups in Shared-Workspace Groupware, *Technical Report 99–1*, Department of Computer Science, University of Saskatchewan, Canada.

Gutwin, C., Greenberg, S. and Roseman, M.: 1996, Workspace Awareness in Real–Time Distributed Groupware: Framework, Widgets, and Evaluation, *in* R. Sasse, A. Cunningham and R. Winder (eds), *Proceedings of the HCI'96: People and Computers XI, London, U.K.*, Springer–Verlag, Berlin, pp. 281–298.

Harrison, S. and Dourish, P.: 1996, Re–Place the Space: the Roles of Place and Space in Collaborative Systems, *in* Ackerman (1996), pp. 67–76.

Hayne, S. C. and Prinz, W. (eds): 1997, *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work, Group'97, Pheonix, USA*, ACM Press, New York.

Henderson, Jr., D. A. and Card, S. K.: 1986, Rooms: The Use of Multiple Virtual Workspaces to Reduce Space Contention in a Window-Based Graphical User Interface, *ACM Transactions on Graphics* **5**(3), 211–243.

Herbsleb, J. D. and Grinter, R. E.: 1999, Splitting the Organization and Integrating the Code: Conway's Law Revisited, *in* ICSE'99 (ed.), *Proceedings of the 21th Conference on Software Engineering, Los Angeles, California, USA*, ACM Press, New York, pp. 85–95.

Herlea, D. E.: 1998, User Participation in Requirements Negotiation, *in* M. Divitini, B. A. Farshchian and T. Tuikka (eds), *Proceedings of the Workshop on Internet–Based Groupware for User Participation in Product Development*, number B 56 in *Working Paper Series*, University of Oulu, INFOTECH Research Center, pp. 25–29.

Herlea, D. and Greenberg, S.: 1998, Using a Groupware Space for Distributed Requirements Engineering, *Proceedings of WET ICE '98: IEEE Seventh International Workshops on Enabling Technologies: Coordinating Distributed Software Development Projects, Stanford University, California, USA*, pp. 57–62.

Horstmann, T. and Bentley, R.: 1997, Distributed authoring on the Web with the BSCW shared workspace system, *ACM Standards View* **5**(1), 9–16.

Hu, C.-H. and Wang, F.-J.: 1998, A Multi–User Visual Object–Oriented Programming Environment, *Proceedings of The Twenty-Second Annual International Computer Software and Applications Conference, COMPSAC'98, Vienna, Austria*, IEEE Computer Society Press, Los Alamitos, CA, pp. 262–268.

Hughes, J. A., Prinz, W., Rodden, T. and Schmidt, K. (eds): 1997, *Proceedings of the Fifth European Conference on Computer Supported Cooperative Work, ECSCW'97, Lancaster, UK*, Kluwer Academic Publishers, Dordrecht.

Iansiti, M. and MacCormack, A.: 1997, Developing Products on Internet Time, *Harvard Business Review* (September–October), 108–117.

Iivari, J.: 1996, Why Are CASE Tools Not Used?, *Communications of the ACM* **39**(10), 94–103.

Jarke, M., Maltzahn, C. and Rose, T.: 1992, Sharing Processes: Team Coordination in Design Repositories, *International Journal of Intelligent and Cooperative Information Systems* **1**(1), 145–167.

Jarzabek, S. and Huang, R.: 1998, The Case for User–Centered CASE Tools, *Communications of the ACM* **41**(8), 93–99.

Johnson-Lenz, P. and Johnson-Lenz, T.: 1982, Groupware: The Process and Impacts of Design Choices, *in* E. B. Kerr and S. R. Hiltz (eds), *Computer–Mediated Communication Systems : Status and Evaluation*, Human communication research series, Academic Press, New York, chapter xx, pp. 45–55.

Kaplan, S. M., Fitzpatrick, G., Mansfield, T. and Tolone, W. J.: 1997, MUDdling Through, *in* HICSS'97 (ed.), *Proceedings of the 30th Hawaii Int'l Conf. on System Sciences, Volume 2 – Collaboration Systems and Technology*, IEEE Computer Society Press, pp. 539–548.

Keil, M. and Carmel, E.: 1995, Customer–Developer Links in Software Development, *Communications of the ACM* **38**(5), 33–44.

Keil-Slawik, R.: 1992, Artifacts in Software Design, *in* C. Floyd, H. Züllighoven, R. Budde and R. Keil-Slawik (eds), *Software development and Reality Construction*, Springer–Verlag, Berlin, chapter 4.4, pp. 168–188.

