NTNU
Norwegian University of
Science and Technology

Selection and use of third-party Software Components

Study of a IT consultancy firm

Martin M Syvertsen

Master of Science in Informatics
Submission date: April 2011
Supervisor: Eric Monteiro, IDI

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

Preface

Being a software developer is as any trade something that harbors its own culture and its very own special understanding of the world and from a very small perspective. This thesis studies the methods of developers in their own world and tries to understand how they work and why they do the things they do.

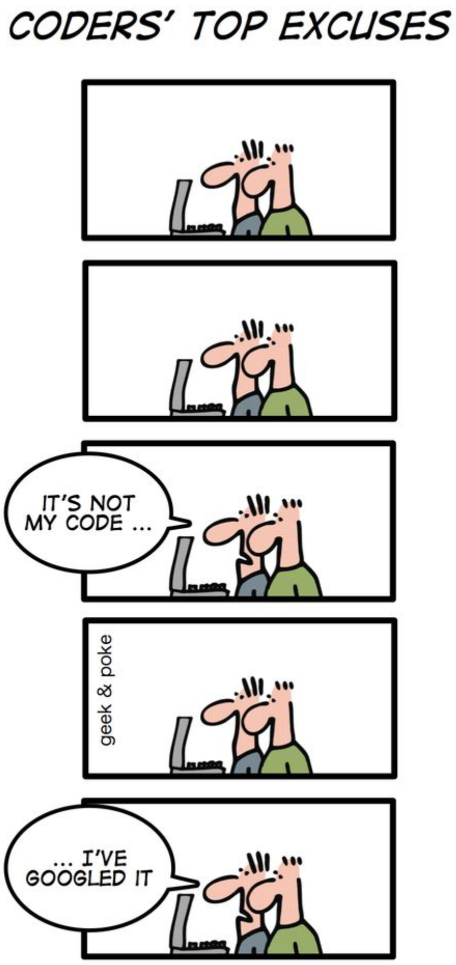


Figure 1: by Oliver Widder under CCAND 2.0 License [6]

Abstract

The use of third party software components is increasing. By looking at developers at a Norwegian IT consultancy firm I find that developers are using components at an individual level and there is no leading agenda that promotes reuse. Developers that are used to finding and using components do so often and with few problems this practice is aiding and improving software development. However the frequency of use is not as high as it could be, either because of limitations of reuse or lack of knowledge and skill on how to find components. By encouraging the use of small software components and proposing simple guidelines on how to do so companies could increase both reuse and the benefits of them.

Keywords: open source, commercial off the shelf, software reuse, knowledge management,

Acknowledgments

I would like to thank my supervisor professor Eric Monteiro for helping me not only write and form this thesis but making it happen in the first place. His feedback has been crucial and without him I would have been lost in how to write and work on this thesis.

I would like to thank Acando and all of it's employes who I have interviewed, observed and talked with during my work and for letting me inside their company doors. A special thanks goes my advisors at Acando for their support, help and for making this thesis possible.

I would like to thank my friends and family for their support and motivation.

I especially would like to thank Ida for her patience, support and understanding.

Contents

List of Figures

List of Tables

1 Introduction

Almost as early as there has been software development there has been an ideal of reusing it efficiently. Seemingly the software developers are often solving the same problems over and over again and the goal of simply reusing previously written software, be it self made or found elsewhere, has a high potential to drastically improve software development. If any new system was made of 90-95% existing code then one could easily see software having better quality, faster development and costing less. although the reuse of software components in professional software development is increasing there is still a long way to go towards achieving both the levels and the effects of reuse that is wanted. This thesis is a case study of small component reuse at a software consultancy firm in Norway and looks at the state of reuse amongst developers. By looking at their methods, frequency of reuse and problems with components this thesis seeks to identify potential problems and how reuse can be increased.

1.1 Research Question

Research Question 0:
The use of software components in enterprise software development is increasing, both of known components and unknown components. With this increased use and availability of components, what problems are developers facing, how are components being selected and what can be done to improve on that process? Why is software reuse not ubiquitous for all developers and all projects? What are the challenges of finding and using smaller software components and what can be done to increase reuse? There are OSS components and COTS components, what is the difference between these categories and how is this affecting reuse? Why has the potential of software reuse not yet fulfilled?

From Req. 0 it can be extracted part-questions that are easier to discern when later discussing the findings and analysis of the thesis:

- **Research Question 1:**
  What is the state of software reuse?
- **Research Question 2:**
  Why are some developers using more components than others?
- **Research Question 3:**
  What are the knowledge-related problems with component reuse?
- **Research Question 4:**
  In what degree is there a difference between COTS components and OSS components?
- **Research Question 5:**
  How can use and selection of software components improve?

1.2 Thesis structure

- Literature - Looking at research on topics related to this thesis
- Research Approach - The research methods that were used
- Case - The case findings and initial results
- Discussion - Reviewing issues, problems and implications of the findings
- Conclusions - Review of what has been found and topics for further work

2 Literature

2.1 Software Reuse

2.1.1 Defining a software component

The problem that this thesis is studying is the use of small software components in software development. It is therefore important to accurately define what is meant by a small software component. For many software projects there will be a predetermined set of components that are familiar and almost staples when it comes to software systems. These components are often big, they have a large feature set, and can be considered industry de-facto standards in their respective fields of development.

For example a DBMS is a component found in most software systems and for the DBMS component there is a handful of well known and well used vendors and technologies available. So when choosing or deciding the main DBMS of a system the software architect will have great prior knowledge and experience when he or she chooses for example MySQL or Microsoft SQL Server, these are not small components and fall outside the scope of this study. Such components will also in many cases be pre-defined from the customer, they come as demands and fundamental requirements from customers who in most cases already have a computer infrastructure and environment. Examples of components that fall outside the definition of small software component are found in table 1.

Name	**Vendor**	**Type**	**Since**
Microsoft SQL Server 2008	Microsoft	DBMS	1989
Apache HTTP Web Server	Apache Software Foundation	Web server	1995
.Net	Microsoft	Development Framework	2002
SAP ERP	SAP AG	ERP	1972

Table 1: Familiar software components

The type of software that thesis is focusing on are often small software components with small feature sets that are chosen not by the major software architects but by the developers that are building the system. If the developer for instance needs a PDF producing component for a system running on the Python programming language and a search for "Python PDF" returns 59 hits on SourceForge [13] and one such component is chosen and used for the system, then that falls under this study. It is important to understand that small does not imply that the component is small in size but rather small in it's function and role in the larger system. It may be important, but not fundamental to the operation and building of the system, those components are usually chosen beforehand and in the planning and modeling phase of the development. Often these small components will be libraries or extensions to existing programming languages or frameworks. Examples of small software components are shown in table 2.

Name	**Vendor**	**Type**	**License**
jsoup Java HTML Parser	one developer	HTML library	The MIT License
MiG Calendar	MiG InfoCom	Calendar GUI	Commercial
Aspose.Pdf for Java	Aspose	PDF library	Commercial
pyOpenSSL	3 developers	python SSL wrapper	LGPL

Table 2: Small software components

To summarize a small component for the purpose of this thesis is a component selected by developers after the overall systems has been specified and modeled. It is often small in functional scope and will not be a critical part of the system itself.

2.1.2 What software reuse means

Software reuse is the method of using existing code when creating new software, a simple and accurate definition of software reuse:

> "*Software reuse is about methods and techniques to enhance the reusability of software, including the management of repositories of components.*" [5]

The basic premise here is the same as for any engineering discipline and that is to create larger systems (buildings, factories, roads) by composing together

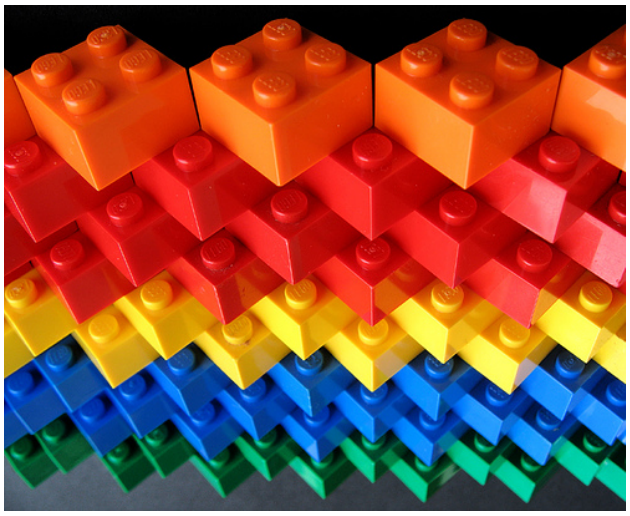


Figure 2: Software components can be thought of as LEGO blocks

existing components that have been used in other solutions. The simplest analogy would be that of LEGO blocks, figure 2 shows some LEGOs. You start with existing pieces of different sizes and functions and simply assemble them together to build the end result. Some projects require that one invents everything from scratch and many projects will require that original problems are solved, but reusing existing and tested solutions will often be possible and preferred. In the early days of software development much focus was on original development, but today it is more common to reuse components, be they purchased, free or self maintained. The goals or wanted benefits of software reuse as proposed by [42] are:

- Better quality software
- Faster development
- Lower costs of development

When using components and actively designing software to be reusable it is important to adopt the software engineering approach to fit as well. Reuse has long been a practice in software engineering but in the beginning this reuse was more ad-hoc [42]. Pressmann presents Component-based software engineering (CBSE) as a method to develop systems by reusing software components that is formalized and developed. By adhering to principles for CBSE one systematically creates a system that is in itself component based and it will be easier to implement third-party components as well as reuse the components that are made for the system in other projects. In the case of CBSE a component can be defined as:

> "*An individual component is a software package, a web service, or a module that encapsulates a set of related functions (or data).*"

By creating components that incorporate loose couplings that are easy to replace and interact by use of simple and well defined interfaces, it is easy to assemble and swap components with better working ones.

although there are several benefits with CBSE these are only theoretical and not guaranteed results. There are several pitfalls, questions and challenges when applying CBSE, for example:

- Can complex and large systems be constructed of different components?
- Can libraries of components be created in such a way that they are accessible to the people who need them?
- Can existing components be found by those who need them?
- How does a developer select the right component for the job?

There are several proposed steps for CBSE development, here are the most important concepts from Pressman [42]:

Component qualification
Component qualification is about ensuring that candidate components will perform the functions that are required by the system. By considering aspects like application interface, requirements for tools and integration, runtime environment, resource usage (CPU, memory etc.), programming language, one ideally picks the best component suited for the task. It is easy to evaluate internally developed components as they are familiar. Third-party components are harder to evaluate because they may be unfamiliar, poorly documented or closed-source.

Component adaptation
Component adaption occurs when the components have been selected and integration towards the final systems begins. Even tough the qualification process selects the best possible and compatible components there is often much work to do in building bridges and so-called wrappers around the components so that they are usable in collaboration with the rest of the system.

Component composition
Component composition is the final step where adapted components are fitted together in the larger system often using a central infrastructure of data-busses and coordination. This is the final step for integrating the acquired components into what will become the final system.

The problem of selection is one that is easy to spot but hard to solve. How can a developer know or check if a component is trustworthy or "up to scruff" when he or she has no previous experience with the component? As the practice of reuse increases the market becomes more and more saturated with more and more components that overlap in functionality, domain, programming language and runtime requirements, making the list of candidates larger and larger. With the addition of open source components the number of components available are approaching the tens and hundreds for almost any purpose of component.

To this problem comes several solutions, most of which involve methodic and formal approaches like [25] [2] . Many of these involve the basic steps of gathering as many components possible, applying different weighted metrics and then choosing from an end result list that is sorted by numeric values. There has been a lot of research on these kinds of formalized methods and some companies and organizations provide frameworks like the Open Source Maturity Model [47]. Also as suggested in [27] such reuse models are exhaustive and little efficient because they will require more time as more components are added to the market.

Hauge points out in [17] that this research has been more academic than realistic. He notes that in reality developers seldom use such formalized approaches as they either take to much time or they simply do not know that they exist. Often research points to two kinds of selection methods, formalized or ad-hoc. Hauge notes that the ad-hoc method is a simplification of developers simply applying experience and resource use that is not ad-hoc nor poor in use and results. He proposes a third approach that he calls situated where the developer chooses from experience, knowledge and resources available in the organization and on the internet. It is situated because it is adapted to the constraints of the project in terms of time, money and resources available.

2.1.3 Software reuse in practice

One important question is if the premise and promise of software reuse has been fulfilled or not. In 1995 Clements claims:

> "*These and other concerns make CBSD a trap for the naive developer. It requires careful preparation and planning to achieve success. Interface standards, open architectures, market analysis, personnel issues, and organizational concerns all must be addressed. However, the benefits of CBSD are real and are being demonstrated on real projects of significant size.*" [5]

In other words that faster, cheaper and better software systems can be made with the correct use of components. And yet the fact remains that software systems require time, effort and money in almost the same way as before, if not more because the expected features and uses of systems is steadily increasing. From the same year as Clements [5], in 1995 a report talks about the success and failure rates of software projects where 16% succeeded 53% challenged (failed to meet deadlines/budgets/requirements) and 31% were canceled [18]. In 2009 these numbers were 32% success, 44% challenged and 24% failure [49]. Such numbers show a slightly positive development, but not enough to not still be disheartened by the state of software development. It is not clear off course if these projects take use of the principles of software reuse, but it would be safe to assume that if reuse had been proven to be a silver bullet to failed software development it would certainly have been widely adopted.

Gartner predicts that OSS components will be used in 90% of all new software in 2012 [14]. So while reuse is increasing, the effects of reuse are perhaps not matching that of the theory. It could be argued that reuse is helping, but not by metrics that are being considered like quality in relations to features or time spent in relations to features.

So the premise of reuse is that if applied successfully it will increase software quality and development speed. But exactly how is one supposed to find the software to reuse? Several proposals and studies have been made on selection methods, often formalized and strict that utilize statistics and assumes collecting larger lists of candidate components. But what about components that someone else already has used to great success? Complex and potentially expensive selection methods could easily be replaced, in theory, by a large database of known and tested components. Such a database could either be internal and private, external and private or external and public. This is a problem related to knowledge and with companies that have a large and distributed workforce in which people are working on different projects but perhaps solving the same problems, selecting the same (or different) components for reuse the challenge is not to simply employ reuse practices in a good way, but rather to exchange knowledge of reuse, both components and methods, across the company.

2.2 Commercial Off The Shelf

Commercial off the shelf software (COTS from here on) is simply put ready made software that is available for purchase. It's purpose is to aid and supplement software development by either providing complete solutions or smaller components like software libraries or GUI components. COTS software is professionally developed by a software vendor and they are sold with different strategies like single license, volume licenses, with support. The potential benefits of COTS are reduced development time, reduced costs and increased software quality. [28]

It has been a long standing goal in computer science to create software that is so modular that creating a new system is simply the assembling of existing components and COTS is the marketplace result of this goal. A good example of the goals and ideas behind COTS software would be:

> *"as government budgets shrink and the desire for increasingly complex systems continues, there is rising interest in leveraging the use of commercially available products whenever possible."* [3]

The government mentioned here is the US government that increasingly emphasizes the use of COTS in development of IT solutions and systems. A small list of COTS components that illustrate typical COTS components is shown in table 3.

Name	**Vendor**	**Type**
Word 2010	Microsoft	text editing
Windows 7 Professional	Microsoft	Calendar GUI
jPDFViewer	Qoppa Software	PDF library
OfficeHTMLFilter	Antenna House	Data conversion

Table 3: Example of COTS software

Word and Windows can be considered the ultimate COTS product because they provide such a generic functionality that almost any business can buy and use the exact same software. Such standalone components are not the focus of this thesis however. For IT systems that are less generic original software is often needed, but here COTS software components can aid with standalone functionality that within the custom made system is generic enough that several projects have use for it.

The potential benefits of using COTS software are huge, but the drawbacks and challenges of COTS are not insubstantial. In fact if the goal is extensive use of COTS one would have to change and re-learn how to develop and maintain software systems. In [3] the authors present several new and key aspects to consider when working with COTS:

Development Lifecycle Changes
Explained as re-examining and re-thinking several lifecycle activities of the development process, regardless of model being used (waterfall, spiral or iterative).

Developing requirements
When writing the requirements for the system the author must be aware of the existing components in the marketplace. In other words the requirements for the system should closely match that of existing features available in the current COTS marketplace.

Selection
The selection of components must occur in parallel with development of architecture and requirements, this to ensure that a larger set of components will be used. Applying a more traditional sequence of requirements, architecture and then selecting components may exclude a large set of components.

Testing
Testing of systems with COTS components is different because you are essentially performing “black box” testing. As such the need and requirement to test the COTS components will have to be assessed and how to test them will differ from traditional testing.

Maintenance
The maintenance of a system with heavy COTS use will require additional insight into the update schedules of the different COTS vendors and also the different licenses that the COTS component are purchased under. Some may come with a time-based license and will require re-licensing, others may have a pay-for-upgrade scheme that requires additional payment to be able to update to the latest version of the component. Such factors must be addressed when planning for maintenance with COTS in the system.

These are the main concerns when it comes to the development of the system with COTS, in addition there are business and management concerns:

- Knowing and evaluating technologies and products
- How to manage systems that use COTS
- Building new business cases
- Building metrics for business assessments
- Knowing the COTS marketplace

The article also points out that even tough COTS has potentially many benefits, there is no simple and bullet-proof method to COTS based development, it still requires a solid development team that are good at integrating different components together to a larger system. The publication "Development with Off-The-Shelf Components: 10 Facts" [28] shows that the use of OTS, both OSS and COTS, is widely used as found in the IDC survey from 2007 shows that more than 50% of developers used components in their software development projects. Further it discusses several myths about the use of software components and criticizes much of the research and work that has been done on the subject, as does Hauge in his PHD thesis [17]. The critique comes to the research that has been done on the selection methods that have been proposed by academia. Much of this research has focused on formalized methods that often employ metrics and time consuming collection of components that meet the requirements specification, these methods have been shown to never se much actual use in the software industry as either they are two time-consuming and expensive or they are simply not known outside of academia [28].

