Problems with Component-Based Software Development

- A comparison between the theory and practice

A master thesis in software engineering at the Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby

By

Christian Bucanac
Software Engineering Class of 1995

August 1999
Acknowledgements

I would like to thank my supervisor, Per-Olof Bengtsson, for supporting me in my work. He gave me a lot of good advice and ideas while writing this master thesis.

I would like to thank all the personnel at Infocenter [Info] for helping me in acquiring all the literature that I needed for this master thesis. I would also like to thank all the companies that accepted my interview invitation. Without their help and participation, this master thesis would not have been possible to write.

I would like to thank my fellow software engineering classmates, Andreas Folkler and Claes Månsson for reviewing this master thesis. Without their comments and suggestions, this master thesis would not be of as high quality as it is.

Ronneby, August 1999

Christian Bucanac
c.bucanac@computer.org

Abstract

Problems with component-based software development are identified in this master thesis. Both theoretical and practical problems are identified. A literature study was done in order to identify the theoretical problems, while interviewing different companies that develop component-based software identified the practical problems. The identified problems are described together with references to the literature the problems were found in and what companies identify the problems. A comparison of the identified problems is made from the practical problems' side of view. The practical problems that are also identified by the theoretical problems are compared to see how well they match. Practical problems that do not have any correspondence in the theoretical problems are identified and an explanation is given of why they may not have been identified by the theoretical problems.

Keywords:
Component-based software development, theoretical problems, practical problems, literature study, interviews, comparison.
# Table of Contents

1 Introduction.................................................................................................................. 1
2 Research method............................................................................................................... 2
  2.1 Literature study ............................................................................................................ 2
  2.2 Industry interviews ......................................................................................................... 2
  2.3 The comparison ............................................................................................................ 3
3 Problems found in literature........................................................................................... 5
  3.1 Vocabulary .................................................................................................................. 5
  3.2 Loss of control ............................................................................................................. 5
  3.3 Lack of information ....................................................................................................... 6
  3.4 Security .................................................................................................................... 6
  3.5 Trustworthiness ........................................................................................................... 7
  3.6 Selection problems ...................................................................................................... 7
  3.7 Component adaptation ................................................................................................. 8
  3.8 Component composition ............................................................................................... 9
  3.9 Component interoperation ............................................................................................. 9
  3.10 Component mismatches ............................................................................................. 9
  3.11 Testing problems ..................................................................................................... 10
  3.12 Component-based system failure and debugging ..................................................... 11
  3.13 Maintenance problems ............................................................................................. 11
4 Problems found in industry............................................................................................ 13
  4.1 Organizational problems [EST99, Mas99] ................................................................. 13
  4.2 Reusability [Flu99, Mas99, Sym99] ............................................................................ 13
  4.3 Quality [Flu99, Mas99, Sym99] .................................................................................. 13
  4.4 Documentation problems [EST99, Flu99, Mas99, Sym99] ........................................... 14
  4.5 Black box components [Flu99, Mas99] ....................................................................... 14
  4.6 Selection problems [Flu99, Mas99, Sym99] ............................................................... 15
  4.7 Adaptation problems [Mas99, Sym99] ...................................................................... 15
  4.8 Integration problems [EST99, Flu99, Mas99, Sym99] ............................................... 15
  4.9 Maintenance problems [EST99, Flu99] .................................................................... 16
5 A comparison between the problems found in literature and industry ......................... 17
  5.1 Differences between the problems found in literature and industry ......................... 17
  5.2 Problems found both in literature and industry ......................................................... 17
  5.3 Industry problems that were not found in literature ................................................. 21
6 Conclusion .................................................................................................................... 22
7 Glossary ...................................................................................................................... 23
8 Appendix...................................................................................................................... 24
  8.1 Appendix A ................................................................................................................ 24
  8.2 Appendix B ................................................................................................................. 26
9 References.................................................................................................................... 41
10 Bibliography ................................................................................................................. 47
  10.1 Read literature ......................................................................................................... 47
  10.2 Unread literature ........................................................................................................ 50
Introduction

Problems with component-based software development are identified in this master thesis. Both problems found in literature and in industry are identified. Problems found in literature are considered to be theoretical problems, while the problems found in industry are considered to be practical problems. A comparison is also made in this thesis between the theoretical and practical problems. The purpose of the identification and comparison of the problems is that it will guide researchers and practitioners in finding solutions to these problems.

A literature study was done to identify the theoretical problems and interviews with different companies were performed to identify the practical problems. The identified problems were analyzed and a comparison was made from the practical problems' side of view that identifies the differences between the practical and theoretical problems.

The identified theoretical and practical problems are described in sections 3 and 4. There are thirteen theoretical and nine practical problems. They are listed and described in each section together with references to literature the problems were found in and what companies identify the problems.

The practical and theoretical problems are compared mainly in two ways. The practical problems that are also identified by the theoretical problems are compared to see how well they match. Practical problems that do not have any correspondence in the theoretical problems are identified and an explanation is given of why they may not have been identified by the theoretical problems.

There is a glossary in section 7 that explains what some terms, used in this master thesis, mean. Hopefully will the explanation of these terms help in the understanding of this master thesis.
2 Research method

2.1 Literature study

The search of literature about component-based software development was done in January of 1999. The literature search was done with help of the search tools Compendex [Com], INSPEC [INS], Computer Society Digital Library [CSDL], SEI [SEI] and the internet search engine Altavista [Alt]. Copies of literature that could not be found in electronic form were ordered via Infocenter [Info].

All abstracts and introductions of the found literature were read and the literature was prioritized. The prioritization had to be done because there was not enough time to read through all the literature. The majority of the literature discusses other aspects of component-based software development than just the problems. The literature was critically read in order to identify the theoretical problems, which are described in section 3. About two thirds, counted in pages, of all the found literature was read. The reference and bibliography sections list what literature was read and not read.

2.2 Industry interviews

The interview candidates were selected from the software companies located at Soft Center [SC] in Ronneby. All the companies web sites were visited in order to try to get a view of their business. Six companies stated on their web site that they developed software with help of components. All of them were invited for an interview. Two of them declined. The four companies that accepted the interview invitation were:

- Ericsson Software Technology
- Flux Software Engineering
- Massive Entertainment
- Symbian

Even though a small number of companies were interviewed, they represent a wide range of software engineering companies, from small local companies to large multinational companies. It was tried to identify a wide range of practical problems instead of trying to identify as many practical problems as possible. There is no assurance that the identified problems exist or may apply to other companies, which were not interviewed.

2.2.1 About the interviews

The interviews were conducted at the companies' offices. The questions were composed with help of the literature that was read. They may be found in Appendix A. All questions were not relevant for every company. The way the companies develop software with components varies. While interviewing, only questions that were applicable to a company's situation were asked. The interviewees were also given some time to brainstorm other problems that they have with component-based software development, which could not be identified by the questions.

The interviews were taped which made it possible to speed them up considerably. Not having to take notes made it possible to concentrate better on the interview. There were almost no distractions during the interviews.
Problems with Component-Based Software Development

Research method

The interviews may be found in Appendix B. The interviewees have checked the interviews to ensure that they are accurate and that no company secrets or other sensitive things have been revealed. The interviews do not just describe the problems in the industry, but also general aspects about how the companies develop systems with help of components. This helps better to understand the problems that the companies have. The interviews were carefully read in order to identify the practical problems. These problems are described in section 4.

2.2.2 About the interviewees

Ericsson Software Technology

Ericsson Software Technology is one of the leading software companies within the telecom industry [EST]. Ericsson Software Technology employs about 850 people and is located both in Karlskrona and Ronneby. They develop products and services with the latest technology within mobile telephony and enterprise communications.

The interview [EST99] was conducted with Johan Forsman and Christer Åkesson. They are two software developers at Ericsson Software Technology. They work with the Prepaid Service. The interviewed persons have approximately two and a half years of experience of component-based software development.

Flux Software Engineering

The interview [Flu99] was conducted with Young Fogelström. He is the CEO of Flux Software Engineering. Flux Software Engineering is doing consultancy in software engineering and conducts project assignments [BucVal99]. The company was started in the summer of 1997. It is a pretty young and small company with only five employees.

Massive Entertainment

The interview [Mas99] was conducted with Martin Walfisz. He is the CEO of Massive Entertainment. Massive Entertainment is a game development company. It was founded in 1997 and currently has 22 employees [Mas]. Their goal is to become one of the most respected game developers in Europe. Their mission is to create spectacular entertainment products combining state of the art technology with intriguing and challenging game play.

Symbian

The interview [Sym99] was conducted with Peter Molin, the CEO of Symbian in Ronneby, Sweden. Symbian is owned by the world's leading mobile device manufactures, that is Ericsson, Matsushita, Motorola, Nokia and Psion [Sym]. Symbian has offices in Japan, Sweden, United Kingdom and USA. Altogether there are about 300 employees in Symbian. Symbian's mission is to set the standard for mobile wireless operating systems and to enable a mass market for mobile wireless devices [Sym]. Symbian develops an operating system called EPOC for mobile wireless devices.

2.3 The comparison

The problems found in literature and industry were analyzed. It showed that there were some general differences between the problems found in literature and industry. These differences are identified and described in section 5.1.
Problems with Component-Based Software Development
Research method

The problems found in industry are considered to be more important than the problems found in literature. The problems found in literature are theoretical problems that may not exist in practice, while the industry problems are real and currently exist in practice. This does not mean that the theoretical problems have no value or are unimportant. They may exist in other companies that were not interviewed or they may become real in the future for the interviewed companies. With this in mind, it was decided to do the comparison from the practical problems', that is problems found in industry, side of view.

The practical problems that are also identified by the theoretical problems are compared in section 5.2. It is done to see how well they match. Practical problems that do not have any correspondence in the theoretical problems are identified in section 5.3. An explanation is given of why these problems may not have been identified by the theoretical problems. While comparing the problems in sections 5.2 and 5.3 it is stated exactly what is compared. The problems found in literature and industry are not atomic enough so that they can be directly compared. For example, both the literature and industry identify the not-invented-here syndrome as a problem. The literature identifies it as a trustworthiness problem, while the industry identifies it as an organizational problem.

It is important to keep in mind the validity of the comparison. All theoretical and practical problems have not been identified. Only about two thirds of the found literature was read and only four companies were interviewed. The comparison made in section 5 is only valid within the boundaries of the read literature and the interviewed companies.
3 Problems found in literature

Problems that were found in the literature are:

- Vocabulary
- Loss of control
- Lack of information
- Security
- Trustworthiness
- Selection problems
- Component adaptation
- Component composition
- Component interoperability
- Component mismatches
- Testing problems
- Component-based system failure and debugging
- Maintenance problems

These problems are described in the following sections.

3.1 Vocabulary

There is much more talk than understanding in component-based software development [KozBoo98, ThoMan96]. The term "component" is a widely used buzzword with many possible interpretations [CarObe98]. Despite that there are precise definitions of a component, it is still possible for significant misunderstandings to arise between the component developer and component user. Often the same term means different things and is defined differently by different people [ThoMan96].

3.2 Loss of control

Using externally developed components as opposed to internally developed components leads to loss of control [Obe98]. The component users are at the mercy of component developers [Voa98a]. If the system fails because a faulty component, the customer will blame the component user, not the developer. There is no control over a component's functionality or performance [BoeAbt99]. There is a limited access to the components' internal design, predefined options for customizing the components' behavior, no ability to influence the release cycle of new component versions and total reliance on the long-term viability, integrity and ability of the component developers [BroWal98a]. This affects many aspects of the design, assembly, testing, deployment and maintenance of a component-based system.