Kelly, S.: 1998, CASE tools support for co-operative work in information systems design, *in* Rolland et al. (1998), pp. 49–69.

Kelly, S. and Lyytinen, K.: 1996, MetaEdit+ – A Fully Configurable Multi–User and Multi–Tool CASE and CAME Environment, *in* P. Constantopoulos, J. Mylopoulos and Y. Vassiliou (eds), *Proceedings of the 8th International Conference on Advanced Information Systems Engineering, CAiSE'96, Heraklion, Crete, Greece*, number 1080 in *LNCS*, Springer, Berlin, pp. 1–21.

Kendall, K. E. and Kendall, J. E.: 1988, *Systems Analysis and Design*, Prentice Hall.

Knudsen, I. and Solheim, R.: 1999, *Bringing Synchronous Collaboration into ICE*, Master's thesis, Norwegian University of Science and Technology.

Krauss, R. M. and Fussell, S. R.: 1990, Mutual Knowledge and Communicative Effectiveness, *in* J. Galegher, R. E. Kraut and C. Egido (eds), *Intellectual Teamwork – Social and Technological Foundations of Cooperative Work*, Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, pp. 111–145.

Kraut, R. E. and Streeter, L.: 1995, Coordination in Software Development, *Communications of the ACM* **38**(3), 69–81.

Lauwers, J. C. and Lantz, K. A.: 1990, Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems, *in* J. C. Chew and J. Whiteside (eds), *Proceedings of the CHI'90 Conference, Seattle, Washington, USA*, ACM Press, pp. 303–311.

Lee, J. H., Prakash, A., Jaeger, T. and Gwobaw, W.: 1996, Supporting Multi–User, Multi–Applet Workspaces in CBE, *in* Ackerman (1996), pp. xx–yy.

Lie-Nielsen, J.: 2000, *Design and Implementation of CoClust (tentative title)*, Master's thesis, Norwegian University of Science and Technology, Trondheim, Norway.

Lindland, O. I., Sindre, G. and Sølvberg, A.: 1994, Understanding Quality in Conceptual Modeling, *IEEE Software* **11**(2), 42–49.

Mackay, W. E.: 1990, Patterns of Sharing Customizable Software, *in* CSCW'90 (1990), pp. 209–221.

Mackay, W. E.: 1999, Media Spaces: Environments for Informal Multimedia Interaction, *in* Beaudouin-Lafon (1999), chapter 3, pp. 55–82.

Malone, T. W., Grant, K. R., Lai, K.-Y., Rao, R. and Rosenblitt, D. A.: 1989, The Information Lens: An Intelligent System for Information Sharing and Coordination, *in* M. H. Olson (ed.), *Technological Support for Work Group Collaboration*, Lawrence Erlbaum Associates, pp. 65–88.

Malone, T. W., Lai, K.-Y. and Fry, C.: 1995, Experiments with Oval: A Radically Tailorable Tool for Cooperative Work, *ACM Transactions on Information Systems* **13**(2), 177–205.

Mansfield, T., Kaplan, S., Fitzpatrick, G., Phelps, T., Fitzpatrick, M. and Taylor, R.: 1997, Evolving Orbit: a progress report on building locales, *in* Hayne and Prinz (1997), pp. 241–250.

Mansfield, T., Kaplan, S., Fitzpatrick, G., Phelps, T., Fitzpatrick, M. and Taylor, R.: 1999, Towards locales Supporting collaboration with Orbit, *Information and Software Technology* **41**(6), 367–382.

Mariani, J. A. and Prinz, W.: 1993, From Multi–User to Shared Object Systems: Awareness about Co–Workers in Cooperation Support Object Databases, *in* H. Reichel (ed.), *Proceedings of the 23. GI–Jahrestagung: Informatik – Wirtschaft – Gesellschaft*, Springer, Berlin Heidelberg, pp. 476–481.

Marmolin, H., Sundblad, Y. and Pehrson, B.: 1991, An Analysis of Design and Collaboration in a Distributed Environment, *in* Bannon, Robinson and Schmidt (1991), pp. 147–162.

McGrath, J. E. and Hollingshead, A. B.: 1994, *Groups Interacting with Technology – Ideas, Evidence, Issues, and an Agenda*, SAGE Publications, Thousand Oaks, California.

Moran, T. P. and Anderson, R. J.: 1990, The Workaday World As a Paradigm for CSCW Design, *in* CSCW'90 (1990), pp. 381–393.

Moran, T. P. and Carroll, J. M. (eds): 1996, *Design Rationale: Concepts, Techniques, and Use*, Laurence Erlbaum Associates.