The aforementioned formalized methods are several kinds of metric based selection that has focused on objectivity and repeatability in attempts to bring more precise science to the selection process. These frameworks have been used and tested in some projects, notably these projects have been very strict and systems critical projects that had and need the time that such formalized methods require. Other research has proposed more revised formalized methods like one developed by Kar and Hareton [27], they proposed a domain-based model to improve results and efficiency compared to pure intuition (subjective) or direct assessment (expensive) methods. Although such studies show promise, they seldom move from academic papers to actual use in the industry. In reality, as shown in "Ten Facts" [28] the methods actually most often used are ad-hoc or as added by Hauge [17] they used what he calls situated methods that are based on experience, knowledge, internet and organization resources. Hauge talks specifically about OSS components but the same principles of knowledge gathering and selection still applies. With ad-hoc and un-formalized methods we are talking about a developer that simply relies on experience or some form of information retrieval. Methods frequently used are:

- Own experience
- Asking colleagues
- Search engines
- Mailing-lists (both internal and external/public)
- Component database

With the selection methods often being ad-hoc one would perhaps expect unforeseen problems with heavy COTS usage, however findings report that this is not necessary the case:

> “The results show that OTS components normally do not contribute negatively to the quality of the software system as a whole, as is commonly expected” [29]

However this article also notes risks and problems with such studies as they often do not take into account the integration work being done be the developers implementing the components. Some cite the benefits of COTS being so strong that software developers are forced as an economic necessity citing efficiency and quality as major driving points [54]. If the potential benefits are as great as those being proposed then there is absent perhaps evidence of software projects quality increasing and development times decreasing, however such data is hard to find. One answer could simply be that systems are becoming more feature rich and also that although COTS components help software development they only provide generic solutions and original problems and systems will still be a challenge to develop. There has also been concerns regarding long term effects and costs of COTS use but few conclusions have yet to be made [1].

2.3 Open Source Software

2.3.1 History of Open Source

The term Open Source Software (OSS from here on) has meant different things over the course over a few decades and when discussing OSS and it's use it is important to assess the status, role and meaning of open source today. Historically one could say that the concept of open source formally began with the creation of such groups as the Free Software Movement in 1983, but there has always existed open source software. When a student, teacher or author shares code that solves a problem, or a sample program that demonstrates a feature, then that is open source software. From the roots of software development in such academic groups and hacker communities grew a more formalized definition of what open source is and means.

In The Cathedral And The Bazaar [41] the author Raymond presents two kinds of open source development, the cathedral way is open source software that is mainly developed and driven by a company like GNU Emacs. In the bazaar way the software is developed by anyone who wants to participate. This is the idea that open source is a development method, others say that open source is a philosophy in the sense that software should be free and open for change from everyone. This view is held and fronted by the Free Software movement, started by Richard Stallman in 1983 when he launched the GNU Project, a free Unix-like operating system. It's important to explain that free in this context means free as in free speech and not as in free beer. In 1998 the term "Open Source" was chosen by several people involved with the free software movement to label their solutions and software. This as a change from labeling it "free software" and enabled the projects to distance themselves from the ideological meanings of the term "free software". So is OSS a method or an ideology? In the context of this paper and today's general understanding of OSS we can say that OSS is:

> "Open source describes practices in production and development that promote access to the end product's source materials."

From changing the name from "free software" to "open source" came a commercialization of OSS that made it more viable and used by enterprise businesses. In the article "The Transformation Of Open Source Software" [9] Fitzgerald proposes the term "OSS 2.0" as a new definition on a new kind of OSS that is more business oriented. He claims that the open source phenomenon has gained mainstream recognition as valuable contributors of software. This has affected how OSS software is perceived and also how it is developed in many cases. Fitzgerald explains that the term “OSS 2.0” is a new kind of open source software that is more commercially driven and has much more in common with proprietary software development than traditional FOSS/OSS software. This has been a gradual transformation and it is hard to pinpoint, but seeing companies like IBM, Google and RedHat basing business on Open Source Software, not only leveraging but helping and steering the development of it, is a clear shift from hackers in their basement creating software in collaboration over the Internet. With this shift comes more business models that makes it more commercially viable to choose and use open source software and proprietary software is no longer the only choice for enterprise development. This also applies strongly when it comes to the practice of software reuse and smaller software components.

2.3.2 OSS Quality

Several sources claim that open source software has higher quality than commercial or proprietary software [32]. They often cite high profile projects like the Apache HTTP Server [50] and Mozilla Firefox [33] and point to the relative numbers of these projects like use, patches, commits, and so on to show that these open source software projects are creating high quality software. This is not necessarily an assumption that is wrong but it may simply only hold true if you look at such projects like Apache and Mozilla Firefox. These are well established projects that have a strong following both in corporate and non-corporate communities. Some studies like [32] and [57] attempt to quantify and analyze these differences in quality into metrics such as downloads, commits, number of active developers and so forth, however they may omit that these numbers alone does not explicitly mean that the software is of high quality.

In one famous article on open source software quality [41] it was said that “given enough eyeballs, all bugs are shallow”. This is a statement from the Cathedral and the Bazaar in which it is presented as “Linus’ Law” where Linus Thorvalds, the founder of the Linux operating system, stated that “Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix will be obvious to someone.”. This is a concept that prevails in talks about open source but its problematic to generalize any open source project as high quality because of this “law”. The problem being simply that there are a huge amount of open source projects and studies of open source project repositories like Github, Google Code and SourceForge reveal that the average numbers for a single project is not that of even tens or hundreds of developers. In one study of Sourceforge projects the average number of developers were low, on average the number of developers was four and the most frequent number was one [26]. In addition to this there are also studies that show that there is not necessarily any correlation between the number of developers and the quality of the software [36]. The more sensible conclusion is that big projects that have a large number of user that are open source will benefit from the Linus' Law, projects like Linux, Mozilla Firefox, Apache, MySQL and so on. Small projects that have one or two developers and have a following of 10-20 people means that the software quality is not in any way guaranteed to be better just because it is open source. Some studies like [45] suggest that OSS components do have high quality, but this study and others like it have looked at components chosen by software developers, not on the average or max/min quality of OSS components at large. It is like saying that "good components are good" or "some OSS components are good which means that all OSS components are good". The simple truth is that OSS software is as any software, the quality varies from project to project.

2.3.3 OSS development methods

The quality of open source touches on another important aspect, namely the development methods that open source projects use. In the early days the projects would live on the Internet chat-rooms, email lists and simple websites. They were driven as hobby projects by individuals with a desire to "scratch an itch" or simply fill a void in the software library. The development method then was a highly distributed and perhaps ad-hoc one that did not resemble the company driven development projects that employed RUP, waterfall or spiral development methods. In the article "Group awareness in distributed software development" [16] a study of open source development methods reveal simple tools like email lists and forums work as awareness mechanism that make it possible for the project to be developed with a high degree of interoperability even tough the developers themselves never sit in the same room or actually physically interact with each other. There is no real magic development method to be found other than that the developers are perhaps highly motivated and pay attention to what is happening on the information channels of mailing lists, forums and chat rooms. Several studies have been made on these seemingly ad-hoc development methods that have produced several high-quality projects like Linux and Apache, but efforts to extract them and use them in commercial development are few if any although there are suggestions in studies such as [16].

Many open source projects are developed in this distributed and seemingly ad-hoc way even today. Modern revision tools like CVS, SVN and recently GIT have improved on the distributed development working environment and there exists several large repositories and websites that are dedicated for hosting OSS projects. These sites offer free hosting for OSS projects and even features some levels of documentation and management oversight. The projects vary from desktop software to smaller software libraries. Table 4 shows a small overview over some of the biggest repository sites.

Name	**Type**	**Number of projects**	**Since**
Ruby Gems	Ruby libraries	22403	2009
Github	projects and libraries	1 million	2008
Google Code	projects and libraries	5648	2005
Sourceforge	projects and libraries	260000	1999

Table 4: Small software components

Between them these sites contain over 1.2 million projects and the numbers are growing. Most projects are small with one or two developers but some projects manage to gain attention and the bigger projects like [7] have for example 900 contributors and also have a high degree of activity in commits and usage. These kinds of projects represents in many ways the modern version of OSS 1.0 projects in that they usually are 100% free, mostly non-commercial, seemingly ad-hoc and based on simple development methods and tools, development is lead by a hand full of individuals but anyone who has the know-how can apply to commit work into the projects. Projects that are popular, big and become important have often become full-time efforts and business for the creators of the program. They become the “gatekeepers” so to speak as to what is committed and put into the official line of the program. When interests and stakes become a source of dispute and conflict with the management, or within the management, another aspect of OSS projects appear. Should the community or management become unfriendly one or more parties may decide to fork the project. This means that the current code of the projects is taken and started up as a new project under a new name. A recent example of forking was the problems with the Open Office software suit [38], the developers were not please with the Oracle management and forked the project to start a new one, Oracle has now stopped it's development of Open Office [40]. Often the threat of forking will work as a negation tool and force compromise and changes to the project.

2.3.4 OSS and business adoption

With the concept of OSS 2.0 many things have changed in the way that many large OSS projects are developed and maintained. More and more open source projects are being developed by companies that for different reasons base their platforms on open source. Good examples are Google Chrome, Android, Megoo and Mozilla Firefox. All these projects are funded and run by companies. They are open source in that the products source is available, but the development is in many cases much more resembling to ordinary company driven software development, something that is often criticized by free and open source ideologists that claim that this means the projects are not in fact open source, or at least not open [39].

More and more projects are now open source but not in the same way as before. Many commercial products have for example a dual licensing strategy where the free and open source version puts restrictions on how to monetize the use. Many open source products offer enterprise support and consultant services with their solutions. In general the trend has been more and more commercialization of open source products and open sourcing of commercial products. It is unclear as to what or how these projects fit in with relation the traditional 1.0 OSS projects but several projects gain much respect and traction in being open source. Arguments like transparency and being able to look under the hood are prominent when people want to speak highly of OSS programs and systems in comparison with closed ones. Even tough these projects are closed compared to their 1.0 counterparts when it comes to development they still offer positive aspects of OSS. Several projects are spinoffs of company driven OSS projects like the Camino browser, based on Firefox which is developed by the Mozilla Corporation [34].

Some companies are attempting not only to use and build OSS but also actively employ the communities that OSS attracts. In one case Nokia reported an interesting article on developing it's Maemo mobile OS by making it open source and creating a community that would support the development of the OS [23]. Their findings were that using the community was not straight forward and required resources to grow and attract developers. Also they found that when it came down to release schedules and bug fixing the community developers were more interested in creating new and exciting features so Nokia was often forced to do that part of the development themselves to meet deadlines and requirements. Perhaps this is why Android development is in large part a closed one where changes are let into the public open-source a while after big releases are finished, a more closed and insular development practice than many other OSS projects [51].

It is worth mentioning that a big problem when discussing OSS software, be it quality, methods or business use, is that there is no conformity or simple meaning to the term of OSS. For instance you would say that Android is OSS, but it is not open for changes and the development is not transparent and open. Android is OSS only in the sense that the end product is made available in source code form. Linux is open for changes and the development is largely transparent, but the actual code that is entered is closely scrutinized by people of high-rank within the project, gatekeepers of the code if you like. Maybe what is needed is a clearer terminology for OSS or a classification systems of types of OSS software that indicates how they are developed, used and purposed for. A community driven project like Mozilla Firefox still has a larger and formalized organization behind the major decisions whilst a company driven project like Android is not transparent or open in it's development at all.

Open source components are often touted to have benefits over closed source counterparts because of being able to view the code and change the code as needed. However very few developers actually do this for two simple reasons: 1 changing the code requires work and that will often defeat the purpose of using a component in the first place, 2 changing the code will create problems further along when updates for the component will not be compatible with the custom changes. Often when there are problems with small software components it is easier to solve the problem indirectly and not the component itself. However some developers do make changes to the OSS software they use and even commit these fixes or enhancements back to the projects, so even tough this benefit is perhaps overstated it is none the less real.

2.3.5 Licenses

One aspect of OSS that is hard to cover thoroughly are the different kinds of licenses available to publishers of OSS. For simplicity sake there are two main kinds of open source licenses that are classified as copyleft and permissive. Copyleft licenses, the most popular example being the GNU General Public License (GPL) [11], requires that modifications of the software will be published with the same license. The goal is simply to require the developer or modifier to share the work back to the community. Some versions of copyleft licenses actually act viral as they demand that the whole system be open-source and same-licensed as the OSS software itself. This has caused some sensational headlines as it has been discovered that commercial software or products have used copyleft licensed libraries, which in turn required the whole project to be licensed under that same license [59]. These kinds of licenses are usually avoided by several enterprise developers because it is either impossible (the systems use proprietary closed-source code) or it is not business wise to make public the whole solution as open source. The permissive kind of license like the Apache 2.0 license [10], give the user of the OSS component freedom to implement it in closed-source systems and even to release it under a proprietary software license. The permissive licenses are generally much more business friendly in that they do not require modified versions of the software to be made public. For some companies the choice of OSS is a choice of what kind of license it uses, for others the license is irrelevant for various reasons like that it will never be public or will not be sold in a particular way.

2.4 Knowledge Management

The problem of selecting third-party components will often boil down to a problem of knowledge or lack thereof. When you need a component and you have knowledge and experience with one or more components you will quickly be able to choose and use a component based on you'r own knowledge and experience. However if you do not posses any knowledge on either the components available, the problem/technology in question then you will have to spend time finding, learning and evaluating components that you do not have prior knowledge or experience of, thus making the chances of making a poor decision much higher and the process will probably take more time and effort.

The lack of knowledge however can be filled by the knowledge from an external source, like from a coworker, a blog, a forum or a wiki. These different sources of knowledge and information will vary in terms of liability, trustworthiness and usefulness. For example if you ask a colleague from the workplace that has 10+ years of experience with software development you will be more inclined to trust that person than a random forum answer by some anonymous user.

These different sources of information of knowledge clearly has different benefits and disadvantages in terms of:

- Usefulness - how useful is the information in relation to the problem at hand?
- Trustworthiness - how reliable is the information?
- Liability - who is saying what - what are the interests behind the information source?
- Accessibility - how much effort is required to attain the information?
- Resource use - how taxing is the information retrieval in terms of hours spent or other persons hours spent
- Social Threshold - how hard is to ask for help about the problem?

These and many more are challenges of knowledge management and transfer that have been studied for centuries if not since the beginning of knowledge transfer itself in the early days of storytelling and learning to hunt. For corporations the challenge is to develop and spread competencies, knowledge and experience as much as possible about inside the corporation itself, a challenge that is daunting when you are more than a hundred people and spread across several countries. Several attempts have been done at utilizing modern information tools like content management systems, wikis, relational databases and other information systems to maintain and effectively organize this valuable asset, but failure to achieve these goals are frequent if not the norm. Walsham puts it quite clearly:

> "Information and communication technologies are not the answer to improved knowledge- sharing within and between people and organisations." [55]

The problem with trying to solve the problem of knowledge and competencies spreading simply by implementing IT information solutions is that knowledge transfer is not simply to write and read text. In "Knowledge Management: The Benefits and limitations of computer systems" [55] Walsham presents several concepts to further define what the term knowledge means in different contexts. *Explicit* knowledge is found in databases, email and books whilst *tacit* knowledge is a person's experience and culture, the tacit knowledge is most valuable and most difficult to spread trough pure text and data. Walsham goes on to talk about a concept that he calls "communities of practice" that are groups of individuals that share interests and cultural language that makes it easier for knowledge to be transferred between individuals.

Walsham [55] explores why technical solutions to knowledge management has failed to achieve larger success within companies and that the reasons for these lies close to that of tacit and non-tacit knowledge. Simply reading what someone has written in a CMS or wiki solution. There needs to be certain levels of hands-on from people for the knowledge to transfer successfully. Other problems is the incentive problem, why should a person contribute to an internal knowledge database when that same competency is the one that sets him or her apart from other workers? When a company has model for individual rewards and workers report billings pr. project, there may be little reason to voluntarily spend time on such superfluous solutions.

However there are reports of successful knowledge management project that incorporate some levels of technological solutions [8]. The article mentions eight points that are crucial for knowledge management systems to be successful:

- Link to economic performance or industry value
- Technical and organizational infrastructure
- Standard, flexible knowledge structure
- Knowledge-friendly culture
- Clear purpose and language
- Change in motivational practices
- Multiple channels for knowledge transfer
- Senior management support

Not going trough all the points I will mention that I find especially knowledge-friendly culture, motivational practices and multiple channels interesting as they point to problems that need to be addressed for the business operation as a whole, indeed the article promotes this several times. To simply "set up a knowledge database" is not enough and knowledge management needs to be addressed from a high level and throughout the company as a large scale effort. It is also mentions that:

> "At one large computer company, a series of ongoing efforts encouraged the reuse of a particular kind of knowledge: component designs" [8]

This company clearly sees component reuse as a knowledge management effort, a very interesting point of view. There is are many pitfalls when it comes to knowledge management and it may seem that capturing the competencies of individual workers at worst may be trying to capturing "lightning in a bottle". However there are ways of improving knowledge management and signs point at technology being able to aid in this, but it is not the solution by itself.

2.4.1 Crowd-sourcing and the Internet

One modern innovation is the Internet. The Internet is the world connected trough it's computers and is easiest to define as computer information spread from everyone to everyone. The term crowd sourcing exposes one very interesting aspect of the Internet which is the sheer volume of users it possesses and what can be done by a larger collective. In the article “The Rise of Crowdsourcing” [21] the author presents several business solutions that are working by simply putting the task out on the internet and then receives solutions and work from the masses to be picked by the company or person that needed the help. Think of it as asking a question to a large crowd, the bigger the crowd the bigger the probability that someone in the crowd knows the problem and has a solution. With an estimated population of 1.9 billion people [15] that crowd is starting to get very big. although not attributed as crowd sourcing, Wikipedia is another good example of how the Internet can solve and create big solutions as long as the numbers are high enough, to date Wikipedia English has 3.6 million articles [58] and those articles are contributed by it’s own users. Open and large forums can also bee seen as a sort of crowd sourcing and sites like Stack Overflow [48] gives software developers a large user-base to ask professional questions to, also the answers of such sites are again indexed by search engines like Google. This makes it probable that if you have a question to ask, someone has already asked the question and gotten an answer and a Google search may reveal it.

