NTNU
Innovation and Creativity

Applicability and Identified Benefits of Agent Technology

-Implementation and Evaluation of an Agent System

Mari Torgersrud Haug
Elin Marie Kristensen

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Harald Rønneberg, IDI
Co-supervisor: Jørn Ølmheim, Statoil
Einar Landre, Statoil

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The focus in the master thesis would be to implement a prototype of a system based on agent technology. An ideal solution should illustrate the benefits of agent technology. The implementation could demonstrate potential application areas for agent technology in the future, and show that agents can be used to solve problems where traditionally development tools have shortcomings. The goal for the students of the master thesis is to obtain practical experience with agent technology and to evaluate the applicability of agent systems. Is the technology as good as asserted?

Assignment given: 20. January 2006
Supervisor: Harald Rønneberg, IDI

Abstract

Agent oriented approaches are introduced with intention to facilitate software development in situations where other methods have shortcomings. Agents offer new possibilities and solutions to problems due to their properties and characteristics.

Agent technology offer a high abstraction level and is therefore a more appropriate tool for making intelligent systems. Multi-agent systems are well suited in application areas with dynamic and challenging environments, and is advantageous in support for decision making and automation of tasks.

Reduced coupling, encapsulation of functionality and a high abstraction level are some of the claimed benefits related to agent technology. Empirical studies are needed to investigate if agent technology is as good as asserted.

This master thesis will give a deeper understanding of agent technology and benefits related to it. To investigate different aspects, an experiment was designed to reveal the applicability and the benefits. Two multi-agent systems were implemented and used as objects to be studied in the empirical study.

As part of the investigation, a proper application area were chosen. The application area can be characterized as a scheduling problem with a dynamic and complex environment. Prometheus and JACK were used as development and modeling tools. Achieved experiences related to the development process will be presented in this report.

The findings of the empirical study indicate reduced coupling and increased encapsulation of functionality. To achieve these benefits, the number of entities and functions had to be increased, and thus the number of code-lines. Further, the results indicate that more entities and lines of code will not have a significant influence on the development effort, due to the high abstraction level of agent technology.

Preface

This master thesis was written as a part of the Master program at Department of Computer and Information Science(IDI) at the Norwegian University of Science and Technology (NTNU). The work is a continuance of a report which concerned the most important aspects of agent technology. The report was written as part of the graduate level course "TDT4735 Software Engineering, Depth Study" during the fall semester 2005.

The problem definition of the thesis was chosen in cooperation with Statoil. The authors have had two supervisors from Statoil, Jørn Ølmheim and Einar Landre. Together they have contributed with their knowledge about agent technology and java programming, and they have shared their experiences with the authors. Harald Rønneberg has been the supervisor at NTNU.

We would like to thank our supervisors Jørn Ølmheim and Einar Landre for their counseling and advices. They have given us insightful and valuable feedback. Their contribution made the task interesting and challenging. We would like to thank Harald Rønneberg, who has been given counseling on the structure of the master thesis.

Trondheim, June 14, 2006

Elin Marie Kristensen

Mari Torgersrud Haug

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

During the summer of 2005, Elin M Kristensen started to work with agent technology as a part of her summer internship at Statoil, and her interest for agents started to grow. Supervisors at Statoil introduced her to a problem definition concerning agent technology for her depth study and master thesis at the university.

The depth study concerned the most important aspects of agent technology. Different agent architectures, multi-agent systems, and application areas were covered. She also explored achieved experiences related to agent technology.

Elin M Kristensen and Mari Torgersrud Haug decided to cooperate with the master thesis, and to continue the work from the depth study. The master thesis is concerned with practical experiences with agent technology. The results of our work is gathered in this report, and include a description of an accomplished research performed on agent technology.

The motivation presented in section 1.1, will introduce the principal objective with our research. The problem definition, composed in cooperation with Statoil, is quoted in section 1.2. Section 1.3 describes the context, and section 1.4 presents the problem approach. Section 1.5 will give an outline of the report.

1.1 Motivation

Today, it becomes more and more important with software systems that require small amount of user interaction, supervision and manual management for enterprises. Schedule- and planning problems do often occur in complex environments. In these kind of surroundings automated systems can be favorable. If the systems are efficient, they might lead to cost reduction and improve the enterprise's production capacity [Ste05].

For instance, a system responsible for monitoring and managing specific parts in an environment, must be able to provide supervision functionality. In a system implemented with traditionally development tools, the supervision functionality includes continuously probing for changes. Much of the resources are therefore occupied and cause unfavorable system utilization. An alternative is to make the passive parts self controlled and more active. Then the parts could be able to signalize and communicate with the rest of the system when relevant changes have occurred. The time occupied by supervision, could instead be utilized to perform other tasks in the system.

Due to their properties, agents can represent parts and indirectly give them active

and autonomous behavior. Agent technology can therefore lead to more dynamic systems. There are many claimed benefits related to agent technology. These benefits can be of great value in different application areas[PW04a].

1.1.1 Applicability of Agent Technology

The use of agent technology is becoming more and more common. Agents are autonomous software entities with a degree of reasoning ability, which make them potential for solving problems in new application areas. They can act as independent entities and introduce automated systems that can handle new problems and increased amount of information.

Agents are able to detect changes in their surrounding environment. Based on these changes they can communicate with other entities in the system or choose suited actions. This property makes agents clearly advantageous in situations with challenging environments.

By monitoring operations, sensors and detectors, agents can collect information from the environment. Agents can also sort and filter the information for a specified purpose. This property makes the agents useful, bringing the right information to the right people at the right time. As a consequence, people involved can make faster and better decisions.

It is of great interest to make further investigations of agent-technology because it may introduce possibilities to solve problems in challenging environments. Within software engineering, the focus is now turning toward dynamic processes. Systems have to adapt more easily to business needs, because they change continually due to instant requirements. As a consequence, enterprises need more dynamic systems to ensure efficiency and satisfied customers[SL03].

1.1.2 Benefits of Agent Technology

Agents are an approach to structure and develop software that offer certain benefits. Agents are proactive and reactive. This makes them human-like in the way they deal with problems[Age06]. The similarity makes their functionality easy to understand. Most agent platforms are built with a high abstraction level, which means that their programming language is close to natural language. This property facilitate the implementation of agents. Other benefits related to agent technology are reduced coupling, encapsulation of functionality and high degree of modularity[PW04a].

1.2 Problem Definition

The problem definition given by Statoil is as follows:

"*The focus in the master thesis would be to implement a prototype of a system based on agent technology. An ideal solution should illustrate the benefits of agent technology. The implementation could demonstrate potential application areas for agent technology in the future, and show that agents can be used to solve problems where traditionally development tools have shortcomings. The goal for the students of the master thesis is to obtain practical experience with agent technology and to evaluate the applicability of agent systems. Is the technology as good as asserted?*"

We have tried as good as possible to meet the challenges of the problem definition in our work with the master thesis.

1.3 Context

Our master thesis is written in cooperation with Statoil. Statoil is a Norwegian oil- and gas company with about 24,000 employees and activities in 29 countries. Statoil has their own IT Department which, among other things, work with development and maintenance of software systems used in the oil- and gas industry all around the world. To be competitive in the oil- and gas market, Statoil continuously explores and takes advantages of new technology.

Agent technology is a new technology in Statoil, and their IT Department stands behind the problem definition of this master thesis. It appears that the use of agents gives a valuable abstraction of systems under certain conditions. Statoil wants to learn more about agents and their advantages. The goal is to verify if the positive results and suggestions related to agents and their usage, are correct. These systems should be able to respond on state changes in the business and take action, either automatically or by assignment of a task to a user for manual processing. Statoil wants to serve their customers as well as possible, and has expectations that agents can improve their services.

1.4 Problem Approach

The objective of this master thesis is to investigate benefits claimed to be related to agent technology. The research will consist of a literature study, development of two multi-agent systems and construction and performance of an experiment.

Our prestudy uses foundings from Elin M Kristensen's depth study performed during the fall semester 2005. In addition have we concentrated about more practical issues. Development and implementation methods have been used to achieve a better understanding of the possibilities and the functionality provided by agents.

According to software literature and theory, agent technology provides certain benefits. To investigate some of these benefits, an experiment was constructed and performed. Two agent-based applications were developed as a part of the experiment. An application area with a complex and dynamic environment was selected in cooperation with our supervisors. The main difference between the systems is that one part is implemented as an object in the first version , and thereafter replaced with an agent in the other.

Development of two agent-based applications have given us experiences that will make us able to perform qualitative evaluations of agent technology. In addition, will quantitative evaluation be performed to provide results with internal and external validity.

1.5 Report Outline

The master thesis starts with a presentation of agent technology and development tools and methods. The experiment process is thereafter described. The process concerns experiment definition, planning, and evaluation of results. System development and examples of implementation will be presented. Finally at the end of the master thesis, experimental results and experiences with agent technology will be given. The organization of the master thesis is as follows:

Chapter 2: Software Agents To understand what agent technology covers, a description of the agent entity is given in this chapter. Agent technology is well

suited in specific application areas, and some of the domains will be presented. Agents have special properties, and there are many claimed benefits related to agent technology. Chapter 2 describes the benefits and presents previous experiences with agent technology.

Chapter 3: Development Methods and Tools Two multi-agent systems have been developed as part of the master thesis. This chapter describes the development methods and tools used in the development process. The methodologies and tools presented are *The Prometheus Methodology* and *JACK*.

Chapter 4: Application Area The application area has been chosen due to its special characteristics and complexity. This chapter introduces facts about the application area and describes the parts to be implemented. Our motivation for the choice is given in the end of the chapter.

Chapter 5: Experiment Approach An experiment has been constructed and performed to evaluate agent technology. Chapter 5 presents the experiment process and different parts of it. The experiment definition and planning are described, together with the experiment construction.

Chapter 6: System Development This chapter describes the system development process of the two agent-based applications. It contains the System Specification, the Architectural Design and Detailed design. Examples of implementation are given in the end of the chapter.

Chapter 7: Evaluation Testing of hypotheses has been performed as part of the experiment. The testing process and the results are presented in chapter 7. A description is given of how agent technology were used to develop multi-agent systems in relation to the chosen application area. The experiences with JACK an Prometheus and the validity of our results are discussed in the end of the chapter.

Chapter 8: Conclusion and Further Work The master thesis is concluded in chapter 8 and suggestions for further work are presented.

Part I

State of the Art

Chapter 2

Software Agents

An understanding of agent technology implies knowledge about the software entity *Agent*. The concept of agents has been around since the 1950's, but the current idea of what an agent is stems from the 1980's, when the properties of independence and autonomy became central to agent hood[Nnw96].

Section 2.1 gives an overview of agent properties. Due to these characteristics, agent-technology provides certain benefits, which is described in section 2.2.

Agent-based solutions are well suited in specific application areas. Examples of some application areas are given in section 2.3, with an argumentation of why multi-agent systems are appropriate. The last section, 2.4, presents previous experiences with agent technology.

2.1 Agent Characteristics

To grasp the extent of the concepts of software agents, it is necessary to identify their main properties and characteristics. There are many taxonomies that refer to different types of agent entities[Nnw96]. Following, one of the most adopted definitions of a software agent is given:

"An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives. An intelligent agent is in addition reactive, proactive and social[Woo02]".

This definition mentions the hallmarks autonomous, reactive and proactive. To make the difference between agents and objects visible, and to clarify the concept of agents, these hallmarks and some other properties will now be explained.

2.1.1 Autonomous

The property of autonomy indicates that agents are independent and make their own decisions[PW04a]. Agents have free will, which influences their behavior when they make decisions about what to do and with whom they want to cooperate. Internal states and goals form the autonomy, and give the agent ability to interact with the environment like a human with a specified goal[Nnw96].

2.1.2 Situated in an Environment

Almost all kind of software are situated in an environment. A possible way to distinguish between agents and objects, is to compare environments where they are appropriate. Agents tend to be used when the environment is dynamic, unpredictable and unreliable[PW04a]. Rapidly changes can occur in this type of environment, and therefore lead to difficulties in predicting the system's behavior on a case-by-case basis[MD02]. Agents can handle such changes in an appropriate way. The following definition illustrate the environment's impact on an agent's performance:

"An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future[FG96]."

Accordingly, the agents use the environment actively to reach its goals.

2.1.3 Reactive

Agents are reactive, and are therefor able to handle changing environments. Reactivity indicates that the agents will respond to perceived changes in the environment before a time limit is exceeded. Events represent changes which the agent must take into consideration when choosing a plan of action. The reaction will be in accordance with the agent's present goal[Woo02].

2.1.4 Proactive

The term proactiveness, indicates that an agents have a goal-oriented behavior. Goals make the foundation for taking rational decisions about what actions to take. The decisions take place when an agent senses changes in the environment and strive to reach a desired system state. Consequently, an agent can make several steps, in a sequence of actions, to reach different goals. In situations where the environment changes or the goal achievement fails, the agent will still try to find a way to reach the goal[PW04a], [Woo02].

2.1.5 Flexible

Agents have flexible behavior, because they can achieve goals in several ways. To reach a goal an agent chooses a plan to follow. If the plan fails for some reason, the agent can choose between other possibilities to obtain the desired result[PW04a].

2.1.6 Robust

To handle complex environments the agent need to be robust. Robustness can be described as the ability to persistently pursue goals even if a failure occurs. The property of robustness is closely related to flexibility. Flexibility introduces the opportunity of recovering from failure by picking an alternative plan[PW04a]. The agent will try all suitable plans until the goal has been achieved or all options are exhausted.

2.1.7 Social

Agents have the ability to collaborate and negotiate with each other to achieve goals. The interactions between agents can be described in terms of human behav-

Benefit	Description
Reduced coupling	No control point to external entities
Encapsulation of functionality	The Autonomy of agents leads to less communication.
High degree of modularity	Easier to modify and expand
Human reasoning	Provided by the BDI model. Substitution of humans.
High abstraction level	Make modeling and implementation easier.
Challenging environments	Handle complex and dynamic in the environment

Table 2.1: Benefits of Agent Technology

ior. The most common interaction types are teamwork, negotiation and coordination[PW04a]. Agents can for instance perform teamwork when several agents have the same goal. Sometimes agents can negotiate to obtain the right conditions in the environment. Reasoning about goals and conditions, and the ability to contact other agents to achieve a goal, make agents social.

2.2 Claimed Benefits of Agent Technology

Despite agent technology is a young field, there exist many claimed benefits about agent technology in the literature. This section will describe the main benefits, and will serve as a foundation for further investigation.

One important benefit of agent technology is that agents *reduce coupling*[PW04a]. They do not provide any control point to external entities. Agents are autonomous, which means that they control how to deal with the messages they receive. The autonomy lead to *encapsulation of functionality*. In addition, agents can be given responsibility and be relied upon in achieving goals, which result in less communication.

Reduced coupling cause more *modular* software systems. Modular systems are easy to expand, by adding several modules. They are changeable because modifications in one module do not cause ripple effects to other modules. Agents can also be used as glue between software applications[PW04a].

In some applications agents can substitute for humans. The reactiveness and proactiveness of agents make them more human-like in how they deal with problems. The Belief-Desire-Intent(BDI) model is used to model agent intelligence by mimicking *human reasoning*[PW04a]. Some agent frameworks provide special agent-languages that take advantage of the BDI-property and make agents more easy to implement. Since the BDI model is based on human reasoning, it constitute a natural paradigm for implementing intelligence. Some agent frameworks take advantage of this and provides a special agent language to fully capitalize on this benefit. These frameworks make programming easier because they are custom-built for the purpose of agents, and the models are close to human thinking. These properties are also characteristics of a model with a *a high abstraction level*[Ølm05].

Agents are clearly advantageous in situations with *challenging environments*. The autonomy of agents bring along the possibility of recovering from failure, making them suitable for unpredictable and unreliable environments. Agents can sense changes in the environment and choose actions based on these changes[PW04a].

The benefits of agent technology are summarized in table 2.1.

2.3 Application Areas

Agents are well suited to solve special kind of problems. This section will present suggested application areas for system development.

2.3.1 Planning and Scheduling

Within logistic and transportation, the most important processes are scheduling, forecasting and dispatching. The processes concern, among other factors, handling of incoming orders and construction of cost optimal transportation plans. The performance of these processes affects the company's income and costs. For instance by focusing on delivery time, quality and services, one can obtain satisfied customers and gain new ones, which again result in more profit[HB04].

Agents are able to monitor situations to make a model of observed performance, which could prepare the company for future situations. The predicted performance can be useful when making plans. Agents can be implemented with multiple plans and choose the most appropriate direction of action for the situation at hand. They are also able to reconsider their direction when changes occur. The latter properties make agents well suited for planning situations.

2.3.2 Business Process Systems

A business process consists of one or several activities that produce results valuable to the enterprise, its stakeholders or its customers[Wik06]. The activities in a business process involve receiving inputs, treat the input with the right method and produce output of the treatment. Today the business processes are becoming more complex and need to adapt to changing circumstances. The traditional models which design the processes before the system is developed, have proved to fail because they are not dynamic[WW05].

Agents can offer dynamic modeling of the environment. They are goal-oriented and flexible, which means that they are able to achieve goals under changing circumstances. There is no need to predetermine the order of activities, because an agent-system can manage the business processes due to their situational awareness. A multi-agent system can handle the current business environment and the underlying business polices. Agent's nature will lead to real-time decisions and coordination of tasks[Kri05].

2.3.3 Decision Support

Decision support involves how to achieve better and faster decisions. By getting the right information to the right people in right time, people involved can come to a good decision faster[Kea04].

Multi-agent systems can be used to gather information from the environment, by monitoring operations, sensors and detectors. To make a decision, information from different sources need to be handled and analysed. A multi-agent system could consist of several agents acting as information seekers that cooperate with agents having an analyst role. Together they can serve as a link between humans and the rest of the system. The decision team could consist of both agents and humans. Agents can propose different alternatives based on the processed information, present this to the user and let the user take the final decision based on experience.

Multi-agent systems can continuously process information from situations and report this to the user. They can gather appropriate knowledge dependent on the events they receive in short time. This could support the users of the system in making proactive decisions, instead of reactive[Kri05].

2.3.4 Critical Situations

In critical situations, time is essential. Minutes and seconds can make a difference between a scare and a catastrophe. The emergency organization has to be formed as fast as possible. Humans get stressed in emergency situations, and can overlook critical events.

Agents have the ability to handle events fast and in accordance with the environment. Multi-agent systems are favorable to help people react faster and more rational in critical situations[Kri05]. Agents can be part of the control system to monitor situations. When they receive an unexpected event, they can gather the right knowledge and make the right people receive the messages. Agents can also coordinate the resources needed to handle the situation[Kri05].

2.3.5 Task Automation

Task automation denotes that a system is capable of performing tasks triggered from internal events without requiring user input[Ølm05]. Tasks that are repeated often and are complicated or tedious to perform, are good candidates for automation. Automation could simplify the work for people in several situations.