Ngwenyama, O. K. and Lyytinen, K. J.: 1997, Groupware Environments as Action Constitutive Resources: A Social Action Framework for Analyzing Groupware Technologies, *Computer Supported Cooperative Work: The Journal of Collaborative Computing* **6**(1), 71–93.

Nonaka, I. and Takeuchi, H.: 1995, *The Knowledge–Creating Company*, Oxford University Press.

Nunamaker, J. F., Dennis, A. R., Valacich, J. S., Vogel, D. R. and George, J. F.: 1991, Electronic meeting systems to support group work, *Communications of the ACM* **34**(7), 40–61.

Olson, G. M., Atkins, D. E., Clauer, R., Finholt, T. A., Jahanian, F., Killeen, T. L., Prakash, A. and Weymouth, T.: 1998, The Upper Atmospheric Research Collaboratory, *ACM Interactions* **3**(X), 48–55.

Olson, G. M. and Olson, J. S.: 1991, User–Centered Design of Collaboration Technology, *Journal of Organizational Computing* **1**(1), 61–83.

Olson, G. M., Olson, J. S., Carter, M. R. and Storrøsten, M.: 1992, Small Group Design Meetings: An Analysis of Collaboration, *Human–Computer Interaction* **7**(4), 347–374.

Olson, G. M., Olson, J. S., Storrøsten, M., Carter, M., Herbsleb, J. and Rueter, H.: 1996, The Structure of Activity During Design Meetings, *in* Moran and Carroll (1996), chapter 7, pp. 217–239.

Olsrød, S. P. and Isaksen, T.: 1996, *ICE – Aquarius Database*, Aquarius project report, NTNU, Norwegian University of Science and Technology, Department of Computer and Information Science, N–7034, Trondheim, Norway.

Orlikowski, W. J.: 1992, Learning From Notes: Organizational Issues in Groupware Implementation, *in* Turner and Kraut (1992), pp. 362–369.

Orlikowski, W. J.: 1993, CASE Tools as Organizational Change: Investigating Incremental and Radical Changes in Systems Development, *Management Information Systems Quarterly* **17**(3), 309–340.

Ovum: 1999, Ovum Evaluates: Configuration Management, *Evaluation report*, Ovum.

Parnas, D. L.: 1972, On the Criteria To Be Used in Decomposing Systems into Modules, *Communications of the ACM* **15**(12), 1053–1058.

Pendergast, M., Aytes, K. and Lee, J. D.: 1999, Supporting the group creation of formal and informal graphics during business process modeling, *Interacting with Computers* **11**(4), 355–373.

Potts, C. and Catledge, L.: 1996, Collaborative Conceptual Design: A Large Software Project Case Study, *Computer Supported Cooperative Work: The Journal of Collaborative Computing* **5**(4), 415–445.

Prakash, A.: 1999, Group Editors, *in* Beaudouin-Lafon (1999), chapter 5, pp. 103–133.

Prakash, A. and Shim, H.: 1994, DistView: support for building efficient collaborative applications using replicated objects, *in* Furuta and Neuwirth (1994), pp. xx–yy.

Prakash, A., Shim, H. S. and Lee, J. H.: 1999, Data Management Issues and Tradeoffs in CSCW Systems, *IEEE Transactions on Knowledge and Data Engineering* **11**(1), 213–227.

Prinz, W.: 1999, NESSIE: An Awareness Environment for Cooperative Settings, *in* S. Bødker, M. Kyng and K. Schmidt (eds), *Proceedings of The Sixth European Conference on Computer Supported Cooperative Work, ECSCW'99, Copenhagen, Denmark*, Kluwer Academic Publishers, Dordrecht, pp. 391–410.

Radding, A.: 1999, Join The Team, *InformationWeek Online* . October 4 [last visited July 7, 2000].
URL: *http://www.informationweek.com/755/55adtea.htm*

Ramampiaro, H. and Nygård, M.: 1999, Cooperative database systems: A constructive review of cooperative transaction models, *in* Y. Kambayashi and H. Takakura (eds), *Proceedings of the 1999 International Symposium on Database Applications in Non–Traditional Environments, DANTE '99, Kyoto, Japan*, IEEE Computer Society Press, Los Alamitos, California, pp. 315–324.

Ramduny, D., Dix, A. and Rodden, T.: 1998, Exploring the design space for notification servers, *in* CSCW'98 (ed.), *Proceedings of the Conference on Computer Supported Cooperative Work, Seattle, Washington, USA*, ACM Press, New York, pp. 227–235.