The growth of the Internet is making it a place of a modern problem that it helped create, the problem of information overload. There is simply to much information available and a simple Google search reveals that a basic term like “java programming language” returns 1.6 million hits as of april 2011. As such the value of the Internet probably having good answers is countered with the problem of finding the good data that is out there. Even tough tools and search engines become better there is an increasing need for having the knowledge and skill of how to find the correct information, much like one must learn how a library works in order to find the right book and it’s location.

2.4.2 Software Craftsmanship

In 1999 a book was released called “The Pragmatic Programmer, From Journeyman to Master” [22] which talks about the software developer and how he/she can improve as one. This book can be seen as the start of the ideas and ideals of software craftsmanship movement. It discusses the process of developing systems from start to end and goes into detail on the actual workflow of writing code and solving problems with programming. A excerpt from the book illustrates the mindset behind the book, this is from “The Evils of

Duplication":

> "We feel that the only way to develop software reliably, and to make our developments easier to understand and maintain, is to follow what we call the DRY principle: Every piece of knowledge must have a single, unambiguous, authoritative representation within a system. Why do we call it DRY?"
>
> Tip 11 DRY—Don't Repeat Yourself [22]

The book focuses on creating code that is efficient, smart and easy to maintain. It promotes the idea of being proud of the work. Other books expand on the ideas that became open with "The Pragmatic Programmer" like "Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman" [19] and "Clean Code - A Handbook of Agile Software Craftmanship" [30]. They all promote a proud-ship of software development, blending it with terminology from carpentry, painters and even eastern fighting. A software developer starts out as an apprentice and goes into teaching with a master, developers group in coding dojos and perform code katas [52] to improve or hone their skills.

Software craftsmanship is not a knowledge management tool itself, but it does promote that software developers take pride in their work, share their knowledge and skill to others and promotes a community of practice. Therefore it is a movement that promotes knowledge transfer and acquirement.

3 Research Approach

3.1 Selection of method and techniques

There are several methods for research that differ in the way they operate and for what kind of research they are most purposeful for. A small overview of the two main research methods can be found in table 5 with some techniques, positive and negative aspects.

Name	**Techniques**	**Positive**	**Negative**
Qualitative	interviews, observation, participation	Answers why, small data samples	Results not replicable, subjective in nature
Quantitative	statistics, surveys	Statistical probabilities in results, replicable methods	Surveys have pre-determined questions and answers

Table 5: Research methods

As the table shows there are positive and negative aspects for either method. Quantitative research and analysis for instance often uses surveys. Surveys are great for gathering large amounts of data and asking simple questions which then can be aggregated into statistics and give results and certainty in percentages. A great example of a large survey was the 2010 US Census which was aimed at gathering information on how many people are living in the united states in 2010 [53]. Questions are simple, often yes/no or with multiple choice as options for answers. For the purpose of gathering information on the number of people living in the U.S.A in 2010 the use of surveys and quantitative research methods is both sufficient and accurate. However such surveys have downsides as well, the questions are pre-written and the scope is often narrow. When the questions asked are pre-determined you will seldom get answers to the questions that are not being asked, such un-asked questions are often the most important ones for many researchers. Some questions may even be written in a leading way, knowingly or unknowingly, and this may affect the results. Quantitate research is a rigid and static form which often has a narrow scope when it comes to problems and issues where it is important to know within mathematical certainty the degree of a certain phenomenon or problem.

Qualitative research is a method that aims to answer not only what is happening or with what frequency, but rather why and how things are happening. To achieve this qualitative methods are much less rigid and not wholly dependent on large amounts of data collection, focusing instead on interpreting gathered results and presenting theories as possible explanations. Being flexible means the ability to start out with one problem or question and then going where the research goes, being open to new information that may change the whole perspective and focus of the research. A good example is the method the qualitative research interview [35]. This is an interview that is open ended, allowing the subjects to digress and expand on the initial questions asked, allowing for a more natural flow of conversation and the goal being to discover new questions and aspects on the research topic. The disadvantages of qualitative research is that the focus is very narrow, making it probable that other data would present itself if a larger sample size was utilized and that observations and interviews of this nature is impossible to re-create and re-test, an important factor for many sciences. Also the interpretations of the data collected may not be correct and even strongly biased in the subjective eyes of the researcher.

The problem for this thesis falls under that of systems engineering and specifically on the less technical aspects of software development. Understanding that software development is a team effort done by people in groups adds a social and human component to the problem and research on the subject will therefore lend from more social research branches than that of more traditional computer science. It may be as much about the tools that developers use, as the way that the individual developers are interacting with the customer, the boss or with each other.

It is important when choosing a research method to select one that is suited for the kind of research to be done. In the case of this master thesis the choice was qualitative research. Also there is an element of quantitative research with one survey performed. With this kind of research it is given that the results are not replicable and highly based on cognitive and subjective factors. Empirical or quantitative research was not deemed possible for this thesis as it is not possible to replicate any given conditions. It is also important to define what kind of standing the qualitative research will be based on as is said by Chua [4] that qualitative research can be done with either a positivist, interpretative or critical stance. Klein and Myers [24] defines the interpretative stance as

> “IS research can be classified as interpretive if it is assumed that

our knowledge of reality is gained only through social constructions such a language, consciousness, shared meanings, documents, tools, and other artifacts.” [24]

This is the stance taken by me in this thesis and as such it falls under qualitative research with an interpretative stance.

3.2 Access

The work on this thesis started in the late fall of 2009 when I first contacted Acando to discuss a potential master thesis that I could work with them on. From the fall of 2009 to spring 2010 a problem and assignment was worked out with the collaboration between Acando, myself and the supervisor for this thesis.

When discussing with Acando potential master thesis problems it was always a factor to have the work be of interest and potential benefit of the company. Starting on a theme of use of open source software the problem developed to that of using software components in systems development at Acando. This was a topic of interest from the work group leaders at Acando Trondheim and the thesis would result in two parts, one master thesis and one report or guide to improving software reuse at Acando.

During the fall of 2009 I attended a business presentation of Acando at Acando’s offices in Trondheim with the student organization for Computer Science at NTNU. During this presentation there was presented a list for interested parties to put up their name in regards for either a job or master thesis. I signed and a couple of weeks later got a mail from Acando asking me what kind of problem I would be interested in writing with the company. After a couple of mails back and forth a meeting was arranged with myself and the two current group leaders at Acando. The initial mail from Acando was sent on 21.10.2009.

My initial problem was centered around open source software phenomenon like distributed development and comparing the development of OSS compared to more traditional software development methods in an enterprise setting. When approaching Acando I wanted to work out a problem that would be interesting for them as well so I was open to changing the thesis problem. As such the initial meetings became a sort of negotiation of different topics and how they would fit a master thesis relevant for Acando. From the initial topic of looking at OSS methodology emerged the topic of software reuse itself and how this was an often used technique but not one that had been closer looked at internally. It is here worth noting that the initial thesis problem was more or less put aside as a sacrifice to gain access to the resource that would be Acando and it's developers. This tradeoff poses a potential problem in a research context, is the research biased towards finding something that is of commercial value to Acando? This issue is discussed later in this chapter.

After a meeting with Acando I went back to my supervisor and discussed the options of how to find a problem that would be interesting to Acando. A meeting with myself, the group leaders at Acando and supervisor was arranged where the more or less final thesis problem was worked out. After this meeting a written formulation of the problem was made and sent to Acando for internal approval, this also worked as an agreement between I and Acando, this formulation was sent to Acando on 18.01.2010. The written formulation is found in the appendix, the names of Acando employees have been censored.

With the problem properly formulated I held an introduction presentation at Acando on one of their meeting days where all the developers are gathered at the Acando offices. This presentation explained who I was and what I would do during the next year at Acando which was to observe and talk to developers at Acando about the topic of the thesis. This presentation was held 06.05.2010.

On 03.09.2010 I met with Acando and signed a confidentiality agreement to keep secret any sensitive information I would acquire during my observations and interviews, this formally also started my observations at Acando as I now could visit their offices during their working hours and also talk to the developers at Acando freely.

My access to Acando has been limited to observing and interviewing the employes at Acando. I have attended internal meetings only and no meetings which involved clients or customers.

3.3 Data collection

The focus for the collection of data or information from Acando has been that of the personal developer experiences at Acando. Starting off with smalltalk in the halls and by the coffee machine to more formal interviews. Observation of work culture from meetings and also just by being in the Acando offices have also provided insight into how Acando works and functions.

By observing and interviewing systems developers at Acando I wanted to know how frequently developers were using third party software components and most importantly how they were finding and selecting them. Also key was to try to discover any latent problems with using third party components, either in general or in specific cases of poorly selected or bad behaving components. The primary goal of the study was to try to assess the status of third party component use at Acando and to se if the situation could be improved.

The basis for the results and discussions in chapters Case and Discussions are drawn from data collected from Acando and Acando employes in Trondheim in the period of fall 2010 to spring 2011. During this time I had access to Acando offices in Trondheim and systems developers there. Acando is an IT consultant firm working with advice, system solutions, developers to lend and running and maintenance of systems. Here follows a complete overview of methods used with numbers were applicable. Also included is an overview of the different phases of data collection with a chronological timeline presentation.

3.3.1 Collection phases

During the work on the thesis different phases of collection emerged. Data needs would change as data came in, this is an overview of the the different stages of data collection and their primary goals and rough descriptions of them.

Phase 1 - Is there a problem?
The initial phase was to ascertain if there existed a certain amount of small software component reuse at Acando. If this was not the case then the whole problem would be more or less mute. This was a critical question to answer and was the first data collection to be done. A survey was sent out in september and was the basis for this phase.

Phase 2 - Basic information
Having established that there indeed was a certain amount of software reuse at Acando the next collection phase was focused on gathering basic information on component use from the developers. Who were using and selecting,

what were they selecting and how were they finding the components fall into this category. By initial observations and informal talks with developers more formal and in-depth interviews were performed with developers.

Phase 3 - Unanswered questions
After gathering basic information several unanswered questions arose when looking over the results and discussing them with my supervisor. These questions became the starting points for second and third round interviews were some new and some previously interviewed developers were targeted.

3.3.2 Collection methods

Questionnaire
One questionnaire was sent out during the fall of 2010 to the systems developers at Acando in Trondheim and Oslo. Approximately one hundred developers were sent an email to this online questionnaire, made using Google Docs Forms, and approximately 50% of the recipients answered. The purpose of this questionnaire was to quickly assess the general amount of use of third party software components at Acando. It was important to establish this early, should the response be low it could be assumed that components were not being used at Acando at a significant volume and the problem for the thesis would be void to ask in the first place. It is interesting to note that such a questionnaire falls under quantitative research methods as it asks a preset number of statically formulated questions and achieves quantifiable numbers to present statistics and probabilities for the research. The questions were focused on basics and was aimed at answering if there was significant use and if yes what methods were being used to find and select the components. An example from the questionnaire (translated from Norwegian):

> “Have you been in a situation for a project/customer where you considered different software components?”

The total amount of questions was eight where seven were asked to those that had chosen a component and 3 were asked to those who had not chosen one. It was sent out on a wednesday on 30.09.2010 and the deadline to answer was friday 01.10.2010. Se appendix for the full questionnaire and results.

Observations
From September 2010 to March 2011 I visited Acando 29 times. Some days I would be there from 09:00 to 16:00, other visits were shorter or later in

the day, workshops for instance were from 16:00 to 20:00. There I worked, observed and talked to developers at Acando and I also ate lunch, provided as a courtesy from Acando to me. I performed different kinds of observation during my time at Acando. I performed participant observation when I presented my problem during Acando meeting days and also I joined several workshops that Acando has in part with its knowledge transfer strategy. For the presentations the room would be filled with about thirty to forty people, mostly IT personnel meaning different kinds of developers, advisors, systems architects or team leaders. I would sit trough most of the presentations, be they about economic results or information on Acando social activities and wait for my turn to do my presentation. The workshops would consist of a smaller group of mostly developers, perhaps 10-15 people and start after work hours at about 16.00. They would be topical and have presentations on different kinds of technology that one or two developers at Acando would present. During these presentations questions could be asked from the audience and I, as part of the audience, asked questions at several occasions.

I performed non-participant observation whilst simply spending time in the Acando offices and I observed a meeting of a smaller Acando workgroup. The meeting consisted of one group leader and four developers that held a “sprint” meeting which discussed and summarized the previous sprint and planned the next one. Other observations were made during my time spent at Acando. I would sit down at an empty work desk, Acando has several temporary work desks at their office, and simply work there with my laptop and notes. I would observe people working on desks adjacent to mine or wander around the office floor. In table 6 there is an overview with approximate numbers of frequency.

Type	**Number**
Observations	29
Smalltalk	24
Workshops	4
Presentations	2
Meetings	3

Table 6: Observations

My observations were recorded in different mediums. Mostly I would note whatever I observed with what recording option I had handy. For instance if someone said something I found interesting when I was by the coffee machine and without any notepad I would perhaps go back to my laptop and note it

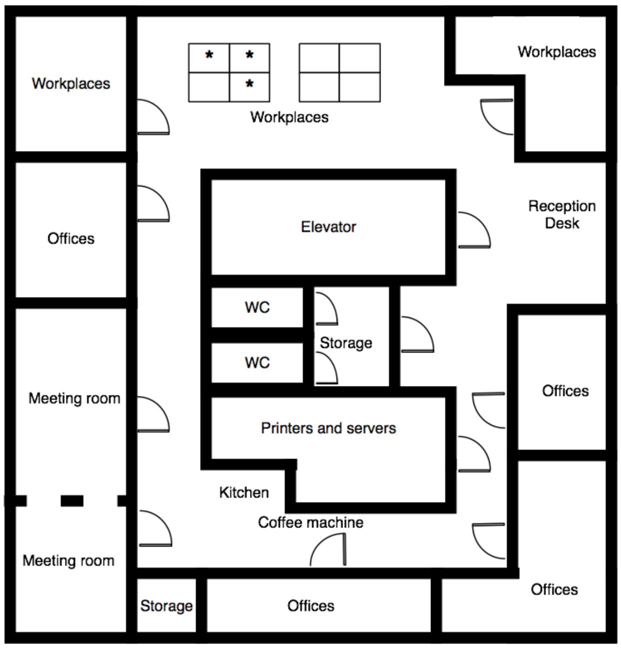


Figure 3: Acando floor-plan

in a digital document. Such notes may therefore note be 100% accurate and it is also worth noting that Acando Trondheim is a Norwegian company that speaks Norwegian, quotes and other related materials have been translated for the purpose of this thesis being written in english. I would take notes with the following media:

Format	**Number of documents**
Google Docs Documents	40
Google Docs Spreadsheets	4
Google Docs Presentations	5
Small notebooks	4
Large notebooks	4

Figure 3 shows the floor-plan of the offices at Acando. The stars are the workplaces which I used during my visits. Workplaces are temporary workplaces for consultants to use, offices are for employes working mostly at Acando. The meeting space is divided by a light-wall that can be slid up to create a larger meeting area. This floor-plan is a sketch and not a precisely drawn or scaled blueprint.

This data collection would quickly become cumbersome and chaotic, several notebooks with almost random notes in them piling up, so I would periodically create summary documents that formulated overall points and key observations, they could perhaps be categories like "Selected quotes" that I found interesting or "notes on selection methods" to make them more easy to organize and look up for further additions should more data be relevant to add later on.

During my time at Acando I had several informal chats about my thesis with several developers and non-developers. This data is more or less unstructured and ended up in notes. This also includes two presentations on the subjects that was held for Acando as a presentation of the work as I began the thesis and neared it's completion. Typical example of these informal conversations would be meeting someone by the coffee machine, a good place to meet developers, and then talk with that person about who I was and what my thesis work was about. In table 9 there is a rough overview of people that are quoted in the thesis that originate from informal talks.

Interviews
From loose talks I would ask developers for more formal interviews. In total I performed ten interviews with seven different developers at Acando. Eight of these interviews were recorded for analysis purposes. The recorded interviews followed the ideas of the semi-structured interview which is a commonly used tool used in qualitative research. Before the interview I had noted some basic questions and several open-ended questions that meant to make the interview subject think and explore the problem. The purpose of this kind of interview is to make it possible to uncover unforeseen aspects or problems that are related to the research goals. An overview over interview types and numbers is found in table 7. During the interviews I would take down notes on interesting topics and points that I deemed important, such notes will not reflect the entire interview precisely but rather catch the pieces and topics that I felt was important there and then. To try to achieve more complete analysis of the interview three interviews were transcribed. This was done to thoroughly go trough them and try to discover points that may have gone forgotten from the rough notes that were taking during the interviews themselves.

During the thesis work I performed three phases of interviews with different goals. The first round of interviews was aimed at getting down the important basics of the problem. How were components being selected, what information channels were being used. An open-ended question example from the first interview template:

> "What experiences have you with using small software components? (problems, good experiences, challenges etc.)"

Interviews would last on an average of thirty minutes and were recorded with my laptop. The subject would be asked if recording was permitted and it was explained that the interview was anonymous as the recording would only be used for analysis reasons. The locations of the interviews would vary from the offices at Acando to other office locations where the given developer would be stationed, it was a point for me to make the interview fit into the developers schedule, this is an important as pointed out by Oates [37].

After some of these first round interviews questions arose that were not being answered (or asked) and a second round of interviews began. For these interviews developers that I had already interviewed were asked again so that I could skip the basic questions and target the problem that stood unanswered. This interviews were also recorded and lasted on average about 30 minutes, the same as the first round. These interviews were more focused than the initial ones but still followed an open-minded style of questions. An example from the second round of interviews:

> "Could you describe a timeline from the moment you recognize that a problem can be solved by using a software component to having selected one and started on implementing it."

After looking at the results from the second round of interviews more unanswered questions arose and a third and final round was initiated. For these interviews both perviously interviewed and new subjects were targeted. When interviewing a new person a condensed version of the initial interview questions were asked in addition to the ones created from the last round.

In table 8 an overview of all bigger interviews and subjects is presented, subjects are anonymous and coded into roles within Acando.

