Implementation and Evaluation of Design by Contract in an Interface Definition Language

Martin Trulsson

Department of Computer Science
Lund University, Faculty of Engineering

Advisors:
Rikard Dahlman, Ericsson Mobile Platform
Martin Höst, Lund University

February 19, 2009
When software systems are developed there is a major focus on quality. The customers who buy a software system expect it to be working flawlessly all the time, and therefore the software developers must always work on increasing the quality of their product.

In an interface definition language, methods that can be invoked by other parts of a system are defined. Today it is possible to write comments for a method but there is no formalized way to specify the constraints of a method. Design by contract is a methodology which enables contracts to be written for a method. A contract means certain conditions that must be fulfilled before and after the execution of the method.

This thesis describes a possible solution of how design by contract may be introduced in an interface definition language. It presents how contracts can be adapted to the specific features of IDL and the differences that exist between a solution based on, for example, an object oriented programming language and an interface definition language.

The thesis also contains an empirical evaluation of design by contract from the viewpoint of different developers. The evaluation studies how the work of a developer is changed when the features of design by contract is introduced into a system, and the difficulties it brings. The work indicates that developers will get benefits from having contracts on the methods in a system, and these benefits are looked into in more detail during the evaluation.
This master’s thesis was carried out at Ericsson Mobile Platform in Lund.

I would like to thank my supervisor at EMP, Rikard Dahlman, for the support during the design and implementation of the developed prototype. I would also like to thank my supervisor from Lund University, Martin Höst, for the help with this report and the valuable feedback from his reviews of this report.

Furthermore I want to thank the developers from EMP, Kjell Andersson and Teddie Stenvi, who helped me with the evaluation of the developed prototype. Their feedback on my work form the basis for the conclusions in this report.

At last I thank the rest of the people working in the Software Architecture and Strategies group at EMP for a pleasant time performing my master’s thesis.
Contents

1 Introduction .......................................................... 1
  1.1 Background ......................................................... 1
  1.2 Objectives of this thesis project .............................. 5
  1.3 Outline of the report ........................................... 5

2 Methodology .......................................................... 7
  2.1 Overview ............................................................ 7
  2.2 Evaluation .......................................................... 8

3 Theory of design by contract ..................................... 13
  3.1 Preconditions and postconditions ............................ 14
  3.2 An example of design by contract ............................ 15
  3.3 Invariants ........................................................... 16
  3.4 Object orientation ............................................... 18

4 Component Model .................................................... 21
  4.1 Component models ............................................... 21
  4.2 Component ........................................................ 21
  4.3 Interface .......................................................... 22
  4.4 UML ................................................................. 23
  4.5 Ericsson Component Model ..................................... 23

5 Interface Definition Language .................................... 27
  5.1 Ericsson Interface Definition Language ..................... 28

6 Design by contract in a component model ....................... 33
  6.1 Preconditions and postconditions ............................ 33
  6.2 Invariants ........................................................ 34
Chapter 1

Introduction

1.1 Background

Software products that are developed today must be very focused on reliability and robustness. It is important that the user can trust the product and know that it does exactly what it is supposed to do and nothing else. In order to ensure the quality of the product it is necessary that the developers are working as a team and follow a well-developed methodology that fits their specific project. This methodology is often a combination of many different techniques that has been invented to improve the software development. One such combination can be a project team that is working according to an agile method [13] when developing a product and also uses a configuration management method to keep track of the different versions of the system. The team can later on follow a specific testing technique when performing the system testing to ensure a certain level of quality of the product before it is delivered to the customers.

The programming languages that are used today are based on methods where the programmer can implement some functionality and later on invoke the method when the specific functionality is needed. This makes the software more structured and it is much easier to get an overview of the software for the programmer than if all code was written from top to bottom. Another benefit is that once a functionality has been implemented it can be reused by the programmer in many different contexts when the specific functionality is requested and thereby avoid duplicated code. It is also possible for programmers to reuse modules that have been written by other programmers and thereby save time during development.

There is a risk when calling other methods and specifically when the methods have been written by other programmers. If the code in the method is written at the same time it is needed and for that specific task the programmer can be almost certain that the code is correctly implemented and it can also
be tested for verification. That certainty can not be guaranteed when other methods are used. This means that the programmer must verify that the output of the used method is correct. The programmer that is implementing the method must also check the parameters that are sent to the method before they are used. Otherwise he can not be sure that the method is used in a correct way with correct input values.

If different parts of a software product are supposed to be reused it is important that they are developed according to a well thought-out and structured design. It should be easy to see the dependencies between different modules and how they communicate. The different parts of the software should have a clear functionality and it should not be possible to use it in another way than what the designer intended.

A lot of methods request input parameters which are used during the execution. Some methods request those parameters to fulfill certain conditions if the execution should be performed according to the developers intention. This means that the developer must verify that the parameters fulfill the conditions before the execution starts, otherwise it is possible that another programmer makes a mistake and invoke the method with wrong parameters. This can be done according to a programming principle called defensive programming [16]. The basic idea of defensive programming is that it is up to the invoked method to verify the values of its parameters. Almost every software developer is working according to defensive programming even if it is not formally stated in the development process. There is hardly any software developer that just accepts the parameters that is sent to a method without verifying them. For example if the input to a method should be the present time, the developers will write code that verifies the hours to be less than twentyfour and the minutes to be less than sixty.

In order to perform all these controls the programmers must implement a lot of extra code. It might happen that the parameters that are sent to a method is checked both before the call to the method and in the beginning of the method before the actual functionality starts. Some of these controls are usually not necessary but they are implemented anyway only to be sure that the method can not be invoked with the wrong input. Meyer describes defensive programming as a technique where the developer should protect all the modules in the software as much as possible, even if it leads to unnecessary code. It is better to have the code implemented than not. Even if the redundant controls do not help, at least they do not harm [16]. The verification might serve a purpose when it comes to the final product that is delivered to customers. The customers will get a product with higher reliability and might therefore be more pleased with the product.

A great part of the software products of today are based on older systems which are extended to get an enhanced functionality to meet the demands
of the customers or are systems that are maintained to get a long lifetime. During the rework of the product it is probable that some components will be changed. A component of a system might also be reused in a new product since it is unnecessary to invent the wheel twice. When components are changed or reused, redundant verification of parameters can be used to ensure that a component only is used with the parameters it is designed for.

The redundant checks might also have a negative impact on the software. The extra code that is implemented increases the complexity and the lines of code. And the risk of introducing errors when implementing the code is greater if the total software is large. Redundant checks will also result in increased execution time for the software.

1.1.1 Ariane 5

An example of problems that may come from reusing software modules without the proper controls of parameters is Ariane 5. Ariane 5 is an expendable launch system which is used to deliver payloads to geostationary transfer orbit or low earth orbit [20]. The first launch of Ariane 5 was made on June 4 in 1996 and resulted in an explosion and the rocket was lost. The explosion cost the developers $500 millions. The European space agency and the French space agency wanted to know the cause of the accident and appointed an international inquiry board. The board consisted of well respected experts and their task was to analyze the accident and produce a report. The board found that the accident was caused by reuse of a module from Ariane 4, which was a predecessor to Ariane 5 [9].

The reused software is a part of the inertial reference system. The software was used before take off and made computations that is used to align the inertial reference system. The computations should normally be finished before take off but the engineers chose to let them continue until 50 seconds after take off instead of just before. The reason for this was that the engineers must reset the module if the computation was stopped. There is always a risk that a launch will be postponed, especially because it was a test launch, and a reset could take several hours [9].

During the computations a 64-bit floating point value was supposed to be converted to a 16-bit signed integer. This could not be done since the 64-bit floating point value is greater than the largest value that can be represented by a 16-bit signed integer and an exception was thrown.

The developers of the software module found during the analysis of the functionality that it is impossible to get an overflow. Because of this no code had been written that could handle the exception. This unhandled exception resulted in a crash of the entire software and all the computers on board. In the end Ariane 5 exploded.
The analysis was correct when the software was developed for Ariane 4 but the analysis was not remade for Ariane 5. There was also a lack of information in the code and specification. It could not be read anywhere that the converted value should fit in a 16-bit signed integer. Therefore, the module was reused in Ariane 5 on the assumption that it should work even if its surroundings had changed [9].

1.1.2 Design by contract

According to Baber [1] the accident of Ariane 5 could have been avoided if verification of the parameters had been implemented by stating preconditions and postconditions. The developers could have implemented verification of the parameters to avoid to start the execution of the functionality and the uncaught exception would not have caused the crash of the entire system. If the accepted values for the parameters had been written in the specification or in the code, it would also have been much easier for the developers to know if the module was reusable or not. It is much easier to know what values the method accepts if it is stated in the code of the method or even better in its specification. It is also better from a tester’s point of view to write the accepted parameter values nearby the code. It is harder to miss a test case for a method if the values to test for are written in the code or specification [9].

One way to avoid similar accidents in the future is to work according to defensive programming. But then it is up to the programmer who is implementing the method to write code for all verifications. This means that there is still a human factor that might introduce an error. Sometimes it is not even the same person who designs the system as implements it. The implementer might not know all the limitations of the method and thereby miss some important controls.