Software agents are ideal for automating tasks since they can react to conditions in the system or initiate the appropriate actions in a proactive manner[Ølm05]. As an example, agents can be implemented to perform search and filtering of information in accordance to specific criteria. Agents are especially useful in generating proposals, based on knowledge processing, for the user to verify and complete. Another situation where task automation can be practical is in relation to maintenance and monitoring routines that must be initiated frequently. Agents could be responsible for initiating these routines with a predetermined time setting.

2.4 Previous Experiences

The purpose of this section is to get an overview of earlier experiences and research results. Previous experiences can serve as a foundation for our research strategy. The results from the experiences can be valuable material for a comparison.

2.4.1 An Example of an Evaluation of a Multi-Agent System

A student at the RMIT Computer Science Department in Melbourne, Australia, has done a comparative analysis of agents and state machines in order to understand how the choice of technology can affect factors as performance, complexity and estimated efforts. The experiment consisted of developing an agent model and a state machine model that where both JAVA based[Bar01].

The results from this study indicate that the state machine implementation has a higher degree of effort in addition to a longer duration. These implications are only estimates and almost totally based on lines of code(LOC). Nevertheless, the results show that using an agent model may *save development time*[Bar01].

Benefit	Description
Save development time	The agent model is estimated to have less code, and therefore shorter time to implement.
Less code	The State Machine model required about 10% more lines of code
Fewer logical errors	Less code, imply fewer errors
Less modification effort	The tendencies show that the modification duration for a state machine module are double that of the agent module.

Table 2.2: Implicated Benefits in RMIT Study

The study also observed that the State Machine model required about 10% more lines of code. This implicates that an agent implementation would lead to *less code* and *fewer logical errors*.

The implementation of the state machine had an increase of modification effort that was more than double than that of the agent. This was used to conclude that the effort would be far greater for the state machine if the system were to be extended[Bar01]. Table 2.2 summarizes the benefits observed in this study.

2.4.2 Multi-Agent System at Sidney Airport

A multi-agent system called OASIS[OAS92] has been used at Sidney airport to make air traffic congestion more easy to manage. OASIS is agent-oriented, and consists of independent agents that solve parts of an overall problem. The distribution of co-operating agents is the central design principle for the system. The system achieves to maximize runway utilization through arranging landing aircraft into an optimal order, assign them a landing time, and then monitor the progress of each aircraft in real time[OAS92].

Observation of the multi-agent system indicates that an agent based tool did not outperform the best air traffic controller, but the performance of the average controllers increased. Therefore, it can be said that the system introduced an overall improvement of performance.

2.4.3 Autonomous Manufacturing Architecture

The Agent Oriented Software Group won in November 2004, the UK Trade & Investment Award, for ”Best New Business Entrant” in UK. They seek to develop leading application products based on JACK agent technology. They have come up with an example that illustrates an innovative way in how to make a manufacturing architecture. Usually, manufacturing architectures consist of robots that transfer a ”part” between them. The alternative architecture has made the ”part” autonomous. The part is represented by an agent, which run on a computer and become active in the decision making. The part-agent communicates with robot controllers via a real-time blackboard. This bring along that the part could send and receive messages directly to the controllers, and answer questions about it self, without needing any external robot investigating its properties.

This example shows that agents are appropriate in situations where a passive object is transfered between active objects. The passive object can be made autonomous and then communicate with the other active objects[Luk]. The gain from this modification are a system with a more dynamic structure and with a shorter reaction

time on unforeseen events. There will be no need for frequently inspection of the object, because the object is able to signal its needs.

Chapter 3

Development Method and Tools

The Prometheus methodology and JACK have been used to develop our two multi-agent systems. The Prometheus methodology provides guidelines for specifying, designing, implementing and testing agent-oriented software systems[PW04b]. The methodology consists of three phases, each outlined in section 3.1. Section 3.2 introduces JACK, which is the agent platform we will use in our implementation.

3.1 Prometheus Methodology

The Prometheus methodology[PW04b] captures the process of developing an agent-oriented system. Design artifacts are used to capture information about the system and its design. The process consists of three phases; the *system specification phase*, *architectural design phase* and *detailed design phase*. The phases will be described in the remainder of this section.

3.1.1 System Specification

The focus of the system specification phase[PW04b][PTW05] is to identify the *goals* and basic *roles*, in addition to *percepts*(input from environment) and *actions*(output to environment). The graphical representation of these artifacts are given in figure 3.1, and related descriptions will now be given.

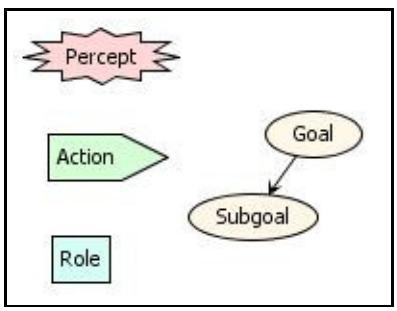

Figure 3.1: Artifacts in System Specification

Actions and Percepts

These issues, rise questions about what input will be available to the agent system from the environment, and what will the agent system do to interact with and affect the environment. Inputs to agent systems are identified as *percepts*, and outputs from the system are identified as *actions*[PTW05].

Goals

The goals [PW04c] of the system should indicate what the system is supposed to do. The identified goals will result in an initial list of system goals. The goals are thereafter refined into subgoals.

Scenarios

Scenarios describe the sequences of progress within the system. Each scenario generates a goal. More specific goals may not require a separate scenario. Scenarios are used primarily to illustrate the normal running of the system, but they can also be useful to indicate what is expected to happen when something goes wrong. The identified scenarios consist of a number of detailed steps, where each step can be a *goal*, *scenario*, *action* or *percept*.

Roles

Roles [PTW05] consist of a grouping of related goals. The term is used for a chunk of behavior the system is going to meet. Each role can be described with *triggers*, which is information about the events and situations that will cause the activity of the role to be activated. The trigger can contain *percepts*, but also prerequisites in the environment. Each role should have percepts or actions allocated to it.

3.1.2 Architectural Design

The Architectural Design phase is based on the artifacts from the *System Specification*. The phase defines the *agent types* and the interaction that will be included in the system[PTW05].

Specifying the Agent Types

This aspect of the *Architectural Design*[PW04d] focuses on the process of deciding which agent types to be used. Agent types are formed by combining roles. Alternative groupings should be considered and evaluated by the the standard software engineering criteria of *coupling* and *cohesion*.

In Agent systems coupling is exhibited primarily in communication between agents, although use of a shared data store is another possible form of coupling[PW04d]. The criteria is to aim for a system that is as loosely coupled as possible. It is not preferable that all agents have to know about all other agents. Cohesion is a property of a single component, and exists if all of the components parents are related. Most often, cohesion in an agent is based on the goals of the agent being closely related[PW04d].

After the suggested design alternatives are reviewed, the most appropriate is pursued.

Specifying the Interactions

This phase considers the dynamic aspects of the system, and focus on designing the interaction between agents. The interaction diagrams are developed by identifying which agents perform the steps in each scenario, see section 3.1.1. The messages that agents exchange to maintain the right sequencing are also considered.

Designing the System Structure

The information about the agent types and the communication between them are brought together in this phase and gathered in a *System Overview Diagram*. This diagram captures the overall architecture of the agent system.

The *System Overview Diagram* contains the agents and their way of communication in term of message exchange, persistent data stores, percepts and actions. A graphical representation of artifacts in the *System Overview Diagram* is given in figure 3.2.

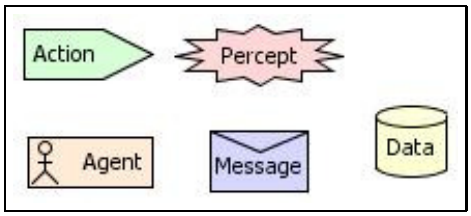

Figure 3.2: Artifacts in Architectural Design

3.1.3 Detailed Design

The Detailed Design phase uses artifacts produced in the *Architectural Design Phase* to define the internals of every agent in the system and to specify how agents accomplish their overall tasks. The internals of each agent are their capabilities, plans, events and data structures. Figure 3.3 show the graphical representation of those artifacts. This phase should provide the information necessary to start the implementation.

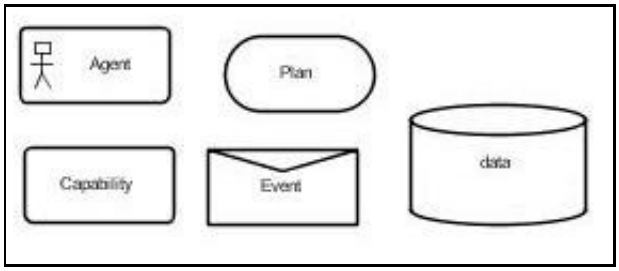

Figure 3.3: Artifacts in Detailed Design

Capabilities

The first part of detailed design deals with the capabilities an agent needs to perform its tasks. Roles are a natural starting point for identifying capabilities. Routine tasks that is required multiple places can be captured in a capability, and be comprised by multiple agents. Each capability should be a well-defined collection of plans, using particular beliefs or data, which address a specific set of goals for the agent. An *Agent overview diagram* shows the relationships between the capabilities of an agent[PW04e].

Events and Plans

The detailed design process continues with focusing on the plans and events generated and handled within each capability. Each incoming message to the capability must have one or more plans to respond to the message. Multiple plans responding to the message provide multiple ways of reacting to a situation. Plans are parts of the specification of the dynamic behavior of the system.

Each plan is triggered by an *event*, which can be the arrival of a percept, arrival of a message from another agent, or an internal message[PW04f]. The plans and events within each capability results in a *Capability Overview Diagram*.

Data

It is important to ensure that all significant data structures are well specified, together with their location. JACK offers some specialized support for data representation, among others *beliefsets*, *views* and resource management[PW04f].

Beliefsets are used in JACK to maintain an agent's beliefs about the world. It describes a set of beliefs that the agent may have in terms of fields. Beliefsets include keys, fields and query methods[Age06].

Views in Jack, provide the means to integrate a wide range of data sources within the JACK framework. The view construct allows general purpose queries to be made about an underlying data model. The data model may be implemented using multiple beliefsets or arbitrary Java data structures[Age06]. For instance can a *Graphical User Interface(GUI)* be wrapped in a view.

Semaphore is a synchronization resource which can be used to establish mutual exclusion regions of processing in JACK plans and threads. A semaphore is a binary resource that plans and threads may wait for, and signal when they have completed[Age06].

These concepts are all represented as *Named data* in the Detailed design diagrams.

3.2 JACK

A range of *agent oriented* programming languages exist today. We will use an agent platform called JACK to develop our agent system. JACK is a product developed by *The Agent Oriented Software Group (AOS)*, and provides tools required to develop autonomous software systems that are both goal-directed and reactive. The JACK Agent Language is a high level programming language that extends Java with agent-oriented concepts[Age06], such as:

- Agents

Entity	File extension	Usage
JACK event	.event	JACK event definition.
JACK plan	.plan	JACK plan definition.
JACK agent	.agent	JACK agent definition.
JACK capability	.cap	JACK capability definition.
JACK view	.view	JACK view definition.
JACK beliefset	.bel	JACK beliefset definition.
Java class	.java	Java class or interface definition.

Table 3.1: JACK Entities

- Capabilities
- Events
- Plans
- Data representations
- Resource and Concurrency Management

Many of the Prometheus concepts map directly to JACK. For instance do a prometheus-agent map into a JACK-agent, and so do the capabilities from the design. Much of the code can be directly generated from a detailed design. The JACK Intelligent Agent environment contains a tool, *Jack Development Environment(JDE)*, which supports the generation of skeletons for JACK code from the detailed design[PW04g].

When developing a JACK application, source code will be created for some or all of the agent oriented entities in table 3.1. Since JACK extends Java with agent-oriented entities, JACK source code is first compiled into regular Java code before being executed. The files that are created for these entities must have the same base name as the entity defined in the file[Age06].

In addition, every JACK application must have a Java class that contains the main method. This class will be the entry point for the Java Virtual Machine and any other Java-file required by this application[Age06].

Chapter 4

Application Area

Claimed benefits and applicability of agent technology have been related to many application areas. Common descriptions for these areas are dynamic environment, handling of complex processes or need for high level reasoning[MD02].

We have found a proper application area to evaluate agent technology. Our chosen application area is jettyplanning at Mongstad terminal and refinery. Section 4.1 gives facts about Mongstad and section 4.2 describes the jettyplanning process.

The motivation for our choice is based on characteristics of the application area and the claimed contribution of functionality a multi-agent system can provide in such a setting. Our reason for selecting jettyplanning as application area is described in section 4.3.

4.1 Facts about Mongstad

One of Statoil's refineries in Norway is situated at Mongstad near Bergen. It provides intermediate storage for more than a third of all oil produced by Statoil in Norway, and has a number of berths where shipment of products and crude oil take place[Sta06].

Oil is brought to Mongstad in shuttle tankers from the Heidrun platform, and through pipelines from the Troll platforms in the North Sea. The crude oil is thereafter exported from the terminal. Mongstad also provides storage capability[Sta06].

Facts on the terminal:

- Six rock cavern stores with a total capacity of 9.4 million barrels
- Two jetties able to handle crude oil and product carriers up to 380,000 dead weight tonnes
- One ship-to-ship jetty able to handle crude oil carriers up to 440,000 dead weight tonnes

Mongstad serves as the terminal for:

- Troll Oil Pipeline I, about 250,000 barrels per day
- Troll Oil Pipeline II, about 150,000 barrels per day
- Heidrun transshipment, about 240,000 barrels per day

Crude oil is also received for transshipment from Statoil-operated fields as Gullfaks, Norne and Åsgard.

4.2 Jettyplanning

The number of jetties at Mongstad restricts the loading capacity of products and crude-oil. The jetties can be characterized as limited resources. To avoid demurrage and extra costs, a preferable utilization of the jetties is as optimized as possible.

One jetty has several berths where ships can receive oil products. Jettyplanning concerns allocation of berths to incoming ships. A description of the jettyplanning process will now be given, followed by present practice in Statoil and an example of a jettyplanning system.

4.2.1 Description of the Jettyplanning Process

Many issues need to be considered when berths are allocated to incoming ships. The following steps describes briefly the jettyplanning process.

- A trader makes an agreement with one of Statoil's partners. The agreement contains information about the shipment of the cargo.
- A ship is delegated to export the oil from Mongstad.
- To receive the oil product the ship needs a port of discharge.
- The berth allocation is given based on shipment information such as arrival date, type of cargo, docking time, the size of the ship, and so on.
- In the berth allocation process, other ships and their allocations also need to be considered.
- An ideal situation is to obtain an optimal allocation of berths to avoid delays and loss of money.
- Unforeseen events related to ships and berths need to be considered, which means that the plan for berth allocation is dynamic.
- When a ship arrives at Mongstad it docks to the given berth and receives the cargo.
- The ship is moved from the schedule when the ship leaves Mongstad.

4.2.2 Present Practice at Mongstad

The jettyplanning process at Mongstad has no tailor-made software system. People responsible for the jettyplanning use Excel-worksheets connected to databases to consider the placement and time allocation for incoming ships before registration.

The partly manual system can be difficult to manage. Complex calculations can be necessary when the number of incoming ships reach a great number. To make the total cost minimal and to be able to make the best decision, many alternative calculations for each ship is required[LO06].

4.2.3 Example of a Jettyplanning System; Seaberth

Statoil has considered a software system called *Seaberth* as an alternative to their partly manual system used today. Seaberth is created by Cirrus Logistics and is used worldwide to assist efficient scheduling and processing of ships[Cir06].

In Seaberth, scheduling is based on the system seeking the best berthing solution within a number of location and user defined, rules and constraints. It seeks to provide the best possible berth plan to meet the prevailing business objectives. The tool helps users make sound scheduling decisions. These decisions are not only based on the immediate requirement of one event, but also on the context of current and future planned operations for the whole facility[Cir06].

Seaberth is the only product of its kind and has a lot of functionality. The system is very expensive. The high cost related to implementation and integration is probably the reason why Statoil has decided not to buy Seaberth[LO06].

4.3 Motivation for Selection of Application Area

Our motivation for selecting jettyplanning as application area is based on its characteristics. Our choice is also influenced by assertions about how multi-agent systems can deal with challenges related to these characteristics. The characteristics and the assertions will now be described.

4.3.1 Application Area Characteristics

Jettyplanning can be characterized by three concepts; planning, scheduling and decision making. Each of these concepts occupies amounts of resources. To obtain a good result, they need to be handled with special consideration.

Planning is hard in changing environments, and the number and frequency of changes make plans outdated in short time. Humans have to spend a lot of time to manually keep the plans up to date. The processes at Mongstad can be characterized as non-linear and dynamic, and they take place in a changing environment. Near real time, from batch to flow, in the right level of detail and with use of simple rules, are some of the main requirements for the planning part[HBH05].

Scheduling is concerned with finding the best possible allocation of berths to achieve the lowest cost at Mongstad. To choose the best allocation alternative, a huge amount of information is required. *Decision making* is used to actually choose the best allocation alternative, and is therefore strongly related to this part.

NP-complete problems can be described as problems which can not be solved in polynomial-time by any algorithm. The presence of the three following key concepts can be used to show that we have a NP-complete problem[THC01].

- *Optimization problems:*Each feasible solution has an associated value, and we want to find the feasible solution with the best value.
- *Decision problems:* The answer can be "yes" or "no", and the decision may change from time to time.
- *Reduction:* Not possible to reduce one problem into another

A jettyplanner application can be concerned with optimization of berth allocation during scheduling. Scheduling is related to decision making. When a new ship

arrives, a berth allocation may lead to reallocation of other ships. Jettyplanning can therefore in some cases be described as a NP-complete problem if complete optimization is required. A consequence of complete optimization is that decision making is needed to avoid a never ending loop, trying to find all possible allocation alternatives.

4.3.2 Agents, Scheduling and Decision Making

It is asserted that agents are especially useful in situations where complex problems need to be handled, or when an exact execution order of activities may not be practical. Agents are proactive, flexible and reactive and try to make the best out of any situation. They can not solve NP-complete problems, but they will try to choose the best solution path in accordance to their current environment [PW04a, Woo02].

Agents use their capability of cooperation to achieve their common goals, which in this case are nearly optimized jetty utilization, loading and maintenance. Their nature leads to real-time decisions and coordination of tasks[PW04a, Woo02]. Jettyplanning consists of complex processes in a dynamic environment, and is therefore a proper application area for our multi-agent system.

Part II

Own contribution

Chapter 5

Experiment Approach

Experiments are a valuable tool in evaluating new software methods and techniques. A good experiment needs to be prepared, conducted and analysed properly to give valid results[CW00].