Official data
I would gather information about Acando trough official channels such as their website, information flyers, economic reports and presentations done by Acando on different occasions. This data was mainly to gather facts about Acando and their business. An overview of such data sources:

Type	Number
Recorded and planned interviews	8
Not recorded interviews	2

Table 7: Interviews

Code	Phases	Role
P1	1,2,3	Developer OSS
P2	1,2	Developer MS
P3	1	Leader MS
P4	1	Architect OSS
P5	1	Developer MS
P6	1	Developer OSS
P7	3	Developer MS

Table 8: Interview subjects overview

Code	Role
P8	Leader
P9	Developer OSS
P10	Developer OSS
P11	Developer MS

Table 9: Smalltalk subjects overview

- Acando interim report january june 2010 - 17 pages
- Friprog tv interview - 9 minute interview of two developers [12]
- Acando helse norge - press announcement 1 page
- Acando på ett minutt - presentation pamflett - 4 pages
- Fri prog i helse sektro - presentation 14 pages

3.4 Writing Process

The process of analyzing the data was a continuing process throughout the data collection and writing of the thesis itself. After one round of interviews

I would try to gather together the overall themes and answers. This made it possible to compare answers and themes with articles and also review them logically. These findings where presented to the thesis advisor and discussed. From these discussions emerged further questions and problems that would spark new rounds of interviews with a new agenda. This process of data collection, analysis and further data collection was continuous for the whole thesis work period. As such the thesis problem started with looking at potential problems regarding small components and ended up looking at why the frequency of use is so relatively low. This was a progression of the work and is reflected in the focus shift from the findings chapter to the discussions chapter.

3.5 Reflection on research method

Klein and Myers provide a set of seven principles for evaluating interpretive field studies [24], these principles are very useful as tools to guide and review research work. They cover several critique angles, here follows these principles and a review of this thesis as to how it stands up against them.

1. The Fundamental Principle of the Hermeneutic Circle
This is the problem of considering something by its smallest components, understanding those parts and then to also understand their roles and positions in the greater whole. For this thesis the problem lies in looking at individual developers on individual software development projects and correlating that to the larger perspective which is Acando as a consultant firm that delivers several projects with several developers. For example if one developer uses lots of components and never has any problems with them, this does not necessarily hold true for all developers in all projects. It is important to understand how one developer fits into the bigger picture and if Acando as a firm holds any larger standard to all it's employes. Hopefully by interviewing several developers I would be able to uncover a trend or tendency when it came to third party component use.

For this thesis there are several underlying topics and themes that have been examined, some closer than others, in an effort to understand the smaller components of the puzzle in order to fully understand the issues. However it is always possible that certain elements have been overlooked or taken for granted in their understanding, used maybe as basic truths and platforms for the thesis to build upon. Having accepted other articles as truths presents this as a problem, if one assertion of a fundamental principle that the thesis relies on is false then does this thesis fall with it? Sadly there is a time and resource limit to this and any academic work, however it is also in the tradition that others may criticize and point out faults with the premises and assumptions.

As for the pieces of problems that have been studied and looked for and during this thesis it has always been a goal to divide and conquer the different topics. Breaking down software component selection into topics of OSS, COTS, software reuse and knowledge management is evidence of this. When interviewing developers it has been a point to not target or follow one single idea from the start, trying rather to identify the different issues and how they stand to create the bigger picture.

2. The Principle of Contextualization
This according to Klein and Myers:

> “[The Principle of Contextualization] Requires critical reflection of the social and historical background of the research setting, so that the intended audience can see how the current situation under investigation emerged.” [24]

For this thesis the reflection comes from the background literature and study. What is the status of research on open source software, software reuse, commercial off the shelf and knowledge management? How does Acando fit in with this research and general status quo of systems development and IS research. This groundwork is important as it a) permits the data from this thesis to be compared to others and b) lets the reader of the thesis quickly assess the state of the different related topics and research areas. It has been interesting to se where Acando fits into the research and where it does not and makes it possible to spot emerging trends and changes that are interesting and relevant for the research question.

> “Interpretivists argue that organizations are not static and that the relationships between people, organizations, and technology are not fixed but constantly changing. As a consequence, interpretive research seeks to understand a moving target” [24]

As such it is important for the work to present the reader with enough information as to form a clear picture of all relevant research views and other contextual views. However there is also here a problem of scope, time and the degree of exhausting all information available. With the Internet it is not only possible but way to easy to burry one self in never ending studies, papers and relevant information. As such the process of forming the academic context is one that has been continuous throughout the work on this thesis and it will be impossible for me to claim that all available sources have been reviewed and looked at. As for the firm that has been the basis of the study it is presented with both facts and observations of work culture and context. However any company is a complex organism and the viewpoint has always been from an insider with one foot inside the door. It can be argued that my access was not free enough, not being able to sift trough the internal IT systems or sit in with customer meetings for instance, this hinderance is however clearly stated and accepted for the results and scope of this thesis.

3. The Principle of Interaction Between the Researchers and the Subjects
This principle regards being critical as to how research data has been acquired and constructed trough social interaction between me as a researcher and the employes at Acando. This is a big concern for me for several reasons. For one I'm studying and interviewing software developers and I myself am a software developer. It has been potentially easy for me to identify with my interview subjects and affirmatively agree with them during the interviews and perhaps not ask the truly hard questions. When for example listening to my own interviews I notice that the conversational flow becomes quick and goes over in "developer" speak, I am perhaps to culturally similar to my subjects. One example would be when a developer explains how he/she find a component, here is an excerpt of the dialog:

> P2 : "Then you do some searches and find the best looking candidate. The site looks good or the documentation is promising."
>
> Me : "Yes, mhmm, ok I understand"

This because the task of searching for a component and finding a promising candidate is something I myself have done and I familiarize all to well with the developer, I should in this case instead have asked "What do you mean best looking candidate? What is a good looking site and what do you mean by promising documentation?". It is very possible that I never asked the hard questions and let the developers of the hook if you like because of my own experiences. However it may also benefit my work as I perhaps already have a slight insight into how developers think and work, making it easier to ask the right questions.

Another big concern is how I am perceived by those that I talk to, I have been accepted for this thesis work by the team leaders at Acando and I therefore have a possible taint of being from the "management" when approaching people. During my interviews I introduce myself as a student working on a master thesis for Acando, making it possible that it is understood that I'm looking for potential problems and faults in the development process of which the developers themselves are easy targets to be blamed for any shortcomings. My impressions from the people I have interviewed and talked to has never been that this was the case, the work culture in Acando being open to criticism and analyzing, without the common worker to be blamed. It is however possible that I have simply not been able to notice this and that answers have been filtered trough a "this goes back to the top" mentality without me being aware of it.

Properly evaluating this principle is hard because it is not easy to say if the social interaction and social constructive way of collecting my data was strongly affected by assumptions, culture, the perception as being sent from management or my own experiences as a software developer. However having that mentality has made it possible for me to look at my findings critically and carefully try to review any prejudice or assumptions that may have been made. Having the materials reviewed by a third party, my thesis advisor, also helps. In the end when reviewing my findings and data trough the principle of interaction I would say that there is a possibility that this has influenced my data and results in a negative fashion, but that I don't think that is the case because being familiar with the culture made it easier, not harder for me to ask the right questions. Having a basic understanding of what a software component is and how it operates made it easier for me to ask straight out "was there any problems?" knowing that this could be the case.

4. The Principle of Abstraction and Generalization
From Klein and Myers [24] :

> "[The Principle of Abstraction and Generalization] Requires relating the idiographic details revealed by the data interpretation through the application of principles one and two to theoretical, general concepts that describe the nature of human understanding and social action"

This thesis is a study of one specific company and has viewed this company trough specific people and moments in time, as such there is an understanding that the findings and observations may be in fact unique for this case and not applicable to other research or other companies. However it has been a point to always try to generalize the findings and find the overall themes and trends that the specific data has revealed. Relating the findings with with existing theory and looking and the bigger picture rather than the individual pieces is an attempt at putting these findings in a broader context. As such there is a goal of looking at something for the value of not just one business but for other research and even other companies. In the conclusions there are discussions of interesting general ideas that this thesis has looked at.

5. The Principle of Diological Reasoning
From Klein and Myers [24] :

> “[The Principle of Diological Reasoning] Requires sensitivity to possible contradictions between the theoretical preconceptions guiding the research design and actual findings (”the story which the data tell“) with subsequent cycles of revision.”

In the discussion part of the thesis the findings from the study are compared to the literature presented in the background chapter. Such differences are discussed and examined as they are both interesting and problematic. Why does this data contradict existing claims? Is the case unique and not related to other findings, are previous findings wrong or has something happened since the literature was written? The literature section has subsequently also been rewritten and changed when new topics and questions would arise that touched on topics that had not been looked at in the literature. For instance the section on software craftsmanship was added late in the process as this topic was brought in at a very late stage. This cycle of looking at the findings and the need to correlating and comparing them with the existing literature has been ongoing and important for the whole process of writing this thesis. There is the problem of time and resources and again it is possible that papers and articles have been overlooked, especially for topics and themes that were discovered late in the process.

6. The Principle of Multiple Interpretations
The social context that may have influenced this thesis is how the thesis itself came to be and was formulated. It was a problem created to cater to the company that was to be studied and gained access to. As such there is a viewpoint that this thesis is to be of benefit to the company and also a problem of not being critical enough at the same time. Looking into a business and finding big problems may not be a popular action and as such there may have been to little critique as to how things operate. No radically

different viewpoint has been presented and how consultancy companies work is in a way taken for granted as the only available view. One could argue that this is not something this thesis has had the option to do, resulting perhaps in the termination of the thesis itself and the cooperation with the company in question, however it stands as a noteworthy critique of this thesis.

It is worth mentioning that aspects and lines of thought that could be considered problematic and as critiques as to how the consultant firm operates have been explored and followed. Having a thesis advisor to review the findings have helped in adding a viewpoint that is in a sense protected from having to be friendly and compliant with the company.

7. The Principle of Suspicion
This principle aims to revel the effects of socially created distortions and psychopathological delusions [24]. Being critical only to the findings and how they are interpreted will not cover the problem of basic assumptions and fundamental issues being misconceived.

For this thesis the principle of suspicion may fall on the fact that developers are claiming to successfully use components without bigger problems. There are hints towards this not being true with small side comments like:

> "Smaller issues are fixed around the component itself, we make it fit" - P1

This is played as a small issue from the developers but may in fact take more time than they realize and maybe even mute the point of using components in the first place. However the overall impression is that the developer are positive towards component use and there is little reason for them to express this if it is not the case. Trying to catch such false assumptions, from either the subjects or the researcher, is however a difficult task and there may exists flaws within any thesis work. For this thesis the tool for trying to catch and uncover such delusions have been transcribing and re-reviewing interviews, making it possible to re-listen to the conversations and critically reviewing what is being said and understood explicitly and what is being dismissed or accepted too easily.

4 Case

4.1 Introduction

This thesis is based on working with the software consultant firm Acando and specifically the branch in Trondheim Norway. Starting with talks in the fall of 2009 the thesis and problem evolved trough meetings with Acando, myself and the supervisor for this thesis. This section will cover the background of the company Acando and then present the results of enquiry and studies that were performed there from the fall 2010 to spring 2011.

4.2 Background of Acando

4.2.1 Acando - Facts and numbers

Acando is a software consultant firm located in Norway, Sweden, Finland, England and Denmark. The whole company has about 1100 employes. Acando is a consultant firm that perform counseling, development, implementation and operation of information technology for it's clients. Acando presents their core values as: team spirit, passion and results. Their most important business partners are Microsoft and SAP.

Acando Norway consist of 100 consultants and are situated in offices in Oslo and Trondheim. The Norwegian branch was established as Consult IT in 1997, merged with eScienza in 2004 and Addcom Innovation in 2005 and became Abeo AS. The modern day Acando is the result of Acando purchasing Abeo AS in 2007. The Norwegian branch works with the development of processes and organization by leveraging IT combined with business processes. Their areas of deliveries include:

- Architecture
- IT-Solutions
- Business Intelligence
- Information Management
- Applications Management

The latest entry to business at Acando is the added service of maintenance and running of services. This as opposed to handing over the solutions that Acando creates to third parties or the clients themselves. Acando has a strong position with public services like public health and central administration services. In the private sector Acando works with telecommunications, energy companies, bank and finance.

Acando Norway has a workforce that consists of over one hundred consultants in Trondheim and Oslo. Of these there are 95% with IT as their education major. The majority are between 30 and 45 in age and have over ten years working experience with IT. This masters thesis is written in co-operation with Acando in Trondheim. The developers at Acando Trondheim are grouped in two groups where one group is focused on working with Microsoft technologies like .Net, SharePoint, Microsoft SQL and other Microsoft related technologies. This group works mainly with customers that either operate on Microsoft technology today or they come to Acando with certain requirements to use of Microsoft related technologies or products. The other developer group works with technologies that don't relate to Microsoft technology. A often used name for this group is "Open Source" even tough they have no restrictions or qualms with working with closed sourced or proprietary software. There is no strong cultural divide between these groups and it is mainly a organizational divide.

Most recently Acando developed a project for Helse Sør-Øst (south-east public healthcare in Norway) called "PRO". The development was here influenced by a desire from the Norwegian government o increasingly use open source in government related software development. The project was a short and quick project and agile scrum development was chosen as the method. Acando presented a solution which put together several different software components, some COTS and some OSS, in a "best of breed" practice that didn't discriminate on if the technology was open source or not. The project has become a showcase for Acando in proving that Open Source and software reuse is a viable strategy that can give good results in a short and cost-effective manner.

4.2.2 Acando work structure and ideals

As mentioned earlier the two groups of development have divided between them Microsoft-related technologies and all that is not directly Microsoft related. Also the company is showing a growing interest in open source technologies, they are looking more and more at solutions based on open source where traditionally there has been only commercial enterprise software. It is important to remember however that much of this open source software is made for commercial settings and often come with enterprise levels of developer and customer support plans, making them less different from their commercial counterparts than one might initially think.

In a interview with Friprogsenteret, a state funded center for use of free software in the public sector, a software architect at Acando talks about how Acando is approaching open source technologies and component reuse when assembling solutions for customers. He mentions that there is a growing interest in use of open source, especially from the public sector, but there is still a lot of insecurity and little knowledge, especially from the customers, about open source software. He and another architect that is also interviewed in the video talk about how there needs to be developed a framework for working with open source in a more standardized way. When it comes to the open source software itself, he believes that it is the same as any software in that it varies in quality and robustness from project to project, but that with the community and open source aspect there comes a lot more information to access and use during both consideration of components and support during development. He puts it as simply that if the software meets the requirements of the customer, then that is the important part, not if it's open source or not.

It is here interesting to mention the correlation between these statements and business endeavors and the article by Fitzgerald on OSS 2.0 [9]. This fits in with his findings on new business models using and leveraging open source software by simply using it to create complete IT solutions from them.

4.3 Culture

My visits, interviews and general studies of Acando gave me insights into the work structure and different work roles at the company. The work culture in Acando is one that in my opinion reflects what is a typical "norwegian" workplace. With this I mean that from initial visits to Acando it was hard to spot the CEO from the rest of the consultant workforce. Sure enough he was the one that sat in his own office (almost his own anyway), but still the culture of Acando is that of equals. It is an informal workplace yet also a decentralized workforce that does not necessarily meet at a weekly basis. One employe had been at a project for five years and sometimes almost forgot that it was Acando he was working for and when people asked he answered that he worked for the current client. This kind of decentralized work environment can make it hard to build close relations to all other co-workers but Acando is aware of this and tries to counter it by having in-door days where all consultants meet at the main office for meetings and general socializing. Other measures include workshops, conference trips and interests groups for biking, exercise and more.

The work culture of the programmers and developers that I have interviewed and observed is a task focused yet a not an overly serious approach to the work in the sense that it's perfectly allowed not to be perfect but learn from the mistakes made under-ways. When talking about this topic of people being people and not all knowing one employe said:

> "At home we ask the stupid questions" - P3

Which means that as consultants towards customers it is expected to be experts on every IT questions, but internally at Acando there is an open culture for asking questions and learning from each other. There is also a culture for discussions and room for being critical as to how things are done, an important value for any company that wants to innovate and not be stuck within old ways. A quote that illustrates this is:

> "Here at Acando there is a high ceiling" - P3

A norwegian saying pertaining to a place where critique and discussions are welcome and encouraged.

The teams I have observed are about 5-6 people with a group leader and currently the scrum and agile methods of work are often used. This implies frequent and relatively short sprints where tasks and objectives are set for two weeks of work. I observed one such meeting and noticed especially an interesting method of estimation of tasks. For each defined task all the developers would choose card from their own deck of planning poker cards. These are cards ranging from 1 to 21 and when all members had chosen a card they presented them at the same time. More often than not they would show different estimates and from that starting point discuss what the most realistic timeframe was, this often included a discussion on what the task actually required.

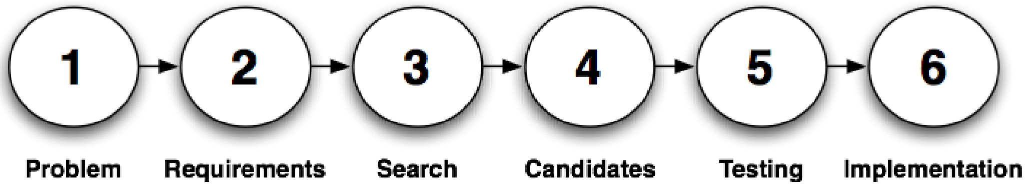


Figure 4: Process model for selection of components

When it comes to knowledge and skill Acando has had several projects for sharing such information on a corporate wide manner. These include email lists, CV databases, wikis and internal blog/forum solutions. The workshops are also intended as a way of exchanging experiences and ideas between the geographically spread workforce.

4.4 Findings

In this section the results of surveys,interviews and observations of Acando are presented. This research is qualitative and not absolute. Theories will be presumed on relatively little data however this is within the understanding of qualitative research.