Another way to avoid this kind of error is to develop software systems according to design by contract [15]. Design by contract means that every method has a contract which specifies the conditions that must be fulfilled before a method is invoked. Conditions can also be specified that must be fulfilled after the execution of the method is finished. The contract should be specified in the design phase and made by the developer that analyses the functionality and usage of the method and later on writes its specification. Therefore, the conditions of the method are more likely to be correct than if the developer who implements the method specifies them. Another advantage is that the conditions are reviewed when a programmer implements the method and if there is an error in the contract it is probably detected during the implementation phase.

In most implementations of design by contract, the code that performs the verification is generated by the compiler or a preprocessor. This means that
the system follow the specified conditions and one human factor is removed from the process.

1.2 Objectives of this thesis project

Design by contract is a software principle that might help software developers to improve the quality of the product due to verification of the parameters which are sent to a method. So far design by contract is only fully integrated into one programming language, which is Eiffel [14]. That rise the question why none of the most used programming languages uses such functionality.

Even if design by contract is not integrated into any other programming language there are tools that have been developed for different languages, for example Jose [7] and iContract [11] for Java.

The aim of this master’s thesis is to investigate the possibilities and difficulties in introducing design by contract into a component based specification language. The main focus is on quality and if design by contract will increase the quality of the product. The functionality of design by contract is used by the software developers and therefore the evaluation is made from a developers perspective. The developers must be able to trust the code that verifies the parameters and not write their own code for verification. Otherwise design by contract will only bring a decreased performance to the product.

When developing a software product, testing is an important phase. Therefore design by contract is also evaluated from the view of unit testing. The focus is on the developers attitude to unit testing and whether the developers will change their attitude if design by contract is introduced.

1.3 Outline of the report

- Chapter 1 gives the reader an introduction to the subject and the objectives of this report.
- Chapter 2 describes the methodology that was used during this project. The evaluation is also described to give the reader the knowledge of what to evaluate.
- Chapter 3 introduces the theory of design by contract and goes through the different features that are used later in the prototype.
- The prototype implements design by contract in an interface definition language and a component model. Chapter 4 give the reader the background needed to understand what a component model is.
• Chapter 5 contains a short introduction to an interface definition language with focus on the Ericsson Interface Definition Language.

• In chapter 6, the features of design by contract are discussed from the perspective of a component model.

• In chapter 7, the developed prototype is described to give the reader a detailed description of how design by contract can be used in a component model.

• Chapter 8 presents the evaluation part of this report. This chapter also summarizes the conclusions from the participating developers.

• In chapter 9 is the result from this report and some future work discussed.
Chapter 2

Methodology

2.1 Overview

The primary aim of this thesis was to investigate the impact of design by contract when used in a component model and an interface definition language. One way of evaluating functionality is to build a prototype and use it hands on. It is then possible to draw conclusions from the experiences you get. The prototype simulates only a part of the system but the experiences from working with a small system can show strengths and weaknesses of design by contract when it is applied to the final product.

The first part of the project aimed at getting a deeper understanding of design by contract and how it can be used. It was important to study already existing alternatives to see different approaches that have been tried and the experiences that came out of it. It was also necessary to see different problems that have been encountered and how those problems were solved.

The next part of the project was to develop a prototype that made it possible to use design by contract in a component model. The prototype was based on the already existing platform that is developed by Ericsson Mobile Platform. The main part was to study the existing system and examine how design by contract could be introduced into an already existing system.

The last part of the project was to evaluate the prototype that was developed and from this evaluation some questions were supposed to be answered. The evaluation was made by developers that used the prototype in their daily work. It was important that the prototype was used in an actual working situation in order to get a result that is as valid as it could be. The results are based on interviews with the developers that used the prototype and they have a good understanding in the daily work at Ericsson Mobile Platform and also a deep knowledge of the problems that could arise in software development.
2.2 Evaluation

2.2.1 Objectives

A lot of the software developed today are large systems that has been on the market for a long time or has predecessors that have been maintained and later on updated with new functionality in order to provide a new and better product. This means that the developers of the product must be certain of the benefits and consequences of a change before they wish to introduce it into the system, especially if the change affects a majority of the files in the system. One question that must be answered in this report is the cost of introducing design by contract into an existing system that is based on a component model.

It is therefore important to look at the benefits of introducing design by contract. It is possible that design by contract might bring great benefits to a system and outweigh the costs of introducing it but it is also possible that the cost of introducing it is too high. There are some factors that must be investigated to see whether design by contract is going to bring a positive effect to a system or not.

It is important to see what happens with the quality of the system, are we going to have an increased quality or will it be the same. The quality of a product is a great factor when it comes to how pleased the customers are and the quality must be maintained on a high level. The customers expect a product to be almost flawless and the developers must always strive for an improved quality. There is otherwise a great risk that the customers will choose an alternative product which fulfills their demands on quality and functionality. The users must feel that they can trust the product to do what it is expected to do. In the case of design by contract the users will be programmers that call upon a method which is provided by an interface. The functionality will be the verification of certain conditions on parameters before and after a method call.

When it comes to reliability there are a lot of questions that must be answered. One question is if the programmers will trust the generated verification code and not write their own. Since a lot of code already exist it is also important to know how the users will react if it does not exist conditions on all interfaces. Is there a risk that the programmers believe that there is verification of conditions on all interfaces and not only on those which have been developed after design by contract was introduced into the system. There will be a great cost to specify conditions on all already existing interfaces and it is therefore important to know if the users will see a benefit from using verification of conditions or if it will be too complicated to know if there exists a contract or not and in the end the development will be slowed down.
The theory of design by contract is to verify the parameters of a method before and after the execution of the actual code. It might therefore be interesting to see if the testing is different when design by contract is used. It might be possible to skip the verification of the parameters in the unit tests because they have already been verified, but will the person that writes the test cases rely on those checks or will those unit tests be written anyway. Another question which might have an interesting answer is whether the developer start to think differently when they test the software and if it is in a positive or negative way.

Another question that must be answered is what happens with the time it takes to find an error. An error is not always shown at the place where it originated. It is possible that an error is accepted by the software according to the compiler but generates the wrong result which then propagates when the result of one method is used in another method. It might also in the end produce a run-time error.

A method that has conditions according to design by contract does not accept an unsatisfied parameter. Therefore it might be easier for a developer to find a fault if a printout of the method, its parameters and the conditions is produced directly when a parameter does not satisfy the conditions of a method instead of waiting with the debugging of the code until the fault has propagated and produced a greater error.

The following research questions can be derived from this section:

1. Is there any change in the quality of a software product when all parameters of a method are verified?
2. Are specified conditions for a method enough for software developers to trust the correctness of the parameters?
3. Is there a risk in introducing design by contract in just a part of a system and not the whole system?
4. May design by contract lead to a decrease in the number of unit test cases?
5. Will a software developer spend less time on debugging if the system stops executing and show a printout of the unsatisfied parameters?

2.2.2 Execution

In order to answer the questions above, a prototype was developed [8]. The prototype is an extension to an already existing compiler and makes it possible to read different conditions that must be fulfilled for each method in an
interface from an interface specification file. The compiler can then generate code that verifies the conditions before and after the execution of the method.

This prototype is used by different developers to implement the functionality of an interface. The test persons are experienced developers on Ericsson Mobile Platform who has good knowledge in the product that is developed and the implementation of new functionality in it. They have also great programming skills with the ability to see advantages and disadvantages in different programming techniques.

During the development on Ericsson Mobile Platform it is not the same person who designs the interface and later on implements it. The evaluation will therefore be divided into two parts. The first part include, the design of the interface which is performed by a designer. He follows the standard procedure which is used when a new interface is developed but he also specifies the conditions that should be fulfilled before and after the execution of the methods. After the design of the interface he deliver the files to the developer who should implement it.

This developer implemented the interface according to the process that is used in the daily work and the only difference will be to use a different compiler which supports design by contract instead of the ordinary compiler. The actual using of the prototype is performed by the developer who implements the functionality since he is the one who compiles the code. Before the evaluation starts, the developer is well informed of design by contract and how the conditions and the new functionality works. It is necessary that the test person has a complete understanding of what is possible to do with design by contract to get accurate result. It is also important that the test person follow the normal procedures of development in order to get a correct evaluation of design by contract compared to development when it is not being used.

2.2.3 Followup

The test person was asked to have a few questions in mind during the development with the prototype which provides design by contract. The most important is whether design by contract might be an asset or not, like if it might provide something extra to the developers of a software program when it comes to implementing the functionality. If it can make it easier for the programmers to implement the correct code but also if it is easier to find an error. Another important question is what the developer thinks about the quality. Is design by contract a method to increase the quality of the final product or are there other, more simple methods which can be used to improve the quality.

After the implementation of the interface the test person was asked to answer questions concerning the experiences of working with design by con-
tract. The questions are derived from the subject that was described in the beginning of this chapter. The main subjects are quality, reliability and testing. All those three subjects are important areas in software development and the evaluation will therefore be concentrated on them along with the cost of implementing design by contract into the development process.
The principle of design by contract was first introduced by Bertrand Meyer [15] and the basic idea is that modules in software systems should be designed to follow contracts. The contracts shall specify under which circumstances a method or even a class in object oriented programming can be used. Most methods request some parameter to be sent to the method when it is invoked. Sometimes it is necessary that the parameter has a certain value for the method to be able to execute and produce a correct output. The contract is all about specifying acceptable values for the parameters both before and after the execution of the method.

A contract in software development can be related to contracts between humans in real life. There usually exist two or more parties when a contract is signed. One part say that a product or service can be supplied and the other part say that it is needed and consumes it.