The component user loses certain amount of control and acquires additional dependencies when using externally developed components [HCF97]. The effect of having multiple dependencies, that is a number of externally developed components and different component developers, may cause short-term inconvenience or the instability of the total system [Car97]. What capabilities a component will have depend on the market forces and the component developer's objectives in the market [Obe98]. Over time the functionality provided by a component will not remain precisely what is needed [Voa98d]. It may not be upwardly compatible because old component releases become obsolete and unsupported by the component developer [BoeAbt99]. The capabilities that a specific component user needs may not be the highest priority for the component developer. If the changes or fixes do not suit a specific component user's system, it may become incompatible with the system [Voa98c].
Problems with Component-Based Software Development

Problems found in literature

The development of a component may be discontinued [CarObe98, Obe98] and the component's functionality may become frozen [Voa98c]. The component developer may go out of business or decide to stop developing a component. This results in the discontinuation of a component that a system depends on [McD98] and leads to a large impact on the maintenance of a system. The component has to be maintained by the component user, who has not written the source code of the component [Voa98d].

There is a danger of a system becoming overly dependent on a particularly externally developed component [CarObe98]. This may occur when a component has a number of desirable but non-standard features. These features are often attractive, but the price to pay is that the system will rely on them and it may result in a vendor lock. The more non-standard features are used, the less portable the system becomes [OBMS97].

3.3 Lack of information

The lack of information about the externally developed components leads to major problems in the development and maintenance of component-based systems [Pou98]. The components should not only meet the system's functional and quality requirements but also be compatible with other components in the system. There are many undocumented system and environmental assumptions made by the components [OBMS97]. These often lead to conflicts when multiple components are integrated.

An externally developed component may be extensively documented, but it does not mean that it is sufficient [CarObe98]. The documentation of the externally developed components may be insufficient or incorrect [LinJon98]. The component developer might not provide the component user with the documentation needed to correctly and securely integrate a component into a system. Information about a component's design assumptions and quality attributes are not always available for the component users [Pou98]. The documentation is often not well written and accurate [CarObe98].

Often the externally developed components are delivered as black boxes, where the only information available is their interfaces [Pou98]. There are restrictions that keep the component users from looking inside the black box component [Voa98a]. The component's licensing forbid decompilation back to source code [Voa98b].

3.4 Security

Using externally developed components in a system may introduce significant security risks [ZhoEdw98]. Inspecting externally developed components, for malicious code or bugs, is infeasible. The source code is often unavailable. Even if it is available it may be too costly to inspect it. Component users have to anticipate security problems and have a strategy for dealing with them [LinJon98]. This is important because component users often do not have any control over the externally developed components.

Any externally developed component may have an impact on the overall system security [LinJon98]. There is a question of what level of security one can attain by composing a system of different components. There might be mismatches between the component security levels. The common problem when integrating different components is that the security level has to be set to the lowest common denominator to make the components work together.
Problems with Component-Based Software Development

Problems found in literature

Some of the security risks originate from the design of the externally developed components and are beyond the control of the component user [LinJon98]. The component design may be intentionally flawed. It may contain intentional security flaws as backdoors, viruses or Trojan horses [Voa98b, Voa98d]. This is almost never discovered until it is too late. The component may have many more features than the component user knows about. The component user may not know the true security implications of including the component into a system.

Many externally developed components that provide the required functionality may not provide the necessary security [OBMS97]. The question is how data is handled by a component. High security data must be encrypted and secure. There might be insufficient understanding of a component's functionality [LinJon98]. For example, the component users may not know if a component transmit passwords without encryption over the network. Unexpected behavior of a component may pose security threats [ZhoEdw98]. The component may access unauthorized resources and services or it may access a resource in an unauthorized way.

Building wrappers to control a component is not foolproof [Voa98b]. Illegal outputs can sometimes bypass a wrapper. For example, if it is unknown that a component calls the operating system to delete a file, the wrapper will not be designed to control it.

3.5 Trustworthiness

Considering that most externally developed components are black boxes [Pou98], the question is if it is possible to trust them to be secure and reliable [ZhoEdw98]. Components are selected on the basis of their credentials, that is, published specifications of their capabilities and constraints [Del98]. The specifications are often incomplete and imprecise descriptions of a component.

Most component users are suspicious of externally developed components [Voa98d]. This is called the not-invented-here syndrome, which encourages many component users to develop components from scratch rather than using externally developed components.

3.6 Selection problems

In the traditional development model, like the waterfall model, the requirements are specified according to the systems requirements. In component-based software development the components often determine the requirements on the system [BoeAbt99]. The component selection often tends to take place early in a project where the system requirements may be fuzzy [Kon96]. At the same time, the technical characteristics of the components are very concrete. This leads to a bottom-up approach [MMM95] of the component selection, because it is much easier to evaluate what you happen to see than taking the effort to determine and search for important characteristics. Having a bias toward a component worsens this problem [CarObe98, MaiNcu98]. It is a very difficult problem because the properties of a system are determined by the components that it is composed of [OBMS97].

Identifying externally developed components requires extensive effort and time spent in attending conferences, reviewing literature, training, traveling and communicating with component developers [Tra98]. Selecting an inappropriate component can result in an enormous amount of extra time and effort to re-evaluate and re-implement the system with another component. The component market is changing fast [Bri98]. This means that the component evaluation is not just done once, but several times during the lifetime of a system.
3.6.1 Selection criteria
The large number of selection or evaluation criteria makes it very difficult for the human mind to make an objective and unbiased decision [Bri98, MaiNcu98]. When the selection criteria is defined, it is seldom well defined [Kon96, MMM95]. The criteria may be listed and documented with one or two words, leaving the exact meaning of each criterion very open to subjective interpretations. The criteria may not be detailed enough to enable an effective selection of components [MaiNcu98]. The documentation of the components is often not sufficient enough. There is sometimes a need to ask the component developers detailed questions about the components. Gathering relevant information from the component developers may be prohibitively expensive [Bri98].

It may be difficult to measure how well a component meets a certain criteria and lead to difficulty in selecting a component [MaiNcu98]. This is especially true if there is no prior knowledge about the components or prior extensive experience of creating measurable test cases.

Evaluating non-functional qualities of a component is hard because of incomplete understanding of them and inadequate tools and techniques for determining if they meet the expectations [OBMS97]. It is hard to determine the reliability, security and performance of a component, yet these non-functional requirements have huge impact on the overall system.

3.6.2 Selection process
The selection process is often not defined [Kon96] and there is no "one-size-fits-all" selection process available [OBMS97]. It is reinvented each time and is performed inconsistently, which makes the reuse of experience from previous selections difficult. Considering that the selection criteria is not always well defined [Kon96, MMM95] leads to that it may be interpreted in many ways, which results in inaccuracy and ambiguity in the selection process.

There is a lack of qualified evaluators [OBMS97]. There is often lack of sufficient time and resources for performing the selection process. It is often done under pressure and the evaluators are often first timers to the task [Kon96]. They may not have time to or experience to plan the selection process in detail. Using experienced evaluators is not good either [OBMS97]. The evaluation may succeed but the integration of the components may not succeed if average people perform it.

Selecting a component is a very complex activity [MMM95]. Not only the qualities of competing components must be considered [SEI], but also whether the technologies on which the components are based are sufficiently mature for general use and whether the technologies are likely to remain viable over the lifetime of the system.

3.7 Component adaptation
Externally developed components are seldom built to plug into each other easily [Car97, CarObe98, Voa98d]. There is a lack of commonality between the externally developed components [Obe98]. The components have different interfaces and properties that are not common with other components or systems. This results in problems of composing the components. There might be a need for extra glue in order to make them work together, which may require rather complex plumbing [SLP98]. The harder it is to put the components together, the higher is the impedance among them. The usual way to overcome this problem is to adapt the components by building wrappers, adapters or changing the component.
Problems with Component-Based Software Development

Problems found in literature

A wrapper is a type of middleware that limits a component's functionality [Voa98c]. Writing wrappers can be a complex activity [McD98] that requires expertise both at the detailed system level and in the component being wrapped. The problem is to know which behaviors of the component to protect against, especially when you do not know how a component behaves [Voa98c]. The component user rarely gets a specification of a component that makes it possible to write good enough wrappers [McD98]. Writing wrappers is a point-to-point solution which means that when a new version of a component is released they will have to be rewritten [CarObe98]. The net result may be that this can increase rather than decrease the system's overall cost when components are used to build systems.

A component may be adapted by changing it. The changes are often unanticipated or poorly planned [MMM95]. Changing reusable components may defeat both the quality and the productivity advantages of using components.

3.8 Component composition

Component composition is about putting components together into larger whole usually without changing the components themselves and often via some rules or mechanisms of composition [ThoMan96]. The result might yield a special purpose composition or a general purpose one.

The component interaction specification may be very complex [MMM95]. It may vary from simple call dependencies to full-fledged behavioral specification including interaction logic, preconditions, postconditions, invariants, etc. The more dependencies the greater the coupling between the components [VigDea98]. The greater coupling between the components, the more difficult it is to compose the components. This problem applies especially in the maintenance phase when a component replaces another one in the system. The testing effort for securing the system is significant, since there are many ways the other components in the system may be affected.

3.9 Component interoperation

Even if it is possible to compose components into a system, it may not be likely that they will be able to interoperate [CarObe98]. Most externally developed components are not designed to interoperate with each other [BoeAbt99]. There is never any assurance that a component will fit smoothly into a system without causing any problems [Voa98b]. The components might not behave perfectly at runtime [Voa98a].

Component interoperation is considered to be a design problem because there is a lack of expressive means for representing interdependencies between the components [Del97]. [GAO95] provides a case study and explanations for why component interoperation problems may cause large cost. This case study is described in section 3.10.

3.10 Component mismatches

There is a paper [GAO95], that is referenced by many authors [BoeAbt99, BroWal98a, BroWal98b, MaiNcu98, Tra98, Wey98], which describes what component mismatches can appear when developing a system out of externally developed components. These mismatches and their causes are described in the following section.
3.10.1 An example of component mismatches [GAO95]

A system called Aesop [GAO94] was built from externally developed components. The system assists the users in constraining their design alternatives and verifying the correctness of their designs. It took about five person years to build the first working prototype of the system. The system was huge, even though the component users contributed with a relatively small part of own code. The performance was poor and the maintenance of the system became impossible without detailed knowledge of the system's implementation.

The problems encountered can be summarized as architectural mismatches. Some of them were:

- Too much code - The size of the system was huge. A lot of the functionality provided by the components was not needed, but it was there and contributing to the size of the system.
- Poor performance - The system did not fulfill its performance requirements. The code size of the components was one of the reasons for this problem. Another reason was too much overhead in the communication between the components.
- Component changes - The components were tested to see if they could be composed. The first tests showed that it was possible to compose them without any changes. When the whole system was composed, it showed that the components could not interoperate. Significant component changes were needed to make them interoperable.
- Re-implementation - Changing the components was in some cases not enough. Some of the functionality did not work as required. It had to be re-implemented to suit the components that were using it.

The cause of the problems was that various assumptions were made by the components. These assumptions were not clearly documented and could therefore not have been dealt with. Some of the assumptions were:

- The components assumed that they were responsible for the execution control. Every component provided a main thread of control that was not compatible with other components' execution control.
- Each component made assumptions about the structure of the system that it was to be integrated into. They made assumptions about how the overall system would be designed and what components it would include.
- Every component assumed that it was supposed to provide the infrastructure for the system. Every component's infrastructure was not needed, which contributed to excessive code and made the system huge.
- The database, that was used, assumed that the communication would form a star topology with the database in the center. It assumed that all components would be completely independent of each other and that there would not be any direct communication between the components.

3.11 Testing problems

Externally developed components are often black box components [Pou98]. They have to be tested without opening the black box [OBMS97]. Current black box testing techniques are not sufficient. It is very hard to determine what types of testing are possible or necessary [Car97]. Without the source code and access to the personnel and expertise used to create the component, testing and debugging is significantly difficult [Voa98b, Wey98]. The component users have to generate inputs, create an oracle that decides the result and build a test driver. All these tasks are expensive.
Problems with Component-Based Software Development
Problems found in literature

3.11.1  A real world example - Ariane 5
The Ariane 5 Flight 501 failure report [Lio96] starts with “On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded.” The Inquiry Board determined that the explosion resulted from insufficiently tested software reused from the Ariane 4 system in the Ariane 5 system without substantial re-testing [Wey98]. It was assumed that there were no significant differences in the reused portions of the two systems.

