Technical and Organizational Aspects on a Migration from Procedural to Object-Oriented Programming

Fredrik Lundberg

Advisors:
Anders Eriksson
Ericsson Mobile Platforms AB

Lena Karlsson
Björn Regnell
Department of Communication Systems
Lund Institute of Technology, Lund University

August 2003
Technical and Organizational Aspects on a Migration from Procedural to Object-Oriented Programming

Keywords

Object-Orientation, EC++, C and C++ compatibility

Abstract

During the last decade object-oriented programming has gained popularity in the industrial community and many organizations have switched to object-oriented programming. A transition from procedure-oriented to object-oriented programming can give many benefits but is also associated with a considerable risk. The advantage a transition has the potential to give is presented in the report. The risks associated with a transition will also be addressed and, where possible, guidelines on how to minimize them will be given.

A survey of the differences and similarities between C and EC++ is presented. Both the differences in the language specifications and runtime performance are examined. This part attempts to verify that EC++ meet the performance requirements and can be used in the current platform. A brief introduction to object-orientation and the benefits gained when using the object-oriented paradigm is given.

A new language will not only have an impact on the product and raise technical issues. A transition to object-orientation will also affect the organizational structure and the way people work. When a significant change takes place, a company will go through a period of stress before the change has been carried out and the situation will be stable again. Some measures must be taken to reduce the negative effects of the stress during the change phase. The change also means that new knowledge must be gained and spread throughout the company.
Contents

1 Abbreviations 4
2 Introduction 5
2.1 Background 5
2.2 Objective 5
2.3 Method 6
2.4 Outline 6
3 Software engineering & development 9
3.1 Waterfall Model 9
3.2 Spiral Model 10
3.3 XP 12
3.3.1 Key concepts of XP 12
3.3.2 Shortcomings of XP 12
3.4 EMP process model 13
3.5 UML 13
4 Object-orientation 13
4.1 Long-term goals 13
4.2 Object oriented design characteristics & features 15
4.2.1 Inheritance 15
4.2.2 Aggregation 15
4.2.3 Encapsulation 16
4.2.4 Reliability 16
4.2.5 Reusability 16
5 Object orientation in real life 16
6 Reengineering 17
6.1 Restore structure in old code 17
6.2 Benefiting from new insights 19
6.3 Adapting to future conditions 19
6.4 Effects of low staff turnover 19
6.5 Refactoring 19
6.6 Reengineering tools 20
7 Vindicating a restored structure 20
7.1 Coding standard 21
7.2 Object-Oriented design metrics 21
7.3 Documentation 22
8 Changes in organizations 23
8.1 Factors that often increase the resistance to changes 23
8.2 Characteristics of successful changes: 24
8.2.1 Establish support for the change early 24
8.2.2 Perceiving the change as an opportunity 24
8.2.3 Motivating the change 24
8.2.4 Confidence in new technology 24
8.2.5 Support from management and gurus 25
8.2.6 A motivated leader 25
8.2.7 Start small and show immediate results 25
8.2.8 A need of changes is perceived 25
9 Technical aspects on porting C to Embedded C++ 27
9.1 Embedded C++ 27
9.2 Embedded C++ compared to C 27
9.3 Embedded C++ compared to C++ 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>OSE Delta operating system and Embedded C++</td>
<td>29</td>
</tr>
<tr>
<td>11</td>
<td>ARM and Embedded C++</td>
<td>29</td>
</tr>
<tr>
<td>12</td>
<td>Development environment and Embedded C++</td>
<td>29</td>
</tr>
<tr>
<td>12.1</td>
<td>Lauterbach Jtag debugger for ARM</td>
<td>30</td>
</tr>
<tr>
<td>12.2</td>
<td>IAR Embedded Workbench</td>
<td>30</td>
</tr>
<tr>
<td>12.3</td>
<td>EMP specific limitations</td>
<td>30</td>
</tr>
<tr>
<td>12.3.1</td>
<td>Macro</td>
<td>30</td>
</tr>
<tr>
<td>12.3.2</td>
<td>new – delete</td>
<td>30</td>
</tr>
<tr>
<td>12.3.3</td>
<td>Enumerations</td>
<td>31</td>
</tr>
<tr>
<td>12.3.4</td>
<td>ECM</td>
<td>31</td>
</tr>
<tr>
<td>13</td>
<td>Impact on efficiency</td>
<td>32</td>
</tr>
<tr>
<td>13.1</td>
<td>Initial testing</td>
<td>32</td>
</tr>
<tr>
<td>13.1.1</td>
<td>Creating classes</td>
<td>32</td>
</tr>
<tr>
<td>13.1.2</td>
<td>Creating larger classes</td>
<td>33</td>
</tr>
<tr>
<td>13.1.3</td>
<td>Call to virtual function</td>
<td>33</td>
</tr>
<tr>
<td>13.1.4</td>
<td>Call to class function</td>
<td>34</td>
</tr>
<tr>
<td>13.1.5</td>
<td>Direct call from C/C++ to C++/C</td>
<td>34</td>
</tr>
<tr>
<td>13.1.6</td>
<td>Creating a vector of objects</td>
<td>35</td>
</tr>
<tr>
<td>13.1.7</td>
<td>Sending OSE signals</td>
<td>35</td>
</tr>
<tr>
<td>13.1.8</td>
<td>Interface between architectural modules</td>
<td>35</td>
</tr>
<tr>
<td>13.1.9</td>
<td>Linking C++ code with third party object code written in C</td>
<td>35</td>
</tr>
<tr>
<td>13.2</td>
<td>Main test</td>
<td>36</td>
</tr>
<tr>
<td>13.2.1</td>
<td>Size of executable</td>
<td>37</td>
</tr>
<tr>
<td>13.2.2</td>
<td>CPU time</td>
<td>37</td>
</tr>
<tr>
<td>13.2.3</td>
<td>Understandability</td>
<td>37</td>
</tr>
<tr>
<td>13.3</td>
<td>EC++ conclusions</td>
<td>38</td>
</tr>
<tr>
<td>14</td>
<td>Semi structured interviews</td>
<td>38</td>
</tr>
<tr>
<td>14.1</td>
<td>Interview setup</td>
<td>39</td>
</tr>
<tr>
<td>14.2</td>
<td>Interview outline</td>
<td>39</td>
</tr>
<tr>
<td>14.3</td>
<td>Opinions about C++ / EC++</td>
<td>40</td>
</tr>
<tr>
<td>14.4</td>
<td>Knowledge and experience from object-oriented programming</td>
<td>40</td>
</tr>
<tr>
<td>14.5</td>
<td>Resistance to change</td>
<td>41</td>
</tr>
<tr>
<td>14.6</td>
<td>Perceived transition risks</td>
<td>41</td>
</tr>
<tr>
<td>15</td>
<td>Estimation of reengineering effort</td>
<td>41</td>
</tr>
<tr>
<td>16</td>
<td>Estimation of software aging</td>
<td>42</td>
</tr>
<tr>
<td>17</td>
<td>Different approaches to reengineering</td>
<td>44</td>
</tr>
<tr>
<td>17.1</td>
<td>Outline of different approaches</td>
<td>44</td>
</tr>
<tr>
<td>17.2</td>
<td>Approach suitable for EMP</td>
<td>44</td>
</tr>
<tr>
<td>17.3</td>
<td>Maintaining quality and reliability during changes</td>
<td>45</td>
</tr>
<tr>
<td>18</td>
<td>Effects on the development process</td>
<td>45</td>
</tr>
<tr>
<td>18.1</td>
<td>Object-Oriented testing</td>
<td>45</td>
</tr>
<tr>
<td>18.2</td>
<td>Code Reviews</td>
<td>47</td>
</tr>
<tr>
<td>19</td>
<td>Introducing a new programming language</td>
<td>48</td>
</tr>
<tr>
<td>19.1</td>
<td>Issues that must be resolved</td>
<td>48</td>
</tr>
<tr>
<td>19.2</td>
<td>Training</td>
<td>50</td>
</tr>
<tr>
<td>20</td>
<td>Discussion</td>
<td>50</td>
</tr>
<tr>
<td>20.1</td>
<td>Problems and further work</td>
<td>50</td>
</tr>
<tr>
<td>20.2</td>
<td>Conclusions</td>
<td>51</td>
</tr>
</tbody>
</table>
1 Abbreviations

C  Procedure-oriented language
C++  Object-oriented language
COM  Component Object Model

Language-independent binary level standard defining how components can interact with each other developed by Microsoft.

EC++  Embedded C++
ECM  Ericsson Component Model

The Ericsson Component Model is a subset of COM.

EMP  Ericsson Mobile Communications AB

PROPS  Development process created by Ericsson.

UML  Unified Modeling Language

Graphical language model language used to describe software.

XP  Extreme Programming

Programming methodology explained in section 3.3.
2 Introduction

2.1 Background

Object-oriented programming was introduced in the late 60s but got little attention outside the academic community at first. This has however changed and the situation is quite different today. Throughout the last decade, object-oriented programming gained popularity outside the academic community and was adopted by many organizations. It is believed that the object-oriented paradigm can give substantial advantages over procedural programming and help organizations to produce software with higher reliability at a lower cost. Although the advantages of object-oriented programming compared to procedure-oriented programming are widely accepted there is little hard evidence of this. Since there is a growing interest about object-orientation within EMP, an attempt to quantify the advantages of object-oriented programming is interesting. It is also interesting to investigate if switching to object-orientation is a suitable approach for EMP and how such a transition could be implemented.

This situation is hardly unique and many organizations have already faced this question. Unfortunately, no articles concerning the information that has formed the basis of other organizations decisions have been published. Neither has any information about the outcome from such changes been published. Therefore EMP cannot rely on case studies from other organizations but has to make a study of their own. Even if such reports had been available some work would have been necessary to find and solve organization specific issues.

2.2 Objective

The main goal is to investigate if switching to object-oriented programming can help EMP to achieve its mission:

"To make our customers first, best and profitable through innovation, quality and commitment."
This is important to keep in mind, the question at hand is not whether object-orientation is “cooler”, more modern or more academically correct but if it can help EMP to become even more competitive. There is not a simple answer to this question. Switching to a new language might appear to be a simple task only affecting “programming” but as we will see this is not the case. A transition from procedure-oriented to object oriented programming would have a deep and profound impact on the whole organization. Not only technology but also the way people work will be affected by the change. Thus, the organization will face a period of stress, both technological stress and organizational stress. This makes it crucial to take both organizational and technical aspects into account when evaluating the costs and benefits from a transition. This report will attempt to answer the question in three parts. We verify that is possible to change language from C to EC++, we capture the advantages of the object-oriented paradigm and finally we discuss how to carry out the transition.