After initial and second-interviews I had gained a clearer image of the extent of software component selection by developers at Acando and also some insights in which way these components were selected. I have made a simple process model for how, in a general sense, most components are selected in figure 4. Explanation of figure 4:

1. Problem In step one a problem encountered in the software development process is identified by the developer to be a general one and that most likely there exists a component that will aid or solve it.

2. Requirements From the initial problem comes a quick identifications of requirements that the potential component must fulfill.

3. Search Quick web searches are performed from the most used web-search engines like Google, Yahoo and Bing.

4. Candidates A shortlist of potential candidates is stacked up.

5. Testing The candidate components are downloaded and quickly tested by the developer.

6. Implementation One component is chosen and will then go strait to implementation into the main project.

The timeline for this selection process is short, often within a day or half-a-day of work. Bigger components that are more crucial and perhaps harder to switch later in development will take longer time and involve more developers, but for smaller (but important) components are chosen within a short timeframe and often by one developer on his/her own.

4.4.1 Identifying the problem

There is a subtle issue when it comes to the use of software components and that is the task of knowing when to look for components. During software development the developers themselves must stop and ask themselves “Is this problem potentially solved with a software component?” and if that answer is “Yes this problem is probably solvable with a software component.” then starting a process for finding and using such a component may start. How does a developer know this? Is there any general way of defining problems that can be solved with software components? Looking for solutions that probably don’t exists would be a waste of time, better to just start making the solution than to be going on any wild goose chase, but if a solution exists there is time to be saved by using it.

When asked about this it was hard for any developer to answer it without pondering the question. More or less they seem to have had enough experience with software development and software components that they a sort of feel for when a problem may be solved by using a component instead of starting from scratch. One developer answered:

> “When the problem or aspect of the system is very generall and common for most systems it is reasonable to believe that there exists a component. For instance almost all end-user systems employ GUI (graphical user interfaces) and therefore such components, be it the framework or general widgets, often exists as component solutions.” - P2

Whilst another, after mulling the question over for a while and actually suddenly coming back to the topic, said:

> “Protocol implementations will often be available as components

> and those will often be prefereable to implementing you'r own solution. Protocols will also almost always be available for a wide range of languages and plattforms." - P1

So for identifying if a problem can be solved with a software component there cain be said that the following will apply:

- Generic problems that apply to all or many systems
- Protocols and file formats
- GUI frameworks and components
- Databases
- API wrappers

But these are generall points and it is hard to say anything about a common identifier for component reuse.

4.4.2 Requirements

Having decided to try and find a component the developer then needs to set some basic requirements for the potential component. This step is important to try to accurately define what the component is supposed to do and how it should do it. Many developers also mentioned that these requirements should be as closely matched as possible, the component should ideally not do much more than needed functionality as this ads complexity to the component that is un-needed and potentially a bad sign. A simple component with at focused feature set is more likely to perform well at those few functions than a complex one with a plethora of features. As such the goal is a perfect set of required features, not a sub-set and not a super-set. Examples of requirements could be for instance a PDF generator component that needs to be able to output tables, take in 10000 characters, page numbers and so on.

4.4.3 Search

The search for a possible component is done mainly by internet searches. Developers value their colleagues knowledge and will use them for help in many situations, but for finding a suitable component a developer will often resort to Google and other search engines to gather information about software components. But why aren't the developers using their colleagues, perhaps their most valuable asset, for finding suitable components? When I asked if there was any threshold or reservation for "spamming" the mailing lists of Acando with questions that may be considered "stupid" that would be perhaps embarrassing to ask, everyone answered that it was quite OK to ask the mailing-list and that there was a culture to ask about anything. The reason for using search engines, dedicated websites and the Internet in general instead of colleagues lies in one developers answer:

> "For the specific problem at hand there is little chance that someone internally at Acando has encountered the excact same problem. With the Internet you have much larger group of developers and it is more likely that someone else has encountered the same problem." - P2

Also the use of the Internet meant that the searching went pretty fast, as opposed to sending out mails that may or may not be answered within an uncertain amount of time. So the Internet in this case works as a fast tool to gather information on available packages. It also serves as a tool to gain knowledge on the use of the potential components. Several sites are dedicated for developers that seek help from colleagues across the net like Stackoverflow [48].

4.4.4 Candidates

For the actual candidates that are selected initially developers mention several key criteria for what they are looking for in a software component. One colorful quote suggests a preference toward OSS components in general:

> "It must be open source and the website must be ugly." - P10

Others preferred COTS whilst most developers claimed to be of the pragmatic kind stating that they selected on the assertion of the component at that it wasn't really that important if it was OSS or COTS. But there were slightly different criteria for OSS and COTS. For OSS components it was important to se that the development of the project was fairly active by checking the timestamp of the last commit logs and gather some overall quality impressions from forum talk and download statistics. Rule of thumb being that the more usage a component has the more likely that the component is useful and of a minimum quality. The usage statistics also applies for COTS components, but additionally the developers were looking for good documentation, support possibilities and an assertion of the reputation of the company delivering the software.

From these criteria the developers would make a shortlist of 2-3 possible components that would be downloaded and tested.

4.4.5 Testing

Not all developers mentioned any specific testing of components. Some would simply pick the best one from the "Candidates" step and begin with implementation. But many cited that they downloaded the components and ran trough them with some quick prototype testing. This would quickly identify if the component actually meet the requirements and give impressions for a last kind of criteria that many mentioned, namely the "feel" of the component. This "feel of use" is an informal criteria that is based on perhaps seemingly unimportant qualities like how easy the component is to use, how the methods are structured and so on. From these quick and simple prototype test follows the actual pick of a single component, if no component stands out then one is picked in seemingly random fashion.

4.4.6 Implementation

From the quick test and picking of a single component begins the implementation of the solution. This means integrating the component with the software system that is being developed. From this point on the component becomes a part of the main system either by a loose coupling or perhaps more entangled use that makes the component difficult to "yank out" should it be necessary. Most developers claimed to use loose couplings, but as one of them pointed out these may be components that don't easily swap. GUI components for instance are often hard to simply replace later on in development.

4.5 Fast and easy selection process

My observations at Acando shows that software components are being selected and used regularly by software developers at Acando. The process for selection of these components is simple, informal and done over a short period of time. The components are small and simple, being “one task” pieces of software. Seemingly these choices often go well. During a presentation of these findings to Acando I asked the developers how often components were replaced because of problems, only one out of thirty raised his hand. From this comes naturally several important concerns and questions:

- Why is this informal process the dominant way of choosing components and not formal and standardized methods
- Why do so few cases result replacing the components initially picked / Why does this method seem to work so well?
- What are possible implications of these findings?

4.5.1 Fast and easy selection of components

The theory or assumption is:

> *The informal and quick selection of software components at Acando works as few or no problems will occur.*

And yet this conclusion is far from the theories of proposed methods of finding, evaluating and picking software components in academic and software engineering circles like CBSE proposed in [42]. The reason for using this method is simple, software developers said they simply do not have the time for formalized methods that take longer time. When faced with a problem that is possible to solve with a software component this has to be done quickly and efficiently for the software component to actually present any benefit over coding the solution from scratch. The main benefit of using software components, as perceived by most developers at least, is that of time spent on problems that can be drastically reduced. A trivial example to illustrate:

For a certain project a developer needs to be able to output data as a PDF document. For such a solution the developer can either a) read up on PDF document standards and create a similar document from scratch, this is the manual "do it yourself" method of development. Alternative b) would be to find and select a software component that does this. For there to actually be any point of using the software component it comes down to a question of time, at least for the developers point of view. For most projects time will be the most costly of all expenses. As such the point of using a component is lost if the process of selecting and implementing it is longer than it would be to make the solution from scratch. Also as one developer responded when asked why this quick method was used:

> "How many developers do you know that likes doing systems design?" -P8

So the quick selection is partly a result of conditions and a preferred way of working, time is important and speed is preferred. But still this process and frequency of use still demands that problems do not occur to often. Had nine out of ten components failed during the development then either other selection methods would be applied or the use of components would be lower. As such the quick selection process is also a result of it actually working in a satisfactory manner for developers. The reason for such low failure rates is one that is hard to identify, but some theories can by made as of why these software components aren't causing big problems.

Simple components :
The components are so simple and "focused" that they often are to be considered "one trick ponies" and as such should, at least, perform their single task in a satisfactory manner. Someone has made and used this component and it is to be assumed that it at least does that single task it was made to do.

Faults are captured by development process:
The components themselves are subject to systems development as the rest of the project is. With quality assurance, test driven development and acceptance testing any problems that the software components would be causing will be caught and dealt with during the development.

Small problems are fixed ad-hoc:
When asked why components rarely were replaced during the development of the project one developer answered:

> "If the component doesn't do everything it needs to, if it has

faults or shortcomings, I usually solve those issues myself" - P9

These kinds of "ad-hoc" fixes will be added by the developer to fill the gaps or solve the problems with the component and so the component does not need to be replaced.

OSS components can be changed:
All tough perhaps rare, some developers have added functionality or fixed issues with OSS components. Some of the developers at Acando has done this and it must be considered a viable option, at least for OSS components. In some of those cases it will also be possible for the changes to be committed back to the OSS project, thus including the fix and making the change unproblematic for maintenance and updates.

4.6 Roles and developer profiles

During my interviews and observations I noted that not only can one group developers into their respective assigned role, developer, group leader, software-architect etc. but also their personalities as developers. Here I present some archetypes of developers at Acando and a radar graph illustrates and compares them in figure 5. It is important to make clear that this chart is meant only to illustrate the different types of developers and their differences, these numbers are not factual but interpreted impressions of the different kinds of developers that have been observed and interviewed. It is important to note that these are personas, characters that don't exist but represent the outermost traits of aspects of the developers. Think of these personas as parts of a scale system where any one developer is in between these personas.

The OSS Developer
The OSS developer has an affinity for open source software in general, he or she uses OSS components, operating systems and solutions frequently in their professional and personal IT lives. He or she may have different reasons for using OSS like ideological, experience, preference or cultural background. Some schools for instance have a strong culture of OSS or OSS related technologies. Whatever the reason this developer thrives with OSS technology and finds good support in forums, chat-rooms and mailing-lists and likes to have the ability to dig down into software to se how it actually works, either to understand it or fix it. This developer believes that OSS software is just as good or better than their commercial counterparts. An OSS developer is more likely to choose an OSS component or solution than a COTS or one

that is proprietary and closed. As one developer answered when asked what kind of components he looked for:

> “It must be open source and the website must be ugly” - P10

A comment that got a round of laughs from the surrounding developer crowd as it was a reference to many OSS projects that spend a lot of time developing software, but spends little or no time on website design.

The Commercical Developer
The commercial developer is one that values professional software development with rules, structure and a certain guaranteed value for the money that is exchanged for software. He or she likes that software is well documented, has enterprise support options and that software is updated at predictable timelines and schedules. There is no problem with software that is open source, but it is strongly preferred that this is commercially and professionally developed software. The reasons for preferring commercial software over community driven software vary, some only care for the documentation and support, others believe that community driven software lacks the focus and discipline of professionally made software.

The Pragmatic Programmer
The pragmatic developer has a practical and agnostic approach to components as he or she does not find any particular brand or type of components to be superior to the other in a general sense. Software is software, it’s either good or bad, and there are plenty of bad software examples in both the closed commercial software world developed by professionals as there is in the community driven hacker world of open source projects. The pragmatic approach does not exclude options, but rather picks the best suited component for any given task or situation.

The Freshman
The freshman is a relatively new developer and most likely fresh out of school. As a new developer it is not often easy to know what kind of options are available, for example “Can I purchase this expensive piece of software?” and it will often be much easier to try to solve problems by programming. Inexperienced developers like this may not be familiar with using software components and can lack knowledge about their general existence, availability and usefulness when it comes to software development. Even tough schools promote the ideals of software reuse with the idea of object oriented programming, where making smart objects and methods is encouraged to write

less and more efficient code, the notion of seeking out software libraries and components is not one that has found much academic focus when it comes to programming classes and curriculums. The freshman developer may grow out of this group and become more proficient in using components, but some developers in some ways never leave this group as they either spend a lot of time within projects that don't allow or require components.

In addition to these developer profiles one can group them into roles in the company structure. These are based on answers given by developer during interviews and talks.

Group Leader
Group leaders at Acando are responsible for several teams of developers and have a role and position that puts them between the developers and the actual bosses.

Team Leader
Team leaders have responsibilities to guide and follow up on the actual developers and are a part of the development process, but don't necessarily program or develop themselves. They are important towards customer relations and steering the developers.

Senior Developer
Has worked with the company for a long while, five years or more, but is comfortable with being a software developer and sees no need or reason to become team leader.

Junior Developer
A newly hired developer that may also be fresh out of school.

Software Architect
The software architect plans the major outline for the larger systems and chooses the initial and big components for the projects. The systems structure and overall design is the main responsibility and this requires a larger look mentality and also to closely follow what the market has to offer when it comes to components.

Advisor
Advisor consultants are hired for the purpose of advice and overall knowledge and competencies in the world of IT, they are usually involved during the initial stages of development or even before it is decided that something is to be developed.

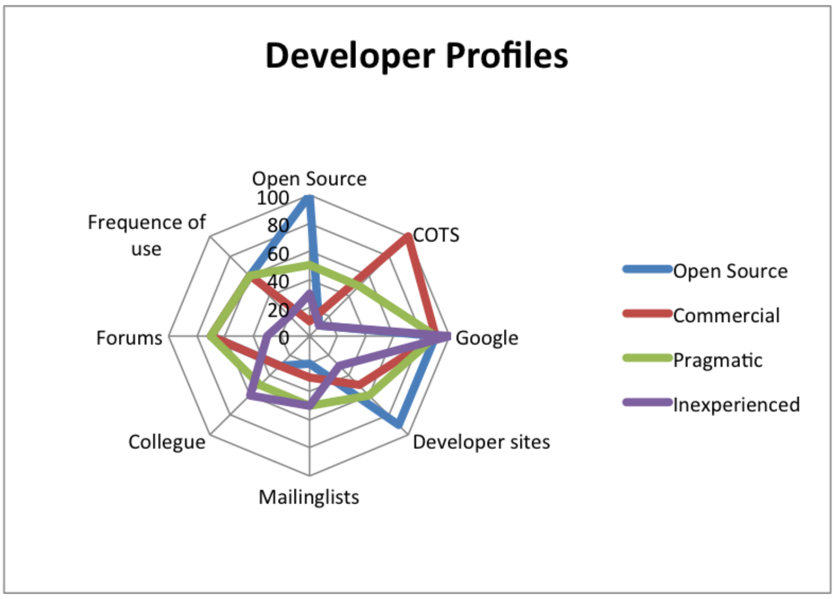


Figure 5: Developer profiles

4.7 Projects and phases

When talking to developers at Acando and looking at case studies of software project there emerges different types of software development projects and different phases that they go trough. Graphs that illustrate the different phases and profiles of projects are found in figure 6 for the profiles and figure 7 for the phases. Here follows descriptions for the illustrated projects and phases. It is important to note that these presentations are approximate and based on the impressions gathered from developers, they are not based on actual numbers.

Planning, modeling and requirements

During the planning and modeling phases there is little component use, however the big and fundamental components are usually selected and planned during these phases and the platforms chosen does have implications for the further use of components.

For the requirements phase there may occur some small component use as they can fill functional gaps during this phase if the architect or developer involved already know of such components.

Development and maintenance

It is during the development phase that the highest amount of component use

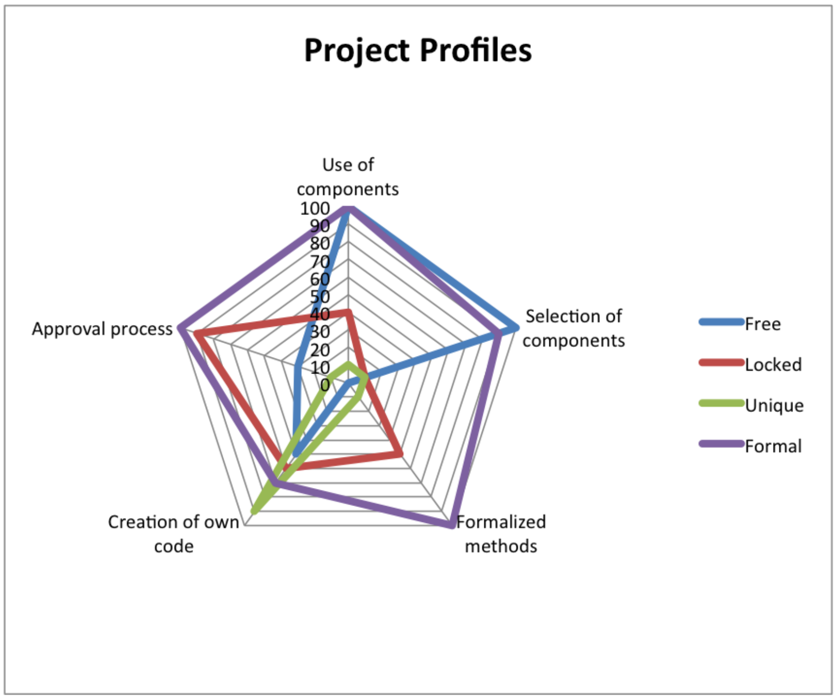


Figure 6: Project profiles

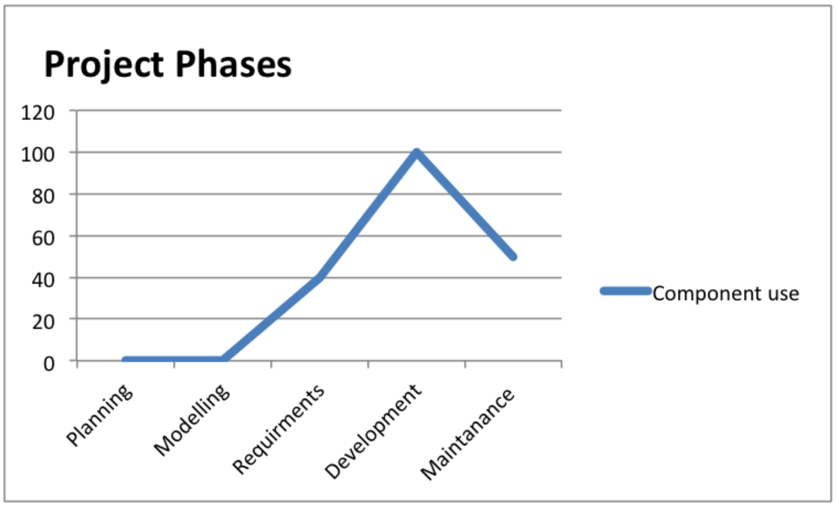


Figure 7: Project phases

takes place. This is natural because it is during development that the need for smaller components may occur and become apparent and the developers working on the system may choose to implement them as the solution for those needs.