The process of performing an experiment can be divided into different main activities. The experiment process is described in section 5.1.

The first step in the process is to define the experiment. A definition is necessary to ensure that important aspects are considered. Our experiment definition is given in section 5.2.

After defining the experiment, the planning take place. The definition gives the foundation for the experiment, while the planning prepares for how the experiment is conducted. Our plan for the experiment is described in section 5.3. The related experiment construction is presented in section 5.4.

5.1 Experiment Process

Results from our study will be of both qualitative and quantitative character. *Quantitative results* are most preferable, because conclusions can be drawn from comparisons and statistical analysis. *Qualitative data*, for instance explanations from people, can give valuable information that support the quantitative results[CW00].

Statoil's solution for jettyplanning is partly manual and a comparison between the current system and a multi-agent system would not be consistent nor practicable. Our research strategy is therefore to perform a *Quasi-experiment*. Quasi experiments can not be called true experiments, because the participants or the objects to be evaluated are not selected by random[CW00]. The objects to be evaluated in our experiment are two implemented multi-agent systems for jettyplanning.

We will use the experiment process suggested by Claes Wohlin et al[CW00], which consists of the following steps:

- **Experiment Definition**
  The experiment is defined in terms of problems, objective and goals.
- **Experiment Planning**
  The design and instrumentation is determined. Possible threats are evaluated.
- **Experiment Operation**
  The subjects to be evaluated is prepared. The experiment is being executed

and data is collected.

- **Analysis and Interpretation**
  Measured data is analysed and gathered in descriptive statics.

Each step defines important aspects necessary for our research. We will use them as a guideline to assure we perform the experiment as good as possible. The two first steps are described section 5.2 and 5.3. The work related to the step *Experiment Operation* is described in chapter 6, and the *Analysis* is given in chapter 7.

5.2 Experiment Definition

The purpose of a goal definition is to ensure that important aspects of an experiment are defined before the planning and execution take place. We have decided to use the "Goal Question Metric"(GQM) template to define our goal.

The GQM template is taken from the book "Experimentation in Software Engineering" [CW00] and is written as follows:

Analyse <*Object(s) of study*>
for the purpose of <*Purpose*>
with respect to their <*Quality focus*>
from the point of view of the <*Perspective*>
in the context of <*Context*>

Objects of Study The objects of study are the entities that are studied in the experiment. We will compare two different multi-agent implementations of a jettyplanner system. See section 5.3 for a brief description of the two systems.

Purpose The purpose defines the intention of the experiment. We want to evaluate the benefits and applicability of agent technology. We believe we can measure the applicability and the benefits by changing ship objects within the system to ship agents, and then evaluate the result and the consequences of these changes.

Quality Focus Quality focus is the primary effect under study in the experiment. We will have focus on benefits and applicability.

Perspective The perspective presents the viewpoint from which the experiment results are interpreted. We have chosen a developer perspective.

Context The context is the environment in which the experiment is run. It defines the personnel involved in the experiment(subjects) and the software artifacts involved(objects). In our experiment we as students are the subjects. The objects of context are the development tools described in chapter 3.

Our definition for the experiment then turns out as follows:

Analyse *two different implementations of the Jettyplanner*
for the purpose of *evaluation*
with respect to *benefits and applicability of agent technology*
from the point of view of *developers*
in the context of *students using Jack and Prometheus to implement a multi-agent system.*

5.3 Experiment Planning

The experiment planning prepares for how the experiment is conducted. This section will present our context selection and hypothesis formulations. The selection of variables and subjects are described, followed by our experiment design.

5.3.1 Application Area Selection

Our application area is jettyplanning at Mongstad as described in chapter 4. Statoil's current system at Mongstad is partly manual, and it is used in a step by step fashion where humans take all the important decisions.

We have decided to implement two multi-agent systems for jettyplanning. A comparison can be performed based on these two versions. The versions are as follows.

- JettyPlanner1: Ships are represented as objects within the system.
- JettyPlanner2: Ships are represented as agents within the system.

Both versions have agent types like *JettyPlanner*, *Berth* and *GuiManager*. For detailed system design, see section 6.3.

In JettyPlanner2, the ship-object is changed into a ship-agent, which means it gets the agent properties described in chapter 2. We want to compare the differences between the two versions to discover possible benefits and applicability of agent technology.

5.3.2 Hypothesis Formulation

Testing of hypothesis make the foundation for analysis in an experiment. A hypothesis is stated formally and the data collected during the course of the experiment is used to, if possible, reject the hypothesis.

To evaluate the two multi-agent systems, we have written a number of hypotheses statement related to agent technology. For each issue we want to evaluate, we have formulated two kind of hypotheses, a null hypothesis(*H0*) and one or two alternative hypothesis(*HA1 or HA2*). The null hypothesis states that there are no real underlying trends or patterns in the experiment setting. If the null hypothesis is rejected, one of the alternative hypothesis is chosen[CW00].

The hypothesis tries to characterize aspects of multi-agent systems, and can therefore be related to questions in the GQM approach[CW00]. Table 5.1 presents the hypothesis.

Id	Hypothesis
H01	The functionality of the the two versions will be implemented with approximately the same number of code lines.
HA1.1	JettyPlanner2 will implement the same functionality as JettyPlanner1 with fewer lines of code.
HA1.2	JettyPlanner2 will implement the same functionality as JettyPlanner1 with several lines of code.
H02	The number of entities will be the same for the two JettyPlanner versions.
HA2.1	JettyPlanner2 will have more entities than JettyPlanner1.
HA2.2	JettyPlanner2 will have less entities than JettyPlanner1.
H03	Both versions will use the same number of functions to complete the different use-cases described in chapter 6.
HA3.1	JettyPlanner2 will complete the use-cases with fewer functions than JettyPlanner1.
HA3.2	JettyPlanner2 will complete the use-cases with several functions than JettyPlanner1.
H04	Both versions will have the same number of couplings between the components in the system.
HA4.1	JettyPlanner2 will have fewer couplings between the components than JettyPlanner1.
HA4.2	JettyPlanner2 will have several couplings between the components than JettyPlanner1.
H05	JettyPlanner1 and JettyPlanner2 have the same number of external operations changing their internal state.
HA5.1	JettyPlanner2 has a fewer amount of external operations changing the internal state than JettyPlanner1.
HA5.2	JettyPlanner2 has several external operations changing the internal state than JettyPlanner1.
H06	Use of agent-technology will not provide a higher abstraction level for modeling and implementation of applications.
HA6.1	Use of agent-technology will provide a higher abstraction level for modeling and implementation of applications.

Table 5.1: Formulated Hypothesis

Some of the hypothesis require an *objective measure*, a value only dependent of the object that is being measured. Other attributes have *subjective measures*, were judgment from people come into account. The testing of the hypothesis is given in chapter 7.

5.3.3 Experiment Design

An experiment design consists of a series of tests of the treatments[CW00]. Our experiment design consists of one factor, which is the jettyplanner system, and two treatments, namely JettyPlanner1 and JettyPlanner2. We want to compare the two treatments against each other.

5.3.4 Validity Evaluation

One important question during the whole experiment process is the validity of the results.

Wohlin et al. proposes four types of threats to the validity of experimental results[CW00]. The threats are related to *conclusion-, internal-, construct-* and *external validity*. Figure 5.1 illustrates how each threat is related to different parts of the experiment process.

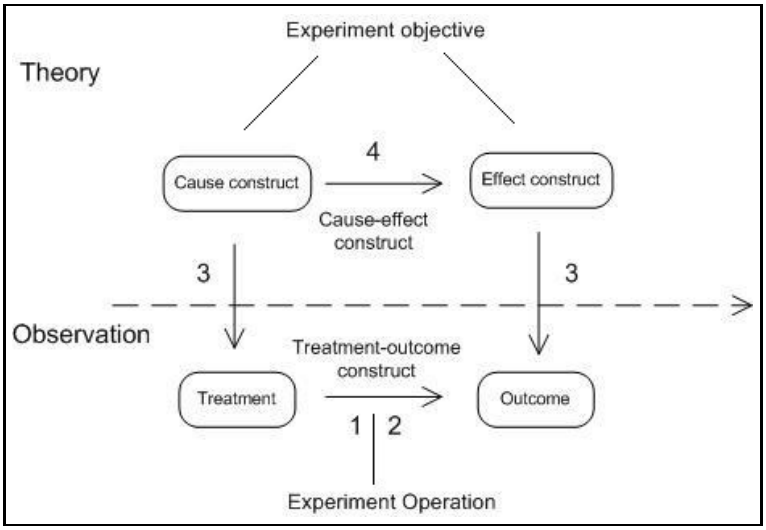

Figure 5.1: Experiment Principles[CW00]

The figure is divided in two parts, the *Theory area* and the *Observation area*. We want to draw conclusions about the theory defined in the hypothesis, based on our observations.

1. **Conclusion Validity** This validity is concerned with the relationship between the treatment and outcome. Conclusion validity means that there is some kind of statistical relationship with a given significance.

2. **Internal Validity** This validity is concerned with the causal relationship between the observed treatment and the outcome. Internal validity means that the treatment causes the outcome.

3. **Construct Validity** This validity concerns the relation between theory and observation. If we have construct validity the treatment reflects the construct of cause and the outcome reflects the effect construct.

4. **External Validity** This validity is concerned with generalization. We have external validity if the cause-effect construct has been generalized and the same results has been achieved.

Validity Threats

We have tried to find possible validity threats related to our evaluation. We have to address, or in some cases, accept these threats to obtain a valid result. We have found the following threats applicable in our evaluation.

Threats to Conclusion Validity:

Low statistical power It can be difficult to reveal a true pattern in our data since we only will perform one comparison of two multi-agent systems. The two systems only present a small fraction of the diversity of multi-agent systems. We will accept this threat and take into account that some of our conclusions can be erroneous.

Fishing We will perform a quasi-experiment which means that the objects to be evaluated are not selected by random. We must be careful to not influence the result by implementing specific outcomes. We will address this threat by implementing the two versions without thinking of preferable results.

Reliability of measures The reliability of measures are dependent of how we decide to perform different measurements. Objective measurements are more reliable than subjective measurements. We will address this threat by using metrics that involve a small degree of human judgment.

Threats to Internal Validity:

Maturation Maturation is the effect of that the subjects, in our case ourselves, will react differently as time passes. The two versions of our Jettyplanner application will be implemented at different time intervals and will therefore be affected by our knowledge about agent technology. We will accept this threat and take into account that our maturation of knowledge can influence the causal relationship between treatment and outcome.

Selection We only have one treatment to evaluate which means that our selection of objects may not be representative for all possible outcomes. We will accept this threat and take into account that our conclusion may not contain all possible outcomes.

Threats to Construction Validity:

Poor experiment constructions The experiment is dependent of the its constructions. If the constructions are not sufficiently defined, it will be difficult to generalize the result of the experiment to the theory behind the experiment. We will address this threat and define measurements and treatments as good as possible.

Mono-operation bias Our experiment may under-represent the construct and may not give the full picture of the theory because we have a quasi experiment. We will accept this threat and take into account that our construction may not give the full picture of the theory.

Confounding constructs A low level of construct can affect the outcome. The cause construct in the theory must be reflected in the treatment and the effect construct must be reflected in the outcome. We will address this threat by constructing an experiment with a clear relation between theory and observation.

Threats to External Validity:

Interaction of selection and treatment A condition that limits our ability to generalize the results of our experiment, is that we may not be representative of the population we want to generalize to, namely the developers. We will accept this threat and take into account that we are only students with lack of experience.

Interaction of setting and treatment The experimental setting and the material shall be representative for the industrial in practice to make the experiment generalizable. We will address this threat by using development tools and methods that are up to date.

5.4 Experiment Construction

The experiment construction connects hypothesis and metrics with benefits. The result is presented in this section, and can be characterized as the underlying structure for our experiment.

5.4.1 Benefits

We will examine some of the claimed benefits for agent technology by relating them to relevant hypothesis. The benefits we want to examine are as follows.

- Reduced development effort
- Reduced coupling
- Encapsulation of functionality
- High abstraction level

We will use *reduced development effort* as a generic term for reduced amount of code, number of entities and functions. Reduced development effort was given as one of the benefits in section 2.4. *Reduced coupling, encapsulation of functionality* and *high abstraction level* are benefits described in section 2.2.

5.4.2 Metrics

Quantitative research is concerned with quantifying a relationship or comparing two or more groups. Entities to be measured are denoted the term metrics[CW00]. The following quantitative metrics will be used in the evaluation:

M1 **Lines of Code (LOC)**
The number of written code-lines. Code-lines will be counted as the number of semicolons in the source-code.

M2 **Number of Entities (NOE)**
The number of events, plans, capabilities, agents, views, beliefsets and java-classes.

M3 **Number of Functions(NOF)**
The number of get/set methods, other methods and plans used by the ship-object and the ship agent.

M4 **Number of Couplings between Entities (NOCBE)**
Couplings in Jettyplanner1 will be defined as external method-calls to the ship-object from other entities within the system and method-calls from the ship-object to other entities. Couplings in Jettyplanner2 will in addition to in- and outgoing method-calls, be events sent and received by the ship agent.

M5 **Number of External Activations(NOEA)**
External activations in Jettyplanner1 will be defined as external method-calls to the ship-object from other entities. External activations in Jettyplanner2 will be external method-calls and received events.

Benefits	Hypothesis	Metrics
Development Effort	H01/HA1.1/HA1.2 H02/HA2.1/HA2.2 H03/HA3.1/HA3.2	M1(LOC) M2(NOE) M3(NOF)
Reduced coupling	H04/HA4.1/HA4.2	M4(NOCBE)
Encapsulation of functionality	H05/HA5.1/HA5.2	M5(NOEA)
High abstraction level	H06/HA6.1	Qualitative Result

Table 5.2: Benefits, Hypothesis and Metrics

5.4.3 The Relation between Benefits, Hypothesis and Metrics

The benefits, hypothesis and metrics can be connected with each other. The relations indicate how the hypotheses and the metrics will be used to investigate possible benefits in the evaluation. Table 5.2 gives an overview of these relations.

Chapter 6

System Development

We have used the development methods and tools described in chapter 3 to design and implement our two versions of the JettyPlanner. The difference between the two versions is that ships are represented as java objects in JettyPlanner1 and JACK agents in JettyPlanner2. We started with the implementation of JettyPlanner1, and then made the necessary modifications and extensions for JettyPlanner2. Section 6.1 contains the System Specification, section 6.2 the Architectural Design diagrams and section 6.3 the Detailed Design Diagrams. Section 6.4 will give code examples, to illustrate how agent entities are implemented in JACK. We will basically describe JettyPlanner1, but in the cases where the two systems differ, we will also describe the approach to JettyPlanner2.

6.1 System Specification

The goals of the system is given in section 6.1.1 and indicate the main requirements for the Jettyplanner. Section 6.1.2 combines goals into groups of functionality, also called roles. The system specification is similar for JettyPlanner1 and JettyPlanner2. Artifacts used in the diagrams was explained in section 3.1.1

6.1.1 Goals

Agents are proactive and have goals they want to achieve. We have identified system goals to obtain an overview of what the JettyPlanner should be able to do.

The main goal of the Jettyplanner is to *Optimize jetty allocation*. This means that the agents try to find the best solution for the current situation, and make the jetty allocation as good as possible. This goal was further refined into sub-goals, as depicted in the figure 6.1

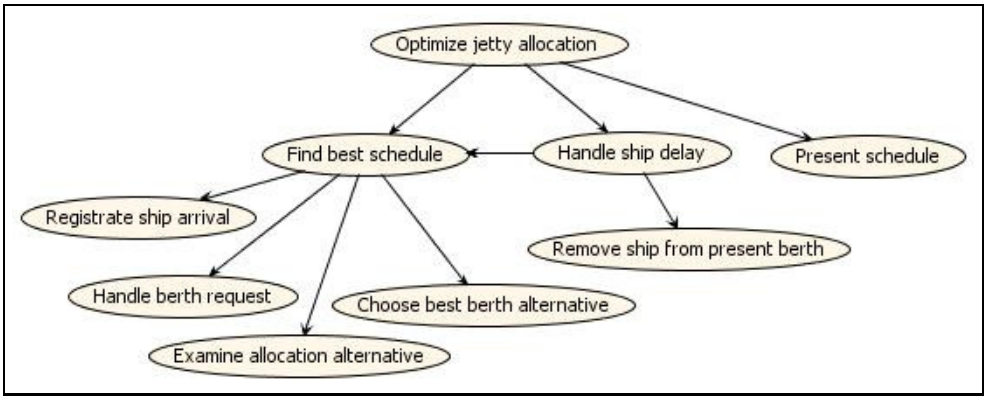

Figure 6.1: Goal Overview for JettyPlanner1 and JettyPlanner2

Optimize jetty allocation is divided into *Find best schedule*, and *Handle ship delay*. In the situation of a ship delay, a new arrival must be considered for a ship. *Find best schedule* will therefore be a subgoal for *Handle ship delay*. We have refined the *Find best schedule* goal into the following subgoals.

Registrate ship arrival; the system must be able to receive a new ship arrival. *Handle berth request*; means that the system must do a request to the berths to ask for the best allocation for a ship. *Examine allocation alternative*; denotes that the cost of allocation for a ship should be examined for every appropriate berth. *Choose best berth alternative*; the system should choose the best allocation alternative for a ship in accordance to relevant variables. The jettyplanner system should also be able to *Present schedule*, in other words present information to the user of the system.

After refining the goals, they were rearranged and similar goals were moved together in groupings. The gropings provided the basis for examining which roles to be fulfilled in the Jettyplanner.

6.1.2 Roles

The functionality of the system is described by using the term *Role*. Each role consists of a grouping of related goals.

The roles for the Jettyplanner are given in figure 6.2, followed by a description of each of them.

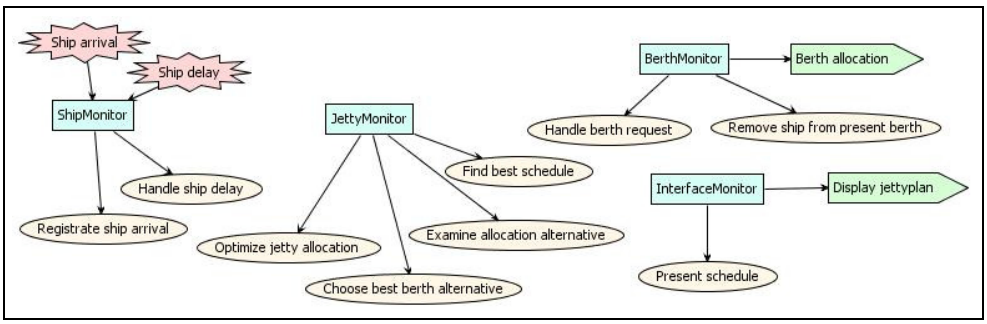

Figure 6.2: Role Overview for JettyPlanner1 and JettyPlanner2

ShipMonitor The ShipMonitor is responsible for registrating ship arrivals and han-

dle ship delay. **Percept:** *Ship arrival* and *Ship delay*.