2.3 Method

As the objective of the study is to predict the outcome of a transition to object-oriented programming, a crucial part of the report is to find all items in the technical and organizational systems that will be affected by a change. Information about the technical system has been gathered through informal interviews with employees at EMP and document analysis. After an area that will be affected by the change has been identified, the next step is to find out how the area will be affected and if there can be a problem. The second step has been performed in two ways. Where available, literature about the issue at hand has been read and, where appropriate, a test verifying that the area examined will work in an object-oriented environment has been done.

The organizational aspects have been addressed by gathering information from literature about problems normally encountered during change phases. Then these problems have been put into the context of a language transition. An understanding of these common problems importance at EMP has been gained through interviews with employees at EMP.

2.4 Outline

The report has three main sections:

1. The first part gives a brief introduction to software engineering, and organization theory. The software engineering sections focuses on the object-oriented paradigm and its advantages compared to the procedure-oriented paradigm but process models and reengineering is also explained. The section about organization theory describes problems that frequently occur during changes in organizations. Ways to minimize the negative effects of the problems are also described.
2 In the second part technical and organizational aspects of a transition is covered. EC++ is compared to both C and standard C++, emphasized on the syntactical differences. Two feasibility tests were performed. In the first, small pieces of EC++ code is run in order to measure the performance of EC++. In the second test, a minor part of the system is rewritten in EC++ and integrated with the rest of the system. This section also examines the tools currently used at EMP and verifies that they can be used after a transition to EC++. Results from the semi-structured interviews and other experiments are also included in this section.

3 The third part describes the effects a change will have on the design of the system and on the software development process. This part also describes the effect a transition from procedure-oriented to object-oriented programming will have on the organization, and the way the staffs work is examined. The results from the experiments are also discussed in this section.
Part 1

Part one covers the theory needed to understand both the technical and non-technical problems of a transition from procedure-oriented to object-oriented programming and design.

The section is based on books and papers about software engineering and organization theory.
3 Software engineering & development

As one of the objectives of this report is to explain how the current development process will be affected by a transition from procedure-oriented to object-oriented programming, a basic understanding of software engineering and software development processes is needed. This section will also provide a context for some of the issues discussed in section 7. If the reader is already familiar with these topics it is possible to skip most of this section but section 3.4 about the process model used by EMP could still be interesting. First, a brief survey of development processes currently used in industry and advocated by researchers is given. Then, an overview of the development process currently used at EMP is provided.

This chapter does not give more than a brief summary of a broad and complex field and the interested reader is recommended to consult a book dedicated to this topic such as [1]. Three models are presented here, the waterfall model, the spiral model and XP. The waterfall model is interesting since it is has a lot in common with PROPS, a process developed by Ericsson (see 3.4). The spiral model is discussed since it explicitly recognizes risks in the development process. XP is the youngest of the three models and is interesting as it addresses problems with the waterfall model and spiral model and offers some interesting ideas.

3.1 Waterfall Model

The waterfall model is the oldest software process model. It was originally introduced in the early seventies in an attempt to make the software process more visible and easier to manage. The idea is to divide the whole development effort into smaller phases and deliver a document after each phase is finished. The document from the previous phase will form the basis for the next phase. After a phase is finished the result has to be validated, tested or verified. Having good methods for inspection (see 18.2), testing (see 18.1), and metrics gathering (see 7.2), will be an essential requirement to do this. Detailed documentation of not only the design but also the entire development process will be crucial to have in the maintenance phase. It is important to remember that the maintenance phase is usually both the longest and most expensive phase [1,2].

Ideally the project team starts with the user requirements, moves from one phase to the next, and finishes the product in one sequence. In reality this is often not the case, instead it is frequently necessary to go back one or several steps and change the document from a previous phase. Since the changed document formed the basis for the following phases much rework can be the result, particularly if it was one of the early phases that had to be redone.
3.2 Spiral Model

At the same time as the division of the entire effort into smaller phases made the development process more visible and easier to manage it also made it less flexible. Thus is the waterfall development process vulnerable to both errors in earlier phases and changed requirements. In an endeavor to combine the advantages of a visible process with a flexible process, Boehm developed the spiral model [3]. Basically the spiral model can be thought of as a sequence of waterfall processes after each other. This allows the high-risk areas to be solved first and when they are stable low, risk areas can be implemented. The first task in each lap is a risk analysis. The result from that analysis is used to plan the work for that phase in a way that minimizes the risk of failure. Besides making the process visible, the spiral model makes the risks in the development effort visible.

Another advantage is that it is possible to use different processes for each lap in the spiral. One example could be using a rapid prototype model for the first lap to verify the requirements and then use a waterfall model for the main development in the next cycle.
The iterative nature of the spiral model offers some interesting advantages since the software developed at EMP is constantly modified according to changed and added user requirements.

![Figure 2 The spiral model [3]](image-url)
3.3 XP

The development of XP started in the mid nineties as a reaction to the process models used at the time. The people behind XP felt that many of the models had grown too big and contained so many rules that it was hardly possible to follow all of them. Instead, they proposed a process model with a much smaller set of rules and a less well-defined process. Their point is that it is better to have a less rigorous model with fewer rules that are not broken than a rigorous model with a large set of rules that might be broken every now and then. This section is based on information provided by [4, 5].

Although the XP approach is not suitable for an organization of EMP’s size there are some interesting ideas in the XP concept.

3.3.1 Key concepts of XP

Some of the concepts that differentiate XP from the waterfall and spiral model are given below, but keep in mind that this is not a full list of the XP concepts.

- **Frequent testing** Most of the test cases are automated and run often.

- **Pair programming** Programming in pairs is claimed to be as efficient as two programmers working on separate tasks at the same time as better code is produced.

- **Frequent integration** Instead of one big integration phase is the code produced by each pair integrated often. This is claimed to reduce integration problems.

- **Daily-stand up meeting** One brief meeting with all participants is held daily to keep everyone updated.

- **Small releases** Small but frequent releases are made and accepted by the customer, which reduces risk in each release and give fast feedback.

- **Break the rules** If the general rules don’t work in a particular situation they can be broken, but not before everyone has agreed on the change and the new rules are established.

3.3.2 Shortcomings of XP

The written documentation in XP is very limited and the project organization is less well defined than it is in the waterfall model. At the same time as the low level of formal organization allows great flexibility for small groups it also means that the model is difficult to use for larger teams. The model also assumes that the team has constant access to the customer, in many cases this is not the case.
3.4 **EMP process model**

The development projects at EMP are currently planned according to the PROPS model. PROPS is an Ericsson wide process model based on experiences learned from projects carried out at Ericsson. As it is not possible to have one model that suits all projects at Ericsson PROPS isn’t a detailed process model. Instead PROPS can be looked at as a framework that is tailored to suit the needs of different projects and departments.

The PROPS model has a lot in common with the standard waterfall model and the pros and cons for the waterfall model are also valid for the PROPS model. [6]

3.5 **UML**

During the design phase a high level language such as UML is probably needed. UML is not a part of the actual process but a useful tool when developing and communicating the design. Providing a tutorial to UML is out of the scope for this paper and we only conclude that if a transition to an object-oriented language is to be successful an object-oriented high-level language is also needed.

4 **Object-orientation**

4.1 **Long-term goals**

An organization’s long-term goal for any successful program is to keep the program successful and profitable. In order to achieve this goal the program must evolve according to the constantly changing requirements that a successful program must fulfill. The changes a program will face include hardware changes, new operating systems and new or changed requirements. This means that the program must be easy to update, without losing old functionality or introducing new problems. It must also be robust enough to cope with such changes without making future changes more difficult to implement. It is widely believed that the object-orientated paradigm can be very helpful when creating software that has the characteristics mentioned above. Some of the attributes frequently associated with object-oriented software are:

- **Lower maintenance costs** It is believed that object-oriented programs are easier to maintain and improve. The reason for this is that object-oriented software consists of more independent modules, which means that a modification only affects a minimal part of the system [1, 7].
• **Improve reliability** Reusing old and well-tested components instead of writing new code can improve the reliability of the product. The old code is familiar to the developers and they know its strong and weak sides. Thus they are less likely to introduce interaction problems when using the component. It is also possible that they have learned how to use the component in order to optimize performance. Besides this, it is also less likely that an old and tested component contains errors compared to a new component, if the maintenance has been done with great care. As discussed in section 6.1 it is also possible that software becomes less reliable with time if it is changed often. Reuse is not an intrinsic part of object-orientation but it is often easier to reuse object-oriented code than procedure oriented (see section 4.2.5). Encapsulation (see section 4.2.3) is also believed to improve reliability because the state and data in an object is protected from uncontrolled access.

• **Reduce cycle-time** In addition to improved reliability; it is also possible to decrease programming time simply by writing less code through reuse. Instead of rewriting code, the time can be spent on other tasks. A better design due to the object-oriented paradigm can also ease the work and speed up development.

During the last decade, a general transition from procedural to object-oriented programming has taken place. The transition started in the academic world but has now taken place in the industry too. Although procedural programming will remain important for a very long time, the effects of the general transition towards object-oriented programming should not be underestimated. One example of such an effect is requests from customers requiring an object-oriented interface. Supplying customers with an object-oriented interface would be easier if the underlying architecture of the system had been object-oriented. The interface is described in section 12.3.4.

Another and potentially even more important effect is the transition from procedure to object-oriented programming in the academic world. This transition means that most graduates from the universities only have knowledge and experience from object-oriented programming. Thus it will become increasingly difficult to recruit personnel with a background in procedural programming and resources must be spent on training new employees in procedure-oriented programming. In addition to this many of the advances that has been made in the software engineering and software management field are based on the object-oriented paradigm. In order to get the full benefit of these advances a transition towards object-orientation is needed.
Reengineering the system from a procedural to an object oriented design has yet another advantage, and a potential risk. As the program has evolved during its lifetime it is also likely that the design has degraded. As new features are added and errors are removed the immediate result is improved software, but the long run effect of numerous modifications can be a degraded structure. If the original structure degrades, the system is likely to become more and more difficult to maintain. The reengineering process is unfortunately associated with a risk, the transition is complicated and many efforts like this have failed. This issue is further explored in section 17, Reengineering.

The long-term goal is to write good software and the question is if this goal will be easier to reach if an object-oriented language such as EC++ is used.