Contracts in software development also have suppliers and consumers. They are called suppliers and clients in existing literature and these names will also be used in this text. The supplier can be a method or a class that already is written and provides some functionality to other developers. It can be an algorithm or computation or simply a few lines of code that is needed in many places and therefore written in a method of its own to avoid duplicated code. The supplier code can then be called from other parts of the system when it is needed. The code that calls the supplier will be the client.

In order to verify that the contract is followed by both the supplier and the client, assertions have been inserted in design by contract. An assertion is a property of some of the values of a program entity. An example can be that a certain integer should have a positive value or that a reference is void [15]. Assertions are basically boolean expressions because their value are either true or false.

This means that assertions can be combined with each other. The syntax for combining assertions differs between different authors but the most com-
mon are a semicolon [15] or "and" [18]. The assertions that will be covered in this text are preconditions, postconditions and invariants.

3.1 Preconditions and postconditions

The term contract has been defined many times and Traon with co-authors defines a contract as "A method contract is a set of assertions that are evaluated before and after the execution of one method. There is one contract for each method in the program that is composed of a pre- and a postcondition and of the invariant of the class" [18]. The precondition that is spoken of is something that must be fulfilled before the code in a method is executed and a postcondition is a guaranteed state or value that the result must satisfy when the code has been executed. The preconditions and postconditions function as a guarantee for both the client and supplier that the other part follows their part of the contract. They form the actual contract and describe the agreement. A contract should, just as in real life, give benefits to both the supplier and the client. In order to get benefits both parties must give something in return, they have obligations towards the other part.

A simple contract has been exemplified below. The invoker of the method must send a parameter of type int that is smaller than MAX_INDEX, otherwise the requested index will be outside the boundaries of the array. In return the invoker can be certain that the method returns the value placed at the requested position in the array.

```c
int my_array[MAX_INDEX];
/**
 * Precondition: index <= MAX_INDEX
 */
int get_value(int index)
{
    return my_array[index];
}
```

If a method has a precondition which say that a parameter must be positive, then every programmer must ensure that the method is invoked with only positive parameters. That is the obligation of the client. A consequence of this is that the method may start executing its code without verifying that the parameter is positive, which is a clear benefit. The benefit for the programmer that uses the method is that he knows that the method returns a
value or is in a state according to the postcondition. There is then no need for verification of the output. As a summary you can say that if the programmer invoke a method with a satisfied precondition then the method promise to deliver an output that is satisfied according to the postcondition[15]. On the other hand, if the precondition is not satisfied, it exists an error in the client and it is there the error must be fixed. Not in the supplier. A violated postcondition means a fault in the supplier and it must be fixed in the invoked method [16]. The conditions help the developers to find the error and to fix it in the right place.

3.2 An example of design by contract

A concrete example of how design by contract can be used is described with the simple implementation of a stack that can be seen below. The implementation contains a struct with three variables. Contents is an array which contains characters, top is an int that keeps track of the character that is on the top of the stack and maxSize is an int which decides the maximum number of characters that can be placed on the stack. There are only two functions implemented in this example but it is enough for the demonstration of preconditions and postconditions. The functions are push, which places the input character on the top of the stack, and pop, which returns the element on the top of the stack and then deletes it.

```c
typedef char stackElementT;

typedef struct {
  stackElementT *contents;
  int top;
  int maxSize;
} stackT;

/*
 * Precondition: top < maxSize
 * Postcondition: top = @pre(top) + 1
 */
void push(stackT *stackP, stackElementT element)
{
  stackP->contents[++stackP->top] = element;
}
```
16 Theory of design by contract

/*
 * Precondition: top >= 0
 * Postcondition: top = @pre(top) - 1
 */
stackElementT pop(stackT *stackP)
{
    return stackP->contents[stackP->top--];
}

The preconditions and postconditions have been written in the comment of each function. That is the most common way of implementation and can for example be seen in iContract. The push-function has a precondition which say that top should be smaller than maxSize. This corresponds to the verification that the stack is not already full. The postcondition say that the value of top should be increased by one. In order to calculate the new value of top in the postcondition, it is necessary to remember the value of top in the beginning of the method. This value is the same value that was used when the precondition was verified. The technique for reusing a value from the precondition differs between different ways of implementing design by contract. The technique that is used in this example was chosen because it is an easy way to describe that it is an old value of top that is used.

The same notation has been used to describe the precondition and postcondition for the pop-method. The precondition say that top should be greater than or equal to zero. That is true if there is an element on the stack, otherwise top will be -1. The postcondition say that top should be decreased because one element is deleted from the stack and top should therefore be decreased by one.

In this example the stack can be seen as the supplier and the client is the user of the stack. As stated before, both the supplier and the client have obligations and benefits towards each other. A summary of the obligations and benefits from the stack-example can be viewed in table 3.1.

3.3 Invariants

An invariant is certain conditions that must be true for all methods that are grouped together into an entity. Often in software development, especially in object oriented programming, an entity represents an object from the real world. There could for example be an entity which represents a house. A
### Table 3.1: Obligations and benefits of the push-function from the stack-example

<table>
<thead>
<tr>
<th>Obligation</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td>The client only tries to push an element to a stack which is not full</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td>The supplier shall add the element to the stack and increase the value of top</td>
</tr>
</tbody>
</table>

A house can contain one or more rooms which have a certain area. Lets say that there exists several methods that can change the characteristics of a house and a room. Then the house entity will need an invariant which says that the summation of the rooms' areas must be the same as the total area of the house.

Mitchell and McKim writes that an invariant should help the reader to build a conceptual model of the abstraction the entity embodies [17]. A class for example, can have some variables or properties that distinguish it as a certain object. A library object can contain zero or more books but it can never have a negative number of books. Invariants are used to ensure that the properties of the entity always have correct values and that the present state is correct regardless of which method has been invoked. A library must always contain books, otherwise it would not be a library.

Assertions like preconditions and postconditions are used to verify the input and output of a single method. An invariant, on the other hand, must be verified on every call to a method which is connected to an entity to ensure the correct properties. An invariant can be seen as conditions that must be fulfilled before and after every method has been executed and could easily have been written as preconditions and postconditions. Since it would have been necessary to write them for every method in the class, it is better to write them as invariants.

As stated before, the conditions of an invariant must be met before and after the method has been executed. However, there is nothing that controls the invariant during the execution of the method. This means that it is all right to break the rules of an invariant during the execution as long as the properties are restored before the method is finished [15].

One example of a class invariant can be seen in an implementation of an
The array must always contain at least zero objects and it can not contain more than the maximum number of objects. This invariant can be written as:

\[ 0 \leq \text{nbrElements}; \text{nbrElements} \leq \text{maxNbrElements} \]

### 3.4 Object orientation

The basic functionality that has been presented previously in this chapter is applicable to all kinds of programming languages. Today when many programming languages are object oriented, there are new demands on design by contract. One important part of object orientation is the possibility to inherit from other classes.

#### 3.4.1 Inheritance

Inheritance means that one class can be a subclass of another class. The subclass will then get all the methods and variables that are implemented in the superclass and thereby will the subclass also get all features of the superclass. This may lead to some problems when implementing design by contract in an object-oriented language. The subclass must then also inherit all the preconditions, postconditions and invariants from the superclass. It can also have new ones which are specific only for the subclass, but then they must be combined with the inherited ones.

Since the subclass inherits all the methods and variables from the superclass it is also important that it has the same preconditions and postconditions for those methods. This is because the dynamic binding [4] that can occur in object-oriented languages. The principle of dynamic binding is that the compiler will choose the correct method to execute depending on which class the object is an instance of. It is therefore important that all the subclasses of a class and the class itself has the same preconditions and postconditions. If that is not the case it might happen that the superclass accepts parameters for a method but the subclass does not. That would violate the rules of dynamic binding.

As stated before, the inherited methods must accept the same parameters as the method does in the superclass but it is possible that the method can accept more parameters than the original method. Then it is said that the precondition is weaker than the original precondition. The same rules apply to postconditions but in reverse. The method in a subclass must at least guarantee the same output as the method does in the superclass but it could guarantee a higher level of security by making the postcondition stronger [15]. One example can be seen in the square method below.
The method returns the square value of an integer which comes as a parameter to the method. The integer must be a value greater than or equal to zero. The method, as it is written here, is implemented in a class that is inherited by another class. The subclass want to change this method to not only accept integers that are greater than 0 but accept all integers, including negative. If the precondition is changed and the square method accept all integers, including negative, it will be weaker than before since the method in the subclass accepts more input values.
Chapter 4

Component Model

4.1 Component models

Along with other new programming languages which are based on object orientation, component based programming languages have evolved. All component based software engineering is based on a component model but different authors have their own definition of a component model. Bachmann et. al. [2] discusses a few of those definitions and defines a component model as a specification of standards and conventions imposed on developers of components. Since there is no global definition of a component model it does not exist any agreement on the features of a component model either.

Bachmann et. al. has defined some features for a component model. One is the interaction between different components. It is necessary that two components have the same assumptions about the interaction between the two components to be able to interact. They must both agree on what the other requires in order to provide its service and also what the provided service consists of. The standards might also specify which communication protocol to be used, how the control flow is synchronized or how the data is encoded.

A component model should also specify quality attributes. An example of quality attributes is security, like encryption. One advantage of component models is that components can be reused or bought from a third part. Then it is important to know if the component keeps the same level of the quality attributes as the rest of the system.