JettyMonitor The JettyMonitor is responsible for a nearly optimized jetty allocation, and should therefore be able to find the best schedule, examine allocation alternatives and choose the best berth alternative.

BerthMonitor The BerthMonitor is responsible for handling berth requests. **Action:** The result of this functionality is a *Berth allocation*, a relation between ship and berth.

InterfaceMonitor The InterfaceMonitor is responsible for presenting a model of the schedule to the user. **Action:** *Display jettyplan*

6.1.3 Scenarios Illustrated by Use Cases

We have written scenarios to illustrate the sequence of performance steps within the JettyPlanner systems.

UseCase1: Ship Arrival describes the sequence for handling a ship arrival, see table 6.1. **UseCase2: Ship Delay** include the sequence for handling delay in arrival time, which indicates that the schedule must be reorganized. This use case is found in table 6.2. The use cases are similar from step3 in UseCase1 and step5 in UseCase2.

Trigger	**Description**	
A ship arrives	A ship arrives Mongstad and reports its arrivaltime in hours. The ship will be allocated a berth with the right product and capacity. The schedule will be updated.	
Number	**Step type**	**Description**
1	Percept	Arrival information about the ship received
2	Goal	Registrate ship arrival
3	Goal	Handle berth request
4	Goal	Examine allocation alternative
5	Goal	Choose best berth alternative
6	Action	Berth allocation
7	Goal	Present schedule
8	Action	Display jettyplan

Table 6.1: UseCase1: Ship Arrival

Trigger	**Description**	
Delay in arrival time	A ship reports that it will arrive later than first assumed. The ship will be reallocated in accordance to the changes and the schedule will be updated.	
Number	**Step type**	**Description**
1	Percept	Delay information about the ship received
2	Goal	Handle ship delay
3	Goal	Remove ship from present berth
4	Goal	Registrate ship arrival, with updated arrival time
5	Goal	Handle berth request
6	Goal	Examine allocation alternative
7	Goal	Choose best berth alternative
8	Action	Berth allocation
9	Goal	Present schedule
10	Action	Display jettyplan

Table 6.2: UseCase2: Ship Delay

6.2 Architectural Design

In the *Architectural Design* are the agent types identified, the interaction between the agents are described with sequence diagrams, and the overall system structure are presented. From now on, and throughout this section, we will separate the design of JettyPlanner1 and JettyPlanner2.

6.2.1 Agent Types

The agent types are identified based on in the roles from section 6.1.2. Note that an agent in a system can have several roles.

Agents in JettyPlanner1

JettyPlanner has role *JettyMonitor*

Berth has role *BerthMonitor*

GUI has roles *ShipMonitor*, *InterfaceMonitor*

Agents in JettyPlanner2

JettyPlanner has role *JettyMonitor*

Berth has role *BerthMonitor*

GUI has role *InterfaceMonitor*

Ship has role *ShipMonitor*

6.2.2 Interaction Diagrams

Based on the use case scenarios in section 6.1.3, the messages that must be exchanged between the agents to obtain achievement of goals in the right sequence have been identified.

Interactions in JettyPlanner1

Figure 6.3 illustrates the interaction between the JettyPlanner1's three agent types in the *Ship Arrival* scenario. Figure 6.4 depicts the *Ship Delay* scenario.

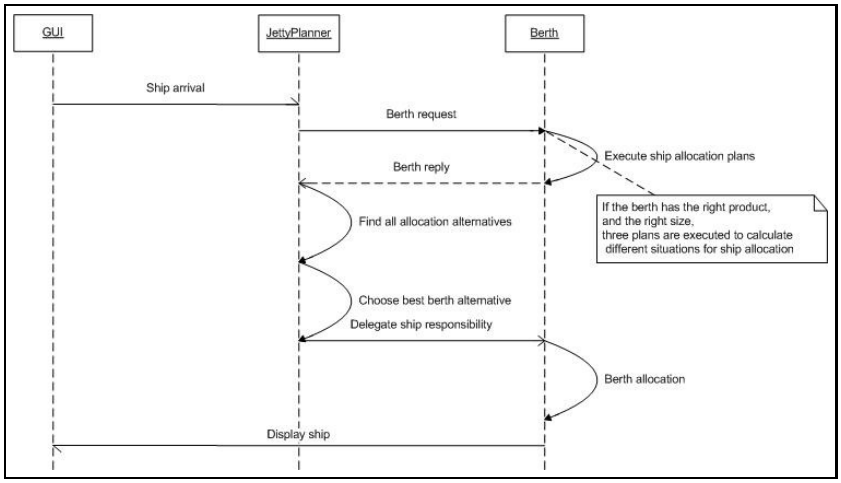

Figure 6.3: Sequence Diagram for UseCase1: Ship Arrival, JettyPlanner1

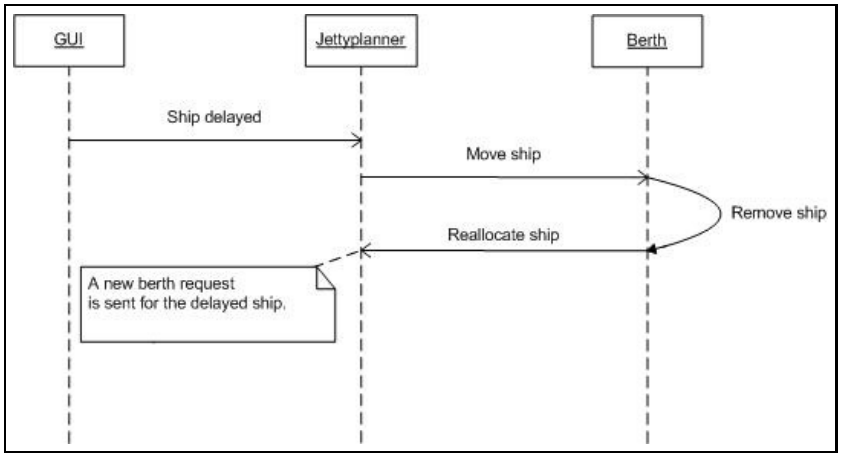

Figure 6.4: Sequence Diagram for UseCase2: Ship Delay, JettyPlanner1

Interactions in JettyPlanner2

Figure 6.5 illustrates the interaction between the JettyPlanner2's four agent types in the *Ship Arrival* scenario. Figure 6.6 shows the *Ship Delay*.

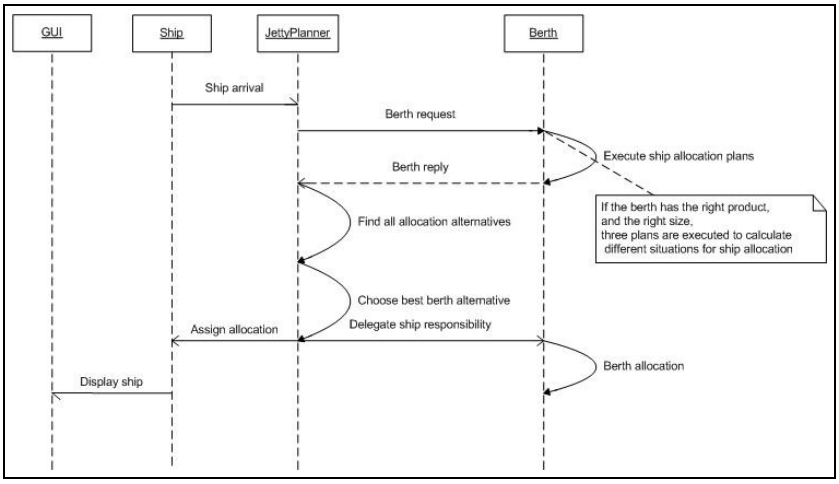

Figure 6.5: Sequence Diagram for UseCase1: Ship Arrival, JettyPlanner2

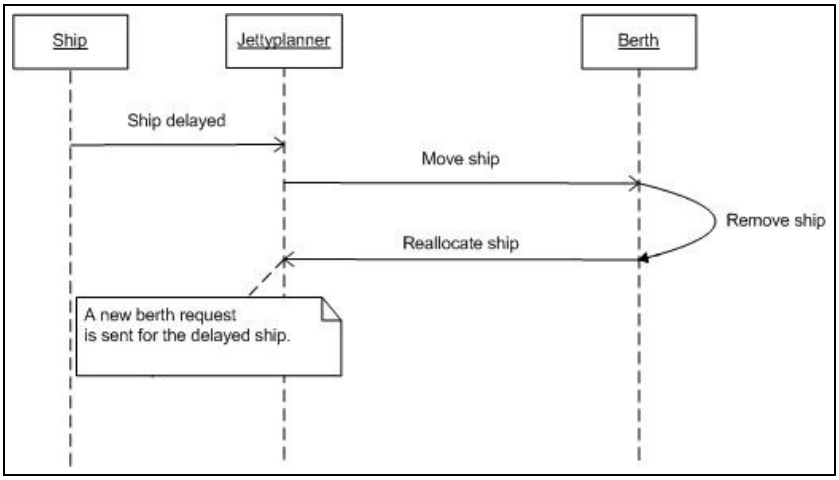

Figure 6.6: Sequence Diagram for UseCase2: Ship Delay, JettyPlanner2

6.2.3 The System Structure

The system overview diagrams gather all the entities in the system and give a brief overview of the systems main entities and the communication between them.

System Overview Diagram for JettyPlanner1

Figure 6.7 presents the system overview diagram for JettyPlanner1. The system includes the agent types; *JettyPlanner*, *Berth* and *GUI*. The *GUI* receives information from the environment of the system in term of the *Ship arrival* and *Ship delay* percepts. As a consequence, messages will be sent from *GUI* to *JettyPlanner* to inform about the environmental changes. If a new *Ship arrival* occurs, the *JettyPlanner* will send a *BerthRequest* to the *Berth*-agents to find an appropriate berth for the ship. The *Berth* will check its allocation opportunities, by using their *Berth data* and send a message to the *JettyPlanner* with a *BerthReply*. The *JettyPlanner* will then delegate responsibility for the ship to a chosen berth.

If a *Ship delay* occurs, the *JettyPlanner* will ask the present responsible *Berth* to remove the ship by sending *RemoveShip*, followed by sending a new *BerthRequest* to all the *Berth*-agents. After the *JettyPlanner*-agent has delegated the ship responsibility to a berth, it informs the *GUI* that it should update the user interface. This results in an action; *GUI* displays a graphical representation of the jettyplan.

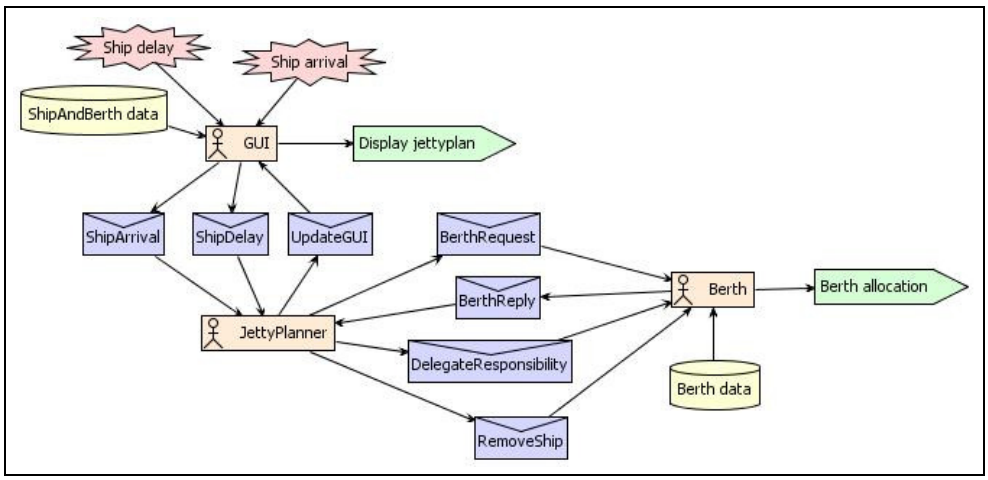

Figure 6.7: System Overview JettyPlanner1

System Overview Diagram for JettyPlanner2

Figure 6.8 presents the system overview diagram for JettyPlanner2. The system includes the agent types; *JettyPlanner*, *Berth*, *GUI* and *Ship*. The *Ship* receives information from the environment of the system in terms of the *Ship arrival* and *Ship delay* percepts. As a consequence, messages will be sent from *Ship* to *JettyPlanner* to inform about the environmental changes. If a new *Ship arrival* occurs, the *JettyPlanner* will send a *BerthRequest* to the *Berth*-agents to find an appropriate berth for the ship. The *Berth* will send a *RequestShipInformation* to the *Ship*-agent, that will give information about the berth in a *ShipReply*. The *Berth* will compare ship and berth data, and inform about its allocation opportunities in a *BerthReply* to the

JettyPlanner. The *JettyPlanner* will then delegate the responsibility for the ship to a chosen berth and inform the *Ship* about the allocation with a *AssignAllocation* message. The *Ship* will then inform the *GUI* to update the user interface, which result in an action; *GUI* display a graphical representation of the jettyplan.

If a *Ship delay* occurs, the message from the the *Ship*-agent will bring along the *JettyPlanner* to ask the present responsible for the *Ship* to *RemoveShip*, followed by sending a new *BerthRequest* to all the *Berth*-agents.

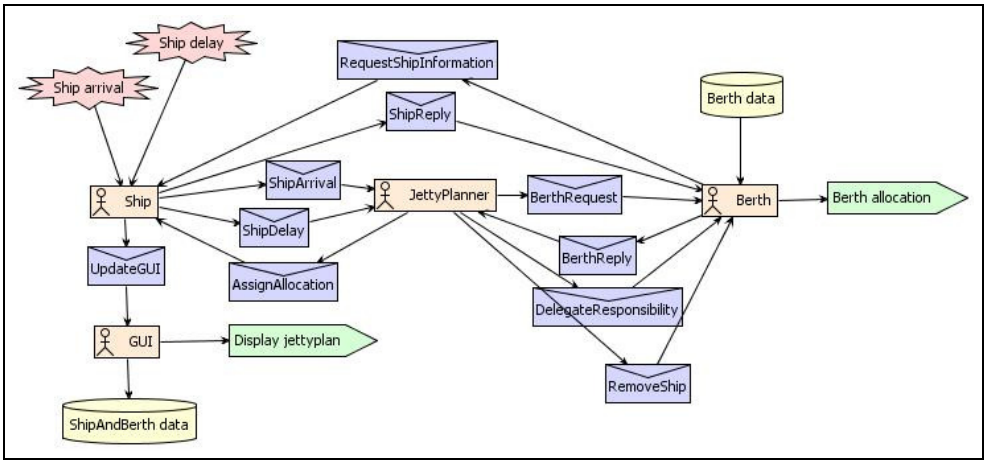

Figure 6.8: System Overview JettyPlanner2

6.3 Detailed Design

The *Detailed Design Phase* deals with the capabilities an agent needs to fulfill its tasks. The *Agent Overview diagrams* take each agent into account, and show the relationship to their capabilities. The *Capability Overview Diagrams* describe each capability separately and include their plans, events and beliefs.

6.3.1 Agent Overview Diagrams

This type of diagram shows the top level view of the agent's internals. The diagram includes the capabilities of the agent, the messages related directly to the agent and the data internal to the agent.

The diagram should also contain the artifacts from the *System Overview Diagram*. Since we have used the Prometheus tool in the *Architectural Design* and JACK in the *Detailed Design*, we only use artifacts from JACK. See figure 3.3 in section 3.1.3.

The *Agent Overview Diagrams* from the two versions do not have many differences. Two different versions will be presented in the cases where they differ.

The *Berth*-agent, in JettyPlanner1 and JettyPlanner2, is given in figure 6.9. The agent has the following capabilities:

InitiationOfLiftingArm Initiates the variables internal to every *Berth*-agent.

RequestHandling Handles a request about whether this berth can make an allocation of a specified ship.

ReceiveShipResponsibility Handles a delegation of responsibility for a ship, and do the necessary reallocations.

ShipRemover Remove the allocation of a specific Ship.

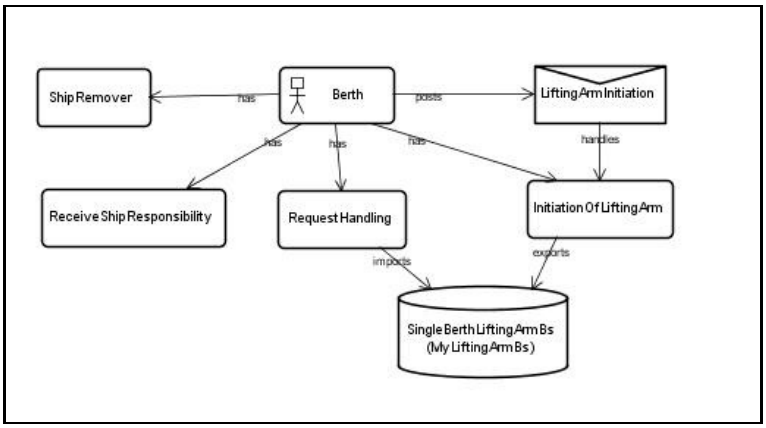

Figure 6.9: Agent Overview, Berth in JettyPlanner1 and JettyPlanner2

The *JettyPlanner*-agent, has the same *Agent Overview Diagram* in both versions. The diagram is presented in figure 6.10. The agent has the following capabilities:

BerthRequesting Handles the case where an incoming ship requests for a berth allocation. The capability uses a *Semaphore* to ensure that only one ship is inquired at the time.

DelayHandler Handle the case where an already allocated ship has reported a delay in arrival time.

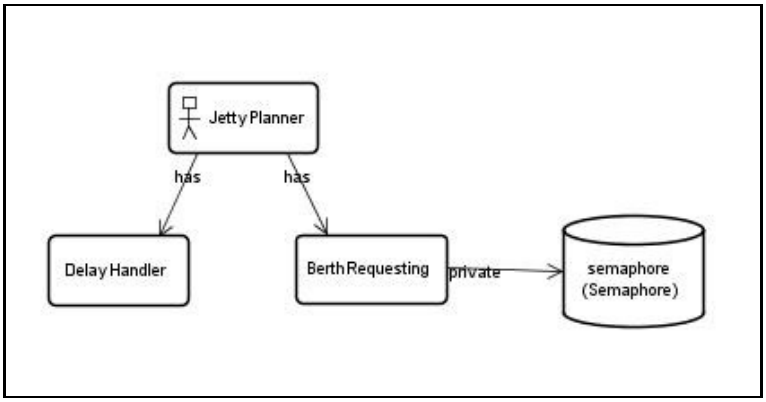

Figure 6.10: Agent Overview, JettyPlanner in JettyPlanner1 and JettyPlanner2

The difference between the two *GUI* agents' overview diagrams, is that the *GUIChange* capability is left out in JettyPlanner2. In JettyPlanner2, the *GUIManager* has the responsibility related to *GUIChange*, in addition to its primary responsibility. The responsibilities of all GUI capabilities will now be explained. JettyPlanner1's *GUI*-agent is given in figure 6.11.