4.2 Object oriented design characteristics & features

4.2.1 Inheritance

Inheritance is a powerful tool and one of the most important features in an object-oriented language but it can also be dangerous if used without care. Inheritance makes it possible to create a base-class with general functionality and sub-classes that use this functionality without actually implementing it. This can be very helpful in many cases, it is frequently convenient to use a general concept such as a shape and not worry about if it is a square or a triangle. Inheritance can also be a convenient way to reuse old code. It is possible to add both new functionality and change the behavior of the old functionality. Inheritance can improve understandability, reliability and adaptability of software but it can also do the opposite. If inheritance is used incorrectly it can be very difficult to understand how the program works.

4.2.2 Aggregation

Aggregation is often a good substitute for inheritance. While inheritance should be used if the relationship between two classes is of the type “is a” aggregation is used if the relation is of the type “has a”. A car is a vehicle and thus should car be a subclass to vehicle. A car is not an engine, instead it has an engine, in this case aggregation should be used. In some cases it is less obvious if one should use inheritance or aggregation and the choice might even change over time.

Aggregation is implemented by putting one of the classes as an attribute in the other class, in the example above “engine” would be an attribute in “car”.


4.2.3 Encapsulation

Encapsulation is intended to separate behavior from implementation. Each module should have an implementation independent interface that does not change even if the internal representation of the module is changed. This eases modifications since one module can be replaced with another without any side effects, provided that they fulfill the same interface. Encapsulation also means that the data that belongs to one module is protected from the modules it is interacting with. This means that the state of an object can be protected and all changes to it must be done through functions defined in the interface.

4.2.4 Reliability

Reliability is not an inherent feature of object-oriented software, it is certainly not true that an object-oriented design automatically guarantees reliability. Nonetheless are there several features in the object-oriented paradigm such as encapsulation that can contribute to more reliable software. In order to take full advantage of the object-oriented paradigm a deep understanding of object-orientation is needed.

4.2.5 Reusability

Reusing old code is nothing new, functions has been reused for a long time e.g. the string manipulation functions in C and mathematical libraries. Reuse on this level is quite possible in an object-oriented design too but now more powerful ways to reuse old code is available too. Instead of just reusing an old function, a set of functions and data containers associated can be reused. Reusing an object instead of independent function should reduce integration problems. In [12] it is shown that reuse can improve quality of the system in terms of lower defect rates. In their study the lowest defect rates are reached for components with a high degree of reused code.

5 Object orientation in real life

Although object-oriented programming has been around for quite some time now there is still not a lot of solid evidence that demonstrate the advantages of object-oriented programming. In [8] the authors state that their organization has been able to reduce development time after switching to an object-oriented language it is however difficult to find success stories from the industry. It is not known if this shortage depends on companies being reluctant to publish such material or if there is a true shortage of successful transitions to object-orientation.
It is somewhat easier to find material from the academic community, experiments that attempt to investigate the difference between object-oriented programming and procedural programming has been performed at universities [9, 10, 11, 12]. These experiments don’t give a straight answer however, some of them indicate that object-orientation achieves its goals and shows that object-oriented software is indeed more maintainable than procedure-oriented software. Other claims that procedural programming is just as good and in some cases even better than object-orientated programming. When evaluating these results one must keep in mind that it is hard to tell if the results from such experiments are valid in an industrial setting. Some problems in the experiments are:

- **Project size** The projects used in the experiments are not very large. This could be a disadvantage for the object-oriented style, if the program is fairly small a functional design is pretty easy to grasp and the complexity of object-orientation is avoided. For a larger project the situation might be the reverse.

- **Skill** The participants in the experiments have been students, mostly undergraduates but also some graduates. In some of the experiments the subjects has been untrained in object-oriented developing.

- **Time frame** The experiments has been performed as “one time tests”. None of them has tried to determine what the effects would be over a longer period.

The experiments indicate that object-oriented programming doesn’t automatically give better software but can be a helpful technique if it is used correctly. One thing that could explain why it is easy to misuse object-orientation is the introduction of new design features; there are more features in object-oriented languages than most procedural languages. This means that it can be more difficult to learn and understand an object-oriented language than a procedural language. Besides being languages difficult to grasp it is probably even more difficult to learn how to design in object-oriented languages. This means that it might be discouraging for an experienced procedural programmer to switch to object-oriented programming. It also means that sufficient training must be offered all employees that are going to start programming in an object-oriented language.

### 6 Reengineering

The transition from a procedural to object-oriented design allows the developers to restore the structure in the old code and take advantage of the lessons learned in the past when it is redesigned. This advantage must also be considered when deciding if a transition is beneficial or not.

#### 6.1 Restore structure in old code
Over the years the code currently used has been improved over and over again, bugs have been found and removed and new functionality has been added. This work has resulted in a proven system when it comes to both fulfilling explicit functional properties and non-functional properties such as performance and tuning settings. However, nothing comes for free, so the total effect of all the small changes that have been made might be a brittle structure that is increasingly difficult to change. Two common problems with a brittle system are that solving a problem frequently takes longer time than necessary and a new problem might be introduced when the old bug is removed. Yet another problem is that some of the assumptions that influenced the original design might not be valid anymore, they might even be in conflict with assumptions valid today. This could result in complicated solutions that will be difficult to maintain for two reasons. First, the code might be complicated and difficult to understand. Second, the person that updates the code might not be familiar with the underlying problems and make erroneous changes. This phenomenon is called software aging and is discussed in several research papers [23, 24]. An estimation of this phenomena indicates that a system age rapidly. According to the results in [25] implementing a change got 20% harder to make per year. The study performed in [23] shows significant signs of decay such as more complicated changes and error introduction. This indicates that some resources should be dedicated to maintaining the structure of the system. The purpose of such maintenance is not to change the behavior of the system but to improve the structure and ease future changes of the system. A restored structure might give better performance and a more robust system even if it is not the main objective.

It is important to understand that the goal of a reengineering effort is not to make individual pieces of source code easier to read but to consider the entire system. Improving the readability of the source is a good start but one must not stop here, instead the work should continue and attempt to improve the general structure and design of the entire system.

Figure 3 Software might decay and need reengineering
6.2 Benefiting from new insights

As new functionality is added and the conditions that influenced the original design become obsolete the original design might not be the optimal choice anymore. For example could an algorithm or data structure that was highly efficient with the original hardware be less efficient if cache sizes, memory bus width or similar hardware dependent assumptions change. Redesigning the product would also allow the designers to take advantage of the experiences gained from the old design. Thus, it should be possible to both learn from past mistakes and make the new design more suitable to current requirements and future development.

6.3 Adapting to future conditions

The requirements and underlying technical assumptions are not the only things that are changing, people are changing to. New graduates from universities become less and less likely to have education in or experience from procedure-oriented programming and design. Teaching people with an object-oriented background to program in a procedure-oriented manner will require some training during the first part of their employment. This means that it will take longer time before they become efficient programmers at EMP and contribute in the development process. It also means that they will require assistance from experienced employees and take up some of their time. Although this will cause an implicit cost it is most probably not the most difficult part of the new employees training. In addition to programming in a procedure-oriented language they must also be trained in procedure-oriented design. Unless they are given proper training in procedure-oriented design, there is a significant risk that their code will be unnecessary difficult to maintain. This part of the training could very well turn out to be expensive and time consuming but if this cost is avoided, the indirect costs might be even higher. The long-term effect of poorly designed implantations is high maintenance costs.

6.4 Effects of low staff turnover

Some parts of the staff have been working at EMP for a long time and have a thorough understanding of the software. It is absolutely crucial to get those individuals involved in the change effort as early as possible. If they take an active part in the planning of the change process and contribute with their knowledge, the risk of failure decreases. They will also play an important role as champions, see section 8.2.5. If they don’t take an active part in the change process, important decisions about the change process has to be made without access to the best information. It will also be difficult to establish confidence in the change process without the support from champions.

6.5 Refactoring

One way to implement a reengineering effort is refactoring. In [28] refactoring is described as:
“Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.”

The idea is to improve a degraded structure in several small steps. Each individual step doesn’t have a significant impact but the cumulative effect is significant. In [28] the author assumes that the original structure is object-oriented and all examples are based on object-oriented code. Although this means that their catalogue of refactorings doesn’t apply directly, it doesn’t mean that the general concept of refactoring isn’t interesting even if the old code is procedure-oriented.

6.6 Reengineering tools

There are some tools intended to automate as much of the work as possible during reengineering efforts. The tools have slightly different purposes, some are intended to aid the developer when making small and medium sized changes in the code [26] others are intended to find parts of the code that could benefit from reengineering [27]. Yet another category of tools is used when major changes in the design and structure of the program is undertaken [2].

Each type of function mentioned above could be useful, but unfortunately there doesn’t seem to be a tool that covers more than one function. If the tool that find pieces of code suitable for refactoring could have been used to perform the actual work it would have been far more useful than two separate tools. Another problem is that they aren’t a part of the standard development environment and switching between different tools is often perceived as tedious and therefore avoided [28].

The tools intended to aid when the code is redesigned is probably most useful in aiding the understanding of the code. Some tools include features that automatically find class candidates in procedure-oriented code. These features appear to be difficult to use and have poor performance.

This is an interesting field, automating tedious work allows developers to focus on more challenging tasks and it also reduces the risk of inserting bugs. However, this field is still rather young and there do not seem to be any commercial tools available.

7 Vindicating a restored structure

The strive towards a better design must not end after the actual transition has been made. Instead the design of the system must be kept in mind at all times. When a modification of the system is done the negative effects the change will have on system structure must be minimized. Preferably, a change made today should not make future changes harder. A company-wide coding standard, design metrics and documentation are important not only during the development phase but during the maintenance phase.
7.1 Coding standard

Defining and using a coding standard has two main advantages. First, the code will be easier to read, maintain and debug because a common style is used and error prone programming styles can be avoided. Second, the coding standard can also be a useful tutorial that gives advice on how common problems are best solved. This should be helpful particularly during the first part of a transition from C to EC++ when programmers are still gaining knowledge about EC++ and familiarity with object-oriented programming but also an important guide for new employees.

There are currently two company-wide coding standards for C++ in place at Ericsson [30, 32]. As these standards are intended for standard C++, some changes will be necessary before they can be successfully used at EMP. These changes should consider the differences between C++ and EC++, that we are migrating from old C code and the fact that the code is intended for embedded applications. There is also a C coding standard for EMP available [31]. If there is a conflict between the C++ standard and the C standard, the C++ standard should be determining.