Rein, G. L. and Ellis, C. A.: 1991, rIBIS: A Real–Time Groupware Hypertext System, *International Journal of Man Machine Studies* **34**(3), 349–368.

Rittel, H.: 1972, Structure and Usefulness of Planning Information Systems, *Bedriftsøkonomen* **34**(8), 398–401.

Robertson, T.: 1996, Embodied Actions in Time and Space: The Cooperative Design of a Multimedia, Educational Computer Game, *Computer Supported Cooperative Work: The Journal of Collaborative Computing* **5**(4), 341–367.

Robinson, M.: 1991, Double–Level Languages and Co–operative Working, *AI & Society: The Journal of Human and Machine Intelligence* **5**(1), 34–60.

Robinson, M.: 1993, Design for unanticipated use....., *in* De Michelis et al. (1993), pp. 187–202.

Robinson, M. and Bannon, L.: 1991, Questioning Representations, *in* Bannon et al. (1991), pp. 219–233.

Rodden, T.: 1991, A survey of CSCW systems, *Interacting with Computers* **3**(3), 319–353.

Rodden, T.: 1996, Populating the Application: A Model of Awareness for Cooperative Applications, *in* Ackerman (1996), pp. 87–96.

Rodden, T., Mariani, J. A. and Blair, G.: 1992, Supporting Cooperative Applications, *Computer Supported Cooperative Work: An International Journal* **1**(1–2), 41–67.

Rogers, Y.: 1993, Coordinating Computer–Mediated Work, *Computer Supported Cooperative Work – An International Journal* **1**(4), 295–315.

Rolland, C., Chen, Y. and Fang, M. (eds): 1998, *Proceedings of the IFIP TC8/WG8.1 Working Conference on Information Systems in the WWW Environment, Beijing, China*, Chapman & Hall.

Rømma, F. and Skjønhaug, S. T.: 2000, *Design and implementation of SWAL (tentative title)*, Master's thesis, Norwegian University of Science and Technology.

Roseman, M. and Greenberg, S.: 1996, TeamRooms: Network Places for Collaboration, *in* Ackerman (1996), pp. 325–333.

Sande, A. M. T.: 1998, *Support for collaborative design in contemporary CASE tools – framework and evaluation*, Master's thesis, Norwegian University of Science and Technology, IDI, NTNU, Trondheim, Norway.

Schmidt, K. and Bannon, L. J.: 1992, Taking CSCW Seriously – Supporting Articulation Work, *Computer Supported Cooperative Work – An International Journal* **1**(1–2), 7–40.

Schmidt, K. and Rodden, T.: 1996, Putting it all Together: Requirements for a CSCW Platform, *in* Shapiro et al. (1996), chapter 11, pp. 157–175.

Schön, D. A.: 1983, *The Reflective Practitioner – How Professionals Think in Action*, Basic Books, New York.

Segall, B. and Arnold, D.: 1997, Elvin has left the building: A publish/subscribe notification service with quenching, *Proceedings AUUG97, Brisbane, Australia*.
URL: *http://www.dstc.edu.au/Elvin/doc/papers/auug97/AUUG97.html*

Seltveit, A. H.: 1994, *Complexity reduction in information systems modelling*, PhD thesis, The Norwegian Institute of Technology, Univ. Trondheim.

Shapiro, D., Tauber, M. and Traunmüller, R. (eds): 1996, *The Design of Computer Supported Cooperative Work and Groupware Systems*, Elsevier Science.

Sharon, D. and Bell, R.: 1995, Tools that Bind: Creating Integrated Environments, *IEEE Software* **12**(2), 76–85.

Sharples, M. (ed.): 1993, *Computer Supported Collaborative Writing*, Springer–Verlag.

Shim, H. S., Hall, R. W., Prakash, A. and Jahanian, F.: 1997, Providing Flexible Services for Managing Shared State in Collaborative Systems, *in* Hughes et al. (1997), pp. 237–252.

Signer, B., Erni, A. and Norrie, M. C.: 2000, A Personal Assistant for Web Database Caching, *in* Wangler and Bergman (2000), pp. 64–78.

Sikkel, K.: 1997, A Group–based Authorization Model for Cooperative Systems, *in* Hughes et al. (1997), pp. 345–360.

Simone, C. and Bandini, S.: forthcoming, Integrating awareness in cooperative applications through the reaction–diffusion metaphor. Unpublished article.

Sivesind, L. E. and Grimstad, I.: 1997, Distance Education on the Internet – Courses in Aquarius, *Technical report*, Norwegian University of Science and Technology.