The Inquiry Board recommendation number two [Lio96] for preventing the disaster in future flights says “Prepare a test facility including as much real equipment as technically feasible, inject realistic input data, and perform complete, closed-loop, system testing. Complete simulations must take place before any mission. A high test coverage has to be obtained.” This real world example shows how important it is to test previously developed components before they are used in a new system [CarObe98, McD98].

3.12  Component-based system failure and debugging
When a component-based system fails it may be hard to identify which component failed [His98, OBMS97]. The first hope is that the component developer will provide some assistance to solve the problem [His98]. This is often a remote possibility. There is a high probability that the component developer will argue that the fault lies in another component than the developer's. This leads nowhere else than to finger pointing and forces the component user to debug the system.

The debugging of a system with externally developed components is different than for a system where all the source code is available [His98]. The component user often does not have any visibility into the component and no control of its workings. The available information is often incomplete and often inconsistent. If the failure was caused by one component, the question is which one? If the failure was caused by an interaction of components, the question is which pair? The difficulty of isolating the fault stems from difficulty of finding out how the components work and then finding out why they do not work.

The debugging may be done with help of the scientific method of observation, hypothesis and experimentation described in [His98]. A hypothesis is formed on one or more observations. To test the hypothesis, a prediction is made on the basis of some input. The experiment is performed and it either supports or contradicts the hypothesis. If the hypothesis is not supported, a new iteration of the method is made with refinements of the hypothesis. The problem of using this method is that it may require a lot of iterations to find the fault [His98]. Even if the fault is found, it may not be much of help. The fault can often not be fixed in an externally developed component [OBMS97]. Waiting for the faults to be fixed by the component developer can often result in significant delays and down time.

3.13  Maintenance problems
Maintaining a component-based system means that a system incorporates new releases of components [CarObe98]. Using externally developed components may be cheap in the first place, but it may become costly when the system needs to be upgraded [McD98, Obe98]. The component developers frequently upgrade their components based on error reports, market needs and competition [HCF97, Obe98]. The frequent changes result in a constant change for the system that uses the components [Obe98]. This may lead to unanticipated and unpredictable changes to the system.
Problems with Component-Based Software Development

Problems found in literature

Component developers often tend to only support a limited number of versions of a component [CarObe98]. Ignoring the component developer's releases may lead to that the system can not survive in the long-term. The technology that the system was built with may have become out-of-dated and not supported by the latest versions of a component. It is therefore preferable to keep the system up to date with the components that are currently supported by the component developer.

A system with several externally developed components has a heavy dependence on the various release cycles of the component developers [Car97, CarObe98]. Different components in the system are upgraded at widely varying intervals. Component upgrades are often not done in a simple plug and play approach [HCF97]. Keeping the glue code current and up to date for a system of any complexity can become a maintenance nightmare [CarObe98].

Replacing a component often has rippling affects throughout the system [HCF97], especially when many of the components in the system are black box components. The component user does not know how a component is built and how it will act in an interdependent environment. Component upgrades can result in numerous unforeseen problems [Car97]. The upgrade of a component with a new version often requires enhanced version of other components [HCF97].
4 Problems found in industry

Problems that were found in the industry are:

- Organizational problems
- Reusability
- Quality
- Documentation
- Black box components
- Selection problems
- Adaptation problems
- Integration problems
- Maintenance problems

These problems are described in the following sections.

4.1 Organizational problems [EST99, Mas99]

In large companies the development can be geographically distributed throughout many different offices [EST99]. The larger distance between the offices, the harder it is to keep close contact with other developers. This makes it harder to coordinate the development. It is especially hard to coordinate changes in a component's interface.

There are some problems with the not-invented-here syndrome [Mas99]. The reason is that the developers are hostile to code that they have not invented, until they get to a certain level of insight in component-based software development.

4.2 Reusability [Flu99, Mas99, Sym99]

Reusability of components does not come for free [Flu99, Mas99, Sym99]. The components have to be designed and developed to be reusable. It costs to make the components reusable. Both [Flu99] and [Mas99] believe that it pays off later when the components are reused. None of them have any concrete metrics to show how much it pays off. Seeing the benefits already in the maintenance phase strengthens their belief of making the components reusable [Flu99]. [Mas99] expects the time invested in making the components in their first game reusable will be paid off manifold when they start to develop their next game.

There is a conceptual problem of deciding where the border goes between a general and specialized component [Mas99]. A component is almost never as general as possible so that it easily can be reused in other systems. There is always a need for modifying a component to make it possible to use it in another system.

4.3 Quality [Flu99, Mas99, Sym99]

Quality does not either come for free. It costs to make a component with high quality. The quality work is not just done now, but also in the future [Mas99]. The more a component has been reused the more it has been tested and the more error fixes have been accumulated. The quality assurance is not done just now, but also in the future.
Problems with Component-Based Software Development

Problems found in industry

The problem with externally developed components is that you never know what quality they have [Sym99]. The quality is often not measurable. The only things you can look at are how much they have been used and how widespread they are. The more they have been used, the higher quality they might have. The majority of the externally developed components are not certified in any way [Sym99]. It may be hard to trust a component because you do not have any guarantee of what they are capable of. Much time is spent on testing the components to see what they can do [Flu99] and how reliable they are [Sym99]. When exchanging a component in a system, regression testing is made to make sure that the earlier implemented functionality still works [Flu99].

4.4 Documentation problems [EST99, Flu99, Mas99, Sym99]

4.4.1 Internally developed components [EST99, Flu99, Mas99, Sym99]

The idea with the component documentation is that another developer should easily be able to understand and start to work with a component faster [EST99]. There is a problem for someone else than the developer to understand a component [Mas99]. The reason for this is that there is a lack of documentation. Experience has shown that it is hard to specify components in a way so that any developer can start to use a specific component [Sym99].

All interviewed companies have different opinions of what needs to be documented about a component. [EST99] says that the functionality of the components is the most important thing to document. They do not believe that documenting calling sequences is so important. [Sym99] believes on the other hand that this is the most important part about a component documentation. [Flu99] says that the design of the components is important to document, while [Mas99] says that the interface of the components is important to document.

There is a strict development process when developing highly security or performance critical systems [EST99]. This development process is often so strict that no exceptions can be made. Everything that the development process says has to be documented must be documented, whether it is necessary or not. Following such strict development processes results in more documentation than source code. Some of the documentation is never read.

4.4.2 Externally developed components [Flu99, Sym99]

Externally developed components are often poorly and too little documented [Flu99, Sym99]. Wrong things about the components are sometimes documented [Flu99]. This makes them much harder to use and understand how they work.

If the externally developed component is a white box component, then the only source of information may be the source code [Sym99]. It is not easy to understand a component by just reading the source code. It takes more time than if the component was well documented with a visual design and explanations in natural language.

4.5 Black box components [Flu99, Mas99]

Using black box components often leads to loss of control [Mas99]. If a black box component has poor performance you can not do anything about it. This is a serious problem for companies that develop high performance systems. [Mas99] have learned from experience that there is a risk of using code that someone else has developed.
Problems with Component-Based Software Development
Problems found in industry

It is very hard to test a black box component, especially if it is large and complex [Flu99]. It is hard to understand how it works and find out if it fulfills the needed requirements. On the other hand, it may be easier to test a small and simple black box component. The fact that the externally developed components are often poorly documented [Flu99, Sym99] does not make the testing easier.

4.6 Selection problems [Flu99, Mas99, Sym99]
None of the interviewed companies have any well-defined process for selecting components. Selecting the right component from the company’s repository is not a problem because the components in there are internally developed. Almost everyone knows what a component is capable of. If they don’t know, then there is always somebody you can ask [Flu99, Mas99, Sym99].

Nowadays the companies do not have so many components in their repositories. In the future it might become harder to select the right component. [Mas99] therefore plans to implement a process for selecting components.

There is a problem of selecting components among the externally developed components. The externally developed components are often poorly documented [Sym99]. This makes it hard to evaluate if any of them suit the particular system. If a component can not be found, the companies are forced to develop the component themselves.

Another problem is that it might be hard to find the right component to use [Flu99]. Several components may match the selection requirements. Which one of them should be used? The components have to be thoroughly examined and tested to determine this. The question is if it pays off doing this.

4.7 Adaptation problems [Mas99, Sym99]
The externally developed components do not always fit in smoothly into a system [Mas99, Sym99]. Adapters or wrappers have to be built for the components or the components themselves need to be changed to fit into the system. Changing a component may result in that the component becomes specialized, so that it only can be used in the environment it was adapted to [Sym99]. [Mas99] says that the components should only be changed if they are not general enough. Components are not changed if they become less general.

4.8 Integration problems [EST99, Flu99, Mas99, Sym99]
The companies solve most of the integration problems in earlier development phases before the integration phase [EST99, Flu99, Mas99, Sym99]. They therefore do not have so many integration problems, but there are some. All kinds of problems have arisen. It is not hundred percent painless to integrate the components. The companies do not want to reveal many of their integration problems.

Developing software with components puts high requirements on the design of the system and its architecture [Flu99]. The architecture of the system must be designed for using components. It is not so easy to do this. A divide and conquer technique is used to handle this problem.
4.9 Maintenance problems [EST99, Flu99]

The maintenance phase is not just about updating the system by exchanging older components with newer ones. The components themselves have to be further developed and updated too [EST99, Flu99], so that they can fulfill future requirements on the system.

A component may grow and it may become hard to get a general view of it [EST99]. It is important to remember not to have too large components. It is very easy to neglect this and do nothing about it. It is hard to get some time over to review the component structure. Maybe it needs to be redesigned or restructured in some way to better support future requirements.

[Flu99] feels that it is not easy, as they would like, to update a system in the maintenance phase. They work on this problem and are trying to improve the composition and interoperation of the components.
5 A comparison between the problems found in literature and industry

5.1 Differences between the problems found in literature and industry

There is a difference in where the components come from. The literature problems assume that the used components are externally developed. On the other hand, the industry problems show that both externally and internally developed components are used. This is one of the reasons to that there are problems in the industry that are not found in the literature. These problems are identified in section 5.3.

The problems found in literature are more technically oriented and detailed than the problems found in industry. Technical and detailed literature problems as described in sections 3.10 and 3.12 are not found among the problems identified in industry. Another example is that in section 4.3 it is said that you never know what quality externally developed components have. While in section 3.4 many quality problems are brought up so detailed as saying that there might be backdoors, viruses and Trojan horses in the components. These kind of differences are described in section 5.2.

5.2 Problems found both in literature and industry

5.2.1 Not-invented-here syndrome

What is compared?
Problems found in Literature | Problems found in Industry
--- | ---
Section 3.5, second paragraph | Section 4.1, second paragraph

The problem described by the industry says that the developers are hostile to code that they have not invented, until they get to a certain level of insight. The literature goes a little bit further by saying that the component users are suspicious of externally developed components and that they would rather develop the components from scratch. The industry does not state that the not-invented-here syndrome leads to that the components are developed from scratch.

5.2.2 Loss of control

What is compared?
Problems found in Literature | Problems found in Industry
--- | ---
Section 3.2, first paragraph | Section 4.5, first paragraph
Section 3.4, third paragraph

Both the literature and industry say that the use of externally developed components leads to loss of control. If there is something wrong with the component, a component user can not do anything about it. As the literature says, the component user is at the mercy of the component developer. Both the literature and industry say that there is no control over the component's performance. The literature goes further and says that there is no control over the functionality either. There might be security flaws as backdoors, viruses or Trojan horses in the components. Not being able to control the performance of a component is a serious problem that the industry has experienced.
5.2.3 Quality

What is compared?
Problems found in Literature: Section 3.5, first paragraph
Problems found in Industry: Section 4.3, first half of the second paragraph

It is stated both by the literature and industry that there is a question of what quality a component has. The literature states the quality in terms of security and reliability, while the industry speaks about quality in general terms. The quality is known on the basis of the component's credentials. For the literature, the credentials are the published capabilities and constraints of a component, while for the industry it is how widespread and used a component is.