In addition to the Ericsson company-wide standards, a book such as [32] that give more detailed explanations on why certain programming styles should be avoided or encouraged could be used. Although the coding rules are defined by the Ericsson standard, such a book could be helpful in providing proven programming concepts.

7.2 Object-Oriented design metrics

The coding standard provides guidelines for how the code should be written but it is not always easy to live by the book. Providing the developer with a set of metrics and tools that collect the metrics automatically could be a convenient way for the developers to see if their code actually follows the style provided by the guidelines. If the metric suite highlights a certain class, it could indicate that the class should be redesigned and would be easier to understand and maintain afterwards, but it could also be because the class is inherently complicated. If the case is that the module is as simple as it can be, the metric suite can still help the testers to find modules that have similarities with modules that have proven to be error prone in the past.

Chidamber & Kemerer suggest one suite of object-oriented metrics in [33]. Their metric suite was examined by Basili, Briand and Melo in [34], according to that validation, the metric suite suggested by Chidamber & Kemerer was useful.

A metric suite on its own is not very helpful; the suite must be combined with a tool. Unless a tool, that gathers the information and presents in an easily accessible way, is available it will be too difficult and expensive to gather and analyze the data.
Although a standard suite of metrics is a good start but in order to get the most out of this approach, the suite probably has to be tailored to the specific needs at EMP. Particularly should constraints associated with real-time applications and embedded systems be measured by the metrics.

It should also be clear that metrics are to be used by the developers to aid them in their help and not a tool for managers to supervise and evaluate the staff. Even if the data and result generated from the metric tool are available to everyone it must be understood that the result is not used to grade the code nor is it used to grade the staff.

7.3 Documentation

Documentation has two purposes; it should aid the implementation phase and the maintenance phase of a software project. As simple as it sounds these goals are often not reached and many software projects suffer from inferior documentation. Investigations indicate that in software projects, particularly during the maintenance phase, documentation was one of the main problems during the 70s, 80s, 90s and continue to be so today [35].

Thus one cannot say that the requirements on documentation are commonly met today and the requirements will be even tougher to meet in the future. Common reasons given for the poor documentation today are tight schedules and high cost to both develop and maintain the documentation. Although documentation is believed to be cost efficient in the long run adequate resources are not always allocated for documentation. An explanation to this that focus is often on the short-term costs and activities that will reduce costs in the long run, such as documentation, are neglected. These problems are not likely to go away but they can be overcome if focus is directed on long-term goals.

In addition to current problems, a transition from procedure-oriented to object-oriented programming will introduce even higher demands on documentation quality. Reuse of code from previous projects is one of the main drivers for the object-oriented paradigm and it is hard to see how this goal can be reached without excellent documentation. Another reason for an increased need for documentation of is the delocalized nature of object-oriented software. As mentioned in section 16, a common problem with object-oriented programming is the increased complexity. The complexity is caused by functionality being separated into smaller pieces, possibly in different files, instead of one continuous piece of code. Documentation for object-oriented software must not only explain how each part of the program works in isolation but also take this delocalization into account and also explain how each part is interacting with the rest of the system. Delocalization is not only present in object-oriented software but e.g. inheritance and composition makes it easy to introduce high delocalization in object-oriented software. [36]. Careful documentation of implicit dependencies will make it much easier to understand and update the software in the future.
Documenting the underlying design decisions will also make it easier to avoid introducing errors when implementing new functionality or using old code in a new context.

8 Changes in organizations

When a major change is carried out, a large part of the staff is affected by it. Some might be involved in the implementation of the change. Others might not work directly with the implementation of the change but still have to change the way they work because of it. Others might not be affected by the change at all but still depend on the outcome of the change. Thus it is important to see that the change is not only a technology issue, it will also be an organizational issue. This section will explore organizational problems frequently encountered when implementing changes in organizations.

8.1 Factors that often increase the resistance to changes

Even if the change will give significant benefits it is not always easy to make the change happen. Some questions that are known to increase the resistance against changes are listed below [13]. If these issues are not given attention, it will be difficult to get full support from the staff.

- **Radical changes** Radical changes are difficult to carry out since people often prefer to stick with an old system rather than switching to a very different and unknown system.

- **Sudden or unexpected changes** It is often difficult to gain support for a change if it is introduced suddenly.

- **Potential negative effects for individuals** Many changes will alter the organizational structure and this can be a threat to some individual’s positions.

- **There is no clear motivation** There must be a clear reason to why it is desirable to leave the current state and move to a somewhat unknown new state.

- **Confidence in the old structure** If there is a very high confidence in the current system it is more difficult to motivate the change.

- **Similar changes in the past has not been successful** If there have been unsuccessful changes in the past, the staff are less likely to have faith in the new change.
8.2 Characteristics of successful changes:

There is no bulletproof plan to implement changes but some factors that have been present in successful changes are given in the literature [13] and a brief introduction is given here. Sections 8.2.1 - 8.2.4 discuss some of the issues raised in the previous section.

8.2.1 Establish support for the change early

If the staff is involved at an early stage in the planning of the change process it is easier to gain support for the change and the risk of failure decreases significantly. Deciding the point from which the staff should be involved is not obvious, at the same time as early involvement reduces the risk for failure it will also require more work and time. A guideline here is the impact the change will have on the staff, if the change is profound early involvement is best.

8.2.2 Perceiving the change as an opportunity

It is absolutely necessary to avoid a situation where the staff feels threatened by the change. If employees feel that the change will harm their position and career they are very unlikely to contribute to a successful change. Instead management must ensure that the members of the staff feel that they are an important part of the change and have control over their own situation. A significant part of this is to involve the staff in the change work early, if a small project management team plans the change and then force it upon the staff, the risk of failure will increase. Instead the staff should be involved in planning from the beginning and have a chance to affect their own part in the change work.

8.2.3 Motivating the change

The motivation for the change has to be clear to everyone involved in the switch from the old system to the new system. The motivation should form the foundation for the change process. If the motivation isn’t understood and agreed on by the participants, it will be difficult to implement the new system. There might even be a formal decision to implement it and the switch might even be formally implemented but in reality little has changed.

8.2.4 Confidence in new technology

Confidence in the new technology’s ability to not only meet but surpass the existing technology must be established. One way to do this is to use the “start small and increase approach” mentioned in 8.2.7. Another way to do this is to show success stories from other organizations that has benefited from the new technology. In addition to this it is also essential that they are introduced to the new language early, otherwise it is likely that they will feel that there is no room in the organization for them after the change.
If confidence in the new technology is established it will make it easier to both motivate the change and make the change less radical.

8.2.5 Support from management and gurus

In [14] three types of participants particularly important during the change phase is mentioned: change agents, champions and sponsors. The change agents are leading the actual work and guide the other participants through the change. Their most important tasks are to find out what parts of the system that will be affected, handle resistance to the change and make sure that the needed skills are available. Champions are people with significant experience from the technologies being changed and regarded as experts on the system. Their main task is to advocate the project and reduce the uncertainty that some participants might have and increase their faith in the project. Sponsors are the managers that allocate resources to the project and in addition to that they must also work hard to make the participants comfortable during the change.

8.2.6 A motivated leader

In [13] two types of leaders are described, the “icebreaker” and the “great leader”. The icebreaker does not advocate a particular change, instead he creates an atmosphere that encourage innovation and new ideas. In this atmosphere the staff will generate new ideas and suggest changes they feel are needed. The great leader on the other hand has a clear vision of where the organization is moving and actively works to move it in that direction. The leader could be both the change agents and the sponsors mentioned in the previous chapter. Sponsors are more likely to act as icebreakers while change agents have more in common with the great leader role.

8.2.7 Start small and show immediate results

Although it will take time to perform the entire change it is preferable if some benefits can be shown quickly. If this can be done it is easier to get continued support from management because they can see that change actually give benefits. Early successes are also important symbols during the change phase since they make it easier to gain confidence in the change process.

8.2.8 A need of changes is perceived

If the staff in the organization feel that there is a pressing need for the change it is easier to get their full support. Illustrating the advantages of the change is important as we saw in section 8.2.3, and this could be combined with showing the disadvantages of not changing. Showing that not changing also introduces a risk can reduce the fear of changing.
<table>
<thead>
<tr>
<th>Prepared (also subject responsible if other)</th>
<th>No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fredrik Lundberg</td>
<td>Uen</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Approved</th>
<th>Checked</th>
<th>Date</th>
<th>Rev</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>6/23/03</td>
<td>PA1</td>
<td></td>
</tr>
</tbody>
</table>

**Part 2**

Part two contains the results from the performance tests and the interviews carried out during the work with the thesis.
9 Technical aspects on porting C to Embedded C++

9.1 Embedded C++

Embedded C++ (EC++) is a subset of standard C++ defined by a committee consisting of several corporations that are developing embedded applications and hardware. Their goal has been an object-oriented language suitable for embedded applications. Thus CPU consumption and memory efficiency has been a major concern. In addition to efficiency they also wanted a language that is compatible with legacy code written in C, predictable and easy to learn. Their solution was to take standard C++ and remove language features that cause a performance penalty, are difficult to use or cause unpredictable behavior.

9.2 Embedded C++ compared to C

Although EC++ is mainly a superset of C and most valid C code is also valid Embedded C++ code, some differences do exist. None of the differences obstructs a porting effort from C to EC++ but the code must be reviewed and some minor changes must be done. Several of the differences are potentially dangerous but legal C code that is not allowed in EC++. Thus EC++ can be considered as a better or safer C code.

The developers of EC++ provide some guidelines on how the differences between C and EC++ should be handled. The full list of guidelines can be found at the official EC++ web page [15], the examples below is a subset of that list.

Size of type char

C char has the size sizeof(int), the size of char and an int is the same.

C++ char has the size sizeof(char), this can be smaller than an int.

Dynamic memory

C malloc / free are used to allocate / return dynamic memory.

C++ new / delete is used to allocate / return dynamic memory.

malloc / free can be used but must not be mixed with new / delete.

Declaration of constants

C A constant in file scope has automatically
A constant in file scope must be explicitly declared as constant in order to get external linkage.

Although some of the features in C++ are considered to cause a performance penalty either in larger executables or slower execution, many of them can be used without any penalty. Introducing classes, function name overloading and operator overloading does not cause a penalty. Using virtual functions will give a small penalty, an additional memory access to a virtual function table is needed to find the proper function. Exceptions and runtime type information are more expensive features in C++ therefore both of them are excluded in EC++. Templates can cause a penalty if used without care but by simply not using the feature this risk is avoided. If they are used properly it is quite possible to reduce the size of the executable since some parts of the code can be shared by several instantiations of the template [9]. Templates are not a part of EC++ but they can be used anyway since the compiler supports templates.