Sølvberg, A.: 2000, Co-operative Concept Modeling, *in* S. Brinkkemper, E. Lindencrona and A. Sølvberg (eds), *Information Systems Engineering – State of the Art and Research Themes*, Springer, Berlin, pp. 305–317.

Sølvberg, A. and Kung, D. C.: 1993, *Information Systems Engineering – An Introduction*, Springer–Verlag.

Sommerville, I.: 1992, *Software Engineering*, Addison–Wesley, Reading, Mass.

Sommerville, I. and Rodden, T.: 1993, Environments for Cooperative Software Development, *Conference X*, IEEE Computer Society Press, pp. 144–155.

Spellman, P. J., Moiser, J. N., Deus, L. M. and Carlson, J. A.: 1997, Collaborative Virtual Workspace, *in* Hayne and Prinz (1997), pp. 197–203.

Star, S. L. and Griesemer, J. R.: 1989, Institutional Ecology: 'Translations' and Boundary Objects: Amateurs and Professionals in Berkeley's Museum of Vertebrate Zoology, *Social Studies of Science* **19**, 387–420.

Suchman, L.: 1997, Centers of Coordination: A Case and Some Themes, *in* L. B. Resnick, R. Säljö, C. Pontecorvo and B. Burge (eds), *Discourse, Tools, and Reasoning – Essays in Situated Cognition*, Springer, Berlin, chapter 2, pp. 41–62.

Taivalsaari, A. and Vaaraniemi, S.: 1997, TDE: Supporting Geographically Distributed Software Design with Shared, Collaborative Workspaces, *in* A. Olivé and J. A. Pastor (eds), *Proceedings of the 9th International Conference on Advanced Information Systems Engineering, CAiSE'97, Barcelona, Spain*, number 1250 in *Lecture Notes in Computer Science*, Springer, pp. 389–408.

Tang, J. C.: 1991, Findings from observational studies of collaborative work, *International Journal of Man–Machine Studies* **34**(2), 143–160.

Tang, J. C. and Leifer, L. J.: 1988, A Framework for Understanding Workspace Activity of Design Teams, *in* CSCW'88 (1988), pp. 244–249.

Tellioğlu, H. and Wagner, I.: 1997, Negotiating Boundaries – Configuration Management in Software Development Teams, *Computer Supported Cooperative Work: The Journal of Collaborative Computing* **6**(4), 251–274.

Tichy, W.: 1985, RCS: A System for Version Control, *Software Practice and Experience* **15**(7), 637–654.

Trevor, J., Koch, T. and Woetzel, G.: 1997, Meta Web: Bringing synchronous groupware to the World Wide Web, *in* Hughes et al. (1997), pp. 65–80.

Trigg, R. H. and Bødker, S.: 1994, From Implementation to Design: Tailoring and the Emergence of Systematization in CSCW, *in* Furuta and Neuwirth (1994), pp. 45–54.

Turner, J. and Kraut, R. (eds): 1992, *Proceedings of the Conference on Computer–Supported Cooperative Work, CSCW'92, Toronto, Canada*, ACM Press, New York.

Vessey, I. and Sravanapudi, A. P.: 1995, CASE Tools as Collaborative Support Technologies, *Communications of the ACM* **38**(1), 83–95.

Walz, D. B., Elam, J. J. and Curtis, B.: 1993, Inside a software design team: Knowledge acquisition, sharing, and integration, *Communications of ACM* **36**(10), 63–77.

Wangler, B. and Bergman, L. (eds): 2000, *Proceedings of the 12th International Conference on Advance Information Systems Engineering, CAiSE'2000, Stockholm, Sweden*, number 1789 in *LNCS*, Springer, Berlin.

Wastell, D. G.: 1993, The Social Dynamics of Systems Development: Conflict, Change and Organizational Politics, *in* S. Easterbrook (ed.), *CSCW: Cooperation or Conflict*, Springer–Verlag, chapter 2, pp. 69–91.

Waterson, P. E., Clegg, C. W. and Axtell, C. M.: 1997, The dynamics of work organization, knowledge and technology during software development, *International Journal of Human–Computer Studies* **46**(1), 79–101.

Winograd, T. and Flores, F.: 1986, *Understanding Computers and Cognition – A New Foundation for Design*, Ablex Publishing Corporation, Norwood, NJ.

Wulf, V.: 1995, Negotiability: A Metafunction to Tailor Access to Data in Groupware, *Behaviour & Information Technology* **14**(3), 143–151.

List of Figures

List of Tables

Index