During the maintenance phase there is some component reuse but not as much as during the main development. Perhaps components selected during development needs to be replaced because of lack of updates or new functionality is need and the old components don't match the new needs.

Free project
With a free project the developers are simply asked to create a solution at a certain budget (sometimes the budget restrains are also very loose) and everything else is up to the developers. There are no constraints as to what kind of technology that can be used and as such this project type is open for developers to use a wide array of components, platforms and existing solutions.

Locked project
A locked project is less prone for heavy use of components as there is a stricter frame for technologies and solutions to be used from the customer (or it may locked down by the architect). There is a stricter approval process for including components and this lowers the use.

Unique project
A unique project presents a problem or system that is unlike anything else that has been previously done. It may also be a project done on a very narrow framework which may be outdated or so proprietary that there exists little software to be reused. For such project it is almost impossible to find any software to reuse and the problems may be to specific for generic solutions to exist.

Formal project
A formal project has a decent amount of software reuse but a little lower than a free project because any component included into the system must be thoroughly justified and approved, be it OSS and free or COTS and costly. This bureaucracy hinders software reuse because it makes it more of a bother for the developers.

5 Discussion

5.1 Introduction

Here I will discuss potential implications of the findings that where presented in the Case-Findings chapter.

5.2 Component use at Acando

During the data collection questions would arise that needed answering. Returning to developers for further interviews and targeting specific problems helped to answer these issues, but often the answers would bring more questions. During this phase the topics and themes of the thesis were analyzed and discussed by me and the thesis advisor.

The findings from Acando show in short the following:

- To some extent components are being selected and used quickly and easily by developers
- These components are not causing problems
- These components are working and being used to positive effect in systems development

So assuming that these points are true one could further present that Acando is benefitting from the use of software components. The selection process of "quick and easy" seems to be sufficient and efficient in selecting components. This method also seemingly has a high cost to gain ratio as often minimal time is spent finding and selecting components. And if Acando is benefitting from the use of components then perhaps Acando are being to careful when it comes to the extent in which components are being used. A claim could be made that:

> "Acando is currently using software components in a non-systemized fashion. Developers are solving problems by using third-party components, but this is neither sanctioned or forbidden."

Seeing that software components are being used successfully at Acando one would expect that the use of components would gradually increase. This does not seem to be the case as most developers report to using a certain amount of components, when they seem appropriate, but never increase the frequency of use beyond a certain point. Some developers would answer that they use the "proper" amount of components and don't go beyond that. They also suggest that there is such a thing as "to much" when it comes to software component reuse. But why is this? If use of components give a positive net effect then why not use as many as possible? In the following sections I present some theories as to why component reuse stagnates at a certain point and also why some developers use more components than others.

Software craftsmanship
Software craftsmanship is a term or idea coined during the late 90s early 00s and was a response by several prominent software developers to problems regarding software deliveries failing and economic interests being the priority and not the quality of the software. The principle idea or philosophy behind software craftsmanship is that programming and creating software is a craft that is learned and developed over years and that it is a skill that needs to be developed. This in contrast to software development being a perfect and replicable science. In short it presents the idea of a software developer being an individual that creates something unique and solves an equally unique problem, with many possible solutions. By applying the ideas of craft and skill, like that of art or carpentry, one can create a purpose of being proud of the work of programming, aiming for the creation of code that looks good and acts good.

When interviewing and talking with developers at Acando there is a sense of craftsmanship ideals present in the developers. They are tinkerers of code that create solutions for the customer trough the end product and have a sense of pride in this task. It is perhaps here that a reason for component reuse stagnating at a certain level. Developers don't won't their job to "simply" be gluing together different components. As one developer put it in comparison to remodeling his own home:

> "In the beginning I was using mostly finished components, but after I got better at it I wanted to use less of the complete solutions and rather make my own from scratch using basic materials." - P1

This attitude and mindset points to a reluctancy to go "all the way" with component use in that it would not be fun or challenging work to simply download an X amount of components and make them work together as one larger solution. To much reuse would ultimately make the work less interesting, challenging and compelling work. There would also be a lower sense of craftsmanship if everything was simply readymade pieces put together. In a sense it adds up to a kind of reverse cognitive dissonance, the psychological effect of being to attached to once own work making it hard to accept faults. Here the developers are reluctant to distancing themselves to far from the solution, not being able to make enough of the solution makes it not a creation at all. As such it seems that developers that use components do so at a slightly increasing curve but stop at a certain point where they feel they are using the "right" amount.

Whilst at an Acando-workshop on Ruby, a programming language, the course presenter showed a built-in library system made for Ruby that makes it easy to install third-party libraries made with Ruby. For example a module that makes it easy to implement authentication will be installed by calling a command like "gem install auth-pack". When presented with this an attendant at the course quickly asked the question:

> "Is there any central rating system for these components? Is it easy to find good ones?" - P11

This is an indication of wanting to use components, seeking good places to find them and perhaps showing a demand for better systems offering ratings and reviews of components. I would imply that attending workshops and looking at new technology horizons is a sign of the software craftsman, an individual that wants to improve his or her skill as a developer. This is not in conflict with using components, it may just be the opposite. A craftsman might think "why reinvent the wheel" that is not an effective and smart way of working at all. As such the ideals of software craftsmanship may hinder component reuse if the goal is 100% reuse and minimal development efforts, but to achieve a relatively high degree of reuse it is a positive incentive.

Incentives

Using third party components has the potential benefit of saving time and utilizing better software that does it's specific task very well compared to a home-made solution. Time as everyone knows is money and this is especially true when it comes to IT consultants. It is interesting to note that many contracts and projects are billed at an hourly rate, the almost infamous "writing hours" principle applying to the developers assigned to a project.

If the use of components indeed reduce development time does that mean that they also cut down on billable hours for the firms? Should the price or solution be based not on hours spent developing but a fixed price for a fixed amount of features than using components would increase profits as hours spent for the developer would go down. These two different models seem to incentivize differently. A feature based model would benefit from extensive reuse whilst for hourly billing it would be a drawback. Can it be suggested that component reuse in consultant IT firms isn't increasing because there is no incentive to do so?

This negative incentive requires that the use of components truly does shorten development time, but do they really do so? There is an estimation model for software development that says that for a project that has available X amount of money, Y amount of time and Z amount of people, the project is estimated to use all of X,Y and Z. It is therefore perhaps to easy to claim that the use of software components actually decreases development time. Maybe it is simply less time spent on that particular part of the system, giving extra time and attention to other aspects and problems. Perhaps there is more time for testing and quality assurance, creating better and more valuable software. One developer explained the use of components in an interesting manner:

> "Think of it as gaining developers for free. If we were to make the solution ourselves we would perhaps spend a lot of time making a solution that in the end is worse than that of a component. Behind any component, that is good, stands one or more developers that have already iterated trough poor solutions and ended up with a good one. By using that component you are gaining those developers effort into the project." - P1

The point is that the negative incentive theory is only applicable if components truly lead to significantly lower development times and that consultancy firms know this and therefore try not to use them. My findings at Acando show that there is little incentive in either direction, no agenda that promotes or discourages reuse. However there is an agenda that promotes quality solutions in an efficient manner, meeting deadlines and budget constraints will always be an important goal, and this agenda is indirectly positive to component reuse. Should this be true then this theory should be further examined. It is off course a troublesome and potentially controversial problem, but a possible one none the less. Why would IT consultant firms extensively use components if they end up loosing business?

When presented with the idea of hourly billing being a negative incentive to component use an Acando-employee the response was that even tough hourly billing was the prominent form of income, as opposed to fixed prices, but that this did not mean consultants were reluctant to use methods or technology that cut down on development time. Effective development delivers more value to the systems, more features and better quality. Also hours are often bought in bulk or periods of time and having effective consultants is important as to deliver value and not expenses to the customer.

> "If we were using hammers instead of nail-guns to prolong the time and write more hours, we would quickly loose clients and integrity as a consultant firm that delivers value to our costumers."
> - P3

This again is an indication that firms are not negative to the use of components when it comes to a potential business model conflict.

Why more components are not the answer
When talking about the subject with a software developer and architect at Acando one theory as to why reuse hasn't been the solution and never will be the solution, is simply that any system developed for a specific purpose, a system tailored for one company for one specific domain, will require time and effort no matter how much you reuse software. Such software is inadvertently complex and as computers and software become more advanced they will become increasingly complex. Such systems are helped by reuse, making it easier to implement protocols, GUI, and solve other generic problems that the systems may encounter, but the systems themselves will always demand complex software development. Simply put the LEGO approach to making systems will never be a complete solution that will drastically change the way software is made, not for the foreseeable future and not for the systems that are being built today anyway.

Maintenance and further development
Several developers mentioned concerns for extensive use of components when thinking on software maintenance. The cost of software maintenance is a problematic one because it adds another perspective to systems development that is rarely, if ever, discussed namely the further development of the system. Even tough good software engineering patterns and systems modeling create loosely coupled systems where parts and pieces can be exchanged, there is seldom a true plan for updating software. Patches are common yes, but a software patch is mostly rewritten parts of code that is added to the system.

For systems that use components they are individually developed by other parties, hence third-party components. Should a problem arise with these components when the system is being upgraded or simply bug-fixed then one has an added dependency that the component is fixed by the third party developers. If the component is causing problems and is not fixed, there may be several reasons for this to happen, then the component will perhaps need to be replaced, either by finding another component or creating the solution from scratch.

One developer mentioned that he was reluctant to use to many components because it caused an added overhead to the next developer that would be working on the code. Not talking about the updating problem but the problem of reading and understanding legacy code. Systems that extensively use components may be harder to read and understand, an added overhead to the next developer so to speak.

With the problem of maintenance there is perhaps good reason for developers to limit the use of components to a certain level, however it assumes that maintaining original code and solutions is easier to maintain than a component equivalent. If the problem is real then the use of components is perhaps that of a short term benefit and a long term disadvantage when it comes to software development. If the firm delivering the solution has a short term perspective then perhaps components are a benefit of them and not the customer. If a longterm perspective is held also for the firm should then use of components be restricted? Such problems remain speculative until studies of long term component use have been made.

5.3 Knowledge, OSS, COTS and software reuse

5.3.1 Knowledge

My findings revealed that a potential problem and hinderance to component reuse is related to knowledge and knowledge management. Here I discuss these issues.

Keeping up to date
The sheer number of available components today is staggering and the rate of which new components are created and added does not seem to be slowing down. If it is required to stay up to date in this market it is understandable that some developers simply don't have the time or energy after work hours to put into this task. As one developer at Acando answered when I asked him about how he could increase his use of components:

> "I believe I could use more components, but I simply don't keep up to date on what that is out there. It's a basic lack of knowing what's available." - P7

The use of components in software development is naturally linked to the problem of knowledge, if you do not know there exists components then you are less likely to use them. Having knowledge about these components is an overhead that adds to the existing overhead of keeping up with modern software development practices and it is perhaps to much to ask of developers to read up on yet another aspect of programming such as this. This then applies to those developers that less frequently using components, they are possibly using less than the "right" amount that other developers claim to be using. When asked in a direct follow-up to the answer on how he could increase his use of components the answer was simple:

> "I would just have to read up more about components, dig trough sites and pay close attention to what is happening and what is out there." - P7

It could of course be argued that this is not needed to use components, several developers seem only to use internet searches and their own experience when it comes to finding components. Being efficient and quick at finding and identifying components regardless of language or project is certainly a much more valuable technique than having to be familiar with a huge number of components at any given time. Nevertheless the task of knowing about components is a big one and potential stopper for new developers that are unfamiliar with finding and using components.

This skill of finding and using components is one that is hard to define in writing and as such poises a problem for knowledge management that is not new, to transfer tacit knowledge to a geographically spread workforce. It may simply not be possible for any one developer to write down "do such and such to find and such and such to identify good components", the attempt could certainly be made, but articles like [55] shows that many skills need to be transferred by learning and doing from an instructor.

The use of components, unlike other developer traits, may not be the biggest talking point when discussing the trade with other developers, be it colleagues at the same company or friends that also work within IT. At Acando there are several interests groups and there is about once or twice a month held workshops for new perspectives on technology. Topic examples includes Ruby, Security or the Azure Cloud platform. These groups fit well with the idea of communities of practice as proposed by Walsham [55] in that here people that have the same interests meet and share their knowledge in groups.

It is possible that the subject of smaller software components is such as small one that it goes under the radar of regular tech-speak, it is a subtle topic that many developers are familiar with, but don't talk that much about. The reason may be simple, pr. component it is not that an interesting or groundbreaking topic when compared to more paradigm shifting ideas like SOA or cloud computing. Component reuse doesn't have this big impact, at least not when you think of them one component at the time.

As such the reuse of smaller components is not a great enough topic that it spreads trough developers like other development techniques or bigger pieces of software like say the .Net framework or a programming editor like IntelliJ. By going under the radar of such arguably important sources of knowledge and information amongst developers, as presented in [56] with the coffee machine story, perhaps this is a reason for the lack of viral and normal collegial spreading when it comes to software reuse in firms such as Acando.

Lack of technical solution
Acando has tried several technical solutions, wiki and CMS being the most prominent and it can be argued that their current CV system employs the

possibility to make available knowledge about components. However these solutions may not have been the best suited for the job and may have failed in their executions. For these solutions there is an overhead of learning how to use them and a barrier of entry to use them, for instance for any wiki article there is perhaps expected a certain amount of text or completeness. So even tough Acando has tried and arguably failed with a technical solution to its knowledge management, perhaps the solutions have not been mature or right for the job.

Acando has tried both an internal wiki and internal CMS systems as technical solutions. The wiki system makes it possible for anyone to add articles to a knowledge database on any topic whilst the CMS system has billboards and discussion groups, making it easy to post questions, news or discuss topics internally within Acando. These systems have failed at achieving widespread use or at least the critical mass of use to make them truly valuable as knowledge resources. They have simply not been used in the extent that they need to have achieved critical mass as such crowd sourcing tools require.

In the case of the internal wiki there is clearly a time consuming component of filling inn information on a continuing basis and there are few if no incentives to actually do so, other than being helpful and kind. Several developers commented on this topic and the those with management positions were well aware that the tendency was that such tools were to time consuming and gave little back to the developer taking the time to write entries. The theoretical gains of such solutions remains high whilst the real life use and gain remains low seems to be the verdict. Many still feel that such systems could work as long as they manage to gain that critical amount of usage and that developers themselves understand the value of the system, not for the short term gain but the long term gain. One developer had an experience with writing such an entry:

> "I looked up something about MS SQL and I found my own entry about some tricks when it came to migration that I found really usefull, I had forgotten that I had written that entry myself"

There is knowledge that sticks and knowledge that is quickly forgotten when you stop using it on a daily basis and clearly such knowledge databases may be useful to storing such forgettable, but not useless, knowledge. There is a clear value to wiki and CMS systems, but how does one encourage and incentivize higher use from developers?

There is also a thought that the technical tools themselves are not mature enough, several developers suggested more "modern" approaches to knowledge notation and several recent articles and trends point to interesting developments when it comes to a technical solution to the problem of knowledge management. The recent year has seen the explosion of social networking sites, FaceBook and Twitter are the biggest examples but also worth mentioning is LinkedIn and Tumblr.

Twitter is an interesting example because there is not a strong focus on being friends or socially connected to the people you follow or tweet against, you simply either follow or don't follow people. Twitter in a nutshell is microblogging with a limitation of a 140 characters pr. update or tweet as it is called. Tweets can contain URLs which again can be to a site, to a picture or to a video. Tweets can also contain hashtags, words with a "#" in front of them which indicates the topic the tweet is about and also works as a search-term to view other tweets with the same hashtag. To tweet is something done fast as the goal is to proclaim something in the shortest amount of chars possible. I myself have a twitter account and I follow several developers, I have noticed that many of them will use twitter and, trough either their network of followers or simply by using a hashtag, will ask developer related questions trough simple tweets and often receive several answers to them.

It is therefore easy to conceive of something of an internal twitter system for developer firms like Acando that promotes simple updates or questions. They can be either informative tweets like "I used component X for making #pdf in #java and it was great" or questions like "Does anyone know a good #PDF component for #java?". Such tweets would be live streamed to other developers at acando, or perhaps a larger professional network of developers, and also stored in a larger database where the hashtags could be used as metadata making the tweets easier to organize and search trough for further use. In fact you could generate automatic knowledge databases if users were connected to their tweets, showing that "Developer A has 20 tweets with #java and 1 tweet with #pdf". Such a system would be quick, easy and painless to use, making a tweet with the right client software takes about 10 seconds, and could even create greater developer interaction between a geographically spread workforce like that of Acando. Microsoft is working on a module for it's Sharepoint solution that could easily be mistaken for this proposal - they call it Office Talk and has a Twitter- and Facebook-like functionality built into the Sharepoint suite [31].

However even tough such proposed systems may sound great and have seemingly great potential it is also worth remembering that wikis and CMS systems also had and still have this same potential promise. A technical solution is therefore simple to think and theoretically create and use, but harder to make a reality. All tough it is worth mentioning that SAP and IBM have been promoting the use of social tools lately, for instance one report claims that:

> "This turned out to be highly motivating. Last year we delivered 160 000 projects, with shorter development time, lower costs and a higher reuse of components" [44]

This was an effect of incorporating social networking technologies, specifically an effort where developers were encouraged to suggest which projects they would work on themselves. IBM is now pushing social tools as a new business booster, but there are no short cuts as they also explain:

> "The training is a part of our hundred year anniversary program. Everyone were to advance the company by strategic and responsible use of social tools."