9.3 Embedded C++ compared to C++

In order to adapt standard C++ to a language suitable for embedded applications, some of the features in C++ were removed. The features were removed since they reduced efficiency but didn’t provide any functionality essential for embedded applications. None of the removed features is a necessary requirement for an object-oriented design. The main disadvantage of EC++ compared to standard C is the absence of runtime type identification. The runtime type identification can be valuable for applications that use polymorphism a lot but have little importance otherwise. Removing the runtime type identification means that it is no longer possible to see which type of subclass a superclass pointer actually points to. A brief introduction of the differences is given below and a full specification of the syntax with comments to deviations from the C++ syntax can be found at the website for EC++ [15].

Templates

Templates are used to define generic objects that can be instantiated with a particular type.

Multiple inheritance

Multiple inheritance is not allowed but single inheritance is supported.

Exception handling

Exceptions have been removed since they are considered to cause a runtime penalty. They can also cause the code to be less predictable.

Runtime type information

The runtime type information has been
removed since it increases the memory consumption without providing critical functionality.

New cast syntax
Dynamic_cast, static_cast and reinterpret_cast conversion is no longer possible because the runtime type information is not available.

Namespaces
Not available. Embedded software is usually not large enough to justify namespaces.

STL
STL are based on templates, thus the STL library is not available in Embedded C++.

Streams, strings and complex numbers
The standard implementations of these classes are based on templates but they are considered important for embedded systems and have been implemented without templates.

10 OSE Delta operating system and Embedded C++

The OSE Delta operating system fully supports C++. Further information concerning C++ / EC++ support can be found in [16].

11 ARM and Embedded C++

The ARM processor can be used for applications written in C++ and thus applications written in Embedded C++.

12 Development environment and Embedded C++

When porting from one language to another not only the feasibility of the porting effort itself must be considered but also if the current tools can be used for the new language. Otherwise the staff will not only have to adapt to a new language and a new way of programming but also a new set of tools. The development environment currently used, Microsoft Visual C++, is as the name suggests intended for C++. Microsoft Visual C++ was used to write the small test cases described in section 13.1. No problems were encountered when the EC++ files were compiled and linked and the code ran as expected. Combining C and C++ code in the same project should be possible since C++ and EC++ are intended to be backwards compatible with C. Even if there are not supposed to be any problems doing this it was necessary to verify that it worked in reality.
During the work with this thesis some problems with the linker has been encountered. It has been possible to work around the problems but the underlying issues with the linker have not been solved. Identifying and removing these issues is an essential part of the future work but not within the scope of the thesis.

12.1 Lauterbach Jtag debugger for ARM

The debugger currently used handles both C and C++.

12.2 IAR Embedded Workbench

The development tool EWARM from IAR supports Embedded C++. Full information about the tool can be found in [17]. The compiler currently used is included in this package, it supports EC++ instead of the full standard C++ language. The compiler is the only development tool that does not support full C++.

12.3 EMP specific limitations

12.3.1 Macro

Some of the macros will have to be rewritten due to the stronger static type checking in EC++. This will cause some additional work but should also be considered an improvement of the old code. The stronger type checking can detect errors that would not have been noticed otherwise.

12.3.2 new – delete

The new and delete operators have to be defined. This can be done in two ways, both the global operator new and delete can be defined once for all classes or they can be defined for each class. In the code used for the tests in section 13 the new and delete operators have been defined for each class. Defining the global operators would not have been more difficult but defining them for each class made it easier to add debugging information. These operators are straightforward to implement and are not more than a few lines long.
12.3.3 Enumerations

Two different types of enumerations are currently used, the standard C enum and the enumeration created by the TYPEDEF ENUM macro. They have similar behavior but are implemented differently and will be affected in different ways by a transition from C to EC++. In C it is possible to assign an enumeration an integer value, which is not possible in EC++. Instead, the integer value must be explicitly converted to an enumeration before assigning the value to the enumeration. Thus a conversion must be added where a standard C enumeration is assigned an integer value in order to make it EC++ compatible. In addition to this, there is no standard implementation of some operators, e.g. operator ++. Unfortunately the operator ++ cannot be defined for enumerations in EC++ and the code must be rewritten at each place where the operator ++ is used on an enumeration. The enumeration created by the TYPEDEF_ENUM macro is not a true enumeration and will not be affected by a transition from C to EC++.

The enumerations in C++ might seem more complicated to use because all operators aren’t defined but there is a reason for this. Forcing the user to do an explicit conversion reduces the risk of carelessly assigning an integer value that is outside the range of the enumeration. Using the operator ++ on enumerations might be convenient but it is not always safe to do so. Normally the values associated with each enumerated constant is assigned values starting at zero and then increased by one, but it is possible for the creator of the enumeration to assign any value to them, not necessarily in an obvious way. User defined operators are more likely to behave the way the user of an enumeration thinks they should work. The example bellow illustrates how dangerous it could be to treat an enumeration as an integer.

```c
enum SHAPE {SQUARE=5,RECTANGLE,TRIANGLE=17,CIRCLE,ELLIPSE};

Here, SQUARE equals 5, RECTANGLE equals 6, TRIANGLE equals 17, CIRCLE equals 18 and ELLIPSE equals 19. Assigning an explicit integer value to or using operator++ on an enumeration like this one could easily cause very strange errors. Thus it makes a lot of sense to use the enumeration values instead of explicit integer values.

12.3.4 ECM

The interface between software from EMP and software from customers, e.g. SonyEricsson, is based on the ECM model, a subset of the COM model developed by Microsoft. The middleware layer implements this interface and some special care must be taken when this layer is migrated to EC++ [18].
• **Memory allocation** The standard memory handler does not handle the memory used in the middleware layer. The middleware memory handler provides extra protection against access violations and reduces the risk of disturbing time critical processes in the lower layers. Thus must operator new and operator delete be implemented with this in mind and use the middleware memory handler. Overriding the standard operators is not technically difficult but everyone working on the middleware layer must be informed about the need to do this.

• **Alignment of binary files** The interface is based on the alignment of the binary files and it is absolutely necessary that alignment of the binary files produced by the EC++ compiler is identical with the binary from the C compiler.

### 13 Impact on efficiency

#### 13.1 Initial testing

In order to test EC++ features such as inheritance, over loaded functions and dynamically created classes combined with the old C code, a small test application was written. This initial testing is intended to verify that EC++ runs properly on the platform, that the old code can be combined with new parts written in EC++ and provide some indication of the penalty associated with some EC++ features. A more general discussion of C++ performance can be found in [18].

Most test cases consist of a few lines inside a loop repeated 100 times and then the total time is divided by 100 to get the average time. This means that the overhead caused by the loop will be included. The total time to run an empty loop was measured 10 times and the average time was 17 ms. In the following results this time has not been removed from the results so one should keep in mind that given numbers include the time caused by the loop. The results are not intended to be seen as absolute values of the cost for certain operations but to illustrate the difference between them. Where appropriate the time required to do the same thing in C is provided to give a hint of the difference between the languages.

The source code for all tests in sections 13.1.1-13.1.7 and the test setting is provided in Appendix A.

#### 13.1.1 Creating classes

This test highlights the cost of creating and destroying an instance of a class. The class contains no data, constructor, destructor, operator new, operator delete and one member function. Thus the time to create this class can be seen as a best case and creating instances of classes with more data will require more time. The test also illustrates that, when possible, automatic objects should be used because they are more efficient.
A class will take some time to create and require some space in memory, however the same is true for a conventional C struct and the class will require no more space than a struct containing the same amount of data would. Objects that have virtual function will require some additional space used for a virtual function table and runtime type information. In EC++ the runtime type information has been removed and the penalty in EC++ will thus be smaller than the penalty in standard C++. [19]. The table below gives an estimation of the time needed to create a class in EC++ and a struct in C with the same amount of data. The class used here is not the same as the class used in the first test.

<table>
<thead>
<tr>
<th>50 objects</th>
<th>C++ class, automatic memory</th>
<th>C struct,</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>0.00367</td>
<td>0.002527</td>
</tr>
<tr>
<td>Size (Bytes)</td>
<td>204</td>
<td>204</td>
</tr>
</tbody>
</table>

**13.1.2 Creating larger classes**

The time required to create an instance of a certain class is dependent on the amount of data in that class, particularly when the object is created in dynamic memory. Therefore three cases were tried. In the first case an “empty” class was created, in the second case a class with some data was used and in the last case a class with more data and functions was used. The data is initialized in the constructor. According to the result from the test is it faster to create the class in dynamic memory than automatic, this is not the expected result. No explanation to this behavior has been found.

<table>
<thead>
<tr>
<th>100 objects</th>
<th>Automatic memory</th>
<th>Dynamic memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>0.0065</td>
<td>0.0056</td>
</tr>
</tbody>
</table>

**13.1.3 Call to virtual function**

Calling a virtual function will give a higher overhead than a normal function call since the function can not be called directly but has to be looked up in a table. This cost has been measured by measuring the time required to call a virtual function and compare that time to the time required to call a non-virtual function in the same class. We also used a class with inheritance depth two.
100 calls | Non virtual function | Virtual function, depth 1 | Virtual function, depth 2 
--- | --- | --- | --- 
0.004620 | 0.003401 | 0.003560

The second test a virtual function was called as a normal class function. This was done to see if the compiler was clever enough to realize that the virtual function table was not needed since the class type was known. Normally a call to a virtual function is done through a pointer and the system selects the proper method to use through the virtual function table. In this case the method was called directly and allowed the compiler to make an optimization. However, this was not the case. Instead this type of call took longer time than the other two cases.

A detailed discussion on the cost of virtual functions can be found in [21].

100 calls | Call to virtual function, class type known. 
--- | --- 
Time (s) | 0.0056

13.1.4 Call to class function

The time to make a call between two classes, both inline and standard call, was compared to a C function call. The method bodies are empty in all cases since it is the overhead for a function call that should be measured. An inline call in C++ is a little bit like a macro, instead of jumping to a function the function body is expanded everywhere the function is used and thus some of the overhead a normal function call would give is avoided. Instead the code might be larger so this should primarily be used for small methods.

100 calls | C function call | Standard C++ call | Inline C++ call
--- | --- | --- | ---
Time (s) | 0.002138 | 0.004620 | 0.004785

13.1.5 Direct call from C/C++ to C++/C

It is possible to call C functions from C++ and C++ functions from C code. In the test the time required to do 100 function calls in each direction was measured and the average calculated.

100 calls | C -> C++ | C++ -> C
--- | --- | ---
In order to make this possible the compiler must be told to use C linkage conventions, this is done by the extern “C” keyword in the include statement.