4.2 Component

A component model is supposed to specify standards and conventions to be applied on components. As with component models it also exist many def-
initions of what a component is and there is not a standardized definition which everyone follows. Lau and Wang describe three definitions in their article [12]. The three definitions have a few things in common. One is that the component is an independent unit which can be used by other software units. The component should also have a usage description which specifies how the component should be used.

Völter analyses one of these definitions in his article [19]. According to Völter, a component is a unit whose purpose is to be composed with other units in order to build a component-based application.

From those definitions you can say that a component is a unit of a system which provides a service, which can be used by other components in the system. The other components does not see the implementation of the functionality but only the provided service and the result. In this way a client of a software does not rely on a specific implementation of a service. This is an advantage since it is possible to change the implementation of a service or even change the component without affecting the client, as long as the new component is compatible with the old one.

Another advantage of a component being a single, independent unit is that it is easier to reuse. A developer can find an old component which provides the requested services and use that instead of developing a new one. This may also lead to a market of components developed by third parties.

4.3 Interface

A component provides certain functionality to its clients and a basic idea of components is that the client should not depend on the implementation but only know which services that are available and how to use them. The method used in a component model to hide the implementation from the client is to use interfaces. Design and implementation decisions that the client depends upon will be consistent over time and therefore put in the interfaces. An example is which methods the client can invoke and which parameters the method wants. The specific decisions for the implementation that the client does not need to have an insight in to use the service will on the other hand be hidden from the client.

An interface must be implemented in a component if the functionality is to be used by a client and on the other hand, a component must implement an interface. If a component does not implement an interface it is impossible for a client to reach the provided services.

A component might implement more than one interface and it is also possible to implement an interface in more than one component. This makes it possible to avoid duplicated definitions of the services that are provided to
4.4 UML

When the component-based applications grow bigger it is harder to get an overview of the different components that belong to a certain application. It is much easier if the components are visualized in pictures or diagrams. A common technique for drawing diagrams with components is UML [10]. One example of how a component can be described can be seen in figure 4.1.

![Graphical representation of a component.](image)

This figure visualize a simple component. The notation is almost the same as when classes are described in UML. The difference is the two small boxes on the left side. An UML diagram can consist of several components. These components usually have a lot of dependencies, which can be described by using arrows. The arrow goes from one component to another component, which implements an interface that is used by the first component.

Figure 4.2 shows a component that displays the interfaces it implements. This version might be better for the user since it is easier to only read information in one document and not in two as the case might be for the component in figure 4.1. It is also easier to show dependencies in this version since the arrow can be drawn to the specific interface that is used.

4.5 Ericsson Component Model

The customers of Ericsson Mobile Communication, EMP, are cell phone manufacturers who provides cell phones to their customers. These cell phone manufacturers does not develop their own mobile phone platform but purchase one from EMP. The platform provides different services that can be used when a cell phone is developed. When the developers of a cell phone
develops new applications, the services in the platform can be used and in that way the hardware can be accessed in a simple and secure manner.

The customers of EMP are supposed to only access the platform through methods which are specified by EMP. In order to do this, a component model has been developed. The component model follows the theory of component models which has been presented in chapter 4.1 to 4.4. All the services which are provided to the customers are specified in interfaces together with all the information which is relevant to the customers. The component model makes it possible for the customers to only see the services and not the implementation since the implementation is made in the component.

It is specified in the interface which methods the specific interface contains and which parameters the methods must have. The parameters can be of two types. They can be used as an input parameter or as an output parameter. As the name tells, the input parameters are used to bring a necessary value to the methods and the output parameters are used to send return values back to the client. The interface also contains comments for all the methods which specify the purpose of a method and how it can be used.

In the Ericsson component model, ECM, it is possible for an interface to inherit from other interfaces. Inheritance in ECM function exactly as it does in object oriented programming. The interface which inherits from another one get all the methods from the other interface and does not have to declare them one more time. This is a widely used functionality since methods exists that must be implemented in every component.

There is a lack of garbage collection in C and reference counting is therefore used to control the number of references to a component that exist. In this way the components know when there is no longer any references to itself and it is possible for it to delete itself. The methods for adding and subtracting the number of references to an interface must be implemented by all interfaces. These methods, together with a method that can give access to

![Component Model](image-url)
another interface of the same component instance, are put in a special interface, which only has those three methods. Since all interfaces should have those methods is it necessary that all interfaces inherit from this interface or from an interface which inherits from it.

![UML diagram of two interfaces.]

The interfaces can be divided into two parts, non-static and static interfaces. The non-static interfaces are the ones described above and inherit from IRoot. A static interface differs from a non-static on three points. The first is that a static interface is initialized at system start up. The second is that a static interface can only be implemented by one component. The third and last point is the lack of instance data. This means that reference counting, constructors and destructors is not needed. The static interfaces inherit from a special root interface instead of the interface which the non-static interfaces inherits from. This means that the static interfaces only inherits the method QueryInterface, which give access to another interface of the same component instance.

The methods are declared in interfaces but the implementation is made in a component. The component also defines data types for state data. To be able to use the methods in an interface it is necessary that an instance of the component has been created. That instance can be seen as an object in a fully object-oriented system.
Figure 4.4: UML diagram of a static interface.
When components are used to develop a software product it is necessary that the interfaces are specified in a correct way. One way to specify interfaces is to use an interface definition language, IDL. IDL gives the opportunity to define an interface with a textual description [3]. A textual description means that no functionality is implemented in the interface but it is separated from the description. One of the benefits of IDL comes from this separation. It makes it possible to offer a service to a client without having the implementation visible. This means that the implementation can be changed without bothering the client because all the client is interested in is how to use the service and that the result is correct.

An interface consists of declarations of a number of functions or methods. These functions must be implemented if the client should get something out of them. It is therefore necessary that a component implements the functionality of the interface. It is up to the component to implement the code of the methods according to the included interfaces in a correct way [5].

Another benefit of the implementation being in the component and not in the interface is that the interface definition language will be programming language independent. The IDL file which contains the interface is compiled using an interface definition language compiler. The output from this compiler consists of stub files and data type declarations which later on can be used when the method is implemented. The generated code in the stub files can be written in more or less any programming language. A pre-requisite is that the compiler can generate code for the specific language [5].

The compiler is supposed to generate code which is based on the information in the interface. A lot of the code written for a component is the same for all components. The code that is the same for all components can then be generated by the compiler instead of written by the developers every time a component is developed. Instead it is enough for the developers to write code for the functionality of the methods. This makes it possible to shorten
Interface Definition Language

the time it takes to develop a new component since a lot of code writing is made by the compiler. The probability of introducing errors in the code is then also decreased since the compiler generates the same code all the time and the human factor is removed.

Several compilers for IDL exist and two of them are CORBA and MIDL [6] and all have their specific features.

5.0.1 Attributes

In some of the interface definition languages that are used today it is possible to have attributes on interface, methods and parameters. This is for example possible in MIDL [6]. An attribute can be used to define the behavior of for example a parameter. In MIDL, attributes are written between square brackets and this is illustrated below.

```plaintext
[attribute]
interface Name
{
    [attribute]
    void methodA([attribute] int parameterA);
}
```

An example of an attribute is that a parameter can be specified as in or out. This means that a parameter can be used as an input value or an output value.

Attributes will be used in the developed prototype to specify preconditions and postconditions for a method.

5.1 Ericsson Interface Definition Language

Ericsson Interface Definition Language, EIDL, is a variant of IDL which is developed by Ericsson Mobile Platform and is very similar to other interface definition languages. EIDL is used for defining the interfaces and components that build up the platform which is delivered to the customers of EMP.

A developer writes the IDL files during the early development of the component development. As said before in this chapter, those files contains only specifications of the interfaces and no implementation. EIDL is, just as any other IDL, not an implementation language. Even if the interfaces not contain any code, the syntax is very similar to the syntax of C. The same keywords are used to include a file and the declaration of a method follow the same standard with some extensions.
In EIDL it is also possible to specify attributes for interfaces, methods and parameters. The same syntax is used as in MIDL and the purpose is also the same, to define different behaviors of an interface, method or parameter. In EIDL it is also possible to specify attributes for `#typedef` when a type is defined and that means that also types can get certain behaviors. The syntax with square brackets are used here to tell the compiler that an attribute is defined for a type.

EIDL also has support for writing comments in the IDL files. The comments must be written in C-style and can be used for two purposes. The first is to write explanations in the interface or component and the second is to write information that is meant to be in the documentation. Everything that is written in the documentation comments will be included in the generated files and can be used when generating the reference manual.

An EIDL file is divided into four sections. The first section shows all the files that have been imported to the current file. The files that are imported are files for those interfaces which are inherited by the current interface or files which contains types that has been defined. The second section includes constants and type definitions. The third section is the interface section. This section defines the interface with the methods that it contains. An interface contains one or several methods that must be implemented by the component which implements the interface. These methods are specified with the parameters it requests. The parameters can be specified as input, output or both depending on the usage and can also have other attributes. Every interface is also given a unique id, a so called interface ID or IID. The IID is used to identify the interface and must therefore be unique. The last section is the component. This part defines the interfaces which the component implements. A component must implement at least one interface but there is no upper limit of interfaces a component may implement. A component also has, as the interfaces, a unique id. This id is a unique identifier of the component and can not be the same as any other, not even an interface.