Initiation Reads data from the named data shown in figure 6.11 and instantiates *Ship*-objects/agents and *Berth*-agents.

GuiManaging Responsible for communication to and from the *Screen*-view, which is connected to the external user interface. Make sure that the agent system are informed about new user-interactions and the user are informed about relevant system decisions.

GuiChange A variant of the *GuiManaging* capability, but handle the situations where a change in already registrated data are made. (e.g a delay in arrival time for a ship)

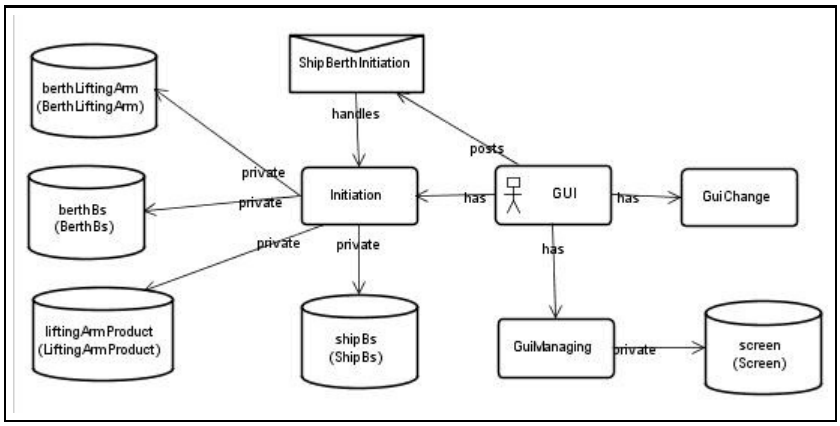

Figure 6.11: Agent Overview, Gui in JettyPlanner1

JettyPlanner2 introduces an additional agent, namely the *Ship*-agent, see figure 6.12. This agent is supposed to substitute the *Ship*-object from JettyPlanner1. To give the two versions of the system the same functionality, the *Ship agent* introduces two capabilities;

ArrivalManaging Manages the occurrence of a new ship arrival, and decides what to do next.

VariableManaging Manages the dynamic variables internal to the *Ship*-agent, receives information about updates and informs other agents about the present value of the variables.

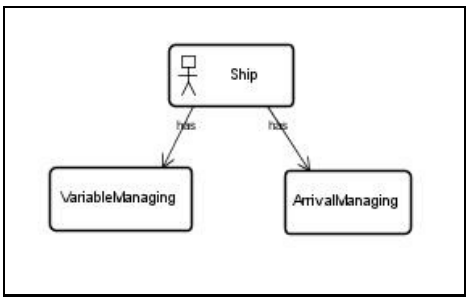

Figure 6.12: Agent Overview, Ship in JettyPlanner2

6.3.2 Capabilities Related to the Berth Agent

This section presents the capabilities related to the *Berth*-agent in figure 6.9. Each will be taken into account and their internal plans, events, and data will be described. Explanations related to capability overview diagrams that are equal for the two versions will only be given once.

Capability: InitiationOfLiftingArm

The *Berth*-agent posts a *LiftingArmInitiation* event to the *InitiateMyLiftingArm*-plan at instantiation. The plan reads data from *MyLiftingArmBs* and relates them to the agent's variables.

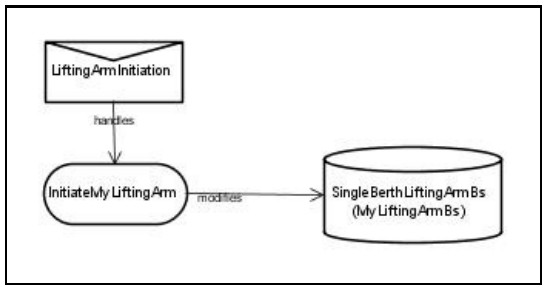

Figure 6.13: Capability: InitiationOfLiftingArm Overview, Berth in JettyPlanner1 and JettyPlanner2

Capability: RequestHandling

BerthRequest events are handled by the *AllocationCheck* plan. The plan checks the *Berth*-agent's variables against the request. If the right conditions are met, it posts an *AllocationRequest* event to itself, which is handled by three different plans. The *AllocateShipWantedArrivalTime* plan, finds the cost and placement of the ship when the ship gets the arrival time it has asked for. The *AllocateShipNearWantedArrivalTime* plan finds the cost and placement of the ship in the case where the ship is allocated behind the ship that occupies the wanted allocation time. *AllocateShipFirstTimeIntervalAvailable* plan, finds the first time interval available, accordingly not occupied by any other ship and large enough for docking duration. After these plans are finished, the *AllocationCheck* plan sends a *BerthReply* with the best alternative that the three alternative plans have produced. The *RequestHandling* capability overview is given in figure 6.14.

Most events in JettyPlanner1 contain a *Ship object* with all information necessary to answer a request. In JettyPlanner2, events do not contain sufficient information and therefore the *Berth-agents* must communicate with the *Ship-agent*. Figure 6.15 illustrates that the three plans; *AllocateShipWantedArrivalTime*, *AllocateShipNearWantedArrivalTime* and *AllocateShipFirstTimeIntervalAvailable* send a *VariableRequest* to ask for information about a ship, and receive this in a *VariableReply* event.

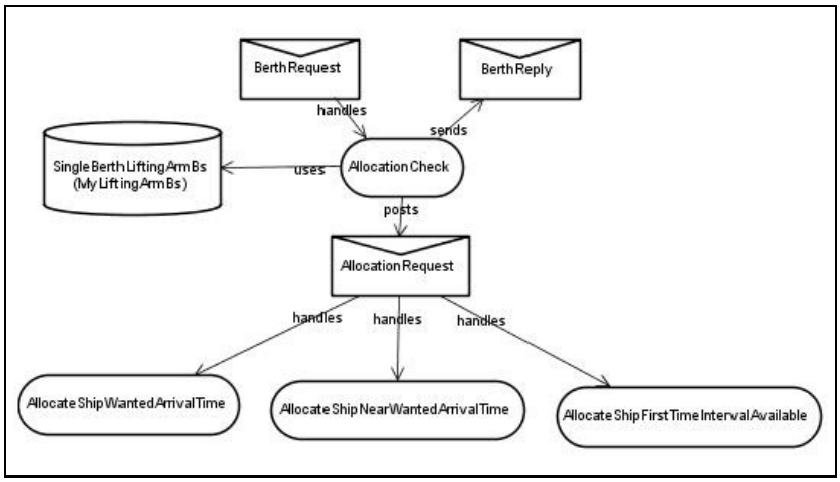

Figure 6.14: Capability: RequestHandling Overview, Berth in JettyPlanner1

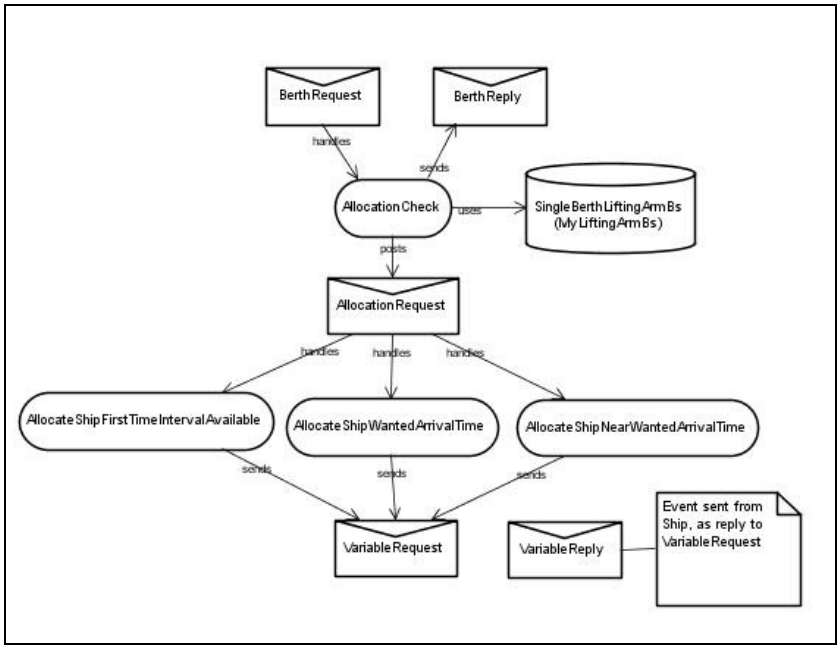

Figure 6.15: Capability: RequestHandling Overview, Berth in JettyPlanner2

Capability: ReceiveShipResponsibility

This capability, given in figure 6.16, is used when a *DelegationOfResponsibility* event is sent to a *Berth*-agent. The *ShipAllocation* plan take care of allocating the delegated ship at the suggested position. In JettyPlanner1, this plan is also responsible for sending out a *DisplayShip* event to inform that the new allocation has taken place. *IncomingShip* events are sent when already allocated ships are knocked out. These events will lead to a new allocation for the knocked out ships.

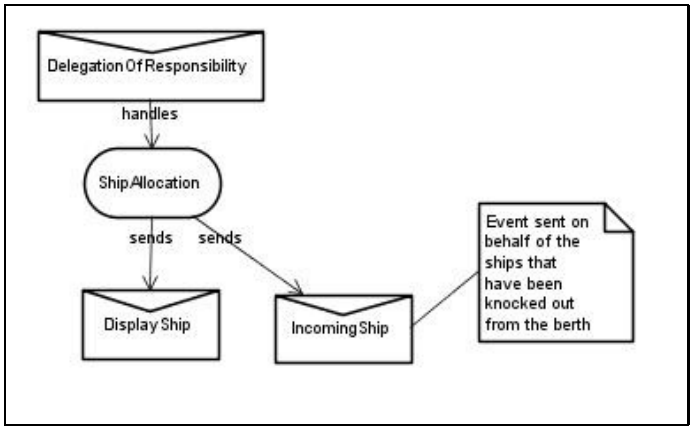

Figure 6.16: Capability: ReceiveShipResponsibility Overview, Berth in JettyPlanner1

JettyPlanner2's version of the capability is found in figure 6.17. What distinguish the two versions are the messages sent from the *ShipAllocation* plan. In JettyPlanner2 a *ResponsibleReply* event is sent to inform that the allocation has taken place. The *KnockOutShip* is sent to the *Ship*-agent.

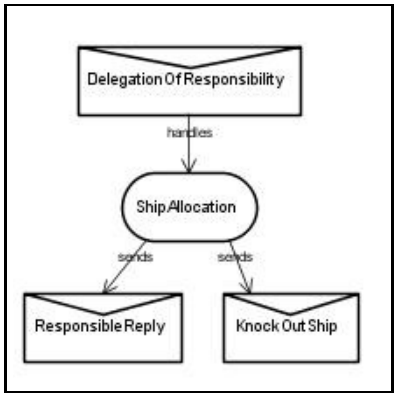

Figure 6.17: Capability: ReceiveShipResponsibility Overview, Berth in JettyPlanner2

Capability: ShipRemover

The *ShipRemover* capability is used in the situation when a ship delay has occurred and a *RemovementOrder* is sent to the *Berth* responsible for the ship. The *Remove-*

Ship plan in JettyPlanner1, figure 6.18, removes the allocation of the delayed ship. The plan sends an *IncomingShip* event to make sure that the ship is reallocated according to the new arrival time. The *ManageRemovement* plan in JettyPlanner2, figure 6.19, handles the same situation as the *RemoveShip* plan, but is not responsible for sending an *IncomingShip* event.

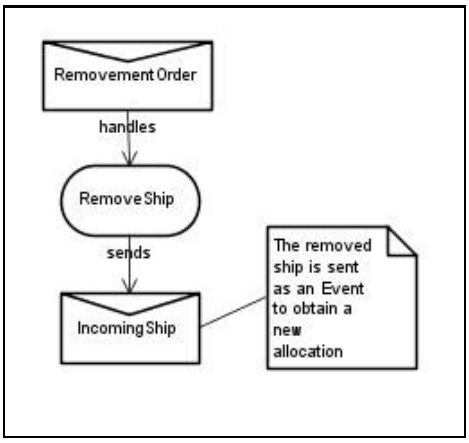

Figure 6.18: Capability: RemoveShipResponsibility Overview, Berth in JettyPlanner1

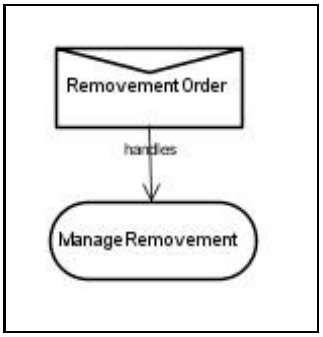

Figure 6.19: Capability: RemoveShipResponsibility Overview, Berth in JettyPlanner2

6.3.3 Capabilities Related to the JettyPlanner Agent

This section describes the *JettyPlanner*-agent's capabilities, with their internal plans, events, and data.

Capability: BerthRequesting

This capability handles the case where an *IncomingShip* event is sent to the *JettyPlanner*. The *RequestForBerthHandling* plan receives the event, and sends out a request to all the *Berth*-agents. When the plan has received a *BerthReply* for all the *BerthRequests*, it posts the allocation alternatives to the *ChooseBerthAlternative* plan. The semaphore is used to ensure mutual exclusion, ergo only one request can happen at one time. The *ChooseBerthAlternative* plan picks out the allocation

alternative which will lead to lowest cost for placing the ship. This part is equal for the two versions.

In JettyPlanner1, see 6.20 figure, the *DisplayShip* will signal not to display the ship in the case of no allocation alternatives. When an alternative exists, a *DelegationOfResponsibility* event will inform the *Berth* that came up with the alternative.

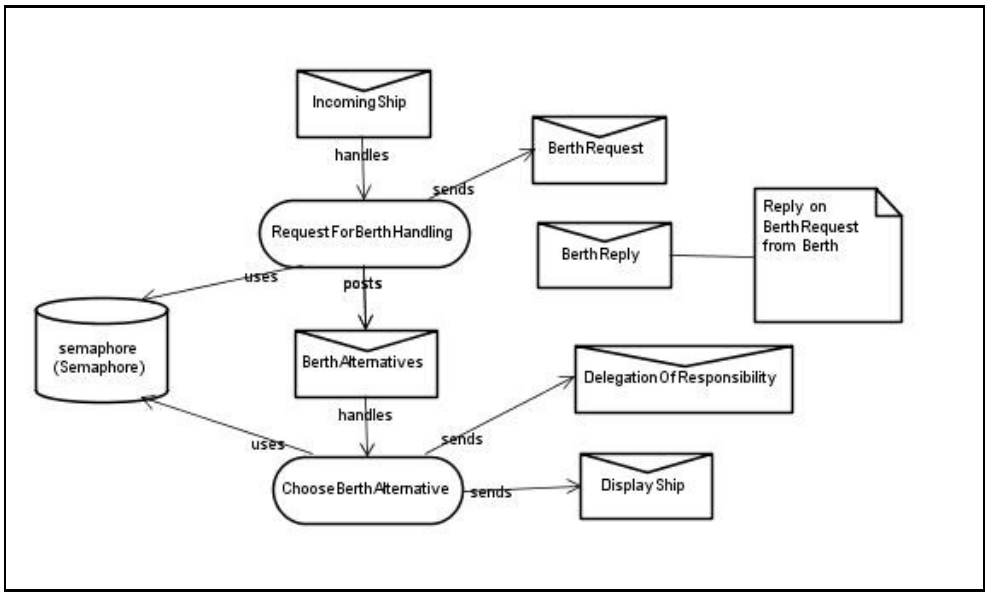

Figure 6.20: Capability: BerthRequesting Overview, JettyPlanner in JettyPlanner1

A *DelegationOfResponsibility* event is sent to the right *Berth* when an alternative is chosen, like in JettyPlanner1. Whether there exists an alternative or not, a *UpdateDynamicVariables* event is sent to the responsible *Ship agent* to inform about the situation. The *BerthRequesting* capability in JettyPlanner2 is found in figure 6.21.

Capability: DelayHandler

The *HandleArrivalChange* plan receives a *MovingShip* event when a ship is delayed. The plan informs, with a *RemovementOrder*, the current holder of the ship. This capability is the same for the two versions, and is shown n figure 6.22.

6.3.4 Capabilities Related to the GUI Agents

This section presents the capabilities related to the *GUI*-agent, shown in figure 6.11.

Capability: Initiation

When the *GUI*-agent is instantiated, it posts a *ShipBerthInitiation* event to the *InitiationOfShipBerts* plan. The plan uses the *shipBs*, *berthBs*, *berthLiftingArm* and *liftingArmProduct* data to read information and relate it to variables needed to establish *Ship*-objects and *Berth*-agents. In JettyPlanner2, *Ship*-agents are established at this stage instead of objects. The results from the *InitiationOfShipBerths* plan, is posted

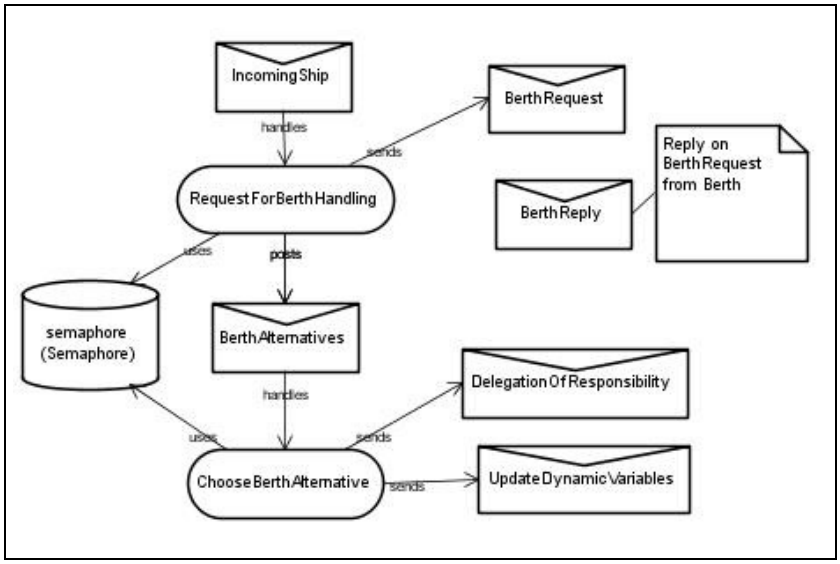

Figure 6.21: Capability: BerthRequesting Overview, JettyPlanner in JettyPlanner2

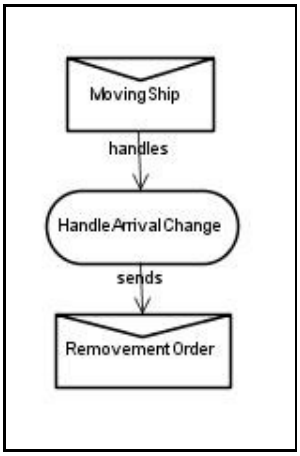

Figure 6.22: Capability: DelayHandler Overview, Jettyplanner in JettyPlanner1 and JettyPlanner2

further in an *InitialData* event to the *GuiManaging* capability, described in the next section.