5.2.4 Documentation of externally developed black box components

What is compared?
Problems found in Literature: Section 3.3
Problems found in Industry: Section 4.4.2, first paragraph

Both the literature and industry agree upon that the externally developed components are poorly documented. The literature goes a little bit further by describing what can be poorly documented. For example it brings up that there might be undocumented assumptions about the system and environment that the components are to be used in. It also says that the documentation of the components may be insufficient or incorrect. This concurs with the problem stated by the industry that wrong things about the components may be documented. The industry says that this leads to that it is hard to understand and use the components. The literature only says that it may be hard to use the components because conflicts between them may occur and it may be hard to integrate them.

5.2.5 Documentation of externally developed white box components

What is compared?
Problems found in Literature: Section 3.4, first paragraph
Problems found in Industry: Section 4.4.2, second paragraph

The industry says that if an externally developed component is a white box component, then the source code is often the only documentation. It is not easy to understand a component only by reading the source code. It takes much more time to understand it than if it would be well documented with a visual design and explanations in natural language. The literature agrees to this by saying that it may be too costly to inspect the source code.
Problems with Component-Based Software Development
A comparison between the problems found in literature and industry

5.2.6 Selection process

What is compared?
Problems found in Literature Problems found in Industry
Section 3.6.2, first paragraph Section 4.6, first paragraph

The industry says that they do not have any selection process defined. This concurs with the literature that says that the selection process is often not defined. It says that the selection process is reinvented each time and performed inconsistently. This makes it hard to use experience from previous selections. The reuse of experience from previous selections does not seem to be an issue that exists in the industry.

5.2.7 Evaluating externally developed components

What is compared?
Problems found in Literature Problems found in Industry
Section 3.3, first paragraph Section 4.6, first half of the second paragraph
Section 3.6.1, second half of the first and second paragraph

Both the literature and industry agree upon that it may be difficult to evaluate externally developed components. It is hard to evaluate the components because they are poorly and insufficiently documented. There is an incomplete understanding of the components, which makes it hard to determine if they meet the expectations. There might be some assumptions made by the components that make it hard to evaluate if they suit a particular system. The literature looks at the problem from another view. It says that the evaluating criteria may not be well defined, which leaves the exact meaning of each criterion open to subjective interpretations.

5.2.8 Component adaptation

What is compared?
Problems found in Literature Problems found in Industry
Section 3.7 Section 4.7

Both the literature and industry agree that the externally developed components do not always fit smoothly together. They are seldom easily plugged into each other because there is a lack of commonality between them. This leads to problems of composing components which is solved by writing some glue code as adapters, wrappers or by changing the components. The industry only changes the components if they are not general enough. Components are not changed if they become less general. This kind of change does not lead to the defeat of both the quality and productivity advantages of using components as the literature says.
5.2.9 Component integration

What is compared?

<table>
<thead>
<tr>
<th>Problems found in Literature</th>
<th>Problems found in Industry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section 3.9</td>
<td>Section 4.8, first paragraph</td>
</tr>
</tbody>
</table>

They industry says that most integration problems are solved in the earlier development phases, before the integration phase. There are therefore not so many problems in the integration phase. The literature concurs with this by saying the component interoperation is a design problem. The integration problems should be solved in the design phase. The literature assumes that the components are externally developed. It is therefore not possible to solve the interoperation problem in the design phase, because the components are usually not designed to interoperate with each other.

5.2.10 Testing

What is compared?

<table>
<thead>
<tr>
<th>Problems found in Literature</th>
<th>Problems found in Industry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section 3.11, first paragraph</td>
<td>Section 4.3, second half of the second paragraph</td>
</tr>
<tr>
<td></td>
<td>Section 4.5, second paragraph</td>
</tr>
</tbody>
</table>

Both the literature and industry problems say that it is hard to test externally developed components. The industry says it is hard because there is no guarantee of what the components are capable of. The components are not certified in any way and the components are poorly documented. It may be easier to test small and simple components because it may be easier to understand how they work. The literature says it is hard to test externally developed components because there is a lack of testing methods and it is hard to determine what types of testing is possible or necessary. Both the literature and industry conclude that it is expensive to test externally developed components.

5.2.11 Maintenance

What is compared?

<table>
<thead>
<tr>
<th>Problems found in Literature</th>
<th>Problems found in Industry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section 3.13</td>
<td>Section 4.9</td>
</tr>
</tbody>
</table>

The maintenance problem is both identified by the literature and industry, but there are different kinds of maintenance problems. This is mainly due to that the literature assumes that externally developed components are used.
The literature identifies the problem of keeping the system up to date with new frequent component releases. It says that these frequent upgrades result in a constant change for the system. These changes may have rippling affects throughout the system that may lead to unanticipated and unpredictable changes to the system.
The industry identifies maintenance problems that have to do with internally developed components. The internally developed components have to be further developed and updated so that they can fulfill future requirements on the system. The components may also grow and become hard to handle which leads to that the components may be needed to be redesigned or restructured to better support future requirements.
5.3 Industry problems that were not found in literature

5.3.1 The reusability, quality and documentation of internally developed components

Problem location: Section 4.2
Section 4.3, first paragraph
Section 4.4.1

The problems with reusability, quality and documentation of internally developed components are not identified by the literature because the literature assumes that only externally developed components are used in component-based software development.

5.3.2 Coordination

Problem location: Section 4.1, first paragraph

The problem of coordinating the development that is geographically distributed is not identified by the literature because it is not supposed to exist in the literature's view of component-based software development. The components are externally developed and acquired, they are not internally developed.

5.3.3 System architecture

Problem location: Section 4.8, second paragraph

The industry says that developing systems with components puts high requirements on the design of the system. The system architecture must be design for using components. This problem is probably not identified by the literature because it does not apply to externally developed components. It may not be possible to design the architecture of a system that is composed of externally developed components because the externally developed components determine the architecture of the system [BoeAbt99, MMM95].

5.3.4 Several components match the selection requirements

Problem location: Section 4.6, second half of the second paragraph

The industry says that it might be hard to find the right component to use because several components may match the selection requirements. The literature seems to assume that the selection of the components results in that one component is selected. This is the case in the PORE selection model [MaiNeu98]. It assumes that the selection results in one selected component even though the last n numbers of templates that are used in the selection process are currently under development. The question is how the PORE model can assure that the selection results in that one component is selected when the selection model is not finished yet.
6 Conclusion

Problems with component-based software development found in literature and industry have been identified and described in this master thesis. The problems have also been compared from the industrial problems' side of view, because the problems found in industry are considered to be more important than the problems found in literature. In the comparison, the problems found in literature and industry were compared to see how well they match. These problems were:

- Not-invented-here syndrome
- Loss of control
- Quality
- Documentation of externally developed black box components
- Documentation of externally developed white box components
- Selection process
- Evaluating externally developed components
- Component adaptation
- Component integration
- Testing
- Maintenance

Problems found in industry that were not found in literature were also identified and an explanation was given of why they may not have been identified by the literature. These problems were:

- The reusability, quality and documentation of internally developed components
- Coordination
- System architecture
- Several components match selection requirements

There are many more practical problems that match, than do not match, the theoretical problems. A reason for this may be that the majority of the companies are young and that the most interviewees are educated at the University of Karlskrona/Ronneby. This means that they might be aware of the literature's view of component-based software development.

The difference between the problems is that the theoretical problems are more technical and detailed. They better describe a particular problem. An explanation for this might be that the companies are not so happy to talk about their problems, especially describing the problems in detail and their technical nature.

There are many more practical problems that do not match, than do not match, the theoretical problems. A reason for this may be that the majority of the companies are young and that the most interviewees are educated at the University of Karlskrona/Ronneby. This means that they might be aware of the literature's view of component-based software development.

The difference between the problems is that the theoretical problems are more technical and detailed. They better describe a particular problem. An explanation for this might be that the companies are not so happy to talk about their problems, especially describing the problems in detail and their technical nature.

There are many more practical problems that match, than do not match, the theoretical problems. A reason for this may be that the majority of the companies are young and that the most interviewees are educated at the University of Karlskrona/Ronneby. This means that they might be aware of the literature's view of component-based software development.

The difference between the problems is that the theoretical problems are more technical and detailed. They better describe a particular problem. An explanation for this might be that the companies are not so happy to talk about their problems, especially describing the problems in detail and their technical nature.

There are many more practical problems that match, than do not match, the theoretical problems. A reason for this may be that the majority of the companies are young and that the most interviewees are educated at the University of Karlskrona/Ronneby. This means that they might be aware of the literature's view of component-based software development.

The difference between the problems is that the theoretical problems are more technical and detailed. They better describe a particular problem. An explanation for this might be that the companies are not so happy to talk about their problems, especially describing the problems in detail and their technical nature.

There are many more practical problems that match, than do not match, the theoretical problems. A reason for this may be that the majority of the companies are young and that the most interviewees are educated at the University of Karlskrona/Ronneby. This means that they might be aware of the literature's view of component-based software development.

The difference between the problems is that the theoretical problems are more technical and detailed. They better describe a particular problem. An explanation for this might be that the companies are not so happy to talk about their problems, especially describing the problems in detail and their technical nature.

There are many more practical problems that match, than do not match, the theoretical problems. A reason for this may be that the majority of the companies are young and that the most interviewees are educated at the University of Karlskrona/Ronneby. This means that they might be aware of the literature's view of component-based software development.

The difference between the problems is that the theoretical problems are more technical and detailed. They better describe a particular problem. An explanation for this might be that the companies are not so happy to talk about their problems, especially describing the problems in detail and their technical nature.

There are many more practical problems that match, than do not match, the theoretical problems. A reason for this may be that the majority of the companies are young and that the most interviewees are educated at the University of Karlskrona/Ronneby. This means that they might be aware of the literature's view of component-based software development.

The difference between the problems is that the theoretical problems are more technical and detailed. They better describe a particular problem. An explanation for this might be that the companies are not so happy to talk about their problems, especially describing the problems in detail and their technical nature.

There are many more practical problems that match, than do not match, the theoretical problems. A reason for this may be that the majority of the companies are young and that the most interviewees are educated at the University of Karlskrona/Ronneby. This means that they might be aware of the literature's view of component-based software development.

The difference between the problems is that the theoretical problems are more technical and detailed. They better describe a particular problem. An explanation for this might be that the companies are not so happy to talk about their problems, especially describing the problems in detail and their technical nature.

There are many more practical problems that match, than do not match, the theoretical problems. A reason for this may be that the majority of the companies are young and that the most interviewees are educated at the University of Karlskrona/Ronneby. This means that they might be aware of the literature's view of component-based software development.

The difference between the problems is that the theoretical problems are more technical and detailed. They better describe a particular problem. An explanation for this might be that the companies are not so happy to talk about their problems, especially describing the problems in detail and their technical nature.

There are many more practical problems that match, than do not match, the theoretical problems. A reason for this may be that the majority of the companies are young and that the most interviewees are educated at the University of Karlskrona/Ronneby. This means that they might be aware of the literature's view of component-based software development.

The difference between the problems is that the theoretical problems are more technical and detailed. They better describe a particular problem. An explanation for this might be that the companies are not so happy to talk about their problems, especially describing the problems in detail and their technical nature.

There are many more practical problems that match, than do not match, the theoretical problems. A reason for this may be that the majority of the companies are young and that the most interviewees are educated at the University of Karlskrona/Ronneby. This means that they might be aware of the literature's view of component-based software development.

The difference between the problems is that the theoretical problems are more technical and detailed. They better describe a particular problem. An explanation for this might be that the companies are not so happy to talk about their problems, especially describing the problems in detail and their technical nature.

There are many more practical problems that match, than do not match, the theoretical problems. A reason for this may be that the majority of the companies are young and that the most interviewees are educated at the University of Karlskrona/Ronneby. This means that they might be aware of the literature's view of component-based software development.