The different parts described above are usually divided into different files where the constants and definitions are in one file, the specification of the interface in one file and the component is in another one. One example of an interface specification that is written in EIDL can be seen below.
import "IInterfaceA.idl"
[
  uuid(2D92EDFF-AB7B-416f-A04D-55C5C16156BC),
]
interface IIInterfaceB : IIInterfaceA
{
  RVoid Method([in] int parameterA,
               [out] int parameterB);
}

The code written above illustrates an interface specification. The interface, IIInterfaceB, inherits another interface, IIInterfaceA, and therefore are the file containing IIInterfaceA imported in this file. IIInterfaceB also contain one method of its own. The name of an imported interface must be written in an import-statement, otherwise does not the compiler know in which file the interface is specified. Upon specification on an interface, the word 

interface

is used to tell the compiler to handle an interface. After this word, the name of the interface is written and if the interface inherits from another one.

The interface has an attribute which is the specific id for this interface, which is here illustrated as an example of how attributes may be used. Attributes are also used to specify that parameterA is an input parameter and parameterB is an output parameter.

Below a component is specified which implements the interface from above. First it imports the file where the interface is specified. It has then, just as the interface, an id as an attribute which identifies the component. The component is given a name which is written after the word 

component

The word 

component

is used to tell the compiler that a component is found. The last part of the component contains its included interfaces. The word 

interface

is used here followed by the name of the interface to be included.

import "IInterfaceB.idl"
[
  uuid(F1DB5987-230C-46cd-9C32-874103B8E4AA)
]
component CComponentC
{
  interface IIInterfaceB;
There are two reasons for the separation of the definitions, the interface and the component specification. One is that an interface can be implemented by more than one component and the types may exist in even more components. Therefore the amount of duplicated code will be reduced if the types and interfaces are written in separate files. The second reason is that a component only has to include the relevant EIDL file when implementing an outgoing interface and not the whole component. An outgoing interface is an interface which is defined by another component.

Ericsson has developed a compiler for the interface definition language. This compiler is developed to suit the specific needs of Ericsson. This compiler takes an EIDL file as input and produces files with declarations, definitions and skeletons for the component to be implemented. These files contain code that has been generated by the compiler in order to support the developer of the component.
Chapter 6

Design by contract in a component model

One way to carry out an evaluation of design by contract is to develop a prototype where the functionality of design by contract can be used hands on. The prototype can then be used in real life and an objective evaluation of the prototype can be performed by other people than the developer. Most of the present solutions for design by contract are developed with an object oriented programming language, which means that many of the published articles on the subject are targeted at object orientation.

A lot of the theory is also applicable on a component model but there are some differences. In this chapter the different parts of functionality that exists in design by contract are discussed from the perspective of a component model.

6.1 Preconditions and postconditions

An important part of design by contract is to verify the value of the parameters which are sent to a method. In a component model are the methods, which are accessible for a client, specified in interfaces. These interfaces should contain all the information the client needs to know to be able to use the service. It is therefore necessary that the preconditions and postconditions are specified in the interface file.

The developer of the component and its interface should only be required to specify the conditions and not write the actual code that verifies them. Since the interface files already are compiled with an IDL-compiler which takes the information in the interface as input and generates stubs for the component which implements the interface, it is a small change to also generate code that verifies the specified conditions in those stubs.
6.2 Invariants

The idea of invariants in object oriented programming languages is to specify a condition which is global for the class. It could be a condition for a global variable or a state that must be fulfilled. Invariants differ a bit in components since no classes exist.

In object oriented programming languages, the implementation of a class is not completely separated from its specification and in some implementations are the conditions even written in the class itself. The idea of interfaces is to separate the implementation of the interface from the client and make the interface independent of the implementation. The client should only know the provided interfaces and not the specific implementation. It is therefore impossible for the client to know the internal data of a component since that data is not to be specified in the interface.

The client of a component will never see the invariant but it is instead used as an error control for the developers of the component. It is the preconditions that must be fulfilled by the invoker of the method. If the preconditions are correctly specified and the implementation of the interface is correct, then it should be impossible for a client to cause an error.

6.3 Old values

A lot of methods wishes the client to send a parameter as input to the method. The value of this parameter is then used in the execution of that method. When a method uses a value during the execution might a developer want to use the value that was sent to the method when verifying the postcondition. If the purpose of the method for example is to make a calculation and return a value to the client or change the value of a variable might the input value be necessary when verifying the postcondition.

The field of application of using old values in postconditions is slightly different in a component model than in object oriented languages. The implementation of the services should be completely hidden from clients and therefore should internal variables of the component not be used in conditions. The only time when old values should be used is when the postcondition depends on a value that come as an input parameter and is modified during the execution.

Variables from the implementation of the component can not be used in the conditions of a method because this will make a part of the implementation visible to the client. If the component is to be changed into another one it is necessary that the new component has the same variables as the old one because those variables are used in the conditions. You must be able to change
a component without affecting the interface because the clients wants to use the same interface all the time and not change it because the implementation of the functionality or the conditions of a method has changed. Otherwise the clients might have to change their own code and that violates the idea of a component model.

6.4 Condition on typedef

In the Ericsson Component Model it is, just as in C, possible to declare a name as a type using typedef. All these names are also declared in files which are compiled with an IDL compiler and code is generated so that the types can be used in the system.

A lot of methods request parameters that is of a type which is defined in the system. When these parameters are used it is possible for them to have conditions. For example must a type named date have the conditions that the value of month is between 1 and 12 and day between 1 and the number of days in the specific month. These conditions apply to every method which has a parameter of that type and instead of specifying the conditions for every method can the developer specify the conditions for the type.

A condition of a type will always be verified when it is used as a parameter to a method. If the parameter is an inparameter will it be verified before execution and it will be verified after execution if it is an out parameter. In this way it is enough to specify conditions in one place instead of in every method that has a parameter of that specific type.
For the evaluation to be representative it is necessary to see the effects of design by contract when it is used in its real environment. Therefore, a prototype has been developed which implements design by contract into the existing component model of Ericsson Mobile Platform.

Ericsson Mobile Platform has developed their own interface definition language compiler which serve as a base for the developed prototype. The new functionality to be added in the existing compiler is the specification of conditions for the parameters of the methods in an interface and the generation of code which verifies the conditions.

It is possible to generate code for several different programming languages with the existing compiler. Some of those are C, C++ and Java. The limited time available for this project makes it impossible to implement a prototype with code generation for all those programming languages and therefore the scope of this prototype is limited to one language, C.

7.1 Design

7.1.1 Pre- and postcondition

An important part of the prototype solution for introducing design by contract into a component model is based on attributes for methods and data types. The attributes are information that specifies a certain behavior or feature and can be defined for an interface, a method or a parameter. In this case, only attributes for methods will be used. The preconditions and postconditions which are specified only refer to a specific method and therefore are they specified only for that specific method.

It is possible to define both preconditions and postconditions in the attributes for a method. To be able to recognize a precondition or postcondition two new keywords are introduced, pre_condition and post_condition.
These keywords have a parameter which is a string and that string should be the condition. An example can be seen below where a method has one parameter, month.

```c
[pre_condition("month >= 1 && month <= 12")]
int DaysInMonth(int month);
```

`Month` is of type int and represents a month of a year where January is 1 and December is 12. `Month` must be greater than or equal to 1 and less than or equal to 12 since a year has 12 months and those limitations are stated as a precondition.

The syntax of EIDL is similar to C. When the conditions are written in the string it is a requirement that they are written in correct C-code to keep the syntax correct. The C-syntax also helps to generate the code since this prototype only generates code for C. The condition will be tested in an if-statement and it is therefore also a requirement that the condition gives true or false.

A method can have several limitations on its parameters. In this example there are two limitations on the parameter and they are written in the same precondition. If there is several limitations it is necessary to write them in the same string and the conditions are merged with `&&` which is the logical and-operation in C and most other programming languages.

### 7.1.2 Conditions on data types

Ericsson uses a lot of different data types. These data types are more or less common types that have been defined to have another name. This name describes the data it represents which a common data type like `int` does not. The defined data types can be a common type or a struct. In this way the data types are more describing about what they represent than an ordinary type.

When a parameter is sent to a method it may have conditions that must be fulfilled regardless of which method is invoked. These conditions might be a part of the properties of the data type of the parameters. For example that a month must be greater than or equal to 1 and less than or equal to 12. The conditions of the data type must always be fulfilled, otherwise the parameter been assigned an incorrect value.

Instead of specifying conditions in every method for a parameter of the specific data type it is possible to specify the conditions when the type is defined. The conditions of a data type will then always be verified when a method has a parameter of that specific type.
When a condition is specified for a data type, attributes are used as well. A new keyword has been added to the already existing list of attributes and that word is `type_invariant`. The attribute has the same features as the ones used for preconditions and postconditions but the difference is that it is used on the `#typedef`-statement.

An example of a condition for data type can be seen below.

```c
typedef [type_invariant("ClockWeekNr >= 1 &&
                         ClockWeekNr <= 53")]int ClockWeekNr;
```

The type is called `ClockWeekNr` and is thought to be the number of the current week. A year has at maximum 53 weeks and therefore the type has the limitations to be greater than or equal to 1 and less than or equal to 53. This has been specified with a `type_invariant` and will be verified every time this type is used as a parameter.

### 7.1.3 Invariant

In object oriented programming languages, an invariant can be seen as conditions which always must be true for every method in a class. In the developed solution where design by contract is implemented in a component model, an invariant is used to specify conditions that must be fulfilled for all methods in a component.