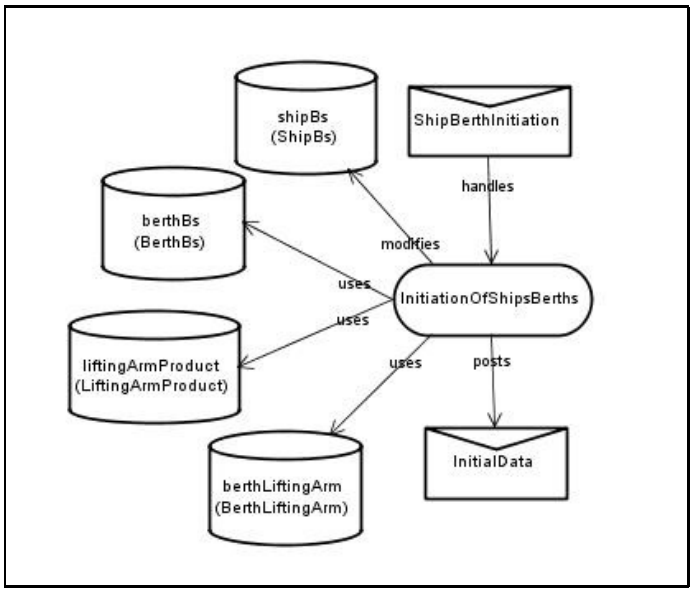

Figure 6.23: Capability: Initiation Overview, Gui in JettyPlanner1 and JettyPlanner2

Capability: GuiManaging

The *GuiManaging* capability's main concern, is to communicate with the *Screen*-view that connects the user interface to the rest of the agent system.

In JettyPlanner1, see figure 6.24, the *RegistrateShipArrival* plan receives a *ShipArrival* event from the *Screen*, and forwards the information by sending an *IncomingShip* event. *InitialData* and *DisplayShip* activate the *InitializeGui* plan and the *UpdateGui* plan respectively. The activation causes different methods to use a reference to the *Screen*-view. These methods initiate updates in the user interface.

JettyPlanner2's version of the *GuiManaging* capability can be seen in figure 6.25, The difference from the other capability is that the *Screen*-view posts a *SubmitShip* event on command from the external user interface classes. The information is thereafter forwarded in a *ShipArrival* event.

Capability: GuiChange

The *ForwardShipDelay* plan related to this capability, see figure 6.26, receives the same *ShipArrival* event as the *RegistrateShipArrival* plan in *GuiManaging*. Only one of the plans is activated at the same time, and both plans have a check to see if it should be activated by the event. The *ForwardShipDelay* plan, forward a *MovingShip* event in the case where the *ShipArrival* corresponds to a ship that is delayed. This capability occur only in JettyPlanner1. In JettyPlanner2, this functionality is covered by the *GuiManaging* capability and the *Ship agent*.

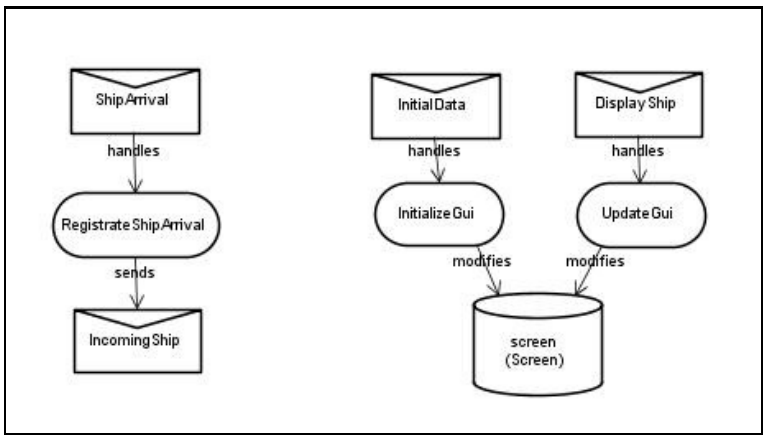

Figure 6.24: Capability: GuiManaging Overview, Gui in JettyPlanner1

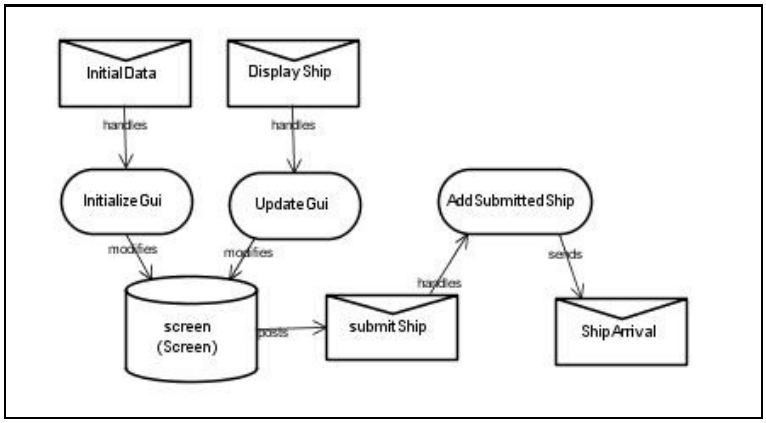

Figure 6.25: Capability: GuiManaging Overview, Gui in JettyPlanner2

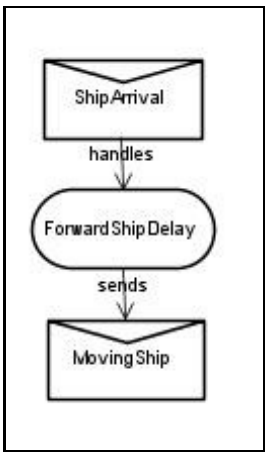

Figure 6.26: Capability: GuiChange Overview, Gui in JettyPlanner1

6.3.5 Capabilities Related to the Ship-Agent

As mentioned earlier, the transformation of the ship-object representation to a *Ship*-agent, is the main difference between the two JettyPlanner versions. The *Ship agent*, from figure 6.12 only occur in JettyPlanner2, and so do the related capabilities.

Capability: ArrivalManaging

The *HandleShipArrival* plan handles *ShipArrival* events. It decides whether this event indicates a new incoming ship. If there is a new incoming ship, an *Incoming-Ship* event is sent. If the ship is delayed a *RemoveShip* event is sent. The *Manage-KnockOut* plan manages the case where a ship receives a *KnockOutShip* message. The plan handles this situation by sending a *ShipArrival* event to make sure that the ship gets a new allocation. The *ArrivalManaging* capability is described in figure 6.27.

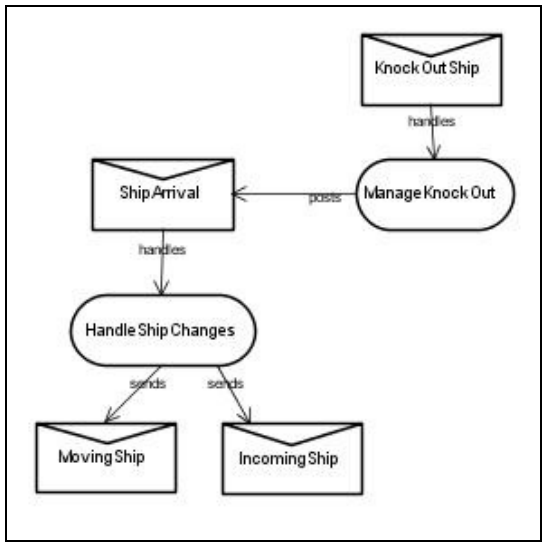

Figure 6.27: Capability: ArrivalManaging Overview, Ship in JettyPlanner2

Capability: DynamicVariablesManaging

The *DynamicVariablesManaging* capability, in figure 6.28, controls the *Ship*-agent's variables. Incoming *VariableRequests* are handled by the *VariableCheck* plan. The plan looks up the requested variables and sends them in a *VariableReply*. The *UpdateVariables* plan is notified when the *Ship* has been delegated to a *Berth*. This will result in an update of the ship-variables, and an outgoing *DisplayShip* event to signal the user interface.

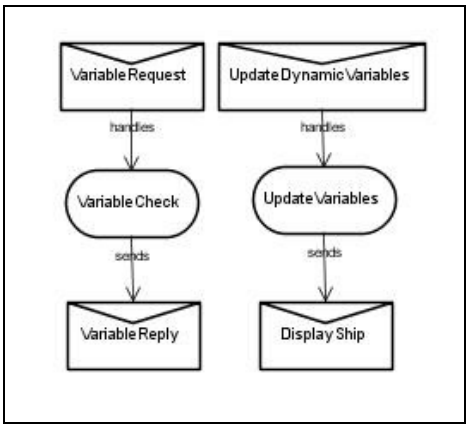

Figure 6.28: Capability: DynamicVariablesManaging Overview, Ship in JettyPlanner2

6.4 Implementation

Code-examples from JettyPlanner1 and JettyPlanner2 are presented in this section to demonstrate how JACK entities are implemented. We will also illustrate the main differences between the implementations of the two applications.

Source code for illustrating different JACK entities is given in section 6.4.1. Section 6.4.2 have intention to illustrate the extensions and replacements that JettyPlanner2 needs to obtain the same functionality as JettyPlanner1.

6.4.1 JACK Entities

Different JACK entities will be presented in this section. These entity types were introduced in section 3.2. Examples of implementations of these entity types are illustrated with source code from our systems.

Agent

Listing 6.1 presents a code-segment from the *Ship.agent* entity in JettyPlanner2. The agent has several get- and set-methods, but these can only be called by plans internal to the *Ship*-agent. It is not possible to activate regular methods within an agent entity from external classes or objects. The reason is that they can not receive a reference to the agent, which is a requirement for making external method calls. The get- and set-methods are not included in the following code-segment.

Listing 6.1: Code-Segment from *Ship.agent* i JettyPlanner2

```
public agent Ship extends Agent {
    #has capability ArrivalManaging cap;
    #has capability VariableManaging cap1;
    public Ship(String name, int length, int width, double draft, int
        capacity, int numberOfHoles){
        super(name);
        System.out.println("Oppretter ship agent");
        this.agentName = name;
        this.length = length;
        this.width = width;
        this.draft = draft;
```

```
        this.capacity = capacity;
        this.numberOfHoles = numberOfHoles;
        visitedBerth = new Vector();
    }
    public String agentName;
    public int length;
    public int width;
    public double draft;
    public int capacity;
    public int numberOfHoles;
    public String product;
    public TimePoint arrivalTime;
    public TimePoint delegatedArrivalTime;
    public TimePoint departureTime;
    public Vector visitedBerth;
```

We have observed that the most exceptional with an agent entity, compared to a java class, is the entity declaration starting with; *public agent...*. Another issue is that agent entities contain declarations of its capabilities.

Capability

The *Ship*-agent has the capability *ArrivalManaging*. The implementation of this capability is shown in listing 6.2.

Listing 6.2: *ArrivalManaging.cap* in JettyPlanner2

```
public capability ArrivalManaging extends Capability {
    #handles external event ShipArrival;
    #sends event IncomingShip ev;
    #sends event DisplayShip ev2;
    #handles external event KnockOutShip;
    #posts external event ShipArrival ev3;
    #sends event RemoveShip ev1;
    #uses plan HandleShipChanges;
    #uses plan ManageKnockOut;
}
```

A capability contains references to all its plans and events.

Plan

The *Ship*-agent uses the plan *ManageKnockOut*. The implementation of this plan is shown in listing 6.3

Listing 6.3: *ManageKnockOut.plan* in JettyPlanner2

```
public plan ManageKnockOut extends Plan {
    #handles event KnockOutShip ev;
    #posts event ShipArrival ev1;
    #uses interface Ship self;

    static boolean relevant(KnockOutShip ev)
    {
        return true;
    }

    context()
    {
        true;
    }
```

```
    #reasoning method
    body()
    {
        self.setDelegatedArrivalTime(null);
        self.setDepartureTime(null);
        self.visitedBerth.add(ev.berthName);
        @post(ev1.newShipArrival(self.getAgentName(), self.product, self.
            arrivalTime));
    }
}
```

A plan entity is declared with *public plan*, and contains references to events that it handles, posts and sends. In addition, a plan can contain a statement, *#uses interface*, which gives access to the agent's methods.

static boolean relevant() is used to determine if a plan is relevant to the actual event. The *context()* is used to determine if the plan should be executed in the current context. The *body()* is the plan's main reasoning method. It describes what an agent actually does when it executes an instance of this plan[Age06]. A plan can also contain other reasoning methods that the main body() method can call when needed[Age06].

Event

The *Ship*-agent sends the event *IncomingShip* to the *JettyPlanner*-agent. The implementation of the event is shown in listing 6.4

Listing 6.4: *IncomingShip.event* in JettyPlanner2

```
public event IncomingShip extends MessageEvent {
    public String shipName;
    public int length;
    public int width;
    public double draft;
    public int capacity;
    public int numberOfHoles;
    public String product;
    public TimePoint arrivalTime;

    #posted as
    shipRequest(String shipName, int length, int width, double draft, int
        capacity, int numberOfHoles, String product, TimePoint arrivalTime)
        {
        this.shipName = shipName;
        this.length = length;
        this.width = width;
        this.draft = draft;
        this.capacity = capacity;
        this.numberOfHoles = numberOfHoles;
        this.product = product;
        this.arrivalTime = arrivalTime;
    }
}
```

All events require at least one *posting method*, starting with *#posted as*. The method must be used whenever an instance of the event needs to be created. It describes how the event can be constructed and posted or sent[Age06].

Beliefset

The *Berth*-agent uses the beliefset *MyLiftingArm* to manage knowledge about its lifting arms. The implementation is shown in listing 6.5.

Listing 6.5: *MyLiftingArm.bel* in JettyPlanner2

```
public beliefset MyLiftingArmBs extends OpenWorld {
    #key field String liftingArmId;
    #value field int capacity;
    #value field String product;
    #indexed query getProduct(logical String liftingArmId, logical int
        capacity, String product);
}
```

The *#key field* declaration is used to describe a beliefset's key fields, and the *#value field* declarations the value fields. An *#indexed query* is used to access the data contained in the beliefset[Age06].

6.4.2 Extensions and Replacements of Code

Code examples from JettyPlanner1 and JettyPlanner2 are presented to illustrate what kind of differences that appear in the implementation when an object is replaced by an agent.

Methods in JettyPlanner1 versus Plans in JettyPlanner2

An agent uses plans, instead of methods, to obtain required functionality. As a consequence, some of the methods implemented in the ship class were moved to JACK-plans when the *Ship*-object was replaced by the *Ship*-agent.

Listing 6.6, and the already given plan in listing 6.3, illustrate how the functionality from the *knockOut*-method in the *Ship*-class is implemented as a plan in JettyPlanner2.

Listing 6.6: Method: *knockOut* in JettyPlanner1

```
public void knockOut(String berthName){
        delegatedArrivalTime=null;
        departureTime = null;
        visitedBerth.add(berthName);
    }
```

The main difference observed between the method implementation and the plan implementation, besides the structure, is the public declaration of the method. This indicate that the method is visible to external entities, which is not the case for a plan.

External Activations of a *Ship*-object versus a *Ship*-agent

An external activation of the *Ship*-object requires a reference to the object. The external references are used to exchange data with the *Ship*-object or use functionality provided by the methods in the *Ship*-class.

External references of the *Ship*-agent is not possible to receive. The references are therefore replaced with event exchanges in JettyPlanner2.

Listing 6.7 and 6.8 illustrate this difference between the versions. The examples are taken from a plan called ”AllocateShipFirstTimeIntervalAvailable”, which is implemented differently in the two versions. In both occurrences, the *Berth*-agent uses the plan. The *Berth*-agent has an *ArrayList* with references to the ships that it is responsible for. In JettyPlanner1, this list contains *Ship*-objects, and in JettyPlanner2 the list contains ship names represented as *Strings*.

The objects in the list in JettyPlanner1 are used to perform a method-call to get the ship's delegated arrival time. In JettyPlanner2, an event is sent to the *Ship*-agent by using its name in the list. If the *Ship*-agent chooses to reply, the berth receives data from the *Ship*-agent.

Entities in JettyPlanner2 can not activate the *Ship*-agent by using a reference. Instead they have to send an event to the agent, which activates a plan, and wait for a reply.

Listing 6.7: Method-Call on a *Ship*-reference in JettyPlanner1

```
for(int i = 1; i < shipList.size(); i++){
    ship = (Ship)shipList.get(i);
    if(!departureTime.isAfter(ship.getArrivalTime())){
```

Listing 6.8: Sending Event *VariableRequest* in JettyPlanner2

```
for(int i = 1; i < shipList.size(); i++){
                shipName = (String)shipList.get(i);

                @send(shipName, varReq);
                try{
                    @wait_for(varReq.replied());
                }
                catch(NullPointerException e){
                    System.out.println("Failed waiting for reply for
                        ship variables at "+self.name());
                }
                VariableReply  varRep = (VariableReply)varReq.getReply
                    ();
```

Part III

Evaluation and Conclusion

Chapter 7

Evaluation

We have performed measurements of our two multi-agent systems by using the metrics described in section 5.4.2. These metrics have been used to test our hypothesis. The results are given in section 7.1.

Jettyplanning as application area has many challenges to be handled during implementation. We have used agent-related development methods and tools to handle these problems. A description of how we used agent technology to develop a solution for jettyplanning is given in section 7.2. Experiences related to *Prometheus* and *JACK* are given in section 7.3.

The validity of our results are discussed in section 7.4. Some of the validity threats have been addressed, while others have been accepted.

7.1 Measurements, Testing of Hypothesis and Results

We have performed measurements for each of the metrics presented in section 5.4. The results of the measurements will be used to test our hypothesis. Each hypothesis is presented with the related metric, followed by a discussion of the result. A conclusion states whether the hypothesis has been rejected or not.

7.1.1 Hypothesis 1

Hypothesis 1 is concerned with lines of code and is given as follows:

H01 The functionality of the the two versions will be implemented with approximately the same number of code lines.

HA1.1 JettyPlanner2 will implement the same functionality as JettyPlanner1 with fewer lines of code.