The difference between the problems is that the theoretical problems are more technical and detailed. They better describe a particular problem. An explanation for this might be that the companies are not so happy to talk about their problems, especially describing the problems in detail and their technical nature.

There are many more practical problems that match, than do not match, the theoretical problems. A reason for this may be that the majority of the companies are young and that the most interviewees are educated at the University of Karlskrona/Ronneby. This means that they might be aware of the literature's view of component-based software development.

The difference between the problems is that the theoretical problems are more technical and detailed. They better describe a particular problem. An explanation for this might be that the companies are not so happy to talk about their problems, especially describing the problems in detail and their technical nature.

There are many more practical problems that match, than do not match, the theoretical problems. A reason for this may be that the majority of the companies are young and that the most interviewees are educated at the University of Karlskrona/Ronneby. This means that they might be aware of the literature's view of component-based software development.

The difference between the problems is that the theoretical problems are more technical and detailed. They better describe a particular problem. An explanation for this might be that the companies are not so happy to talk about their problems, especially describing the problems in detail and their technical nature.

There are many more practical problems that match, than do not match, the theoretical problems. A reason for this may be that the majority of the companies are young and that the most interviewees are educated at the University of Karlskrona/Ronneby. This means that they might be aware of the literature's view of component-based software development.

The difference between the problems is that the theoretical problems are more technical and detailed. They better describe a particular problem. An explanation for this might be that the companies are not so happy to talk about their problems, especially describing the problems in detail and their technical nature.
7 Glossary

The glossary explains what some terms, used in this master thesis, mean. The terms are explained, they are not defined. The same terms may be used differently or may have different meaning in other contexts than this master thesis.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter</td>
<td>Glue code that adapts one component to another component.</td>
</tr>
<tr>
<td>Black box component</td>
<td>A component where only the interface is available to the component user. The implementation of the component is not available to the component user.</td>
</tr>
<tr>
<td>Component</td>
<td>An identifiable building block in a system's architecture. It has a well-defined interface and implements a uniform set of services in a special domain. A component is indented for replacement and reuse.</td>
</tr>
<tr>
<td>Component certification</td>
<td>A legal guarantee of a component's capabilities and constraints in a certain environment.</td>
</tr>
<tr>
<td>Component developer</td>
<td>A software engineer that develops and maintains a component that a component user uses.</td>
</tr>
<tr>
<td>Component user</td>
<td>A software engineer that uses components to build systems. The component user performs the process described by the term &quot;component-based software development&quot;.</td>
</tr>
<tr>
<td>Component-based software development</td>
<td>The literature explains it as a process of selecting, adapting and integrating components into systems [Car97, HCF97]. The system is maintained by updating the components in the system. Component-based software development is also called component-based software engineering.</td>
</tr>
<tr>
<td>Externally developed components</td>
<td>Components that are not developed or maintained by a component user. The externally developed components may be bought from a component developer or acquired for free, from example the internet.</td>
</tr>
<tr>
<td>Framework</td>
<td>The part of an overall system architecture into which variable components can fit [ThoMan96].</td>
</tr>
<tr>
<td>Glue code</td>
<td>Mechanism that permits composition of components in a software architecture [ThoMan96].</td>
</tr>
<tr>
<td>Internally developed components</td>
<td>Components that are developed and maintained by the component user.</td>
</tr>
<tr>
<td>Plug and play</td>
<td>The ability to easily compose and making the components to interoperate. How easy it is to plug and play the components is subjective.</td>
</tr>
<tr>
<td>White box component</td>
<td>A component where both the interface and the implementation are available to the component user. The component user may change the implementation of the component.</td>
</tr>
</tbody>
</table>
8 Appendix

8.1 Appendix A

This Appendix contains the questions that were asked while interviewing the companies listed in section 2.2.2.

8.1.1 General questions
- Tell me a little bit of your company, your business area, number of employees, locations of the company, etc.
- Does the management support component-based software development? What makes them do that?
- What is your definition of a component?
- Why do you develop software with help of components?
- What benefits are there with component-based software development?
- How long experience do you have with component-based software development?
- Developing software with help of components, is it the right way to go? Is there any other better alternative?
- Do you use externally developed components and/or internally developed components? Are they black box or white box components?

8.1.2 Detailed questions

The development process
- Has your way of developing software changed considerably when you started to develop software with help of components?
- Do you have any special development process for handling all the problems and risks that are involved when developing software with help of components?

Internally developed components
- How do you handle the costs for testing and maintenance of the components? Are these costs repaid? Do you have any metrics that shows this?
- Do you spend extra time in making the components reusable? Does it pay off?
- Do you have several versions of the components with different functionality? How costly is it to handle all the different versions?
- Do you miss anything that supports component development or component-based software development? Design experience, tools, methodology, documentation, etc?

Documentation questions
- How do you document the components? What is documented and what is not documented?
- How easy is it to use a component with help of the documentation?
- Do you have any problem with poorly documented externally developed components? How do you handle this problem?

Security questions
- Do you have any security-related problems with components?
- Do you trust the externally developed components? What makes you trust them?
- What do you think about black box components?
- What security level do you achieve when a system is built up with components?
- Do you see the operating system as a security risk?
Problems with Component-Based Software Development
Appendix

Technology questions
• Is object-oriented technology needed for component-based software development? Why? Why not?
• Do you use any component enabler technologies, for example Java Beans, CORBA and D/COM? How do you decide which technology to use?
• How do you model a system than is built up with components?

Selection questions
• How do you select a component? How do you find it? Do you have any well-defined selection process?
• How do you define the selection criteria? Is it well defined?
• How critically do you evaluate a component? Do you spend enough time on the evaluation?
• Do you have any problems with finding components? What do you do about it?
• Are the components certified in any way? If not, how do you know what they are capable of?
• Do you have any component repository? How well does it work? Is it easy to find a component?

Integration questions
• How easy is it to compose the components and make them work together?
• What integration techniques do you use?
• Do you use some kind of contracts between the components when you integrate them?
• Have you needed to change a component in order to make it work with another component? Did the component become specialized?
• Do you build adapters and wrappers for the components? What problems are there when using adapters and wrappers?
• How easy is it to exchange a component with another one in a system? How do you know how the new component affect the system?
• How do you solve the problem if two components do not want to interoperate?
• Do you have any performance problems with the integrated components?
• Do you have problems with components that do not want to give up their execution control? How do you solve this problem?
• Do any components share any data? How does this work? How is the data kept consistent?

Component testing
• How is the testing of the components and the component-based system made?
• What testing methods do you use? Are they different from the one used in traditional software development?
• Do you test externally developed components? Why?
• How do you handle the situation if a system with several externally developed components fails and the component developers blame each other?
• When a system fails, how do you find which component has failed?
• Do you do more testing when a new component has replaced another one? How well do you test the system in order to secure it?
Other questions
• Have you used a component more than once?
• Do you experience the not-invented-here syndrome?
• What do you do if a component developer that you depend on goes out of business? How do you handle your dependence on the component developers?
• Do you have any maintenance problems with component-based system?
• Are there any other problems with component-based software development that you have run into?

8.2 Appendix B
This Appendix contains the result of the interviews performed with the companies listed in section 2.2.2.

8.2.1 Component-based software development at Ericsson Software Technology [EST99]

Introduction
The management of Ericsson Software Technology supports component-based software development. Developing software with components is a way of working at Ericsson Software Technology. It is an organizational issue where every component has an owner that is responsible for its maintenance and further development. A kind of divide and conquer strategy is used.

Ericsson Software Technology develops and uses domain specific components. They only work with internally developed components. They have the source code available and can change and further develop the components themselves. Building systems out of externally developed components is not an option. There are no suitable components to buy and use in the domain that Ericsson Software Technology works. Instead, the idea in Ericsson Software Technology is to build systems out of small manageable well-defined components. These systems are easy to maintain and develop further.

The development process
There is not much new development at Ericsson Software Technology. Most of the development is in the maintenance phase of the component-based software development process. It is about adding new functionality to the existing system.

A project is started when there is something new to be built in a system. It is checked what components have to implement the new functionality and what needs to be updated in the system. It is very easily done because the components are well defined. You know exactly where to look, because every component has a specific well-defined functionality and is in a specific domain.

There is no single customer that defines the requirements. Instead, the market and the competition determine the requirements. The requirements are divided into groups. Every group of requirements is implemented in each iterative step. An iterative step can take two to five months to complete and consists of three steps: implementation, integration and testing.
Problems with Component-Based Software Development

Appendix

It is pretty easy to develop the components. Most of the work with component-based software development is in coordinating and making the interfaces of the different components to work together. A problem the developers experience is that the software development can be geographically distributed throughout many different offices. It makes it harder to coordinate the development of the system so that the components work well together. The larger distance between the offices the harder it is to keep close contact with other developers.

**What is a component?**
The developers at Ericsson Software Technology describe a component as:
- A block of code that is easy to handle.
- Has a well-defined interface.
- Has a special set of uniform functionality.
- Belongs to a special and uniform domain.
- It is easy to identify what it does.
- It is easy to maintain and further develop it.

As mentioned earlier, every component has an owner. Developing software with components is a way of separating the responsibilities of a system. Each developer is responsible for one or several components. It is the developer’s responsibility to make sure that the component works as it should and to maintain and further develop the component/s.

**Benefits**
The main reason for developing component-based systems is because it is easy to maintain. Each component has a uniform functionality and is in a special uniform domain. Because of this, you know exactly where everything is and exactly where to change the system. It is easy to identify where new functionality should be implemented. There is always someone that is specialized in the component, who can easily implement new functionality and make sure that the component works. If the system was not divided into components, then it would be very hard to implement new functionality.

Each component has an owner. This means that there is always someone who is responsible for a component. It is the owner’s responsibility to update the component and make sure that it works well together with the other components. An owner has a commitment to make sure that the component works. Because of the component ownership, every owner is specialized in some functionality and domain. This is an advantage because you always know whom to ask about something.

A component is not larger than that you can get a general view of it. A single person can work with one or several components. Another person that has not worked with the component earlier can easily understand and start to work with the component. It is much easier to work with a component than with a very large system.

**Documentation**
The documentation of the components varies in different projects. In the AXE [AXE] projects, there is a strict process of how to develop the components and the system. There is very much documentation, in some cases more than source code. Everything that the process says to document is documented whether it is needed or not.
In non-AXE projects, there is less documentation. Document templates are used to document the components. These templates describe what should be documented and help not to forget to document something important. The most important in the template is considered to be describing the functionality of a component. Only things that need to be documented are documented. Common sense is used to decide what to document and what not to document. For example, if you describe something in a diagram, then it is not necessary to describe it textually again. Only small spicy things that make it easier to understand the diagram should be written down.

There are two kinds of documentation for each component, a usage and a development documentation. The components and the systems are modeled with help of a visual tool and the text is written in an ordinary word processor.

The usage documentation describes how to use the specific component. It describes its interface and how to integrate it together with other components. The interactions between the components are described with help of class diagrams, since the integration is usually done between two classes. Calling sequences are normally not needed to be documented. Mostly it does not matter in which order the methods are called. The components are designed not to do anything if the methods are called in the wrong way. This simplifies the usage of the components very much. A person that does not own the component can integrate two components with help of the usage specification. The inner workings of the component are described in the development documentation. It describes the methods and how they collaborate. The idea with the development documentation is that another developer should easily be able to understand and start to work with a component faster.

Tools
Ericsson Software Technology has everything that is needed to perform the work with component-based software development, but things could get better. They would like to have Rational Rose [RS] to make the modeling of the components easier. The reason for that they do not have Rational Rose is because they would only use a small portion of the functionality that Rational Rose provides. It would probably not pay off to buy and use Rational Rose. In the future Ericsson Software Technology plans to use the Unified Modeling Language [RS], which is supported by Rational Rose. They will therefore probably start to use Rational Rose in the future.