It is possible for a component to implement several interfaces and since the invariant specifies conditions for a component should the invariant not be specified in the interface. Instead a method is generated in the file containing the implementation of the component where a developer is able to specify the invariant. The method with the invariant will then be called before and after the actual method is invoked to ensure that the specified conditions are fulfilled. This means that the invariant can not be specified in IDL but must be written in the implementation language, which in this case is C.

### 7.1.4 Error handling

If a condition is not fulfilled for a method, the execution of the software will be stopped and a printout to a log will be produced. The information in the printout tells the developer which method got an invalid parameter. The conditions of the method is also printed so the developer does not have to check the documentation of the interface to see the specified conditions. The developer also receive information about the values of the parameters. With
this information, it is possible for the developer to compare the parameters to the conditions. An example of a printout is visualized below.

******** Unfulfilled type invariant!!! ********

Method: CClock_IClock_SetTimeZone
conditions: *pTimeZone >= -48 & *pTimeZone <= 51

pIClock = 0x09bb40a4,
pTimeZone = 55

*******************************************************************************

When the execution of the software is stopped it is possible for the developer to start debugging. The debugger makes it possible to see the code that the method was invoked from and also what happened just before the method was invoked. This makes it possible for the developer to find an error directly when it occurs. This means that the developer does not have to set any breakpoints and go through the code to find the erroneous parameter.

7.1.5 Using input values in postcondition

When a designer of an interface specifies a postcondition for a method it might happen that the value which is guaranteed after execution of the method depends on a value which was input to the method. Execution of the method might change the value of the input parameter. If the input value is to be used during the verification of the postcondition, it is necessary to specify this. The value must be stored temporarily because otherwise the value might be changed during the execution of the method.

When a postcondition uses the value of an input parameter, a specific syntax must be used. The key word $pre(...) is used to inform the compiler that the value of a parameter must be stored and that the stored value should be used when the postcondition is verified. The method in the following example has a parameter of type TUnsigned, which is an integer and represents the number of bytes that is possible to read from a queue. The method is not supposed to change the value of NrOfBytes during its execution and therefore is the postcondition for the method that NrOfBytes should be unchanged after the execution of the method is finished.
Prototype

Void OnReadyForReadEvent([in,out] TUnsigned NrOfBytes,
[in,out] TClientData ClientData);

7.1.6 Which conditions should be verified

When an interface developer implements the functionality of an interface the verification of conditions should almost be mandatory. It is important to verify the conditions in order to find all possible faults during the implementation of the interface.

EMP develops a platform which provide methods with functionality to their customers. When the customers develop their software should, in the ideal world, the platform be free of errors. In that case it might not be necessary for the customer developer to check the postconditions since all methods should return a correct value when the platform is delivered. It is still important to verify the input parameters because a method is only supposed to execute with correct input values. If incorrect values are sent to a method, the programmer is supposed to see the error instantly.

It is possible in the prototype to choose which conditions to verify. The different options are visualized below.

- ENABLE_OPA_ASSERTION
- ENABLE_PRE_CONDITION
- ENABLE_POST_CONDITION
- ENABLE_PRE_INVARIANT
- ENABLE_POST_INVARIANT
- ENABLE_TYPE_INVARIANT

These options are set during compilation with a #define-statement. The ENABLE_OPA_ASSERTION option is used to choose if the conditions should be verified at all. It is meant to be used when the product is delivered and it should not contain any errors. When the software program has been tested and verified and is delivered to the end user it is not necessary to verify the conditions anymore. Then it is possible to maximize the performance of the system by not verifying conditions all conditions. When the software is nearly finished should the methods only get correct parameters, otherwise it would have been detected during the development.

The rest of the options decide which conditions to verify. As the names state, the options must be defined to run the generated code and the options
decide whether preconditions, postconditions, invariants before method call, invariants after method call or conditions on types should be verified or not.

7.1.7 Inheritance

IDL has support for inheritance of interfaces which is widely used in the platform of EMP. When an interface is inherited by another interface, all methods will be included in the subinterface. Then all the specified conditions for those methods are also inherited since a method must have the same constraints in every interface. When an interface is inherited it is not possible to redefine a method because IDL does not support redefinition. It is therefore not possible to redefine any conditions for a method either.

7.2 Design decisions

7.2.1 Attributes

The conditions are written as attributes to methods. This seems to be the natural solution since it is a method that is invoked and the conditions must be fulfilled before the method starts executing.

An alternative solution could have been to specify the conditions for each parameter in a method. This solution would make it necessary for the designer to specify conditions on several places if a method has more than one parameter. Therefore, the chosen solution is preferred since the designer of an interface only have to specify the conditions for a method in one place.

The solution where all conditions for a method’s parameters are specified together also enables comparisons between the parameters. It is for example possible that one parameter must be greater than another one.

It was not always clear that the conditions should be stated as attributes. In some of the existing implementations of design by contract, for example iContract, the conditions are stated in the comment of a method. This technique was also under consideration when designing the prototype but in the existing component model, attributes are used to define the behavior of an interface, method or parameter. If conditions also are stated as attributes will the extended functionality be consistent with the already existing functionality.

7.2.2 Conditions on data types

A parameter type is often used to represent a real object. These objects have properties in the real world which distinguish them from other objects. When an object is represented in a software program it is important to have the
same properties for an object as in the real world. If, for example, a real object has limitations, those limitations must also exist on the data type.

A data type can be used as a parameter in many different methods and the designer could specify the conditions on every method. Then the same information must be stated at many different places and there will be a risk of incorrect conditions. There is always a risk of human mistakes when the same thing is written in many places. If a condition is changed it is necessary for the developer to change it for all methods. Then it is easy to forget to change the condition in some places and the condition for a data type will not be consistent throughout the system.

These problems have been solved by enabling conditions on data types. When a data type is defined using `typedef` the developer can give the type an attribute. In the already existing IDL compiler, it was possible to give attributes to types. It is therefore an easy task to introduce a new keyword to the list of attributes.

Whenever a method which has a parameter of a type with a condition is invoked, the conditions of that type will be verified. This means less work for the developer than if conditions only could be stated for methods since the conditions will only be stated once. It is also easier to change the condition of a type when it exists only in one place.

The word `type_invariant` was chosen because an invariant must always be fulfilled for all methods that are related. The conditions that are specified for a type must also always be fulfilled when a parameter of that type is used. Therefore, the new attribute is called `type_invariant`.

### 7.2.3 Invariant

An invariant of a component can not be specified in an interface. If an invariant should be specified in the interfaces, it would be necessary to know exactly which interfaces a component implements. The combination of interfaces must also be decided when the interface is designed and could not be changed because then there is a risk that the invariant becomes incorrect.

To be able to provide the possibility to specify an invariant, an extra method, in excess of those specified in the interfaces, is generated where the programmer can specify the conditions to be fulfilled in the invariant.

### 7.2.4 Error handling

The whole idea of design by contract is to detect invalid parameters before execution of the methods. If it is to be helpful to the developers, it must produce relevant information and display it to the developer. Otherwise the developer might rather use a regular debugger to find the incorrect parameter.
During the implementation of the prototype, several aspects of which information to print was discussed. The most important information is probably which method received the invalid parameter. It is also important to print the conditions of that method. The developer should not have to use the reference manual to find out which condition has been violated.

If a parameter is invalid, the values of the parameters is also printed. The programmer can then compare the value of the parameters with the conditions with help of the printout to find the invalid parameter.

Sometimes there is a risk that the developer does not know from where the method was invoked. It can also be hard to know how the parameter got the wrong value when looking at the printout without its context. Therefore when a method gets an invalid value of a parameter the execution is stopped and the developer can choose to start the debugger. With the help of the debugger, the developer can go stepwise through the execution to see the code from where the method was invoked. It is then possible to find out the cause of the invalid parameter. This makes it easier to debug the software since the developer does not have to set any breakpoints manually and go stepwise through the program to find an error.

### 7.2.5 Which conditions should be verified

When the software is being developed it is necessary for the developer to be able to choose which conditions to be verified. One thinkable solution is to choose if verification should be made before and after execution, i.e to verify the preconditions or the postconditions.

In the implemented solution the different choices has been further divided. It is possible to choose verification before and after execution but it is also possible to choose between verification of precondition, invariant and type invariant before the execution and postcondition, invariant and type invariant after the execution. This further breakdown was made to give the developer as much freedom as possible when it comes to the verification of conditions. A developer might have situations where it is not necessary to have complete verification and then should he not be forced to have complete verification.

### 7.2.6 Inheritance

In IDL it is possible to inherit from other interfaces. If the inheritance would have worked as it does in an object oriented programming language this might cause problems but there are differences. In the Ericsson IDL can an interface be inherited by another interface but since the methods are implemented in the component and not in the interface the methods will have the same features regardless of which interface it belongs to. The method can only be specified
Prototype

7.3 Limitations

One major limitation of the prototype is that if the designer of an interface writes a condition that can not be tested in an if-statement or if a parameter is misspelled in the condition it will generate a compiler error from the programming language compiler, in this case the C-compiler, and not from the IDL-compiler. This error might be confusing for a developer since it comes from code which was generated by the compiler and not written by the programmer. An improvement would be if the string with the condition was parsed and verified to be a correct statement before the code is compiled.

Another limitation of the prototype is that it is only possible to choose between which conditions to verify before the compilation. It might be desirable for some developers to change the conditions to verify during runtime.