HA1.2 JettyPlanner2 will implement the same functionality as JettyPlanner1 with several lines of code.

We have chosen metric M1(LOC), to test hypothesis 1. Semicolons were counted to find the total number of written code-lines in each JettyPlanner version. The *Find in files* function in Textpad were used to count semicolons.

We have counted semicolons in the JACK source code for the agent entities. We have this because the amount indicates the actual number of lines that have been

written. When the source code is compiled into regular Java code, JACK generates extra lines of code[Age06]. The result of the measurement is given in table 7.1. For more information about code generation in JACK, see chapter 3.

M1: Lines of Code(LOC)		
Package	**JettyPlanner1**	**JettyPlanner2**
gui	593	530
ship	0	159
berth	387	466
jettyplanner	171	183
Other	4	5
Total Lines of Code	**1155**	**1343**

Table 7.1: Results for M1: Lines of Code(LOC)

We have depicted the result in a diagram to illustrate the measurements related to each version. The diagram is given in figure 7.1.

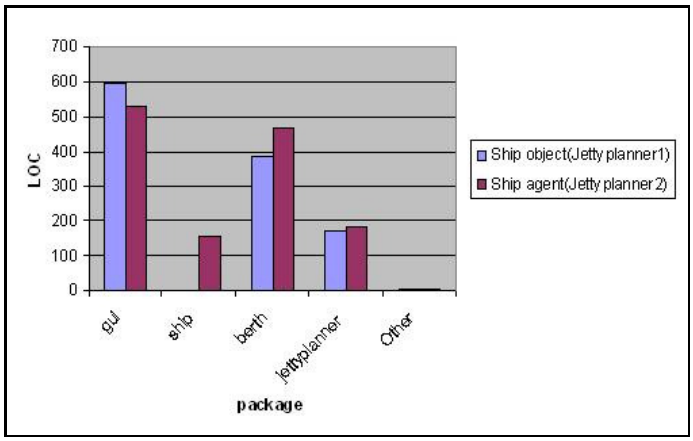

Figure 7.1: Measurements of Metric M1(LOC)

Discussion

The diagram in figure 7.1 depicts that JettyPlanner2 has a larger amount of code in certain packages than JettyPlanner1. JettyPlanner1 has a larger amount of code in the gui-package than JettyPlanner2. The reason for this is that the ship-object belongs to the gui-package in JettyPlanner1. When we transform the ship-object into a ship-agent, a new package is created, the ship package. The amount of code in the ship-package is therefore zero in JettyPlanner1. In JettyPlanner2 the ship package contains the ship-agent, its plans, events and capabilities.

The berth and the jettyplanner package have a larger amount of code in Jetty-Planner2. The reason for this is that ship-references and external method-calls in JettyPlanner1 are replaced with events and plans in JettyPlanner2.

JettyPlanner1 has 1155 lines of code, while JettyPlanner2 has 1343 lines of code. This is an increase of 16.28%.

Conclusion The result from the measurement of metric M1 in table 7.1 indicates that JettyPlanner2 has several lines of code than JettyPlanner1. We therefore reject hypothesis H01 and choose hypothesis HA1.2.

We have seen that turning our ship-object into a ship-agent will give us more code. We believe that this tendency will occur in other cases where agents are used instead of objects.

7.1.2 Hypothesis 2

Hypothesis 2 concerns the number of entities in each version and is given as follows:

H02 The number of entities will be the same for the two versions of the Jettyplanner.

HA2.1 JettyPlanner2 will have more entities than JettyPlanner1.

HA2.2 JettyPlanner2 will have less entities than JettyPlanner1.

We have chosen metric M2(NOE), to test hypothesis 2. We counted the number of entities related to packages in each JettyPlanner version. The results of the measurement are given in table 7.2, and are illustrated in figure 7.2.

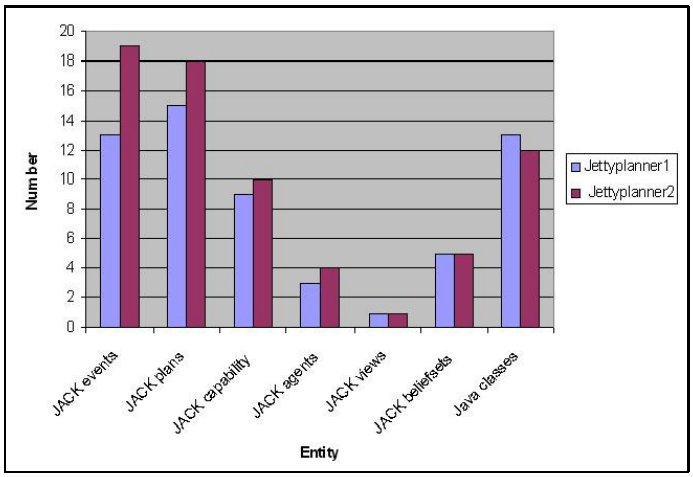

Figure 7.2: Measurements of Metric M2(NOE)

Discussion

We can clearly see in figure 7.2 that the use of a ship-agent in JettyPlanner2 lead to more event and plan entities, than in JettyPlanner1. The distribution of other entities are quite similar for both versions.

The reason for this increase in amount of events and plans, are that a ship-agent needs these entities to implement the same functionality as a single ship-object. An object only consist of one class with methods and variables, while an agent distribute its functionality in plans and events. The difference is illustrated in code examples in section 6.4.2

JettyPlanner1 consists of 59 entities, while JettyPlanner2 consists of 69. This is an increase of approximately 17% in amount of entities.

M2: Number of entities(NOE)			
Entity	**Package**	**JettyPlanner1**	**JettyPlanner2**
JACK events	gui	5	4
	ship	0	4
	jettyplanner	4	5
	berth	4	6
Total:		13	19
JACK plans	gui	5	4
	ship	0	4
	jettyplanner	3	3
	berth	7	7
Total:		15	18
JACK capabilities	gui	3	2
	ship	0	2
	jettyplanner	2	2
	berth	4	4
Total:		9	10
JACK agents	gui	1	1
	ship	0	1
	jettyplanner	1	1
	berth	1	1
Total:		3	4
JACK views	gui	1	1
Total:		1	1
JACK beliefsets	gui	4	4
	berth	1	1
Total:		5	5
Java classes	gui	8	7
	jettyplanner	1	1
	berth	3	3
	no package	1	1
Total:		13	12
Total Number of Entities		**59**	**69**

Table 7.2: Results for M2: Number of Entities

Conclusion

The result for metric M2 in table 7.2 indicates that JettyPlanner2 has more entities than JettyPlanner1. Hypothesis H02 is therefore rejected, and are replaced by hypothesis HA2.1.

We have seen that turning a ship-object into a ship-agent will lead to more entities in terms of plans and events. We believe that this tendency will occur in other cases where agents are used instead of objects.

7.1.3 Hypothesis 3

Hypothesis 3 concerns the amount of functions in each version and is given as follows:

H03 Both versions will use the same number of functions to complete the use-cases

described in chapter 6.

HA3.1 JettyPlanner2 will complete the use-cases with a fewer functions than JettyPlanner1.

HA3.2 JettyPlanner2 will complete the use-cases with several functions than JettyPlanner1.

Metric M3(NOF) has been chosen to test hypothesis 3. We have counted the number of get/set methods, other methods and plans used by the ship-object and the ship-agent respectively. The results of the measurement are given in table 7.3, and are illustrated in figure 7.3.

M3: Number of functions(NOF)		
Functions	**JettyPlanner1**	**JettyPlanner2**
get/set- methods	14	14
Other methods	3	1
Plans	0	4
Total Number of Functions	**17**	**19**

Table 7.3: Results for M3: Number of Functions(NOF)

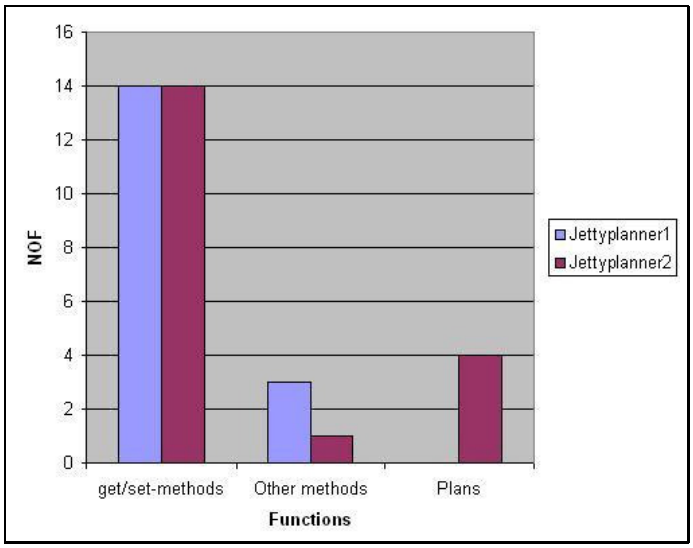

Figure 7.3: Measurements of Metric M3(NOF)

Discussion

Figure 7.3 illustrates that the ship-object and the ship-agent have the same get()- and set() methods. They also have one regular private method in common.

The noticeable difference between JettyPlanner1 and JettyPlanner2 is that two of the ship-object's methods in JettyPlanner1 have been implemented as plans in JettyPlanner2. These methods are public in JettyPlanner1, which means that other entities can call the methods if they have an external reference to the ship-object. In JettyPlanner2, external references are replaced by events. As a consequence, the

ship-agent need plans to handle these incoming events. The ship-agent also uses plans in JettyPlanner2 to reply to requests.

The ship-object in JettyPlanner1 has 17 functions, while the ship-agent has 19. This is an increase of 11.76%.

Conclusion

The results for metric M3, given in table 7.3, depict that JettyPlanner2 has two more functions than JettyPlanner1. We reject hypothesis H03, and replace it with hypothesis HA3.2.

Public methods used by external entities must be replaced by plans when turning objects into agents. The increase in number of functions for JettyPlanner2 may imply that this can be a tendency when turning objects into agents.

7.1.4 Hypothesis 4

Hypothesis 4 is concerned with the amount of couplings between entities in each version and is given as follows.

H04 Both versions will have the same number of couplings between the components in the system.

HA4.1 JettyPlanner2 will have fewer couplings between the components than JettyPlanner1.

HA4.2 JettyPlanner2 will have several couplings between the components than JettyPlanner1.

Metric M4(NOCBE) has been chosen to test hypothesis 4. We counted the number of couplings between the ship-object/ship-agent and other entities in each version. In section 5.4 we defined couplings to be in- and out going method calls and events.

The results of the measurement of metric M4 are given in table 7.4 and are depicted in figure 7.4.

M4: Number of Couplings between entities (NOCBE)					
External package	**Entities**	**JettyPlanner1**		**JettyPlanner2**	
		Coupling to ship-object	*Coupling from ship-object*	*Coupling to ship-agent*	*Coupling from ship-agent*
gui	Java classes	17	0	5	0
	Plans	0	0	1	2
jettyplanner	Plans	3	0	2	3
	Java classes	1	0	0	0
berth	Agent	2	0	0	0
	Plans	37	0	12	1
	Java classes	4	0	0	0
All packages		**64**	**0**	**20**	**6**

Table 7.4: Results for M4: Number of Couplings between Entities (NOCBE)

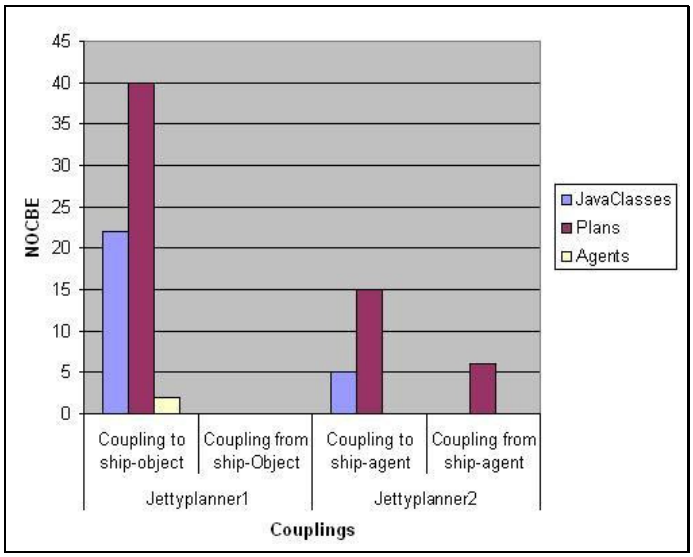

Figure 7.4: Measurements of Metric M4(NOCBE)

Discussion

The results depicted in figure 7.4 indicates that there are a larger amount of couplings between the ship-object and other entities in JettyPlanner1, than between the ship-agent and other entities in JettyPlanner2. In addition, the amount of incoming couplings from java-classes is reduced in JettyPlanner2.

In JettyPlanner1 most of the couplings occur as external references to the ship-object from other entities' plans. It is not possible to have external references to an agent. Therefore, these kind of couplings have to be replaced by events in JettyPlanner2.

Events are used by the ship-agent to communicate with other entities. The agent sends these events from plans. As a consequence, all the couplings from the ship-agent to other entities are implemented in plans in JettyPlanner2. The ship-agent has 6 outgoing couplings. There are zero outgoing couplings in JettyPlanner1 because the ship-object has no external references to other entities.

The ship-object in JettyPlanner1 has a total of 64 in- and outgoing couplings. These couplings are reduced to 26 in JettyPlanner2 when the ship-object is replaced by an agent, which means an reduction of 59.38% in couplings between components.

Conclusion

Based on the results for metric M4 in table7.3, we reject hypothesis H04 and choose hypothesis HA4.1.

We have seen from the results that incoming couplings are reduced when turning an object into an agent. There is no longer need for external references to the object from other entities. Events are instead used for communication, and out-going couplings will therefore increase as a consequence.

The huge reduction of couplings in JettyPlanner2 may imply that this tendency will occur in other cases where agents are used instead of objects.

7.1.5 Hypothesis 5

Hypothesis 5 is concerned with the number of external activations and is given as follows:

H05 JettyPlanner1 and JettyPlanner2 have the same number of external operations changing their internal state.

HA5.1 JettyPlanner2 has fewer external operations changing the internal state than JettyPlanner1.

HA5.2 JettyPlanner2 has several external operations changing the internal state than JettyPlanner1.

We have chosen metric M5(NOEA) to test hypothesis 5. We decided to count incoming couplings that activated the ship-object or the ship-agent to get the results given in table 7.5. The couplings were defined as external method-calls and received events in section 5.4.2.

Figure 7.5 illustrates the results in a diagram.

M5: Number of External Activations(NOEA)			
External package	**Entities**	**JettyPlanner1**	**JettyPlanner2**
gui	Java classes	17	5
	Plans	0	1
jettyplanner	Plans	3	2
	Java classes	1	0
berth	Agent	2	0
	Plans	37	12
	Java classes	4	0
All packages		**64**	**20**

Table 7.5: Results for M5: Number of External Activations

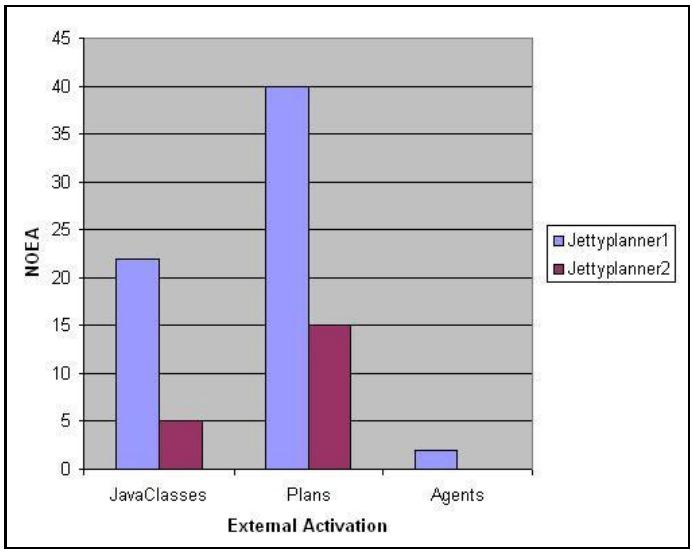

Figure 7.5: Measurements of Metric M5(NOEA)

Discussion

It can be seen in figure 7.5 that there is a larger amount of external activation in JettyPlanner1 than in JettyPlanner2.

To handle the ship-object, plans and java-classes in JettyPlanner1 contain ship-object references which are used to activate the object. The references need to be used each time the ship-object is involved in a process within the system.

In JettyPlanner2, activation occurs when the ship-agent receives events. The agent will then decide how to respond to the event, and the reply can be in accordance with a request. An example of a reference to the ship-object, and an event to the ship-agent, is given in section 6.4.2.

The ship-object in JettyPlanner1 has 64 external activations. These are reduced to 20 in JettyPlanner2 when the ship-object is replaced by an agent. This means that there is an reduction of 68.75% in external activation between components.

Conclusion

Based on the results for metric M5 in table 7.5 we reject hypothesis HO5 and choose hypothesis HA5.1.

We have seen that the number of external activation is reduced when we replace an object with an agent. We believe the reason for this is that the agent is autonomous and independent, and can be a participator in processes instead of being an object to be handled.

We are convinced that the tendency with reduced number of external activation will occur in other cases where agents are used instead of objects.

7.1.6 Hypothesis 6

Hypothesis 6 concentrates about the level of abstraction, and is given as follows.

H06 Use of agent-technology will not provide a higher abstraction level for modeling and implementation of applications.

HA6.1 Use of agent-technology will provide a higher abstraction level for modeling and implementation of applications.

Since its not possible to quantify the abstraction level using metrics, we have decided to do a qualitative assessment for hypothesis 6. We have examined the modeling against the following definition for abstraction.

Abstraction means that we concentrate on the essential features and ignore details that are not relevant. The transition from a problem to be solved to a model or a implementation is dependent of decomposition. The result is a hierarchical structure where the problem is described at different abstraction levels[Vli02].

In connection with implementation of applications, the term ”abstraction level” has the same meaning as ”level of programming language”. Definition for high-level programming languages is given below.

A high-level programming language is a programming language that, in comparison to low-level programming languages, may be more abstract, easier to use, or more portable across platforms. Such languages often abstract away CPU operations such as memory access models and management of scope[Wik06].

Discussion

We have used JACK to design and implement our two JettyPlanner versions. The foundation for JACK is the BDI Architecture[Age06], which is based on the attitudes Belief, Desire and Intention.

In the BDI Architecture, the agents have their own beliefset which contains the knowledge and assumptions about the environment and other agents. The desires of the agent is represented with a set of plans. Each plan can be used to achieve goals according to the current environment. The agent will use the beliefset to select an appropriate plan, when a plan is selected it will represent the intention of the agent[MLD04, Ølm05].