Framework Distributed System
The Framework Distributed System (FDS) is an internally developed universal platform for component composition. It supports multiple systems and is therefore the most reused component. The FDS is the component that controls the execution of the system. Because of this, there is no thread of execution problem in systems that are composed by the FDS.

The systems that use the FDS have different requirements. For example, one system may require higher reliability than another one. This has led to the FDS being developed to support the system with the highest requirements on reliability. Other systems that do not need such a high reliability get this high reliability for free.

The FDS is maintained and further developed by a special group of engineers. The group is composed of engineers from all systems that use the FDS to compose their system. The reason for this is to coordinate the development of the FDS so that it suits the different products. The group makes sure that it supports the new functionality that is required. They also make sure that it is easy to integrate the components.
Security
The FDS is a very critical part of the system. The security requirements on it are therefore much higher than on the components that it composes. The system that uses the FDS can not be more secure than the FDS itself. The FDS always has a backup copy of every component that it composes. If one component fails, the other one takes over.

There are some strict security requirements on the components. It is well defined how a component may be started and fail, if it does that. It is also well defined how a component may be shut down. There is a framework of rules that explains in detail what is allowed to do and what is not allowed to do. This contributes to the security and stability of the system.

Integration
The components are mostly composed with help of the FDS. Another way of composing the components is done by using CORBA [OMG]. The difference between composing the components with FDS and CORBA is how fast and easy it is to compose the components. The FDS is easier to use because it is specially developed for composing the components. It is also developed for handling hard trafficked communication software.

During the design phase the interfaces between the components are designed and coordinated. Later during the implementation, there are no practical problems of getting the components to work together. It is easy to put them together and make them work together. The integration and interaction of the components is automatically solved in the implementation phase since it has been solved in the design phase. Therefore, there are no problems of integrating two components.

All communication between the components is done through the FDS. It knows and is aware of all components. A component sends and receives messages to and from another component through the framework. The components do not share any common data, they only communicate with help of messages. The messages may consist of passing parameters or objects. Telecom protocols and signals are used for communicating between different systems. For this purpose adapters are used between different systems. The adapter translates from one protocol to another. This is common for all communication between different systems and is needed to make the communication possible.

Testing
The testing is done at three levels, that is, component, integration and system testing. There is a test plan for each level of testing. The most important testing is the component testing level. In the component testing the component must always pass a so-called basic test. Fault injection tests are made where the components are fed with faulty data and checked to see how they respond to it. Component traffic testing is made where the purpose is to see if the component can handle the amount of traffic needed.

On the second level of testing, the integration between the components is tested. There is not much integration testing performed because the components are designed and made for being integrated. They are often very easily integrated. On the third level of testing, the whole system with all components is tested. The system is tested to check if the requirements are fulfilled.

Several testing methods are used to increase the confidence of the system. For example, coverage testing and fault injection is used in several test phases. All tests log information from the components. This makes it easier to find a fault in a test. You can directly see what went wrong and if necessary, easily backtrace it to the source of the fault.
Maintenance
If you don't develop software with help of components it may happen that the system may grow and become hard to get a general view of it. This is exactly what Ericsson Software Technology wants to avoid. It is very easy to change and further develop a system that consists of small manageable components that have a uniform and well-defined functionality. It is important to remember not to have too large components. If you do, then the idea of having small manageable components is lost. If a component grows large or out of its functionality and/or domain, it may be needed to review it and possibly redesign and split the component into one or several components.

It is hard to get some time over from the projects to review the component structure. There is no time planned for this. It is very easy to neglect and don't do anything about this problem. There should be somebody that takes care of the problem of making the components easier to work with. This is the key in component-based software development, to have a strategy of how to further develop the components.

8.2.2 Component-based software development at Flux Software Engineering [Flu99]

Introduction
Flux Software Engineering has one year of experience with component-based software development. The management of Flux Software Engineering supports component-based software development in the cases it is profitable. There are mainly two reasons for why Flux Software Engineering develops software with help of components. The first reason is that they expect that it will shorten the lead-time for developing systems. The other reason is that component-based software development fits very well in their evolutionary and incremental development process. Developing software with components has not changed their development process much. They have been able to adapt it very easily.

Flux Software Engineering is both a component developer and a component user. When they are about to build a system, they either develop or acquire the components elsewhere. They put together a system with the components and sell it to the customer. While the time passes, they further develop and maintain the components that the system is built up of. When the customer feels that the system needs to be upgraded, they can turn to Flux Software Engineering. The system is easily upgraded by replacing an older component with a newer one that supplies the functionality that the customer wants. The replacement of the components is done in a plug and play style.

Flux Software Engineering does not develop all their software with components. The software is developed with components when it fits in the system. It depends much on what requirements there are on the system. If there is a requirement for a system to be able to swap the components at runtime, then they develop the system with components. To be able to do this, the components have to be well defined and the system must be designed for it. It is impossible to do it if the architecture does not support it.

A large system with a long lifetime is more suited to be developed with components than a small system with a short lifetime. Developing software with components is an investment in time. It takes some time to find or develop the components that are suitable for a system. Therefore it does not pay off to develop small systems with a short lifetime with help of components. In those cases, the software is made without components.
What is a component?
A component is a collection of modules that together form a certain well-defined functionality or a set of functionality that fits in a uniform domain where the component is to be used. It may be specialized in its functionality and domain, but the interface should have a general structure so that it is possible to reuse it in other systems, which are in the same domain. This also means that a component should be replaceable with another component, which is within the same domain. It may not be possible to use a specialized component in systems of another domain.

Benefits
Developing software with components puts requirements on the system and its architecture. An architecture that is built up of components becomes very flexible. This is one of the greatest benefits. This benefit has a side effect of putting high requirements on the design of the system. A divide and conquer technique is used to handle this side effect. This results in very good designs and support of system flexibility.

As mentioned earlier, a component has a well-defined functionality in a uniform domain. This makes it easy to further develop and maintain a component. It also makes it possible to easily exchange components in a system. It can be made at runtime in an easy plug and play style.

Systems that are built up of components are easier to test. A system that is divided into components makes it easier to isolate the faults that appear during the testing. Another testing advantage is that it is possible to test the components independently before they are integrated. The components have a well-defined interface, which makes the integration very easy and the integration testing minimal.

Reusability
Flux Software Engineering mostly uses general components. Those components have a uniform interface, which makes it possible to reuse them many times. Specialized components are not used so much, since it does not pay off to use them.

Flux Software Engineering has built in the reuse thinking in their software development process. They don't think so much about making the components reusable because it is a natural way of working for them. They automatically design the components to be reusable. They are aware of that it costs a little bit to make the components reusable, but they believe that it pays off fast. The benefits can directly be seen in the maintenance phase.

Documentation
The internally developed components are well documented, especially the design. The design is used for two purposes. First, to make sure that the requirements are fulfilled. Second, to make the maintenance phase easier. It is easy to use, understand and further develop the components with help of the design.

Flux Software Engineering has some problems with the documentation of the externally developed components. The components have too little documentation and wrong things are documented. This makes them harder to use.
Black box components
Flux Software Engineering believes that it is very important to see the test results of a black box component so that they know the quality of the component. They also have to test the component themselves to see if the functionality is suitable for a specific system. It is possibly harder to do this with a black box component.

Flux Software Engineering trusts a black box component if it fulfills the needed requirements. They trust more in a component if it comes from a large company with a good reputation. The components are in that case often well-done and more used. The chance of the component being well tested is much higher.

There is a difference in using small and simple versus large and complex black box components. A simple black box component that has a well-defined functionality can be easily tested to see how it works and if it fulfills the requirements. In this case it is a good idea to use the black box component. In the other case, it is much harder to test a large and complex black box component. It is not so easy to understand how it works and find out if it fulfills the requirements as with a small and simple black box component. In this case it is not a good idea to use a black box component.

Object-oriented technology
Flux Software Engineering believes that object-oriented technology is not needed for component-based software development. On the other hand, they believe that the concepts of object-oriented technology are applicable to component-based software development. They can be applied on larger units of code than a simple object. A component, as well as an object, must have a well-specified responsibility and functionality. For example, information hiding is a good design technique that can also be used in component-based software development. You can encapsulate a component, which then becomes a black box component.

Configuration Management
The components of a system are continuously developed further and updated. There are therefore several versions of one single component. It is important to keep track of all versions of a component so that you know what functionality each component has. There is also a need to keep track of what versions of components build up a certain customers system. This is important to know when the customer wants to upgrade the system. The components may be incompatible with each other since they may support different overall functionality of a system. Flux Software Engineering does not have any problems with the configuration management. They see it as a potential problem as the configuration management gets more sophisticated and more complex when there are several versions and configurations of the components and systems to handle.

Selection
Flux Software Engineering both develop their own components and acquire them elsewhere. They therefore have some internally developed components to choose among. They know these components very well. Selecting the right component among the internally developed ones is not a problem.

The problem of selecting components appears when they try to choose among the externally developed components. It is not hard to find components that can be used, but it is hard to find the right one, since several components can match the needed requirements. The right component is selected by thoroughly examining the components and testing them to see which component is best to use in the long-term.
## Integration
Frameworks are mostly used for integrating the components. The framework is the part of the system that does not change and that does not need to be flexible. It is the general overall architecture of the system. The components, on the other hand, are the parts of the system that are changed and updated. The components make the system flexible, since you can exchange them easily.

In some sense, contracts are used between the components. There is a detailed specification of what preconditions have to be fulfilled for being able to use a component's functionality. There is also a detailed specification of the postconditions too.

A system may be distributed on several machines and consist of several applications. Scripts and functional based languages are used to integrate these parts. These scripts glue the system together and make the whole system work together.

The communication technique that is used depends on what kind of a system is built. The best suited communication technique for the system is selected. A framework mostly integrates the components. Therefore the communication is done through the framework. This does not mean that the communication is just done through the framework. The components can also communicate directly with each other. It has happened that two components have shared data in form of a database. The problem of synchronizing the components, so that the data is kept consistent, has been solved in the database design.

## Testing
Flux Software Engineering tests the components to verify that they work as required. They test both internally and externally developed components. The testing is usually not done to gain confidence in a component. Instead, the functionality is tested to see if it satisfies the requirements that are put on them. Fault injection testing and stress testing is only performed when needed.

Integration testing is a very small part or non-existent part in the testing phase. It is included in the system testing when the functionality of the whole system is tested. All problems with the component integration are dealt with in the design phase. It is too late to deal with them in the testing phase.

The functionality of the components is implemented incrementally. Regression testing is made in the end of each increment to make sure that the earlier implemented functionality still works. Regression testing is also used to secure the system when a component has been exchanged.

If a system fails, it is easy to identify the source of the fault. The systems have a good error handling on several levels. With the help of this error handling, it is pretty easy to identify which component has failed and where.

## Maintenance
A maintenance problem that Flux Software Engineering has discovered is that it is not so easy, as they would like, to update a system at runtime. They are working on this problem and are improving the plug and play ability of the components so that it can be easier to update a system.
8.2.3 Component-based software development at Massive Entertainment

Introduction
Massive Entertainment has already from the start of the company, in 1997, decided to develop their first game Genesis with help of modules and components. The management supports component-based software development, they understand and are convinced by the benefits. Component-based software development makes it easier to update and further develop the system. It is believed that component-based software development makes it easier to reuse the developed components in future systems. Massive Entertainment's development process emphasizes the reuse of design and source code. All modules and components that Massive Entertainment uses have been internally developed, except for one, which is Microsoft Sockets [MS]. The modules and components have been developed with reuse in mind.

Massive Entertainment does not go out on the market and look for externally developed components that they could use in their system. All components are internally developed. This does not mean that they have completely closed the door for externally developed components. They are, in the future, open for using externally developed components if it shows that it fits their needs. The reasons for not using externally developed components are:

- They want to develop their own components to gain knowledge and expertise in the domain of building games.
- They do not want to depend on any external component developer, by licensing the technology. With own developed technology, they have full control over it.
- They have high performance requirements on Genesis. An externally developed 3D game engine may not be optimized for their game. It would be hard or impossible for them to do the optimizations in an externally developed 3D game engine. There would probably be much unwanted functionality and it would probably be very slow.