It is also only possible to decide which conditions to verify for the whole system. An improvement of the prototype would be to give the developers complete freedom in deciding for which components a specific condition should be verified. A thinkable scenario would be to only verify the conditions for the modules that are a part of the new development and not modules that are already finished and tested.

A precondition or postcondition attribute requires a string as argument and this string contain the conditions for the method. This string is already limited to be 200 characters. The length has not been extended and therefore the maximum length of the conditions is 200 characters. In a commercial product should this limit be removed or at least be extended to not be a limitation in the length of the conditions.
The evaluation was performed at Ericsson Mobile Platform, EMP, where different developers, who were not involved in developing the prototype, participated in the evaluation to get an objective view of the prototype. During the evaluation of the prototype, the developers were working according to the normal routines at EMP and the only difference was that the compiler was changed to the one supporting design by contract.

When the developers at EMP are developing a new interface the work is divided into different phases. First the new functionality is evaluated and the new interface and component are designed. After the design phase the functionality is implemented and unit tested. When the interface is implemented, it is fully tested and integrated into the rest of the system.

The different phases in the development process are conducted by different developers who are specialized in their area of working. To get an accurate evaluation of the prototype, different developers were working with the prototype in their development work during the different phases. The developers came from departments within EMP who work with the specific task to evaluate. In that way information was gathered from people who work with development on a daily basis. The gathered information reflects how the introduction of design by contract affects the developers in the different phases of the development process.

Due to the limited time available, the evaluation only contain two phases, design and implementation. The test phase was not part of the evaluation, but after the implementation the implementer wrote unit test cases. The developer who wrote the unit test cases answered the questions that concerned the testing part of the evaluation. The questions for the developers are summarized in appendix A.

Before the developers, who used the prototype, began their work, they were given a list of questions to think about during their work. They were given the questions before the evaluation started to have them in mind during
the work. In this way it was made sure that they knew the focus of the evaluation and were able to answer all the questions. These questions were asked both to get feedback on the prototype and to get feedback on the process of working with design by contract. The questions were asked to get information which is relevant when the research questions in section 2.2.1 were answered.

8.1 Execution

In the evaluation, an interface with new functionality was developed. The process when the interface was developed followed the standard procedures of development in EMP and started with the design phase.

During the design phase, the interface was designed by a designer who evaluated the needed functionality and specified the methods of the interface. The interface should have a name and parameters that matched the functionality. When the methods were specified, the designer also decided which conditions must be fulfilled before and after the execution of the method. This is something the designer must do even with the regular compiler which does not support design by contract but now he specified the conditions in the interface through the new keywords.

When the design of the interface was completed, it was given to a developer with the task to implement the functionality. After the implementation of the interface the new software unit was tested by the same person who did the implementation. In this phase the functionality of design by contract was visible to the developer since error messages were displayed when a method got an incorrect parameter.

8.1.1 Participating developers

The designer who took part in the evaluation is a senior developer with a long time of working experience in industry. He has worked with design for the last couple of years and his programming skills are above the average software developer. He has worked with design of software systems in EMP for more than half his professional working life and is well aware of the development methodologies in EMP and which problems that might occur.

The developer who used the prototype during the implementation of the interface has worked in the industry for a couple of years and ranks his programming skills to be well above the average developer. He has experiences of both design and implementation of software systems but has mainly worked with implementation at EMP. The skills in both implementation and design makes it possible to see benefits and drawbacks from both an implementers
and a designers point of view and then also see the consequences from both angles.

8.2 Results

The evaluation was divided into different phases and was performed by two different developers. The developers were asked different question about their opinion of the prototype and also about the using of design by contract. This section is divided in three parts and summarizes the feedback that comes from the participating developers. The questions for the developers are summarized in appendix A.

8.2.1 Design

The designer thought that specifying conditions for the parameters of a method was a good idea. It is already a part of the work today when an interface is designed and will therefore not bring any extra work for the designer. The designer must think about constraints for a method when it is designed to know how the functionality should work and which parameters a method should have. This information must also be given to both the developer who implements the interface and the developers who call methods in the interface when they are developing their applications.

There is no formal syntax to describe constraints today and to notify the implementers about the constraints of a method. Specification of the conditions for a parameter might not even involve any additional work for a designer since they already have to think about the constraints of a method. It might instead bring a decreased workload for the designers since a formal specification method which can be used to specify conditions will exist.

The syntax in the developed prototype is very simple and is based on a statement which should be evaluated in an if-statement. This syntax might be too simple if design by contract should be really useful. One improvement of the prototype would be to invent a completely new syntax for defining conditions. Then would it also be necessary to make a parser which convert the conditions and produce C-code.

8.2.2 Implementation

The developer who implemented the new interface and used the prototype could not see any changes in his ways of working and the workload was about the same. Even if the designer specified the conditions for a method the developer thought about the constraints and wrote code for verification anyway.
He meant that the amount of written code depends on how well the designer had specified the conditions in the design. The developer must trust the designer to have a correct and complete specification of the conditions in order not to write his own code for verification of the parameters.

When a method got a parameter with an incorrect value, the prototype prints an error message with information about the parameters and the conditions. The developer thought the idea of an error message to be good but the values of the parameters might not be necessary information. An error message with the values of the parameters might be helpful for some developers but it is easier and better to use a debugger to check the values. It is then also possible to go stepwise through the system and see the values before the method was invoked and thereby see what caused the incorrect values.

The printout could be extended with information about the method from where the invalid method call came. This could be substantial information for a developer since a method might be invoked from several different places and then it is good to know where the method was invoked. In some way, this already exist since the debugger is started and then it is possible to follow the execution and see from where a method was invoked.

The implementation developer could not see any difference in the quality of his work. When an interface is developed today, different verifications of the parameters are already implemented by the developer. The designer can describe conditions for the parameters when an interface is designed and if that is the case the developer will implement code to verify those conditions. The difference with design by contract is that those conditions can be specified with a formal language and the verification code will be generated by the compiler.

8.2.3 Unit testing

The developer that performed the unit testing on the developed interface do not think that there will be any major differences in the testing process. The interface is designed by a human developer and there will always be a risk of human mistakes. If one or more conditions are left out during the design of an interface because a designer missed one special case of the method the method will not be verified for that case. It is therefore almost necessary to also have test cases in the test suite that verifies both correct and incorrect parameters to the methods. The reason for having test cases is to prevent human mistakes and errors might still be introduced when the conditions are specified in the design phase.

The prototype that is developed today and used in the evaluation stops the execution if a method receives an invalid value of a parameter. This may become a problem if the test suite involve test cases which test a method with
invalid parameters because then the execution is canceled and the test suite must be restarted. To be able to run the complete test suite the test cases must be removed, otherwise the execution will be canceled again. As a tester you would like to have a pass from the whole test suite and that can not be done without removing test cases, which must be made manually.

8.3 Validity

During the evaluation the prototype was only tested by two developers, one designer and one implementer. To get an even more accurate evaluation additional developers should participate. The short time available for the evaluation limited the number of participating developers to two but still, the evaluation brought an impression of design by contract.

The difference between the prototype compiler and the original compiler, which does not accept conditions for methods, will only affect the developer when the written code is incorrect. The behaviour will otherwise be exactly the same from a developer's point of view. The experiences of the developers might therefore depend on their programming skill. A very skilled programmer might not notice any difference in the compiler but a less skilled programmer might get help in finding errors in the produced code. A more accurate evaluation will therefore need a mixture of developers with varying skills.

The result of the evaluation also depend on the programming skills of the designer. The designer specifies the conditions and it is important that the designer is skilled enough to specify the conditions to completely cover the limitations of the method's parameters. If the designer miss a condition or a condition is incorrect, the programmer might get a incorrect experience from design by contract.

8.4 Conclusions

Given the feedback from the developers who participated in this evaluation is the thought of specifying conditions in the interface a good idea. This can not be done in an interface definition language today but according to the participating designer there is need of a formal method for specifying conditions.

Another conclusion from this evaluation is that if the theory of design by contract could be used in its full extent the implementer must be able to trust the designer to specify correct and complete conditions for the methods. There is a risk that a designer misses one condition and if the implementer
thinks that the conditions are complete he does not verify all possible inputs. There is also a risk when the test cases are written that the developer trust the designer and does not test any parameter values and thereby misses to verify all possible inputs.

The whole test suite could not be run because the execution was canceled when a method got an invalid parameter. This is a clear disadvantage since the test cases might be necessary to detect human mistakes in the design as well as in the implementation. It might not be a proper solution to remove those test cases because then the methods are not tested with invalid parameters and are depending completely on correct specifications from the designer. A better solution might be to introduce different modes that can be chosen. The default mode may result in a cancelled execution and another mode, a test mode, may be chosen during testing and then the execution should not be cancelled.

When it comes to the quality of the product, design by contract does not automatically bring improved quality. As it is today the designer come up with conditions that must be true for the method to execute. Then it is up to the developer to implement the code which verifies the conditions. The change with design by contract is that the code is generated by the compiler but it is still a human developer who specifies the conditions and there is a risk of faulty or missing conditions. The quality of the product might be increased because with design by contract is the designer given a formal method to specify conditions and notify other developers about the constraints of a method.