```
Extern "C"{
    #include "c_functions.h"
}
```

### 13.1.6 Creating a vector of objects

In this test an array with instances of classes was created. Both a plain class and a sub class was used in the test. The intention of this test was not primarily to estimate the time required to create the array but to verify that the proper constructor and operator new were used. The test verified that the array was created properly, both when the array was created in automatic memory and when it was created on the heap (using operator new). The size of the array will be the size of the objects in the array plus eight bytes. The last eight bytes is a constant overhead cost for all arrays, this means that the size of an array with 10 objects that occupy 40 bytes each will be 408 bytes.

### 13.1.7 Sending OSE signals

It is possible for a process written in C++ / Embedded C++ to send signals to and receive signals from processes written in C. The same OS function is used for both C and C++ code so no attempt to measure the time required performing the operation was done.

In the test a process in the current system sends a signal to the test process written in EC++. When the EC++ process receives the signal it sends a response back.

### 13.1.8 Interface between architectural modules

Communication through OSE signals has been verified. Signals have been sent both from C to EC++ code and from EC++ code to C.

### 13.1.9 Linking C++ code with third party object code written in C

Linking C code with EC++ code has been tested and no difficulties was found. As long as the linker can handle the object code it is no problem to link it with EC++ code.
13.2 **Main test**

A minor part of the code has been transformed to an object oriented design in EC++. The conversion was done in two steps, first the C code was converted to procedure oriented EC++ code and then object-oriented design was introduced. The conversion had two goals: to show that an object-oriented component would be sufficiently efficient to handle time critical tasks and to illustrate some advantages of object-oriented design.

Actions taken when converting to procedure oriented EC++:

- Include files were put in an extern “C” block.
- The way enumerations were used had to be changed at some places.
- Some implicit type conversions had to made explicit.
- Direct calls to OS functions were replaced by indirect calls to the OS.

This was done for:

- `ph_tchf.c`
- `ph.c`
- `ph_data.c`
- `ph_pdcch.c`
- `ph_pfull.c`
- `ph_sdcch.c`
- `ph_tchhs.c`
- `ph_tchm.c`

A wrapper class containing the most frequently used OS calls was written too. This class encapsulates the OS system and thus eases switching from one OS to another. Currently most of the OS calls are used through macros, which help portability but also introduce some problems. The macros do not perform any type checking and can have less predictable behavior than proper function calls.

A struct that contains data that is used by all files in the ph module was converted to a class. Instead of allowing direct access to the data in the class methods to get and set the values were written.
Only ph_tchf.c used the os_wrapper class, the other files still use the macros. The reason for this limitation is that the files had to be changed manually and the time required to do this could not be justified. Modifying ph_tchf.c was enough to test the performance issue and changing the other would not provide any additional information.

ph_tchf is also the only file that uses the methods in the dedicated class to access the data, the other files access the data directly. This limitation has the same motivation as the os_wrapper class limitation above.

The files that were only ported to compilable EC++ code required very little effort. No more than one hour was spent on converting any of those files. The converting effort needed depends on how the C code is written, if the code contains many deviations from EC++ it will require more time. The files converted in this test is a too small sample to be a valid estimation for the whole platform but it clearly indicates that converting the source code from C to EC++ is feasible.

13.2.1 Size of executable

The executable that contained the C++ code was somewhat larger, this was expected because this version contain additional methods. It the size of the executable did not explode however and the test does not indicate that size of a C++ version of the code will be to large.

13.2.2 CPU time

When the files compiled as EC++ code were used there was no detectable decrease of performance. One of the files, ph_tchf.c, handles a phone call and there was no detectable change in performance when the new version was used. The benchmark macro was used to measure the performance and it did not show any increase after the files had been compiled as EC++ code.

13.2.3 Understandability

It would have been desirable to include a comparison between a procedural and object-oriented design of a module in the report but unfortunately this was not possible. Due to problems with the linker it was not possible to rewrite the module with a more object-oriented design. The implementation used in the test could be described as a procedural design that uses some object-oriented features.
13.3 EC++ conclusions

In the first part of the section C was compared to C++ and EC++, and we saw that C was with a few exceptions a subset of EC++. Most differences between C and EC++ are a result of the effort to make the language safer than standard C. Compared to C EC++ has also features to support object-oriented programming. Although the features are intended to make programming easier it will take time before an old C programmer master them. Multiple inheritance is not available in EC++ because it was considered as to difficult to use but there still are several features in EC++ that are difficult to use. One example is constructors, it is easy to overlook defining a copy constructor but the result can be very unpredictable behavior. It also easy to forget to specify the destructor in a base class as virtual but doing so can result in problems with the memory handling.

All in all, C++ and EC++ put a lot of responsibility on the programmer. At the same time as this increases the efficiency of the languages it also makes them more difficult to use.

The results from the tests in sections 13.1 and 13.2 show that it is possible to use EC++ in the platform. The results from the performance tests indicate that EC++ can be used without loosing critical amounts of performance. Some of the performance differences can be the result of a more efficient C compiler. The main test shows that it is possible to port the C code to EC++ code in short time.

14 Semi structured interviews

An attempt to make a first survey of the staff’s opinion about object-orientation and their willingness to change from procedure oriented to object-oriented programming has been carried out. The main objective of the study was to verify that the resistance to change among the staff is not an insurmountable obstacle and to get an opinion of the sources to resistance. Some questions concerning their knowledge about and experience from object-oriented programming and C++/EC++ were also asked.

Almost every one of the interviewed individuals regarded object-orientation to be better than procedure orientation. Among those who did not think object-orientation was better no one thought procedure-oriented programming was better for all tasks but it could be equally good as object-oriented programming. Object-orientation was considered to be less suitable for software interacting with hardware or mathematical algorithms.

Among those that have previous work experience from object-oriented programming the support for object-orientation was particularly high. All of those claimed that they would prefer to work with object-oriented programming.
Almost everyone in the study thought that the language and programming paradigm had a minor impact on software quality compared to the experience of the programmer. At the same time many of the participants in the study agreed that it would be easier to create a good structure with object-orientation. This could indicate that they think it is just as easy to introduce ‘bugs’ in an object-oriented program as a procedure-oriented but the design and structure in the object-oriented program is better.

No one believes they would be more efficient if they work with procedure-oriented programming but some believe they would be more efficient if object-orientation was used.

14.1 Interview setup

The most suitable way to survey a complex issue is interviews. The participants in the survey can have very different knowledge and opinions about the topic and it is difficult to cover all possible answers in a questionnaire. The advantage of interviews compared to questionnaire is the greater flexibility in interviews and the opportunity to catch details in the answers. Some of the subtle details that are revealed when the issues are discussed during the interview would not have been revealed if a form had been used. At the same time as the openness of the interview is the method’s greatest advantage it is also a possible source of errors. If the interviewer is not careful he can easily influence the person being interviewed and introduce a “measurement error”. At the same time as the interview must be flexible and adapted to the opinions of the person interviewed, the outline of all interviews must be roughly the same and all issues must be discussed in all interviews. An interview outline can help the interviewer to remember all areas that are supposed to be covered.

The interview setup is based on the ideas given in [20].

14.2 Interview outline

The outline is intended to be a checklist and guarantee that all topics are covered but leave room for changes in the formulation of the questions. Because different respondents have different knowledge about the topic the questions can’t be asked in the same way all the time.

It is also possible for the respondents to give answers to questions that are not explicitly asked but perceived as associated to the topic by the respondent.
14.3 Opinions about C++ / EC++

A common opinion is that C++/EC++ is a complicated language, they feel that seemingly innocent expressions in C++ can have unexpected consequences. Some of the interviewees mentioned Java as an easier and more reliable object-oriented language.

At the same time are many curious about C++ and interested in trying it out.

14.4 Knowledge and experience from object-oriented programming

Most people in the interview have some experience from object-orientation although it is very limited in some cases. Among the participants in the study a about half had work experience from object-oriented programming. The reason for this high number is that a majority of the people in the study is new at EMP. None of them have been working at EMP the whole time since they graduated and they have gained experience in object-orientation at other companies. This is most likely not the case for most people at EMP.
14.5 Resistance to change

There have not been any signs of heavy resistance towards a transition. None of the more reluctant people in the study have significant experience from object-orientation. The main argument against a change is the training needed to successfully adopt object-orientation and that a transition could slow down other projects.

Those who are reluctant to the change motivate their resistance by claiming that it is not clear if object-orientation is better than procedure-orientation for EMP.

14.6 Perceived transition risks

Some of the interviewees feel that a transition is not possible since it would stop other projects for a longer period and allow EMP’s competitors to improve their positions.

Another opinion is that a transition would also require a lot of testing, this increases both the direct cost of a transition and reduces the amount of testing resources that can be allocated to other projects.

A common perceived risk is resistance to a transition, even if most people in the study claim that they aren’t reluctant to a change themselves they feel that others might be.

15 Estimation of reengineering effort

The effort required when transforming the current design to an object-oriented design is considered to be too difficult to estimate at the moment. Instead an estimation of the effort needed to transform a procedure-oriented design in C to a procedure-oriented design in EC++ was performed. This information will be valuable since a transition from procedural to object-oriented programming has to start with turning the C code into compileable EC++ code. When this first step has been performed, knowledge gained from the process will give some additional input that can be used to estimate the cost for the next step, switching to object-oriented programming. The results and other details about the test and estimation can be found in section 13.2.
16 Estimation of software aging

It would have been interesting to measure the level of decay in the software currently used and perhaps use the result as an additional motivation for a reengineering effort. This has not been done for two reasons: a proper analysis would take too much time (it might even justify a thesis in itself) and it is out of the scope of this thesis. Although the result could be an additional motivation for a new design it is not directly connected to the subject of the thesis, which is the benefits gained from a transition from procedure oriented programming to object-oriented programming.
Part 3

Ideas on how a new language can be introduced in the system are described in this section. The effects an introduction of an object-oriented language would have on the development process are also discussed. Organization aspects on a transition to an object-oriented language at EMP are also covered.
17 Different approaches to reengineering

17.1 Outline of different approaches

In [14] three different ways to replace a legacy system with a new system are given:

- **New** Replacing the old system with a completely new system at once. Although some parts from the old system might be reused, the new system is more than a new release of the old system. Instead it is should be looked at as a new application.

- **Incremental** A new design of the old system is created and the old components are mapped to this skeleton. Gradually the old components are replaced by new components.

- **Evolutionary** This approach is similar to the incremental approach but the increments are less clear. A component is phased from one state to another in many small steps rather than one distinct transition. Both changes in individual components and system properties such as modular design can be introduced this way.

17.2 Approach suitable for EMP

An evolutionary transition might be the way that introduces least risk because the new properties are integrated gradually and the system can be regression tested easily after each change. The fact that C and EC++ code can coexist helps during an evolutionary change. This makes it possible to partition the entire change effort into smaller pieces that can be phased into the desired state little by little as the developers gain familiarity with object-oriented programming and EC++. At the same time as this approach will reduce some risks it will introduce a new risk too, if there isn’t a clear migration plan with a timeline the change might never reach the desired state. Instead of resulting in a system redesigned in an object-oriented manner the result could be a mixture of a procedural and an object-oriented design. In that case, not only time and money is wasted but the outcome could also be a system that is even harder to maintain than it was in the beginning.

The refactoring approach explained in section 6.5 is probably the best way to implement an evolutionary transition to an object-oriented design. One particular advantage of this approach is that small improvements are included quickly and benefits from the change can be shown early. This makes it easier to both secure support from management and solve the problems mentioned in section 8.1.
Some tools that can aid the developers during the change phase are discussed in section 6.5. These tools have the potential to aid the developers but introducing them might be too much at one time. However, even if they are not used initially they could be useful after a while when the developers feel that they master object-orientation.

17.3 Maintaining quality and reliability during changes

When code is written, bugs are easily introduced no matter how good the developer is. The number of bugs introduced is naturally lower if it is a good and careful programmer but the number will never be zero. The only remedy to bugs is testing and code reviews. Code reviews are effective but they are also expensive because they require a lot of time. Testing is also a good way to find bugs but not a popular task among developers.

People in the XP community have been forced to find a solution to this problem since they promote frequent changes and therefore must do a lot of testing. Their solution is coding and testing in conjunction. Instead of writing a lot of code first and then test, they make small changes and test them before making the next change. In fact, they try to write test cases before they write the code that they are going to test. They are also integrating and running integration tests often in order to find integration issues as soon as possible. In order to perform that many tests, most of them have to be automated.

All the principles suggested in the XP methodology might not be suitable for EMP but having a set of automated tests should help the developers in their daily work.

18 Effects on the development process

Switching from a procedure-oriented language to an object-oriented language is not just a matter of rewriting the code to compile in a new compiler but to take advantage of the new principles that the language supports and write code in a new way. In order to support the staff in achieving this goal some guidelines on object-oriented development could be useful. These guidelines do not necessarily have to replace the current development process, instead they could be a complement that addresses the object-oriented issues. As mentioned in sections 7.3, 18.1 and 18.2, documentation, testing and inspections methods need some modification in order to fit in an object-oriented development process. The guidelines should help the developers to modify and integrate these methods into their development process.