What is a component?
Genesis is built up of modules and a framework-like module called the application code module. A module is a composite component that consists of several smaller atomic components. The difference between a module and an atomic component is that an atomic component is more reusable than a module. A component is more general and made for being reusable in many different situations. A module is more specialized than a component and can therefore only be used in a certain type of applications.

Benefits
Massive Entertainment's goal is to create a modular and reusable core technology that will shorten the development time and allow them to produce systems with higher quality. A component can be easily tested which makes it easier to attain higher quality. Reusing a component also increases the quality. The more it is reused the higher quality it gets because the error fixes accumulate. The quality assurance is not just done now, but also in the future.

In component-based software development you can easily reuse components after they have been developed. Developing component-based software makes it easier to license out the technology. The 3D game engine is made for this purpose. It is built up with components and has a general interface that may fit many needs.

Component-based software development makes it easier to maintain and further develop the system. It is built up of general components, which makes it easy to exchange them with other components. It is easy to plug and play the components at runtime.
Reusability
Massive Entertainment has the reuse thinking built-in in their development process. Because they will reuse the modules and components, they have to be designed to withstand future requirements. When they build a system they read the requirements specification and identify modules and components that are needed for the system. To the largest extent possible they try to make these modules and components as general as possible so that they can be reused in other future systems.

They spend extra time to make the modules and components reusable. They have not yet seen any concrete evidence of this being profitable. While the interview was done they were still developing their first game. The modules that the game consists of have therefore not been used more than once. In their next game they expect to be able to reuse all modules except for the application code module. The application code module will be thrown away because it is specific for Genesis. Before they throw away the whole application code module, they will look for smaller components that can be reused. Massive Entertainment is convinced that the time invested to make the modules and components reusable will be paid back manifold when they start to develop their next game because so much is expected to be reused.

Not only the source code is reused. The design and knowledge are also reused. Massive Entertainment has built a network communication protocol that is specific for Genesis. It will not be possible to reuse it in future games. On the other hand, the knowledge of building and using it will be reused.

The problem with developing components in the game domain is that you need a lot of knowledge and expertise to succeed. It is not easy to make a component as general as possible. There is a conceptual problem of deciding where the border goes between a general and specialized component. A component is almost never as general as possible so that it can easily be reused in other systems. There is always a need for modifying a component for making it possible to use it in another system.

Documentation
Massive Entertainment has not yet come to the phase of documenting the components, they plan to do this later. The most important thing to document is considered to be the interface of the components. It is important to know how to use the components and what functionality they provide. Documenting how the functionality is implemented comes in second hand.

There has been a problem for someone else than the developer of a component to understand a component. The reason for this is the lack of documentation. This has been solved easily because the responsible for the components has always been available.

Black box components
Massive Entertainment uses Microsoft Sockets for the network communication in Genesis. This can be seen as a black box component that is supplied by an external vendor. They trust this black box component because it is so widely used and therefore well tested. If there is a security problem with it, they believe that it is not their problem. It is an operating system issue. If a problem appears, the customers will blame Microsoft and not Massive Entertainment. Massive Entertainment does not provide the socket component, it is supplied by Microsoft's operating system.
Object-oriented technology
Object-oriented methodology is needed for conceptual reasons in component-based software development. It is easier to design and understand a system if you think in object-oriented terms. Object-oriented methodology is applicable to component-based software development. On the other hand, object-oriented technology is not needed for component-based software development. It is possible to create and integrate components without object-oriented technology.

Selection
Massive Entertainment has no problems in selecting which components to use. There is no problem of keeping track of all components and deciding which component to use because they are so few in the company and because they develop the components themselves. Every developer knows what components there are and how to use them. They develop a component if they discover that they are missing it. Massive Entertainment believes that keeping track of all the components and selecting components will become a problem in the future. They therefore plan to build in support for it in their development process.

Integration
The modules are integrated with help of the application code module. This module is specialized for every game. It is the part that connects all other modules and make up the core of the game. It is a kind of a framework that holds together and executes the game. To build another game, Massive Entertainment will have to develop another application code module, but they hope that they will be able to reuse the other developed modules in Genesis.

The integration of modules is done through their interfaces. The interfaces are well defined which makes it easy to integrate them. It is easy to exchange a module or a component with another one. It can be easily done at runtime in a plug and play like style. For example, a TCP/IP network module can easily be exchanged for an IPX network module. The network interface is so general and well defined that it is painless to change the network module at runtime.

The communication between the modules is done through the application code module. If two components are unable to communicate, Massive Entertainment builds adapters that translate the communication between the two components. Another option is to change the components. This is only done if the communication is prohibited because they are not general enough. The components are not changed if they become less general.

Massive Entertainment builds wrappers for every graphics card standard. The wrappers provide a common interface to the 3D game engine. The generality of the 3D game engine is upheld this way because it does not have to worry about which graphics card is used in a computer. The graphic card wrappers are also a kind of adapters. They translate the 3D game engine's general instructions to specific graphic card instructions. This way the 3D game engine uses the full potential and performance of a certain graphics card.
Testing
Most of the Genesis testing will be done "in action". The game will be played and tested to see if it passes the test specification and fulfills all requirements. The modules and the components will be tested before the "in action" test. The goal with the "in action" testing is to get as much external input of the game as possible. The developers should not do the "in action" testing. This is because they will be subjective and possibly fail to identify some of the faults. It is better to get external input, because the faults will be identified immediately. The "in action" testing will be performed both locally and internationally.

The interfaces of the modules are well-defined, they are also designed to be integrated. There is therefore no problem of testing the integration of two modules. The testing is more focused on testing the functionality of a module and the whole system. If a fault is found, it is identified where it occurred by tracing it in the source code. Because the system is divided into modules and smaller components it is pretty easy to find out the origin of the fault.

The modules have not been thoroughly tested yet. The 3D game engine will be tested to see what hardware needs to be supported. If new hardware needs to be supported, wrappers will be implemented for it. Massive Entertainment's 3D game engine is compatible with the future. Performance testing of the 3D game engine will also be performed. These testing results will possibly be used for optimizing it.

Other
Not-invented-here syndrome
Massive Entertainment has some problems with the not-invented-here syndrome. The reason is that the developers are hostile to code that they have not invented, until they get to a certain level of insight in component-based software development. Another reason is that Massive wants to own their technology, because of all the benefits. They are not against using externally developed components. They only believe that there are more benefits with developing and using their own technology.

Performance and optimization
Because Massive Entertainment develops almost everything themselves, they think that they do not have so many performance problems. If they had, they would optimize it. When Genesis is close to be finished, they will look for and identify possible performance problems. They know already now what can and should be optimized. An example is that the Microsoft's number conversion functions are ineffective. To optimize the number conversion they had to implement it in assembler code. This optimization increased the performance of the game by 15 percent. This has taught them that there is a risk in using code that has been developed by someone else, especially in a performance critical module like the 3D game engine.

Massive Entertainment believes that they would have had many performance problems if they had acquired an externally developed 3D game engine. They would most likely not be allowed to optimize it. And if they were allowed to optimize it, they would not know what and where to optimize the 3D game engine since they have not developed it.
8.2.4 Component-based software development at Symbian [Sym99]

Introduction
Symbian's management supports component-based software development. They are convinced that it is the right way to go and understand the benefits with it. Symbian uses both internally and externally developed components. These components are assembled into large systems like the EPOC operating system. This is done with an own invented development model. This development model also handles all risks and problems that come with component-based software development.

Symbian only uses white box components to avoid all the problems with black box components. They do not acquire externally developed components if they do not get the source code and the right to change and further develop them. Everything is much simpler with white box components. You have control over them, when you have the source code. Having control over the components makes the development much easier than using black box components.

Developing software with components is not the ultimate way of developing software. There are many things that affect software development. Many other techniques complement component-based software development. One example is design patterns.

Benefits
With component-based software development you get more effective and faster development. It is cost effective to develop systems with components. The benefits of reusing components outweigh the costs for adapting, testing and maintaining the components.

Another benefit is that you get a higher quality of the developed system. You never know what quality an externally developed component has. It is not measurable. They are often more used and tested by other component users. They are therefore often of high quality.

Reusability
The goal for any software that Symbian develops is to make it reusable. The extra time to design and implement reusable software is taken into account when planning the work. Very little software is hard coded. It is only done in exceptional cases when there is some time pressure. The hard coded piece of software is later, when there is more time, reengineered into reusable software. Some pieces of the software are not required to or can not be made reusable. That software is therefore always hard coded. An example of this is the glue code.

Documentation
The externally developed components are often poorly documented. It is therefore sometimes hard to understand how a component works. Because Symbian only works with white box components, they get most of the information from the source code. There are sometimes problems with understanding the component when you just have the source code. It takes more time than if it was well documented with a visual design and other explanations in natural language.
Symbian plans to increase the quality of the documentation of the internally developed components. Experience has shown that it is hard to specify components in a way so that any developer can start to use a specific component. The most important part of the component documentation is the use cases. That is, how to use the component and how it works. Questions like What methods have to be called before another method is called? and What do the methods do? have to be answered by the documentation. Perhaps you have to initiate the component before you can use its functionality and thereafter you may have to close it. This also helps you in knowing what is allowed to do and what is not.

Object-oriented technology
Object-oriented technology is not needed for component-based software development. Components can be developed and composed without object-oriented technology. Object-oriented technology only makes component-based software development easier.

Selection
Symbian does not have any well-defined process for selecting the components. There is no need for this. They believe that they put enough time in selecting the right component and evaluating it. The evaluation is done by testing the component to see how trustworthy and reliable it is. It is sometimes hard to find the right component. In that case they are forced to develop the component themselves.

Symbian keeps track of all the components with help of a component repository. When a developer needs to find a component s/he looks for it in the repository. If no component could be found in the repository, then it is looked at what externally developed components could be used. The externally developed components are often poorly documented. This makes it hard to evaluate them to find out if they satisfy the specific need.

Component adaptation
The components that Symbian acquire externally do not always fit smoothly into the systems they develop. Much depends on the specific component and the system that it is supposed to work in. Often they are forced to change the component, which is the common solution to all problems. This is possible because Symbian has the source code to all components they use. They have the right to change and further develop the components. Internally developed components do not often need to be changed. They are developed to fit the environment they are supposed to work in.

Depending on what is changed, the changed component may become a specialized component that can only be used in the system that it was adapted to. The goal is to try to make the component as general as possible so that it can also be used in other systems. Symbian is especially good at this, since many components have been reused several times. Time invested in developing and using components has paid off many times.

Integration
There are some problems to get the components to work together. All kinds of problems have arisen. It is not hundred percent painless to integrate the components. It is impossible to generally say how easy it is to integrate two components. It varies from one situation to another. The goal is to integrate components as easy as possible. Sometimes it is very simple to integrate two components and sometimes not. The amount of glue code used to integrate the components is very small. This means that the components work together very well and indicates that it is relatively easy to integrate the components.
Exchanging a component with another one with the same interface and functionality is not a problem. It is very easily done. The system with the new component is tested again, but not more than usual just because of the exchanged component.

Various integration techniques are used, depending on the components and their environment. The best-suited technique is used. Some components are integrated with a database. The database is kept consistent by synchronizing the components with the operating system primitives or with a server that handles the communication with the database. Creating a server component is the most common way of synchronizing two components that work against a database.

**Testing**

Symbian's component-based software development model is different from the traditional software development model. The difference is that there is more testing in the component-based software development model. This applies especially to externally developed components. You don't know how much you can trust them, since they are not formally certified. They are tested to see how trustworthy they are, how well they work and suit the needs. They are tested as much as needed, that is, as much as it is believed to be necessary.