The understanding of the relationship between the attitudes and how they affect the behavior of agents, provides a natural way of modeling intelligence. The BDI model is based on human behavior and reasoning, and can therefore provide a control mechanism for intelligent action on a high abstraction level [MLD04, Ølm05]. Using terms like agent, capabilities, plans, events and beliefsets makes it possible to describe the problem area from a human viewpoint, ergo high abstractive, and therefore easy to understand.

JACK is a cross-platform development environment written in Java. The programming language is far from CPU operations, which can be seen in the code structure. Source code must be contained in entities, and the entities are quite specific. Therefore, it would be difficult to make a programming language with foundation in JACK. We use this as an argument, for calling the model and programming language high abstractive.

Conclusion

Based on our own experiences obtained during design and implementation of the JettyPlanner versions, we decide to reject hypothesis HO6 and choose hypothesis HA6.1.

We have experienced that modeling and implementation is on a high abstraction level when we use agent technology to develop our systems. Abstraction of the problem area is obtained using terms like agents, plans, capabilities and events. These terms are related to human behavior and reasoning, and are therefore easy to understand. The programming language is in addition portable across platforms and far from CPU operations.

7.1.7 Summary, Testing of Hypothesis

The results of the hypothesis testing can point out tendencies that will occur when agents are used instead of objects.

We have seen that the amount of code lines has increased with 16.28% in JettyPlanner2. Also the amount of entities increased with 17%. JettyPlanner2 has more lines of code and entities because of the necessary extension of events and plans related to the ship-agent.

Another interesting result of the tests is that the amount of couplings between the ship and other entities was reduced with 59.38% in JettyPlanner2. Agents do not provide any control point to external entities, and all communication is based on sending and receiving events. The reduced amount of coupling lead to a higher degree of modularity. Modularity is therefore obtained by increasing the number

of entities in the agent system, and as consequence, the number of code-lines. To achieve the same modularity in JettyPlanner1, without replacing the ship-object with an agent, more java-classes would have to be implemented. Interfaces for the ship-object could for instance have been implemented to make specific functionality accessible[EG95].

Couplings between entities are interrelated with external activation. Our tests of the hypothesis show that external activation of the ship was reduced with 68.75% when the ship-object was replaced with a ship agent. Events are used instead of external references to exchange data with the ship in JettyPlanner2. The agent is autonomous, which means that it can control and decide for itself whether it want to exchange data with other entities. This autonomy lead to reduction in external activation and therefore better encapsulation of functionality.

Modeling and implementation have been on a high level of abstraction when we developed our two systems. The problem area was transformed and decomposed using human terms like capability, events, plans and agents. Due to agent technology, it was easy to describe the problem to be solved. The transition turning the problem into models, and thereafter implementation, was not difficult. A positive impact is that the high abstraction level influences the development effort. The negative effects from the increase in lines of code, entities and functions, can be reduced due to the high abstraction level.

7.2 Experiences with Agent Technology and JettyPlanning

Jettyplanning is our application area. We identified three concepts; *Scheduling*, *Planning* and *Decision making* in section 4.3.1 to characterize jettyplanning. These concepts revealed challenges that we had to handle during implementation of our multi-agent systems. Our experiences with agent-based solutions related to our application area are presented in this section.

The developed JettyPlanner systems solve the *scheduling problem* by taking incoming ship under consideration one by one. Requests are thereafter sent to all berths. Berths with the right capacity and demanded product computes three cost alternatives for suggestion:

1. The cost is calculated for allocating the ship at wanted arrival time. If the berth is occupied by another ship at this time, the alternative will result in the other ship being knocked out.

2. If the berth is occupied at wanted arrival time, the cost is calculated for allocating the ship after the occupying ship.

3. If the berth is occupied at wanted arrival time, the cost is calculated for allocating the ship at first time arrival available.

Since ships can knock out each other in alternative 1 and 2, we had to set up a condition to avoid infinite loops. We decided that a ship can not knock out the ship that knocked it out in the first place.

The berth-agent decides which of the three alternatives has the lowest cost, and sends it to the jettyplanner-agent as a suggested allocation alternative. The jettyplanner-agent receives suggested allocation from all relevant berth-agents. The jettyplanner-agent will delegate the responsibility for the incoming ship to the berth-agent with the best alternative for allocation. The incoming ship will as a result be allocated

the berth that has suggested the lowest cost. The scheduling problem is solved by achieving the best allocation-solution for a ship at the present moment.

Decision making is strongly connected to the *scheduling problem*. Several decisions are made to find the best allocation alternative for a ship. The berth-agent is implemented to make a decision about which of the three allocation alternatives it will suggest for the jettyplanner-agent. The jettyplanner-agent is implemented to collect answers from all the berth-agents, and decides which berth will be the receiver. In our opinion, the agents in our multi-agent systems have proved to be valuable in decision making. They are capable of making decisions on their own, which was very useful during implementation of our systems.

We stated in section 4.3.1 that *planning* is difficult in changing environments. To make a schedule representable for the situation and optimal at all time, it has to be reconsidered each time a change occur. In our system, changes are reported to the system through inputs in the graphical user interface. The input generates events that inform the agents about the changes. The agents are then able to perform a rescheduling every time a change occur. As a result, our two multi-agent systems react dynamically to the environment. They handle changes and keep the schedule up to date.

7.3 Experiences with JACK and Prometheus

We have learned about development tools and methods related to agent technology during the work with our master thesis. We have studied the *Prometheus design tool*, and achieved practical experiences when we used it to design our own system.

JACK Intelligent Agents Agent Practicals was used as a guideline to get confident with the development environment. We got familiar with the most common possibilities JACK provides as a tool.

We have experienced that Prometheus and JACK have an abstraction level which is easy to understand and use to convert situations from the real world into models. We did the detailed design of our systems in JACK, which generated skeletons for entity files from diagrams. This saved us from a lot of work and the diagrams evolved together with the implementation. A positive consequence was that the models were up to date at all time.

The high abstraction level made some useful concepts in JACK easily accessible. We used the *Semaphore* to obtain mutual exclusion for plans. Also, a type of event called *InferenceGoalEvent* were used to obtain calculation of several answers for one event. Another concept that eased our implementation, was the use of a *ParalellMonitor*. The monitor makes it possible to do tasks in parallel by several agents[Age06].

It was challenging to think agent-oriented during the modeling process. Our earlier experiences are broadly speaking related to object-oriented programming. During implementation we sometimes felt that we did not make the most out of the possibilities JACK provides as a tool.

We experienced that JACK generates a huge amount of extra files with code, which was difficult to comprehend the meaning of. In the beginning, we tried to use the version control tool, *Concurrent Versions System(CVS)*, to keep track of our work and changes in the code. It was difficult to keep all the files up to date with all the extra files generated by JACK. Some of the clean-up operations related to code did not work properly in JACK. Not all the generated files where deleted in situations where we renamed an entity. This introduced errors when the application were

compiled and executed. We had to manually delete the files.

7.4 Validity Concerns

We discussed validity threats in section 5.3.4. Some of these threats had to be accepted, while others needed to be addressed to obtain valid results. We will now discuss the validity of our results.

Conclusion Validity *Low statistical power:* We have only performed one comparison. Some of our conclusions can therefore be erroneous due to lack of data.
Reliability of measures: We have used metrics to test our hypothesis. Metrics are quantitative, and our measurements are therefore objective.

Internal Validity *Selection:* Our selection of objects may not be representative for all possible outcomes.

Construction Validity *Experiment construction:* We have defined measurements, hypothesis and treatments to obtain a good experiment construction. We have tried to construct an experiment with a clear relation between theory and observation.
Mono-operation bias: We have performed a quasi-experiment which means that our experiment may not give the full picture of the theory.

External Validity *Interaction of selection and treatment:* We may not be representative of the population we want to generalize to, namely the developers.
Interaction of setting and treatment: We have used development tools and methods that are up to date to have an experimental setting that are representative for the industrial in practice.

Chapter 8

Conclusion and Further Work

In this chapter we will give a conclusion for our research and suggest issues for further work. The conclusion is given in section 8.1, suggestions for the future are presented in section 8.2.

8.1 Conclusion

It is claimed that agent technology has a large degree of utility value for enterprises due to agent's properties and behavior. Achieved benefits like reduced coupling, better encapsulation of functionality and high abstraction level are all related to multi-agent systems.

We have performed an experiment to investigate theory about agent-technology and potential application areas. We have developed two applications for jettyplanning at Mongstad. The difference between these applications is that incoming ships are represented as software-objects in JettyPlanner1 and software-agents in JettyPlanner2.

The experiment was based on claimed benefits about agent-technology. We wanted to investigate what we would achieve within functionality and effort if we replaced software objects with software agents. Several hypothesis were written to cover different aspects. To obtain reliable and valid results, quantitative metrics were defined to be used in the evaluation.

We have tested six hypothesis. The results can be related to four claimed benefits. One of the benefits is *reduced development effort*. According to the results of the testing, the development effort in lines of code and number of entities has increased with approximately 16-17%. The number of functions increased with approximately 12%.

Reduced coupling and *encapsulation of functionality* are also claimed to be attendant to agent technology. In our experiment, the amount of couplings was reduced with approximately 60% when the ship-object was replaced with the ship-agent. The amount of external activation of the ship entity had a reduction of approximately 70%.

It is asserted that agent technology has a *high abstraction level*. We have experienced that modeling and implementation were on a high level of abstraction when we developed our two systems. We used human terms to transform and decompose the problem area. Due to agent technology, it was easy to describe the problem to be solved, and transform it into models and implementation. JACK agents can

be developed and deployed on any platform, and the programming language is far from CPU operations.

According to our results will agent-based solutions have a slight increase in development effort. Due to a considerable reduction of couplings and better encapsulation of functionality, this increase can be viewed as convenient. The results related to reduced coupling and encapsulation of functionality are so good that the gain weights up for the extra effort related to the increased number of code-lines, entities and functions. In addition will the high abstraction level affect the development effort and reduce the negative effects even more.

The number of code-lines for each of our two multi-agent systems are approximately 1100 and 1300. Since some of our results are very distinct, we believe that the discovered tendencies also will be revealed in larger systems. The reduced coupling and the encapsulation of functionality will influence the software architecture to a high extent. It will probably be easy to implement and maintain agent-based systems of larger scale.

Agent entities can be used as natural building blocks in larger systems. The increase in development effort will probably be noticeable, but on the other hand, this increase can be characterized as trivial due to the benefits related to higher abstraction level.

We believe that the results are a consequence of the extra functionality offered by agent-technology. The agent is autonomous, and participate more actively than an ordinary object. It can control and decide for itself whether it want to communicate and collaborate with other entities.

An autonomous agent lead to reduction in external activation and therefore better encapsulation of functionality. The use of events and plans reduces the amount of couplings between entities, and a higher degree of modularity is obtained as a consequence. We believe that our results indicate that this can be a tendency in other cases where software-objects are replaced with software-agents.

Planning, scheduling and decision making can be challenging to provide, and they characterize our application area. Optimization and decision making are interrelated with NP-complete problems[THC01], and traditionally development tools have shortcomings dealing with these kind of problems. Agent technology can not solve NP-complete problems, but can offer new and better alternative solutions.

We have experienced that agent technology provides functionality to handle our application area in a new kind of way. The multi-agent system tries to make a nearly optimized schedule for the jetty at Mongstad at any moment, by taking real-time decisions based on the environment. Due to agent-technology, new dynamic solutions can be found to handle complex application areas.

8.2 Further Work

Agents offer new solutions to already existing and well known problems. They can coordinate work, arrange themselves to changes in the environment and handle huge amounts of information.

The results from our experiment indicate that agent-technology actually can be related to benefits like reduced coupling, encapsulation of functionality and higher abstraction level. Our two multi-agent systems are of a small scale. It would be interesting to perform the same kind of experiment on larger multi-agent systems. If the results turned out be approximately the same, agent technology will be extremely valuable for enterprises.

We have seen that agent technology offers functionality that can deal with complex problems where planning, scheduling and decision making is necessary. These characteristics can be related to many other problem areas.

Multi-agent systems can for instance be used in public health services to provide scheduling of appointments. Nurses and doctors could then use more of their time to help patients. Doctors make decisions every day, a multi-agent system could provide decision support by giving important information and suggest medical solutions. More task automated systems would probably decrease the medical waiting list, which in the longer term would decrease government spending.

We have seen that agent-technology can be valuable in the oil- and gas industry. In addition to jettyplanning, there are several other application areas where agent-technology can give valuable contributions. Agent-based solutions, can for instance be used to monitor pipelines out in the north-sea. The agents could control the pump performance and the substance of crude oil at each pipeline. Due to a high abstraction level, avoidance of centralized software could be possible. Necessary calculations could be performed at each pipeline, and thereafter gathered if necessary. If a system is distributed, problems can be divided into subtask and be solved in parallel. The performance of the system would increase as a consequence of the distribution[Emm97].

We have performed one experiment to demonstrate applicability of agent technology. We have seen that its possible to replace a passive object with an active agent, and obtain a more dynamic system. For the future, several experiments should be performed to validate our results. Another possibility to demonstrate the benefits of agent technology is to compare a multi-agent system against a complete object-oriented system.

Bibliography

[Age06] Agent Oriented Software Group. *The Agent Oriented Software Group (AOS)*, http://www.agent-software.com/, 2006.

[Bar01] Arran Bartish. A comparativ analysis of intelligent agents and state machines: Models for the game domain. Technical report, RMIT Computer Science Department, Australia, 2001.

[Cir06] Cirrus Logistics. *The SEABERTH Solution to effective berth sceduling*, http://www.seaberth.com/, 2006.

[CW00] Martion Host et. al. Claes Wohlin, Per Runeson. *Experimentation in Software Engineering, An Introduction*. Kluwer Academic Publishers, 2000.

[EG95] Ralph Johnson et. al. Erich Gamma, Richard Helm. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley Ltd, 1995.

[Emm97] Wolfgang Emmerich. *Distributed System Principles*. Department of Computer Science, University College London, http://www.cs.ucl.ac.uk/staff/W.Emmerich/lectures/ds98-99/dsee3.pdf, 1997.

[FG96] Stan Franklin and Art Graesser. Is it an agent, or just program?: A taxonomy for autonomous agents. Technical report, Institute for Intelligent Systems, University of Memphis, 1996.

[HB04] Helga Neureiter Christian Herneth and Heimo Bürbaumer. A collection of agent technology pilots and projects. Technical report, Capgemini, Germany, 2004.

[HBH05] Ole Henrik Olsbu Heimo Bürbaumer, Arnt Vegard Espeland and Rune Hovde. Shipment & allocation, agent solution workshop, mapping of business areas to sw agent functionalities. Technical report, Capgemini, Norway, 2005.

[Kea04] Elin M. Kristensen and Sigrun Lurås et. al. The complot: A performance about about statoil world operations. Technical report, Rotvoll Operation Center, Statoil, 2004.

[Kri05] Elin M. Kristensen. Agent technology, tdt 4735 software engineering, depth study. Master's thesis, NTNU, 2005.

[LO06] Einar Landre and Ole Henrik Olsbu. Facts about present practice, 2006. e-mails from Ole Henrik Olsbu and Einar Landre.

[Luk] Andrew Lukas. Microsoft PowerPoint presentation made by Andrew Lukas in AOS.

[MD02] Scott A. O Malley and Scott A. DeLoach. Determining when to use an agent-oriented software engineering paradigm. Technical report, Department of Electrical and Computer Engineering, Air Force Institute of Technology Wright-Patterson Air Force Base, Ohio and Department of Computing and Information Technology Sciences, Kansas State University, 2002.

[MLD04] Ronald Ashri Michael Luck and Mark D'Inverno. *Agent Based Software Development*. Artech House, Boston, London, 1 edition, 2004.

[Nnw96] Hyacinth S Nnwana. Software agents: An overview. Technical report, Intelligent Systems Research, Advanced Applications and Technology Department, Cambridge, http://www.sce.carleton.ca/netmanage/docs/AgentsOverview/ao.html, 1996.

[OAS92] *The OASIS Air Traffic Management System*, 1992. Proceedings of the Second Pacific Rim International Conference on Artifical Intelligence (PRICAI92).

[PTW05] L. Padgham, J. Thangarajah, and M. Winikoff. Tool support for agent development using the prometheus methodology. Technical report, RMIT University, Melbourne Australia, 2005.

[PW04a] L. Padgham and M. Winikoff. *Developing intelligent agent systems - a practical guide*, chapter 1. John Wiley and Sons, 2004.

[PW04b] L. Padgham and M. Winikoff. *Developing intelligent agent systems - a practical guide*, chapter 3. John Wiley and Sons, 2004.

[PW04c] L. Padgham and M. Winikoff. *Developing intelligent agent systems - a practical guide*, chapter 4. John Wiley and Sons, 2004.

[PW04d] L. Padgham and M. Winikoff. *Developing intelligent agent systems - a practical guide*, chapter 5. John Wiley and Sons, 2004.

[PW04e] L. Padgham and M. Winikoff. *Developing intelligent agent systems - a practical guide*, chapter 8. John Wiley and Sons, 2004.

[PW04f] L. Padgham and M. Winikoff. *Developing intelligent agent systems - a practical guide*, chapter 9. John Wiley and Sons, 2004.

[PW04g] L. Padgham and M. Winikoff. *Developing intelligent agent systems - a practical guide*, chapter 10. John Wiley and Sons, 2004.

[SL03] Wei Xu Shaohua Liu, Jun Wei. Towards dynamic process with variable structure by reflection. Technical report, Technology Center of Software Engineering, Institute of Software,The Chinese Academy of Sciences, Beijing, 2003.

[Sta06] Statoil ASA. *About Statoil, our business*, http://www.statoil.com, 2006.

[Ste05] C.A. Rouff M.G. Hinchey J.L. Rash W.F. Truszkowski R. Sterritt. Towards autonomic management of nasa missions. Technical report, NASA Goddard Space Flight Center USA, 2005.

[THC01] Ronald L. Rivest et. al. Thomas H. Cormen, Charles E. Leiserson. *Indroduction to Algorithms*, chapter 34. The MIT Press, Cambridge, Massachusetts, 2 edition, 2001.

[Vli02] Hans Van Vliet. *Software Engineering, Principles and Practice*. John Wiley and Sons Ltd, 2 edition, 2002.

[Wik06] Wikipedia.org. *Wikipedia, the free encyclopedia*, http://www.wikipedia.org, 2006.

[Woo02] M Wooldridge. *An introduction to MultiAgent Systems*. John Wiley & Sons Ltd, 1 edition, 2002.

[WW05] Minhong Wang and Huaiqing Wang. Intelligent agent supported business process management. Technical report, Department of Information Systems, City University of Hong Kong, Hong Kong, 2005.

[Ølm05] Jørn Ølmheim. Software agents, for dynamic java enterprise applications. Technical report, Statoil, 2005.