18.1 Object-Oriented testing

Introducing the object-oriented paradigm does not change the basic need and importance of testing but introduces some new challenges. This section is based on the information provided by [37, 38].
• **Inheritance** Inherited methods must be re-tested even if they have been carefully tested when used previously. Although it is less likely to find errors in a class that has been tested earlier, it has not been tested in the new context, thus its behavior in this context must be verified.

• **Dependencies between member functions** It is not sufficient to test the member functions separately and only verify that they give the proper response given a certain input. The interactions between member functions through state-variables and sequences of function calls must also be tested. This type of testing is not needed in procedure-oriented programming, it could be considered to be testing on a level between unit testing and system testing in a procedure-oriented program [39, 40]. It is also possible that subtle timing errors are introduced when old code is used in a new context and tests checking this are necessary.

• **State verification** Verifying state-dependent behavior can be difficult for two reasons. Encapsulation hides the state of an object and makes it more difficult to observe the state during debugging. The second reason is that the state is often distributed over several classes and they must be tested together and data must be collected from all classes simultaneously in order to determine the state.

• **Iterative development** Object-oriented development is often characterized by iterative development where small increments are added and tested frequently.

• **Non-localness** In addition to the problems mentioned above, the new features introduced by the object-oriented paradigm might, if not used with care, increase the complexity of the program and make it more difficult to create the right test cases. Some observations found by [38]:

1. *It is extremely difficult to understand a given class in a large OO program if that class depends on many other classes.*

2. *Without sufficient “insight”, a tester may not know where to start testing an OO library.*

3. *It is extremely costly to construct test stubs since a tester has to understand the called functions, possibly create and properly initialize certain objects, and write code to simulate the behavior and effects of the called functions.*

4. *It is difficult to identify and test the effect of polymorphism and dynamic binding.*

5. *It is difficult to identify change impact in OO maintenance since the impact may ripple throughout the OO program through the complex dependencies.*
At the same time as object-orientation introduces these challenges it also introduces some benefits. Dividing functionality into several smaller units with well-defined interfaces between them might seem to increase the need of integration testing. This is however not the whole truth. Dividing the functionality into smaller parts just made the interfaces between different parts of the old code, that were implicit, explicit and visible. This can help testing since each unit can be tested against its interface and the errors found are localized to that specific unit.

Another potential benefit is automatic testing based on design documents. The object-oriented paradigm is not only code, it is also design documents written in a language such as UML. In *Automated testing from Object Models* [41] it is suggested that these design documents can be used to aid testing by generating test scripts and then run them. Automated test scripts are naturally not sufficient testing but they can find many errors and allow manual testers to spend more on system and integration testing.

18.2 Code Reviews

Fagan introduced code review in the form of a formal inspection in 1972. Since then code inspection has proven its value and has become an important method to find defects in software at an early stage. Performing the inspections will require some resources but it is believed to be a cost-effective way to find defects [42, 1]. Inspections has several advantages compared to ordinary testing:

- Not only code but also all kinds of document can be tested.
- It can be performed during the whole development process.
- Defects difficult find in ordinary tests can be found.
- Inspecting other people’s code will give a more thorough understanding of the whole product.

However, the development of inspection methods and verification of their efficiency was done with procedure-oriented programming in mind. If the differences between programs written in a procedure-oriented manner and object-oriented manner is not recognized and compensated for, the performance of the method might decrease significantly. The main problem introduced by the object-oriented programming is the division of functionality into several smaller classes and methods. This means that each small part is easier to understand but it also means that it is more difficult to understand how it is interacting with the rest of the program [43, 44].
Currently, inspections are performed in a rather ad-hoc manner at EMP. Suggesting a code inspection method suitable for EMP after a transition is out of the scope of this paper. Nonetheless, it is clear that there is much to gain from inspections and improving the inspection method is an important way to increase software quality. The inspection method used after a transition must take the characteristics of object-oriented programming into account and reduces the disadvantages of the decreased localness. When choosing inspection method more recently suggested improvements to Fagan’s original process such as perspective-based inspections and usage-based inspections should be considered too. A brief introduction to software inspection and a presentation of some inspection methods is given in State-of-the-art: Software inspections after 25 years [45]

19 Introducing a new programming language

When the new programming-paradigm is introduced, EMP will face some of the problems discussed in section 8.1. In [14] a few questions that highlight these problems, in the software-reengineering context, are raised. These questions are discussed in the next section. Answering these questions is one part of the solution to the problems in 8.1.

A small interview study of employees at different EMP departments was carried out. The goal was to find out what the main causes of risk from the staffs point of view could be. The results from the interviews can be found in section 14.

19.1 Issues that must be resolved

- **Why** There must be a clearly defined goal that is understandable to everyone who is influenced by the change. The goal should capture the advantages of the object-oriented paradigm and why they are important to EMP. The goal will be the beacon during the change phase, that tells everyone where he or she is going. Just knowing where to go is not enough, a thorough understanding of why reaching this goal is desirable must also be established. If this is not accomplished it will be very difficult to motivate the participants in the change project and the risk of failure increases dramatically. This is particularly important from EMP as the C language has been used rather successfully for along time and most employees have confidence in C.
- **What** When the goal has been defined and understood it is necessary to determine what has to be changed in order to reach the goal. It should also be clear what must not change, which is the properties of the old system that has to be valid after the change. Defining the scope of the change is naturally a necessary item of the technical planning of the change but it is also important from an organizational perspective. Unless the scope is clearly defined it is possible that the staff will experience the change as more radical and complicated than it really is and increase their resistance to the change. For EMP it is probably important to keep strict control on what parts are changed in order to avoid a situation where random parts are reengineered without coordination.

- **How** A map that describes the migration path to the desired state is needed. Information about the process, e.g. model, task decomposition, quality evaluation and testing facilities, is needed. The staff should be heavily involved in this phase in order to make sure that no part of the system is overlooked. As pointed out in sections 8.2.1, allowing the staff to participate in the planning of the change effort is also an important part to gain their confidence. Among other things the process model, design methods, tools, work breakdown and quality evaluation metrics should be defined here.

- **When** A time schedule that sets the start date, milestones and the end date must be available. The most critical aspect on the time plan for EMP is to minimize interference with other projects.

- **Where and who** Sufficient resources, both manpower and knowledge, must be allocated to the project. It is likely that shortages in knowledge will be found when the tasks are assigned to individual members in the team. The distribution of the tasks and responsibility must be clear. At the same time as a goal for EMP is disturbing other projects as little as possible the project must be given sufficient resources. If enough people, knowledge and equipment aren’t allocated the outcome is jeopardized. Reengineering is not easy which means that it is crucial to allocate experts to the project and not only junior employees.

- **What if** A careful risk assessment should be done and the risks found must be explicitly addressed in the change process used during the change phase. Guidelines that highlight some risks and ways to avoid them should be available to the staff. As problems are encountered and solved they should be published so that they aren’t repeated.
19.2 Training

As mentioned several times in this paper proper training in object-oriented programming and design in EC++ is needed. There are employees with previous experience from object-orientation and C++; this does not mean that they don’t need training. Instead they should be given more advanced training and become “champions” or “change agents” (see section 8.2.5).

The two most obvious ways to acquire knowledge is external courses and consultants. It is important to make sure that the courses teaches object-oriented programming for embedded systems and preferably uses EC++ instead of standard C++. Consultants must not be used to do any actual work, instead they should be used as mentors and teach the EMP employees how to do the work. Combining courses for initial training and consultants for guidance and assistance during the work could be a good solution.

Encouraging the staff to interact with each other in both formal and informal contexts is a good way to spread knowledge and generate new ideas. Management can encourage this by indicating that discussing different ways to solve problems, aspects of object-orientation and EC++ is considered a productive activity and not a waste of company time.

Further ideas on how to acquire knowledge and spread throughout an organization can be found in [46].

20 Discussion

20.1 Problems and further work

The most important parts to complement this thesis with before it can be used as the basis for a decision are cost and benefit estimations. Estimations of the effort needed to switch from a procedure-oriented design to an object-oriented design and a prediction of the cost savings from object-orientation must be performed. The estimated effort needed to perform the transition must be less than the predicted savings from the object-oriented paradigm. The cost might seem high but it should be amortized over a longer period and compared to the savings during the same period. An estimation of the work needed to turn the C code into compileable EC++ code was done but estimating the work to modify the design was considered too difficult.
As mentioned in section 8.2 one of the first steps taken during a change phase should be changing a small part of the change and use the, hopefully, successful result as motivation for the entire change. In addition to being an important way to sell the ideas behind the change this pilot project can also be used to estimate the time required to change the entire system. After a module has been implemented in an object-oriented language it should also be easier to get a fair idea of the benefits from the new design and estimate the benefits of the object-oriented paradigm. Even if the current implementation in C is not redesigned to an implementation in EC++ an object-oriented module can still be useful as an example when new modules are to be developed. Being able to compare an object-oriented and procedure-oriented implementation of an existing module should be helpful when the design of a new module is considered.

Another important issue is how to change the architecture without introducing new defects. Some rules, such as coding standards, that control the work when the architecture is changed is most likely needed in order to maintain a high level of quality during the change. Careful regression testing of the new architecture is likely to be expensive, but necessary, to guarantee a low defect rate. Automated testing could be an important aid here.

During the work with the test cases in section 13, significant problems caused by unstable and incompatible versions of the code was encountered. In most cases re-implementing the test cases in a more recent release solved the problems. Re-implementing the code could not solve one problem, instead an error in the linker had to be removed. After the linker was updated this problem no longer occurred but waiting for the new release slowed down the project. Unfortunately, solving these problems required a lot of time, this unfortunately meant that main test in section 13.2 does not include a comparison between a procedure-oriented and object-oriented design.

20.2 Conclusions

Although some problems associated with the object-oriented paradigm has been discovered and no one believes it is the silver bullet that will solve all software-engineering problems it has some important advantages over procedural programming. At the same time as it is difficult to find success stories about object-oriented programming it is also difficult to find reports talking about organizations that has switched back to procedure-oriented programming.

As discussed in section 13.3 there are error prone parts in EC++ and C++ that can cause problems. Some of these problems are already present and in some cases even worse in C. Other problems e.g. multiple inheritance can be solved by simply not using the feature, multiple inheritance is not allowed in EC++.
It is important to realize that changing the last part of the filename from `c` to `cpp` does not automatically guarantee a great design that will be reusable, easy to modify and less error prone. Writing bad code in EC++ / C++ is perhaps even easier than it is to do so in C because of the new features in the language. Thus it is important to realize that a transition will take time and require some additional resources. If training is not given enough importance, the transition will be very risky and the likelihood of achieving the goals will be low.

Object-oriented software is often claimed to have worse performance than the corresponding procedure-oriented program. This is not necessarily the case, it is true that some object-oriented languages, e.g. java, is slower but it is not true that an EC++ program must have lower performance than a C program. On the contrary it is quite possible to achieve higher performance in EC++ than C. As long as the designer is skilled and the program is written with performance in mind an object-oriented design can be just as efficient as a procedure-oriented design.

There has been a general move towards the object-oriented paradigm, particularly in the academic world, and there are no indications of a move back to procedural programming. This indicates that it will be increasingly difficult to recruit personnel with knowledge about C in the future and some resources must be allocated to training of new staff. Thus a transition to an object-oriented language might be necessary at some point.

A gradual transition from C to EC++ is quite possible since C is essentially a subset of EC++ and the two languages can coexist in the same program. Thus it is possible to start using procedure-oriented programming in EC++ with little effort and not introduce the object-oriented paradigm until enough knowledge has been acquired. It also means that small increments of object-oriented code can be introduced and integrated with the old code.

There is also an opportunity cost associated with a transition. When resources are allocated to the transition, fewer resources will be dedicated to development of new functionality. Thus it is needed to find a period where sufficient resources can be allocated without interfering with development. If new functionality is being developed this downside is not present, besides the fact some training might be necessary.

*Acta est fabula, plaudite.*
References


4. [www.extremeprogramming.org](http://www.extremeprogramming.org)


15. EC++ official website, [http://www.caravan.net/ec2plus/](http://www.caravan.net/ec2plus/).

17. IAR, *ARM IAR Embedded Workbench User Guide*, IAR.


34. V. R. Basili, L. Briand, W. L. Melo, A Validation of Object-Oriented design metrics as quality indicators, IEEE Transactions on Software engineering, 1996.


Appendix A

The files with the source code used for the tests in section 13 can be found at [elektroniskt repository för document] and only the fragments that does the actual measuring is presented here.

- Creating classes