The fact that big companies like IBM [44] and SAP [43] are making guidelines and investing in social tools suggest that there is something to be gained, but I believe it is a little early to be sure of exactly what this is.

The use of crowdsourcing

All of the developer that actively used components mentions Internet as a primary source for knowledge and information about components. It is here worth noting that software component mostly live and exists on the Internet. This is where they are sold, developed, marketed and either grow or disappear. These same developers also mention that they value information and knowledge higher than random internet forums and web sites, but still they are most prone to use the Internet when it comes to finding components. One developer gives a simple and reasonable answer as to why this is the case:

> "There is a much larger probability that someone on the Internet has had my problem and has a component to recommend. A larger probability than someone within this relatively small firm

to have encountered the exact same problem at least. A Internet search is also much faster than contacting someone else." - P2

One could easily argue that this assumption and method is faulty and may even result in loss of potential components, either that are better than the ones chosen or no components were found when there in fact existed components. However it is hard to argue that the Internet search is faster and has a higher probability of returning results, even on obscure software development problems, than phoning a colleague or sending out an email.

In this sense the Internet can be seen as the ultimate crowdsourcing tool, indeed the term crowdsourcing originates from Internet based technologies [20]. The basic idea of crowdsourcing is simple, you put you'r problem out in the open and hope for people to respond and help you out. The more people that can help you the larger the probability that someone will have a helpful answer. For developers at Acando they sometimes do this, they may post a question up on StackOverflow.com on what component other developers would recommend for a specific problem, or they may simply search for the component and find already asked and answered questions on sites such as StackOverflow.com. With modern search engines that are more frequently updated, index more and more sites and give better search results it is increasingly easier to find relevant information for the seemingly most obscure problems. With the large number of users that the Internet possesses the odds of finding help increases dramatically. Internet as a crowdsourcing tool then achieves this critical mass of users that the internal solutions often fail to achieve. One could in fact argue that the Internet is a knowledge management solution that works, if not in a very chaotic way.

5.3.2 COTS, OSS and Software Reuse

Threshold of use

My first observation was that for commercial software development there is an important aspect of threshold for using a component or not. For some projects there will be a need to justify any added costs to the project, so selecting a COTS component is something that requires approval from someone in the project leadership. These costs don't necessarily need to be particularly high or problematic in themselves, but having to ask permission to spend more money is a requirement for COTS components that OSS component don't face. When searching for components, there may simply be a lower threshold to select an OSS component because downloading and using OSS components does not require any budget approval. In fact this low threshold also permits a higher degree of “try and fail” as it is much easier to download something that is OSS and try it as a solution, if it works then there is no need to ask for budget approval and the software itself has proven its value. One developer had a story about this kind of mechanic:

> “One manager had said at some point that ”We will never use OSS in this company“ as to which a employee responded ”But our mail-server is already running on Linux and has been for a long time now“” - P6

This story was presented as a proposal that OSS software was being used more often than managers might even be aware of. Clearly using an expensive COTS component is not that easy to simply do without approval from either customer or management.

For many companies and projects the use of OSS components is potentially problematic when it comes to licenses. Companies that provide software solutions or solutions that use software are more sensitive to the different kinds of OSS licensees.

Culture and benefits
The biggest difference between OSS and COTS is perhaps the cultural aspect. Several developers have a leniency towards either OSS or COTS and has experience or ideology as reasons for this. Many of the benefits of either OSS or COTS are disputed by people with strong cultural affinity, for example someone that appreciates COTS components would often downplay the option of altering OSS components.

Whilst these cultural and perceived differences are many, the actual consequential differences were harder to find. Developers did not report that OSS components were more troublesome or that COTS components were much better. They had perceptions that this was or could be the case, but most were certain that software was software, it is either bad, good or something in between.

The kind of developer will influence how the developer acquires and seeks knowledge about components. Being an OSS oriented programmer will tend to follow OSS news and look for components within that category and the commercially oriented programmer vice versa. Ideally is perhaps the pragmatic developer that doesn’t care if the component is OSS or COTS, giving him or her the biggest pool to select components from. However as presented these different developers are not real and few developers are in one booth, if presented with only one choice then the developers that use components would pick that one as one stated:

> “For that component we were looking for something commercial, but we only found an OSS component and picked that one” - P1

So for any developer that is used to using components there will seldom be such a strong affinity for either OSS or COTS that he or she opts to not use a component when the only option is COTS or OSS.

Benefits that are perhaps underestimated is one of culture. Open source components are created and used by developers that thrive on forums and email-lists. In general there is a culture for answering and exchanging solutions to problems and the support this provides for open source components is often a great deal larger than for pure commercial products. One developer mentioned that they had purchased enterprise support for an open source product but that he felt that the community support for this product was lacking. He felt that when the product had this enterprise support and people were purchasing it the community support disappeared as the questions that would be asked in the forums were being asked to the professional support staff.

The lack of differences
Many articles talk and discuss OSS and COTS as if they are from different worlds when it comes to software. COTS has the perception in several publications as to being the true commercial software alternative. OSS is perceived as an interesting phenomenon and much has been written on the quality of outstanding OSS projects and the commercial value of these.

These terms have in many context become laden with meaning that is hard to actually find in todays component marketplace. There are several commercial vendors that are open sourcing their solutions. They are selling different degrees of their own product, one free without support, one paid with support and so on. Within the traditional OSS world there is a stronger commercial presence, OSS products are monetized with either support, services or in some cases developer expertise for hire. With these worlds blending and becoming more complex the OSS 2.0 term starts to make a lot more sense and the traditional schism between commercial software and OSS software means less and less.

Looking at articles that study OSS like [32] and [57] they are all looking at OSS software as something distinct, something different from the commercial alternative. As the developers present it they see software as software. Yes there are certainly differences between purely commercial software and free open source software, but for a developer it all boils down to the exact same thing which is to find software that works and using that to aid his or her own development of a software system. The issues of ideology, openness, relative costs, development methods and other factors do not matter that greatly when the goal is the same.

Lack of formal approaches
When talking to developers there is a distinct lack of academic influence in the work methods. No developer would say that they employ the values of CBSE or name any formal approaches to software reuse. There is one loosely based principle of loose couplings between software but that is only a small part of any suggested framework or systematic approach to software reuse and is a principle that is found in many programming languages and cultures, not at least that of software craftsmanship in being efficient and coding in a smart way. Looking at articles on selection proposals like BASIS [2] for COTS or OSM [47] for OSS there has not been a single developer to even mention such proposals, this is in accordance with the findings of Hauge in [17].

5.4 Improving at Acando

For Acando this thesis has been interesting to find ways to improve the use of third party components in systems development. This should preferably be a simple, painless and low-cost improvement of methods building on todays practices. Here follows a string of topics on which Acando could improve, some more feasible than others however the main focus here are simple, easy and cost-efficient ways of improving reuse.

Formalizing and incentivizing
As it stands today the use of third party software components is a frequently used practice at Acando but it is not one that is formally encourage by the company itself. As it seems it is something that developers have a practice of at an individual level, even tough it is common enough to be a normal and well accepted practice. Should Acando want to increase the use of components they could simply encourage all of its developers to do so, making it a publicly accepted practice.

Acando has several important and well functioning ways of spreading ideas and knowledge internally throughout the company. The simplest way it might seem is to have interests groups and workshops that make as meeting points for developers to meet after work and discuss technical topics of either commercial interests or simply interests (and often those topics also become commercial at some point). Should Acando want to increase the use of software components it could simply have some lectures on the topic, either at a central meeting of all employes or at a voluntary after-work workshop group. The developers who are familiar with finding and using components could then share simple tricks, favorite websites, things to look for in a component etc. Such groups are good examples of communities of practice [55] and are ideal for spreading and sharing knowledge. This is a simple way of trying to increase reuse at Acando.

By formalizing the use of components there would also be room for guidelines and rules for component reuse. However as to strict rules could potentially hinder reuse it would be important to create guidelines that don't add complexities and added overhead to the development process itself.

Technological solutions
It would be interesting for Acando to look at the social tools that companies like IBM and SAP are using and promoting in order to achieve more internal communication to improve knowledge sharing. Also looking at Office Talk for Sharepoint by Microsoft there are several technical solutions to be considered. Acando already uses Sharepoint and as such the Office Talk technology is especially interesting. However as Acando has already tried new and promising technology before, like wikis, there is little evidence that guarantees any results from such a solution. Also this is a proposal that requires much more time and effort and could be potentially costly for the company.

6 Conclusions

6.1 Conclusions for research

Software reuse
Small software components are being used without big problems at Acando an IT consultancy firm in Norway. The frequency of use amongst developers vary from individual to individual and there does not seem to be a pattern of increased use. With the full potential of software reuse seemingly not being achieved or even attempted there is a question of why that is. Software developers are dealing with making customized solutions for customers, systems that have similar functionality to other systems, but is still tailored for one special task and general software is not enough for the client. Within this system lies requirements for functionality that many other systems, and therefore developers, has already made and solved. Reusing such components aids software development, but will never fully replace original software solutions and it's challenges.

Software craftsmanship
It is interesting to note how individually these developers are working. Sitting with individual tasks on larger group efforts the single developer meets problems and solves them, either by himself/herself or by Internet searches. When these solutions involve third party software components that are found, selected and added to the system without much peer review is this a positive trend or a problematic one? There is also the interesting concept of developers being reluctant to use to many components, becoming simply gluers of code. This may never be the case as original software solutions will be needed in the foreseeable future, but should developers be so proud and connected to their work?

The concepts of software craftsmanship promotes a strong connection and pride towards software development and the goal is better quality software with smarter solutions, it is not clear if this stands in the way of extensive software reuse or not.

Software development Literature and software development courses focus to much on either original development, internal project reuse like object oriented programming or strict formalized selection methods that se little to no use in the real world. Reuse chapters present a simple reality where it is suggested a bottom-up approach, you start off wanting to reuse and design the system around existing components. This is in most cases impossible and the staggering amount of existing components makes the task almost impossible if not extremely time consuming. Such proposals are a stark contrast to the reality where components are added when needed and possible, a method that is simple, quick and gives small incremental benefits to existing development methods, instead of having to redefine it. This point lies close to one statement from the 1999 article "Why Software Reuse Has Failed" [46] :

> "The principles, methods, and skills required to develop reusable software cannot be learned effectively by generalities and platitudes. Instead, developers must learn concrete technical skills and gain hands-on experience by developing and applying reusable software components and frameworks in their daily professional practice."

The author of the article is talking more about internal systematic software reuse, although OSS is mentioned as a prominent model for reuse. Still the point applies here that generalities and platitudes don't apply so well to real world development, especially when evident by this almost under-the-radar practice of software reuse.

6.2 Conclusions for developers

Increase reuse
When the use of components goes seemingly so well there is little reason not to encourage further use. Simple guidelines and practices should be established. Developers with experience should create collections of good sites and trustworthy vendors. However the more overhead such projects require the less likely they are to succeed. Simply encouraging and formally acknowledging the practice of software reuse could be enough to increase frequency and effect of reuse.

6.3 Limitations

This thesis is based on studying one branch of a Norwegian IT consultancy firm in Trondheim and as such what is observed and concluded here may be attributed to local factors. Cultural aspects of Norway and Acando may be unique and not be transferrable to other places and contexts.

6.4 Further work

OSS and COTS
The differences between OSS and COTS software needs to be investigated closer and probably the terminology needs to be redefined. With a steady commercialization of OSS software and increasing open sourcing of COTS these categories are merging and making less and less sense when talking about them as different kinds of software. Looking at not only the differences but the similarities between third party software and identifying new markers that classify them would aid further research and studies.

Component use effect
The premises of software reuse, be it small components or larger ones, is that of creating better software faster and cheaper. However there has been few studies that actually compare reuse projects against original projects. The problem of course being that each software project can be defined as unique and you would almost never be able to develop one project with the same developers twice as a test. Measuring the actual effects of software reuse is difficult but needs to be addressed.

Also the long term effects of software reuse are yet unclear. As it stands today it is hard to say if maintenance is more expensive and further development is more difficult with heavy reuse projects. This should also be a topic of huge interest for further studies.

References

[1] Chris Abts, B.W. Boehm, and E.B. Clark. COCOTS: A COTS software integration lifecycle cost model-model overview and preliminary data collection findings. In *ESCOM-SCOPE Conference*. Citeseer, 2000.

[2] Keith Ballurio and Betsy Scalzo. Risk reduction in cots software selection with basis. *COTS-Based Software Systems*, pages 31–43, 2002.

[3] Lisa Brownsword, David Carney, and Tricia Oberndorf. The opportunities and complexities of applying commercial-off-the-shelf components. *Crosstalk*, 11(4):4–6, 1998.

[4] W.F. Chua. Radical developments in accounting thought. *The Accounting Review*, 61(4):601–632, 1986.

[5] Paul C. Clements. *From Subroutines to Subsystems: Component- Based Software Development*. Software Engineering Institute Carnegie Mellon University, 1995.

[6] Creative Commons. Attribution-NoDerivs 2.0 Generic (CC BY-ND 2.0. *http://creativecommons.org/licenses/by-nd/2.0/*.

[7] Cyanogen. Cyanogen Mod cm-kernel-exp. *https://github.com/cyanogen/cm-kernel-exp*, 2011.

[8] T.H. Davenport, D.W. De Long, and M.C. Beers. Successful Knowledge Management Projects. *The Knowledge Management Yearbook 1999-2000*, pages 89–107, 1999.

[9] Brian Fitzgerald. The transformation of open source software. *Mis Quarterly*, 30(3):587–598, 2006.

[10] The Apache Software Foundation. Apache 2.0 License, 2004.

[11] Inc. Free Software Foundation. GNU GENERAL PUBLIC LICENSE, 2007.

[12] Friprogsenteret. Acando satser påfri programvare. *http://vimeo.com/10976342*.

[13] Geeknet Inc. Source Forge. *http://sourceforge.net/*, 2011.

[14] Gartner Group. Gartner use of OSS in 2012, 2008.

[15] Miniwatts Marketing Group. Internet World Stats. *http://www.internetworldstats.com/stats.htm*, 2011.

[16] Carl Gutwin, Reagan Penner, and Kevin Schneider. Group awareness in distributed software development. *Proceedings of the 2004 ACM conference on Computer supported cooperative work - CSCW '04*, page 72, 2004.

[17] Ø yvind Hauge. Adoption of Open Source Software in Software-Intensive Industry. 2010.

[18] By Duncan Haughey. Why Software Projects Fail and How to Make Them Succeed. pages 1–2, 1995.

[19] D Hoover. Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman. 2009.

[20] By Jeff Howe. The Rise of Crowdsourcing. *North*, (14):1–5, 2012.

[21] J Howe. The Rise of Crowdsourcing. *Wired*, (14.06), 2006.

[22] A. Hunt and D. Thomas. *The Pragmatic Programmer*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000.

[23] Ari Jaaksi and Linux World. Building consumer products with open source communities – the Maemo and 770 experiences. *Agenda*, pages 1–17, 2006.

[24] H.K. Klein and M.D. Myers. A Set of Principles for Conducting and Evaluating Interpretive Field Studies in Information Systems. *MIS quarterly*, 23(1):67–93, 1999.

[25] J. Kontio. A case study in applying a systematic method for COTS selection. *Proceedings of IEEE 18th International Conference on Software Engineering*, pages 201–209.

[26] S. Krishnamurthy. Cave or Community?: An Empirical Examination of 100 Mature Open Source Projects. *First Monday*, 7(6):20–56, 2002.

[27] K Leung. On the efficiency of domain-based COTS product selection method. *Information and Software Technology*, 44(12):703–715, September 2002.

[28] Jingyue Li, Reidar Conradi, Christian Bunse, Marco Torchiano, Odd Petter N. Slyngstad, and Maurizio Morisio. Development with Off-the-Shelf Components: 10 Facts. *IEEE Software*, 26(2):80–87, March 2009.

[29] Jingyue Li, Reidar Conradi, Odd Petter Slyngstad, Marco Torchiano, Maurizio Morisio, and Christian Bunse. A State-of-the-Practice Survey of Risk Management in Development with Off-the-Shelf Software Components. *IEEE Transactions on Software Engineering*, 34(2):271–286, March 2008.

[30] R.C. Martin. Clean Code. *Prentice Hall PTR Upper Saddle River, NJ, USA*, page 448, 2008.

[31] Microsoft. Office Talk. *http://www.officelabs.com/projects/officetalk/Pages/default.aspx*.

[32] Audris Mockus, Roy T Fielding, and James D Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, July 2002.

[33] Mozilla. Firefox. *http://www.mozilla.org/projects/firefox/*, 2011.

[34] Mozilla Foundation. Mozilla. *http://www.mozilla.org/*.

[35] M.D. Myers and M. Newman. The qualitative interview in IS research: Examining the craft. *Information and Organization*, 17(1):2–26, 2007.

[36] Brandon Norick, Justin Krohn, Eben Howard, Ben Welna, and Clemente Izurieta. Effects of the number of developers on code quality in open source software: a case study. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–1. ACM, 2010.

[37] B.J. Oates. *Researching information systems and computing.* Sage Publications Ltd, 2006.

[38] Oracle. Open Office. *http://www.openoffice.org/*, 2011.

[39] Ryan Paul. Android openness withering as Google withholds Honeycomb code. *Ars Technica http://arstechnica.com/open-source/news/2011/03/android-openness-withering-as-google-withhold-honeycomb-code.ars*.

[40] Ryan Paul. Oracle gives up on OpenOffice after community forks the project. *Ars Technica*, (http://arstechnica.com/open-source/news/2011/04/oracle-gives-up-on-ooo-after-community-forks-the-project.ars), 2011.

[41] E. Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.

[42] SP Roger and I Darrel. *Software Engineering A Practitioner's Approach*. 1992.

[43] Erik Rossen. SAP lanserer "Facebook for bedrifter". *digi.no http://www.digi.no/864109/sap-lanserer-%ABfacebook-for-bedrifter%BB*, 2011.

[44] Erik Rossen. Sosiale medier snudde IBM opp ned, 2011.

[45] Di Wu S. Ajila. Empirical study of the effects of open source adoption on software development economics. *The Journal of Systems and Software*, (80), 2006.

[46] D.C. Schmidt. Why software reuse has failed and how to make it work for you. *C++ Report*, 11(1), 1999.