The syntax that is used today for describing the conditions might not be enough to describe the different conditions. According to the designer who took part in the evaluation there is a need to describe more complex conditions than is possible with the current prototype. If that should be possible it might be a good decision to invent a completely new syntax to fulfill the needs of the designers and then also make a parser which generates the code. With the current solution a parser would be needed anyway since it is not just C-code that should be generated. IDL is supposed to be language independent which means that code for any language could be generated. Therefore is a parser needed anyway to transform the condition statement to the correct language.

In the developed prototype, the generated code is not placed in the method where the developer implements his code but in a wrapper method that is invoked before the actual method. The thought behind this design was to have the generated code separated from the code which is written by a developer. This means that a developer only have to look at the conditions in the specification and does not have to think of how they are verified. The developer who used the prototype did not think this was a good idea since he wants to see the verification of the parameters and in that way be able to detect miss-
ing verification of parameters. There are advantages and disadvantages with both solutions and it might therefore be necessary with a larger evaluation with more developers before a final solution is implemented in a product.

Conditions for the methods will at first hand be specified for new interfaces. It might not even be possible to specify conditions for methods in an already existing system because there are too many methods and it will be a very time-consuming task to specify conditions for all the already existing methods in a system. It might give negative consequences if some invalid parameters to some method result in canceled execution and an error message while other methods will accept the parameters and execute. It might not be a big problem for the developers on EMP since they know if it is a new or old interface that is developed and if design by contract is used in that interface or not. The problem is when the product is delivered to the customers who look at it as a unit and do not know if the interface they are invoking has verification of the parameters or not.
9.1 Discussion

A conclusion of this report is that a formal method for defining conditions that must be fulfilled before a method starts executing might be helpful for the designer of an interface. When an interface is designed today, the developer who designs it specify constraints for the methods and he must communicate that information to the developer who implements the functionality. There is no formal language to do this and it is up to the developer to find a way to inform the other developers. A possible benefit of a formal language for specifying conditions is increased quality of the product since there is a decreased risk of misunderstanding between the designer and other developers are decreased.

When the code which verifies the parameters is generated automatically by the compiler the responsibility is moved from the developer who implements the interface to the one who designs it. This change in responsibility does not guarantee an improved quality of the product but it might remove possible misunderstandings between the developers. A possible change in the solution is to generate the code so that it is visible to the developers and give them the opportunity to add more conditions to the ones already specified. If the prototype is extended with this functionality can the developers only be allowed to see and add extra conditions but they can not be allowed to change any of those that are specified by a designer. The conditions which are stated in the interface must also be applied later on when the method is invoked. In this way the implementer is given a certain feeling of control of the execution of the method, something that also was requested by the interviewed developer.

One of the research questions was how design by contract affects the testing of a software product. As discussed in chapter 8, the use of design by contract does not bring any major differences to the testing part of software development. Even if the actual code that verifies the parameters are gen-
erated by the compiler are the conditions which the code is based on still
written by a human developer. It is therefore still necessary to have test cases
which verifies the accepted parameters of a method since all developers might
make an error and an incorrect condition might bring a huge quality risk to
a system.

The developer who participated in the evaluation of design by contract
when applied to an interface definition language did not see any differences
in his ways of working or any changes in the software module he produced.
When compared to his earlier experiences, the quality of the system has the
same level as if the interface would have been developed without design by
contract. The result of this evaluation might not be representative for all
software developers since the programming skills of the participating developer
are well above those of an average developer. The quality of the software
components that are produced by the developer might already have a high
level of quality and it may therefore not be possible to increase the level of
quality. If a less skilled developer had been participating in the evaluation
might it be possible to see an increased level of quality compared to the earlier
components that has been produced by the same developer. Therefore would
it be desirable to extend the evaluation with more developers with different
skills and earlier experiences.

In this evaluation only developers from Ericsson Mobile Platform partici-
pated. This means that the developers only used design by contract when the
functionality of an interface was implemented. Those developers only see the
difference with design by contract when they implement a method in an incor-
crect way and violates a postcondition. Another part of the using of design by
contract is when the interfaces are used in the application development that
is performed by the customers of EMP. An interesting part to evaluate is the
experiences of a developer who invokes the methods with a contract and also
to see the changes in the development work it might bring for an application
developer. It is the developers who invoke the methods who will see the big
differences since they are the ones sending parameters to a method when it is
used, not the developers of EMP.

The customers of EMP have developers who invoke the methods provided
by EMP. If the product sold by EMP provide the functionality of design by
contract those developers will see the advantages in verifying the parameters
of those methods. A comparison can be made with the Ariane 5 project where
the developers did not have knowledge of the limitations of the methods and
the entire system crashed. The methods of EMP will not accept any invalid
parameters and the developers can be sure that the methods are used in a
correct way.

When introducing design by contract in a system there is a cost of in-
troducing the theory behind it to all developers and teach them about the
functionality and possibilities it brings. This cost might be the biggest when
design by contract is introduced in a system, especially in an already exist-
ing system and in a large company with a lot of developers. There will also
be a transition period from when design by contract is introduced until the
developers trust the conditions to be correct.

There is also a one-time cost for extending the compiler with design by
contract but once it is done the compiler is ready to be used by everyone.
This means that when a new interface and a new component are developed
conditions can be specified for the methods. A huge cost of introducing design
by contract into an already existing system would be to specify conditions for
all the methods that already exit. It is not necessary to specify conditions
for all methods in a system, it might also exist methods that does not have
any conditions and therefore must the compiler be backwards compatible.
There might exist a risk when some methods, which have clear constraints,
do not have any specified conditions. An application developer might get used
to have an error message printed when a parameter is incorrect and if that
message do not come from all the methods in a system when an invalid value
is used he might get confused. It is not a problem for a developer within
EMP since they know that conditions only exist on interfaces developed after
a certain date. For a developer on a company which is a customer to EMP the
situation might be different. When a customer buy the platform of EMP, it
looks like one unit and the developers can not know if an interface is developed
before or after conditions for the methods were introduced.

A conclusion of this report is that the greatest benefits with introducing
design by contract into an interface definition language will be for the designer
of an interface since they get a formalized method to specify the constraints
of a method and the developers who invoke the methods in an interface.
Providing conditions for the methods does not affect the developers of the
methods in any major way because they still have to work in the same way
when writing code and later on test the methods.

9.2 Future work

In this evaluation only two developers were participating. To be able to get
a more accurate result it is necessary to extend the evaluation with more
participating developers. The developers should also have different earlier
experiences. For example a couple of senior developers mixed with some
junior developers.

In a further evaluation of the prototype should also developers from the
customers of EMP participate. This means that developers who invoke the
methods which are provided by EMP uses the compiler with conditions. This
might give a new perspective of the subject since those developers see the 
conditions from the other side since they must fulfill them and not trust that 
they have been fulfilled as the developers of EMP do.

For the compiler, which is extended with design by contract, to be really 
useful and give as much support as possible to the developers it must also be 
further developed. The developers who participated in the evaluation asked 
for possibilities to specify even more complex conditions and for that to be 
possible the syntax must be extended. With a more complex syntax a parser 
is also needed which verifies the syntax of the conditions and transform them 
to the code which should be generated.
Bibliography


A.1 General questions about design by contract

Write shortly of your experiences of design by contract. Might it contribute to the product? Does it make the development easier or more complicated? Do you have any thoughts of improvements for my prototype?

A.2 Questions about the participants earlier experience

1. For how long have you worked with software engineering?
2. For how long have you worked at Ericsson Mobile Communication?
3. For how long have you worked with design/implementation?
4. On a scale from 1 - 10, how would you rank your programming skills?

A.3 Questions for the designer

1. From what I have understood are you already writing conditions for methods but not in a formal language. Do you think that that writing conditions for the parameters of method as attributes is a good solution?
2. What do you think about the syntax of the conditions? Would you like to have it in a different way?
3. Is there a need of specifying more information than is possible with the current solution?
4. How much additional work is it for you to specify conditions for the parameters of the methods?
5. How long time does it take to design an interface with conditions compared to without conditions? Answer in percent.

6. Do you think it is easier to inform both the implementers of the interface and the clients about the conditions when they are specified nearby the method they limit?

7. If design by contract should be introduced in the product will conditions at first hand be specified for new interfaces. What do you think about the possibilities of specifying conditions for already developed interfaces?

A.4 Questions for the implementer

1. Will design by contract affect your ways of working when an interface is implemented? I.e. when you know that a method can only get values of its parameters which is valid according to the specification.

2. Is it possible that you still write code that verifies the parameters before the execution?

3. Did it happen during the implementation that a condition was not fulfilled? If yes: What do you think about the print out to the log?

4. Would you like to change anything in the print out that would make it easier to find an error?

5. If you look at the differences in not getting a print out and getting one, do you think it helps in finding an error when you get an error message directly if a parameter does not fulfill its conditions?

6. Do you think the quality of your work will be improved if you know that the methods have conditions that must be fulfilled and that those conditions will be verified before execution?

7. Do you think that verification of parameters contribute to the final product or do you think that controls of parameters is up to each developer to implement if they are necessary?

A.5 Questions for the unit tester

1. Do you, as a tester, think that the specified conditions in the interfaces are enough to trust the controls and not write any test cases for the same thing yourself?
2. Is it possible that you might change view regarding necessary test cases if you as a tester knows that the specified conditions will be verified?

3. Can you see a risk in not only have conditions in parts of the system? I.e. only in new developed interfaces and not those that were developed before design by contract was introduced?

4. Do you think the quality of the product will be improved if the methods have conditions that must be fulfilled and that those conditions will be verified during the execution?