The testing of component-based software is not done in any different way than in traditional development. The normal unit, integration and system testing methods are used, where a unit is a component or a set of components. If the system fails in some way, then the system is debugged as any other non component-based system. This is possible because Symbian has all source code available.
9 References

[Alt]
Altavista
Internet Search Engine
http://www.altavista.com

[AXE]
AXE Product lines
Ericsson
http://www.ericsson.se/wireline/indexpro.htm

[BoeAbt99]
COTS Integration: Plug and Pray?
Barry Boehm, Chris Abts
IEEE Computer, January 1999

[Bri98]
COTS Evaluation and Selection
Lionel C. Briand
International Conference on Software Maintenance, 1998

[BroWal98a]
The Current State of CBSE
Alan W. Brown, Kurt C. Wallnau

[BroWal98b]
An Examination of the Current State of CBSE: A Report on the ICSE Workshop on Component-Based Software Engineering
Alan W. Brown, Kurt C. Wallnau
International Workshop on Component-Based Software Engineering, 1998

[BucVal99]
Software Economics in Practice
Christian Bucanac, David Vallata
University of Karlskrona/Ronneby, 1999
http://home5.swipnet.se/~w-51660/documents/Software_Economics_in_Practice.pdf

[Car97]
Assembling Large Systems from COTS Components: Opportunities, Cautions, and Complexities
David Carney
SEI Monographs on the Use of Commercial Software in Government Systems, June 1997

[CarObe98]
The Commandments of COTS – Still in Search of the Promise Land
David J. Carney, Patricia A. Oberndorf
Carnegie Mellon University, 1998
Problems with Component-Based Software Development

References

[Com] Compendex
Lund University Library
http://hood2.ei.org/eivill/plsqli SWITCH main

[CSDL] Computer Society Digital Library
IEEE Computer Society
http://www.computer.org/epub/

[Del97] Toward a Design Handbook for Integrating Software Components
Chrysanthos Dellarocas
Fifth International Symposium on Assessment of Software Tools and Technologies, Pages 3-13
IEEE Computer Society Press, 1997
ISBN 0-8186-7940-9

[Del98] Toward Exception Handling Infrastructures for Component-Based Software
Chrysanthos Dellarocas
International Workshop on Component-Based Software Engineering, 1998

[EST] Ericsson Software Technology AB
http://www.ericsson.com/SE/epk

[EST99] Ericsson Software Technology Interview
Johan Forsman, Christer Akeesson
University of Karlakrona/Ronneby, April 16 1999

[Flu99] Flux Software Engineering Interview
Young Fogelström
Flux Software Engineering Office, April 22 1999

[GAO94] Exploiting style in architectural design environments
David Garlan, Robert Allen, John Ockerbloom
Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineering,
December 1994
http://almond.srv.cs.cmu.edu/afs/cs/project/able/ftp/aesop-fse2.ps

[GAO95] Architectural mismatch or Why it’s hard to build systems out of existing parts
David Garlan, Robert Allen, John Ockerbloom
Proceedings of the 17th International Conference on Software Engineering, 1995
http://almond.srv.cs.cmu.edu/afs/cs/project/able/ftp/archmismatch-icse17.ps
Problems with Component-Based Software Development

References

[HCF97]
Component-Based Software Development / COTS Integration
Gary Haines, David Carney, John Foreman
Software Engineering Institute, October 7 1997
http://www.sei.cmu.edu/activities/str/descriptions/cbsd_body.html

[His98]
Experience Report: Correcting System Failure in a COTS Information System
Scott A. Hissam
International Conference Proceeding on Software Maintenance, 1998

[Info]
Infocenter Library
University of Karlskrona/Ronneby
http://www.hk-r.se/bib/engindex.html

[INS]
INSPEC
CD-ROM Teknik AB
http://webspirs2.cdromteknik.se/cgi-dos/hkr/hkr.cmd

[Kon96]
A Case Study in Applying a Systematic Method for COTS Selection
Jyrki Kontio
Proceedings of the 18th International Conference on Software Engineering, 1996

[KozBoo98]
Component-Based Software Engineering
Wojtek Kozaczynski, Grady Booch

[LinJon98]
A Map of Security Risks Associated with Using COTS
Ulf Lindqvist, Erland Jonsson
IEEE Computer, June 1998

[Lio96]
Ariane 5 Flight 501 Failure
The Inquiry Board, Prof. Jacques-Louis Lions
ESA, Paris, July 19 1996
http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html

[MaiNcu98]
Acquiring COTS Software Selection Requirements
Neil A. Maiden, Cornelius Ncube
IEEE Software, March/April 1998

[Mas]
Massive Entertainment AB
http://www.massive.se
References

[Mas99] Massive Entertainment Interview
Martin Walfisz
Massive Entertainment Office, April 27 1999

[McD98] The Cost of COTS
John McDermid
IEEE Computer, June 1998

[MMM95] Reusing Software: Issues and Research Directions
Hafedh Mili, Fatma Mili, Ali Mili
IEEE Transactions on Software Engineering, Vol. 21, No. 6, June 1995

[MS] Microsoft
http://www.microsoft.com

[Obe98] COTS and Open Systems
Patricia Oberndorf

[OBMS97] Workshop on COTS-Based Systems
Patricia Oberndorf, Lisa Brownword, Ed Morris, Carol Sledge
SEI, November 1997

[OMG] Object Management Group
http://www.omg.org

[Pou98] Towards Component-Based Software Engineering
Gilda Pour

[RS] Rational Rose, Unified Modeling Language
Rational Software
http://www.rational.com

[SC] Soft Center
Ronneby, Sweden
http://www.softcenter.se
Problems with Component-Based Software Development

References

[SEI]
COTS-Based Systems
Software Engineering Institute
http://www.sei.cmu.edu/cbs/

[SLP98]
A Generative Approach to Componentware
Marcelo Sant’Anna, Julio Cesar Sampaio do Prado Leite, Antonio Francisco do Prado
International Workshop on Component-Based Software Engineering, 1998

[Sym]
Symbian Ltd.
http://www.symbian.com

[Sym99]
Symbian Interview
Peter Molin
Symbian Office in Ronneby, March 31 1999

[ThoMan96]
Component Software Glossary
Craig Thompson, Frank Manola
Object Services and Consulting, 1996

[Tra98]
Component-based Integrated Systems Development: A Model for the Emerging Procurement-centric Approach to Software Development
Vu Tran

[VigDea98]
Building Maintainable COTS Based Systems
M. R. Vigder, J. C. Dean
International Conference on Software Maintenance, 1998

[Voa98a]
The Challenges of Using COTS Software in Component-Based Development
Jeffery M. Voas
IEEE Computer, June 1998

[Voa98b]
Certifying Off-the-Shelf Software Components
Jeffery M. Voas
IEEE Computer, June 1998

[Voa98c]
Maintaining Component-Based Systems
Jeffery Voas
IEEE Software, July/August 1998
Problems with Component-Based Software Development

References

[Voa98d]
COTS Software: The Economical Choice?
Jeffery Voas
IEEE Software, March/April 1998

[Wey98]
Testing Component-Based Software: A Cautionary Tale
Elaine J. Weyuker

[ZhoEdw98]
Security Control for COTS Components
Qun Zhong, Nigel Edwards
IEEE Computer, June 1998
10 Bibliography

10.1 Read literature

This section lists the literature that was read in order to identify the theoretical problems. This
literature is not in the reference section because no problems were found in it and therefore not
referenced.

[Aoy98]
New Age of Software Development: How Component-Based Software Engineering Changes
the Way of Software Development
Mikio Aoyama
International Workshop on Component-Based Software Engineering, 1998

[AllFro97]
Leaping the Obstacles of Component Technology
Paul Allen, Stuart Frost
Object Magazine, October 1997

[AT&T96]
The Promise of Distributed Business Components
AT&T Technical Journal, March/April 1996

[Bha98]
Management of Component-Based Software Engineering
Shahzad Bhatti
International Workshop on Component-Based Software Engineering, 1998
http://www.sei.cmu.edu/cbs/icse98/papers/p2.html

[Bro98]
From Component Infrastructure to Component-Based Development
Alan Brown
International Workshop on Component-Based Software Engineering, 1998

[BRS98]
Componentware – The Big Picture
Klaus Bergner, Andreas Rausch, Marc Sihling
International Workshop on Component-Based Software Engineering, 1998

[Car98a]
Evaluation of COTS Products: Some Thought on the Process
David Carney
The COTS Spot, September 1998

[Car98b]
COTS Evaluation in the Real World
David Carney
The COTS Spot, December 1998
Problems with Component-Based Software Development

Bibliography

[CasLia98]
Development Process for the Creation and Reuse of Object-Oriented Generic Applications and Components
Xavier Castellani, Stephen Y. Liao
JOOP, June 1998

[ClaMur98]
Verifying components under development at the design stage: A Tool to support the composition of component design models.
Siobhan Clarke, John Murphy
International Workshop on Component-Based Software Engineering, 1998

[Dig98]
Business Object Component Architecture
Tom Digre

[Fil98]
Injecting Management
Robert E. Filman
International Workshop on Component-Based Software Engineering, 1998

[Han98]
Characterization of Components
Jun Han
International Workshop on Component-Based Software Engineering, 1998

[Ino98]
From Class Libraries to Component-Based Development
Takeshi Inoue
International Workshop on Component-Based Software Engineering, 1998

[Kar98]
Component Development with a Big ‘C’
Dan Kara
Software Magazine, August 1998

[KKKR98]
Inter-Component Communication as a Vehicle Towards End-User Modeling
Manolis Koutlis, Petros Kourouniotis, Kriton Kyrimis, Nikolina Renieri
International Workshop on Component-Based Software Engineering, 1998

[Kru98]
Modeling Component Systems with the Unified Modeling Language
Philippe Kruchten
International Workshop on Component-Based Software Engineering, 1998
Problems with Component-Based Software Development

Bibliography

[LonSea98]
A Comparison of Component Integration Between JavaBeans and PCTE
Fred Long, Robert C. Seacord
International Workshop on Component-Based Software Engineering, 1998

[Mey99]
On To Components
Bertrand Meyer
IEEE Computer, January 1999

[NadRin98]
A Validated Software Reuse Reference Model Supporting Component-Based Management
Nader Nada, David C. Rine
International Workshop on Component-Based Software Engineering, 1998

[Nin98]
CBSE Research at Andersen Consulting
Jim Q. Ning
International Workshop on Component-Based Software Engineering, 1998

[Pas98]
The Promise and Perils of Component-Based Development
Ed Pasahow
Digital Systems Report, Volume 20, Number 1, 1998

[Por98]
Unification of Components and Objects Through Abstractions
Dan Port
International Workshop on Component-Based Software Engineering, 1998

[SchAss98]
Concepts for Developing Component-Based Systems
R. Schmidt, U. Assmann
International Workshop on Component-Based Software Engineering, 1998

[Tai98]
A Connector Model for Object-Oriented Component Integration
Stefan Tai
International Workshop on Component-Based Software Engineering, 1998

[Ueh98]
Component Architecture for Business Application Systems
Sanya Uehara
International Workshop on Component-Based Software Engineering, 1998
Problems with Component-Based Software Development

Bibliography

[WanMac98]
Architectural Components and Object-Oriented Implementations
Guijun Wang, H. Alan MacLean
International Workshop on Component-Based Software Engineering, 1998

[Wu98]
A Component-Based Architecture for Building and Managing Global Information Systems
Xuequn Wu
International Workshop on Component-Based Software Engineering, 1998

10.2 Unread literature
This section lists the literature that was found but not read because lack of time in the master project.

[Jel98]
Component-Based Software Engineering
Thomas Jell
Cambridge University Press, 1998
ISBN 0-521-64821-1

[Kon95]
OTSO: A Systematic Process for Reusable Software Component Selection
Jyrki Kontio
Institute for Advanced Computer Studies and Department of Computer Science, 1995

[Sam97]
Software Engineering with Reusable Components
Johannes Sametinger
Springer, 1997
ISBN 3-54062695-6

[Szy98]
Component Software – Beyond Object-Oriented Programming
Clemens Szyperski
Addison-Wesley, 1998
ISBN 0-201-17888-5