[47] Open Source Advisery Service. Open Source Advisery Service, 2010.

[48] Stack Exchange Inc. Stack Overflow. *http://stackoverflow.com/*.

[49] Standish Chaos Reports. Software Project Failure Costs Billions.. Better Estimation & Planning Can Help, 2008.

[50] The Apache Software Foundation. Apache HTTP Server. *http://httpd.apache.org/*, 2011.

[51] The Open Handset Alliance. Android Open Source Project. *http://source.android.com/*.

[52] Dave Thomas. Code Kata–How It Started. *http://codekata.pragprog.com/codekata/2007/01/code_katahow_it.html*.

[53] U.S Census Bureau. 2010 Census. *http://2010.census.gov/2010census/*, 2010.

[54] J. Voas. COTS software: the economical choice? *IEEE Software*, 15(2):16–19, 1998.

[55] G Walsham. Knowledge Management:The Benefits and Limitations of Computer Systems. *European Management Journal*, 19(6):599–608, December 2001.

[56] G.M. Weinberg. *The psychology of computer programming*, volume 932633420. Van Nostrand Reinhold, 1971.

[57] D.A. Wheeler. Why open source software/free software (OSS/FS, FLOSS, or FOSS)? Look at the numbers. *David A. Wheeler's Personal Home Page*.

[58] Wikimedia. Wikipedia. *http://www.wikipedia.org/*, 2011.

[59] Windows 7.cc. Microsoft: Windows 7 tool used GPL code. *http://www.windows7.cc/windows-7-tips-tweaks/microsoft-windows-7-tool-used-gpl-code/*.

A Glossary

.Net Framework - Microsoft programming framework

Agile software development - A conceptual framework for software engineering that promotes development iterations throughout the life-cycle of the project

Azure Cloud - Microsoft cloud computing plattform

Black box testing - Testing on a non-transparent and unknown system

CMS - Content management system

COTS - Commercial off-the-shelf

CVS - Concurrent Versions System

Cloud computing - A technology used to access services offered on the Internet cloud

DBMS - Database management system

ERP - Enterprise resource planning

GIT - Distributed revision control system

GUI - Graphical user interface

IntelliJ - integrated development environment for Java

Iterative software development -

LGPL - Open source license

MS SQL - Microsoft SQL Database

MySQL - Open source DBMS

OSS - Open source software

PDF - Portable document format

Python - Interpreted programming language

RUP - Rational unified process

Ruby - Interpreted programming language

SOA - Software oriented architecture

SVN - Revision control system

Scrum - Incremental framework for agile software development

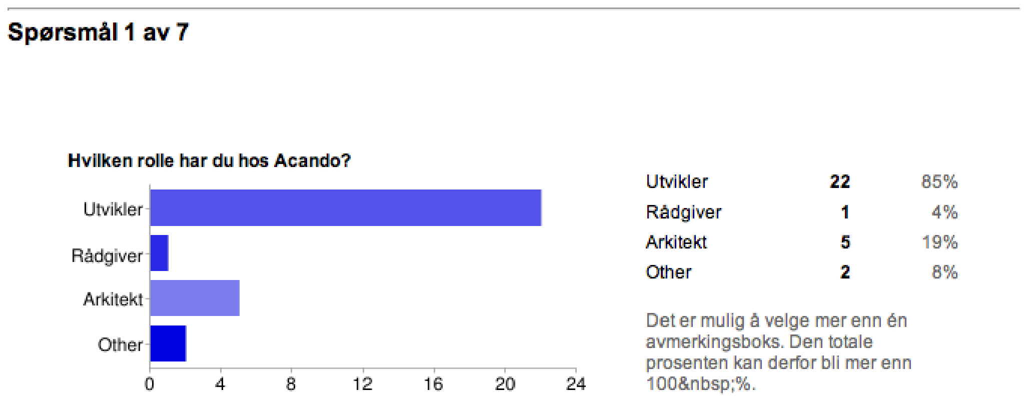
Sharepoint - Microsoft intranet solution

Spiral model - Software development process

Twitter - Microblogging service

Waterfall - Software development process

B Survey



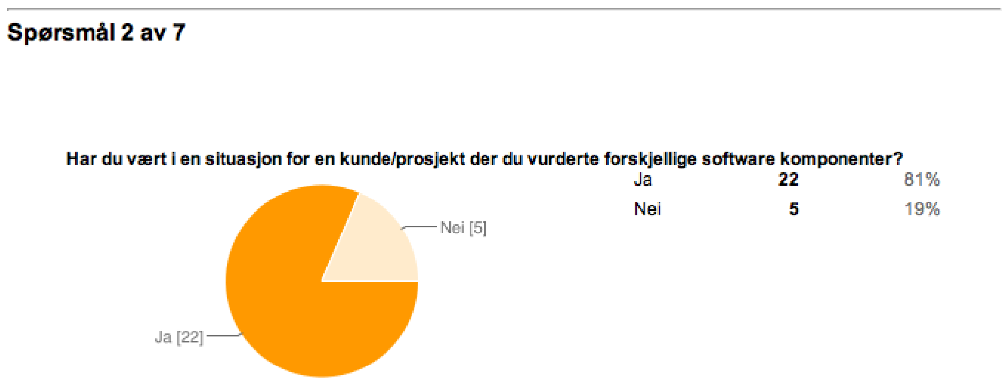
Spørsmål 1 av 7

Hvilken rolle har du hos Acando?

Utvikler	**22**	85%
Rådgiver	**1**	4%
Arkitekt	**5**	19%
Other	**2**	8%

Det er mulig å velge mer enn én avmerkingsboks. Den totale prosenten kan derfor bli mer enn 100 %.

(a) Question 1



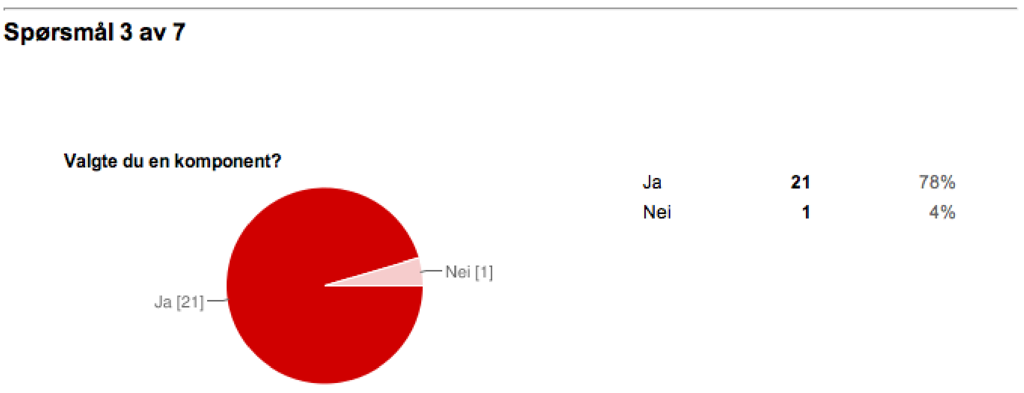
Spørsmål 2 av 7

Har du vært i en situasjon for en kunde/prosjekt der du vurderte forskjellige software komponenter?

Ja	**22**	81%
Nei	**5**	19%

(b) Question 2

Figure 8: Survey sent to developers at Acando September 2010

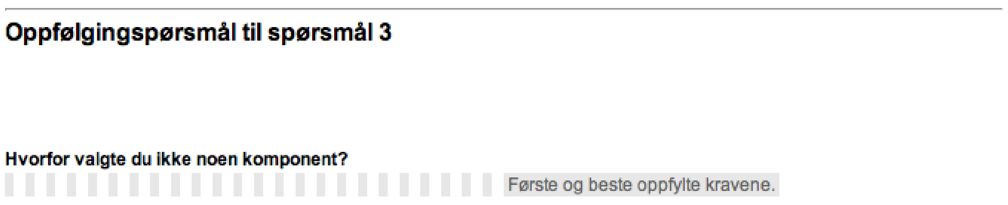


Spørsmål 3 av 7

Valgte du en komponent?

Ja	21	78%
Nei	1	4%

(c) Question 3

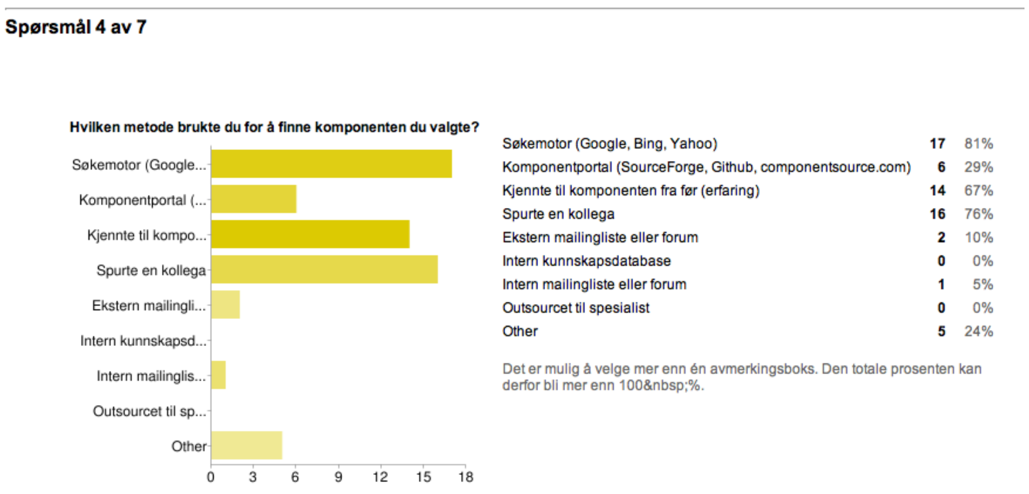


Oppfølgingspørsmål til spørsmål 3

Hvorfor valgte du ikke noen komponent?

Første og beste oppfylte kravene.

(d) Question 3 follow up



Spørsmål 4 av 7

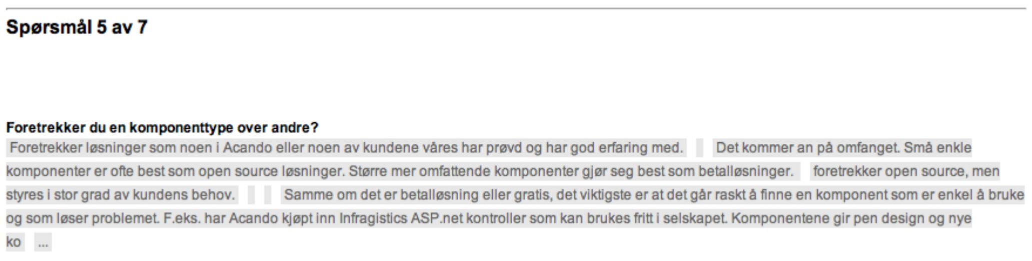
Hvilken metode brukte du for å finne komponenten du valgte?

Søkemotor (Google, Bing, Yahoo)	17	81%
Komponentportal (SourceForge, Github, componentsource.com)	6	29%
Kjennte til komponenten fra før (erfaring)	14	67%
Spurte en kollega	16	76%
Ekstern mailingliste eller forum	2	10%
Intern kunnskapsdatabase	0	0%
Intern mailingliste eller forum	1	5%
Outsourcet til spesialist	0	0%
Other	5	24%

Det er mulig å velge mer enn én avmerkingsboks. Den totale prosenten kan derfor bli mer enn 100 %.

(e) Question 4

Figure 8: Survey sent to developers at Acando September 2010

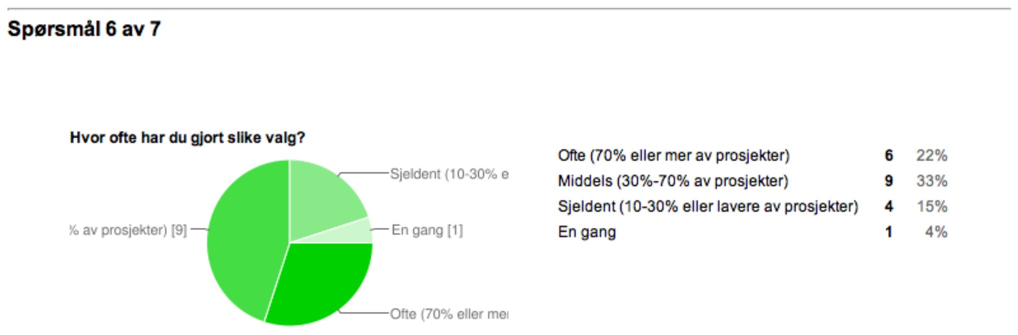


Spørsmål 5 av 7

Foretrekker du en komponenttype over andre?

Foretrekker løsninger som noen i Acando eller noen av kundene våres har prøvd og har god erfaring med. Det kommer an på omfanget. Små enkle komponenter er ofte best som open source løsninger. Større mer omfattende komponenter gjør seg best som betalløsninger. foretrekker open source, men styres i stor grad av kundens behov. Samme om det er betalløsning eller gratis, det viktigste er at det går raskt å finne en komponent som er enkel å bruke og som løser problemet. F.eks. har Acando kjøpt inn Infragistics ASP.net kontroller som kan brukes fritt i selskapet. Komponentene gir pen design og nye ko ...

(f) Question 5

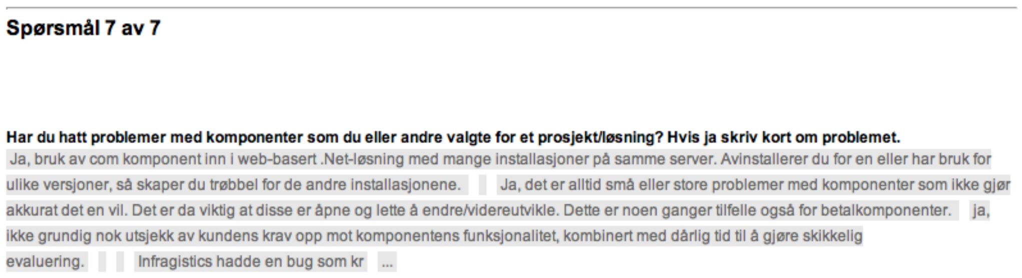


Spørsmål 6 av 7

Hvor ofte har du gjort slike valg?

Ofte (70% eller mer av prosjekter)	6	22%
Middels (30%-70% av prosjekter)	9	33%
Sjeldent (10-30% eller lavere av prosjekter)	4	15%
En gang	1	4%

(g) Question 6



Spørsmål 7 av 7

Har du hatt problemer med komponenter som du eller andre valgte for et prosjekt/løsning? Hvis ja skriv kort om problemet.

Ja, bruk av com komponent inn i web-basert .Net-løsning med mange installasjoner på samme server. Avinstallerer du for en eller har bruk for ulike versjoner, så skaper du trøbbel for de andre installasjonene. Ja, det er alltid små eller store problemer med komponenter som ikke gjør akkurat det en vil. Det er da viktig at disse er åpne og lette å endre/videreutvikle. Dette er noen ganger tilfelle også for betalkomponenter. ja, ikke grundig nok utsjekk av kundens krav opp mot komponentens funksjonalitet, kombinert med dårlig tid til å gjøre skikkelig evaluering. Infragistics hadde en bug som kr ...

(h) Question 7

Figure 8: Survey sent to developers at Acando September 2010

C Problem

Masteroppgave med Acando - Utvelgelse av tredjeparts software komponenter

Problemstilling

Å bruke ferdige softwarekomponenter til IT prosjekter blir mer og mer vanlig. Disse kommer fra store kjente leverandører, mindre kjente kommersielle komponenter eller open source komponenter med eller uten tilknytning til kommersielle firmaer. De større firmaene har man som regel god kunnskap om, men når det kommer til mindre kjente firmaer og komponenter så blir det raskt vanskeligere å velge riktig komponent. Denne problemstillingen eksisterer i dag og ser ut til å stadig øke.

Det er i denne sammenhengen interessant å se på hvordan slike komponenter utvelges. Det kan være meget problematisk å velge feil komponent, det være seg et firma som ikke holder vann på avtaler eller open source komponenter med lite dokumentasjon og uforståelig kode. Det vil være stadig viktigere å utføre en god prosess når det gjelder å velge slike komponenter med tanke på anbud, utvikling og vedlikehold.

Mål

Med problemstillingen å velge ukjente softwarekomponenter vil jeg ha som mål å lage en guide med tanke på å forbedre og kvalitetssikre utvelgelsesprosessen. Det er et viktig poeng at denne guiden skal være så enkel å implementere som mulig med tanke på lave kostnader for Acando. Å se på eksisterende struktur for kompetanseoversikt med muligheter for å utvide disse er en idé som har kommet opp og kan være en mulig løsning. Ordet guide er åpent med tanke på at resultatet av denne oppgaven kan være alt fra en liten sjekkliste til en utvidelse av eksisterende kompetansesystemer. Det er viktig at det skal være enkelt å ta i bruk og eventuelt implementere det som oppgaven måtte resultere i.

Gjennomføring

Oppgaven vil ta utgangspunkt i hvordan utvelgelse av tredjeparts software komponenter foregår i praksis hos Acando i dag. Enten ved å se på utførte prosjekt eller pågående prosjekter. Jeg vil derfor snakke med ansatte hos Acando og utføre intervjuer. Det er viktig at dette arbeidet skal være så lite forstyrrende som mulig for Acando. I første omgang vil jeg være interessert i bli bedre kjent med Acando som bedrift. Jeg vil gjerne besøke bedriften dette semesteret (januar til juni) en til to ganger i måneden, gjerne f.eks på bedriftens innedager om dette er mulig. Derfra vil det gå mot å se nærmere på enkelte prosjekt høsten 2010 - vår 2011. Resultatet av dette arbeidet vil være en masteroppgave samt en rapport og presentasjon for Acando.

Denne oppgaven er en masteroppgave i informatikk innenfor systemering ved IDI fakultet på NTNU, og er utformet på grunnlag av møter og samtaler med **** og **** hos Acando høsten 2009. Selve masteroppgaven skal leveres juni 2011. Studenten er Martin Syvertsen og veileder Eric Monteiro.