```cpp
// Create 100 objects in automatic memory
void cpp_test_functions::measure_time_to_create_automatic_object(){
    printf("\n\n before creating 100 objects automatic mem\n");
    for (int i=0; i<100; ++i){
        cpp_test myTestClass;
        //myTestClass.inline_test_method();
    }
    printf("\n after creating 100 objects automatic mem\n");
}
```

```cpp
//create and destroy 100 objects, dynamic memory
void cpp_test_functions::measure_time_to_create_dynamic_object(){
    cpp_test *myTestClassPointer;
    printf("\n\n before creating 100 objects, 100 function calls\n");
    for (int i=0; i<100; ++i){
        myTestClassPointer = new cpp_test;
        //myTestClassPointer->test_method();
        delete myTestClassPointer;
    }
    printf("\n after creating 100 objects\n");
}
```

- Creating larger classes

```cpp
// Instantiating larger object, automatic memory
void cpp_test_functions::measure_time_to_initiate_larger_object_aut(){
    printf("\n size of cpp test object with some data %d",
            sizeof(data_test_class));
    printf("\n Before creating and deleting object with some data, aut mem");
    for(int i = 0; i <50; ++i){
        data_test_class testObject;
    }
    printf("\n After test of object with some data");
}
```

```cpp
// Instantiating larger object, dynamic memory
void cpp_test_functions::measure_time_to_initiate_larger_object_dyn(){
    data_test_class* dataObjPointer;
    printf("\n before creating AND deleting data obj in dyn. mem");
    for(int i = 0; i < 50; i ++){
        dataObjPointer = new data_test_class;
        delete dataObjPointer;
    }
    printf("\n after creating data obj in dyn. mem.");
}
```

- Class methods
// Calls between c++ classes
printf("\n before measuring time to call function in different class\n");
for(int i = 0; i<100; ++i){
    test_functions.measure_empty_function();
}
printf("\n after measuring time to call function in different class ");

// Between c++ classes, inline
printf("\n innan inline funktions anrop mellan klasser\n");
for(int i = 0; i <100; ++i){
    test_functions.measure_empty_inline();
}
printf("\n efter inline funktions anrop\n");

- Function calls between C and EC++ code

// call to c function
printf("\n before call to c function\n");
for(int i = 0; i <100; ++i){
    c_test_function();
}
printf("\n after call to c function\n");

- Inheritance

// Create 100 instances of a sub class
void cpp_test_functions::measure_time_to_create_sub_class(){
    base_class *myBaseClassPointer;
    printf("\n before creating 100 objects using inheritance\n");
    for (int i=0; i<100; ++i){
        myBaseClassPointer = new sub_class;
        delete myBaseClassPointer;
    }
    printf("\n after creating 100 objects using inheritance\n");
}

// Measure time to create a sub class, depth = 2
void cpp_test_functions::measure_time_to_create_subsub_class(){
    base_class *myBaseClassPointer;
    printf("\n before creating 100 (subsub) objects using inheritance, function calls\n");
    for (int i=0; i<100; ++i){
        myBaseClassPointer = new sub_sub_class;
        delete myBaseClassPointer;
    }
    printf("\n after creating 100 (subsub) objects using inheritance, function calls\n");
}

- Virtual functions

// Measure time to call a virtual function.
void cpp_test_functions::measure_time_to_call_vf(){
    base_class *myBaseClassPointer = new sub_class;
    printf("\n before calling a virtual function in sub class 100 times. d=1\n");
for (int i=0; i<100; ++i){
    myBaseClassPointer->inheritance_test_method();
}
printf("\n\n after calling a virtual function in sub class 100 times. d=1\n");
delete myBaseClassPointer;
}

// Measure time to call a virtual function in sub sub class.
void cpp_test_functions::measure_time_to_call_vf2(){
    base_class * myBaseClassPointer = new sub_sub_class;
    printf("\n\n before calling a virtual function 100 times. d=2\n");
    for (int i=0; i<100; ++i){
        myBaseClassPointer->inheritance_test_method();
    }
    printf("\n after calling a virtual function 100 times. d=2\n");
delete myBaseClassPointer;
}

- Vector of objects

// Testing vector with instances of objects
void cpp_test_functions::vector_test(){
    printf("\n the size of on instance is: %d", sizeof(sub_class));
    printf("\n Before vector test, automatic memory\n");
    {
        sub_class subClassArray[10];
        subClassArray[3].array_test_method();
    }
    printf("\n After vector test, automatic memory\n");
    printf("\n Before vector test, dynamic memory l= 10\n");
    sub_class* subClassArray = new sub_class[10];
    subClassArray[3].array_test_method();
    delete[] subClassArray;
    printf("\n After vector test, dynamic memory\n");
    printf("\n Before vector test, dynamic memory l= 20\n");
    subClassArray = new sub_class[20];
    subClassArray[3].array_test_method();
    delete[] subClassArray;
    printf("\n After vector test, dynamic memory\n");
}

- OSE signals

SIGSELECT Primitives[] = {1, RR_CPP_signal};
union SIGNAL * ReceivePrimitive_p = NIL;
ReceivePrimitive_p = os_functions.receive_signal(Primitives);
os_functions.free_signal_buffert(&ReceivePrimitive_p);
printf("\n received signal from rr. Sending response\n");
SEND_SIMPLE_PRIMITIVE( CPP_RR_signal, RR_Process_ );