Patterns for Injection of Mock Objects in a Modeling Environment

Fabian Wiberg

Tutor, Ola Leifler
Examiner, Kristian Sandahl
Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Fabian Wiberg
Abstract

Capsules are modeling language elements which are sometimes used to develop real-time software. One way to test such capsule elements without relying on dependencies to other units is to use mock objects. The aim of the study was to look at existing object-oriented design patterns and investigate how they could be used for capsules, in order to perform mock testing. The focus was to find solutions that were usable from the programmers’ point of view, meaning that they should promote high user effectiveness, efficiency and satisfaction when implementing them. It was also important that program efficiency wasn’t affected negatively. 5 design- or refactoring patterns were adapted for capsules: Constructor Injection, Setter Injection, Parameterize Method, Factory Method and Abstract Factory. Those patterns were evaluated by 5 programmers in a usability test, where Incarnate Injection (an adaptation of Constructor Injection) and Abstract Factory were considered most usable. Incarnate Injection seemed to be easier to implement and promoted high user efficiency, while Abstract Factory was considered more flexible. The performance tests indicated that Abstract Factory compromises program efficiency when the factory product is resource-heavy and is required frequently by dependent capsules. The study showed that it is possible to adapt design patterns to capsules by looking at conceptual similarities between capsules and classes. However, there are cases when this adaptation is impossible. Furthermore, even when adaptation is possible, it was apparent that different patterns adapt differently well to capsules.
Acknowledgements

This study is a Master’s thesis made at the program Master of Science in Computer Science at Linköping University.

The study was made at Ericsson in Linköping and I owe a big thank you to the Ericsson staff for making it such an enjoyable time for me. In particular, I would like to thank my supervisor, Johan Westling, for giving me the guidance I needed as well as valuable input on my work and the report. I would also like to thank Johan Wibeck for giving me the opportunity to come and do my thesis at Ericsson.

I would like to thank my supervisor at the university, Ola Leifler, for giving this thesis direction and for giving me goals to work towards, as well as motivation during the finalization of this report. I would also like to thank my examiner, Kristian Sandahl, for making sure that my work was up to standard. I want to thank my opponent, Henrik Laurentz, for his feedback which raised the quality of this report.

At last, I would like to direct a big thank you to the Ericsson developers who dedicated valuable working hours to participate in interviews and user tests. This study, as a whole, would not have been the same without your cooperation.

Fabian Wiberg
# Table of contents

Chapter 1  Introduction ........................................................................................................ 7  
1.1  Aim............................................................................................................................... 7  
1.2  Research Questions ..................................................................................................... 8  
1.3  Limitations .................................................................................................................. 8  
1.4  Definitions ................................................................................................................... 8  
1.5  Report Disposition ....................................................................................................... 8  

Chapter 2  Background ...................................................................................................... 11  
2.1  The Modeling Environment ......................................................................................... 11  
2.1.1  Modeling Concepts ................................................................................................. 11  
2.1.2  The Test Framework ............................................................................................... 12  

Chapter 3  Theory ............................................................................................................ 13  
3.1  Design Patterns .......................................................................................................... 13  
3.2  The Dependency Inversion Principle and Related Design Patterns ......................... 13  
3.2.1  The Dependency Inversion Principle ..................................................................... 13  
3.2.2  More on Dependencies .......................................................................................... 15  
3.2.3  The Dependency Injection Pattern ........................................................................ 15  
3.2.4  Factory Patterns .................................................................................................. 16  
3.2.5  The Service Locator Pattern ................................................................................ 17  
3.2.6  The Parameterize Method Refactoring Pattern ..................................................... 18  
3.3  Differences and Similarities between C++ and RSA-RTE ......................................... 18  
3.4  Mock Objects ............................................................................................................. 21  

Chapter 4  Method ............................................................................................................ 23  
4.1  Pre-study: Choosing Evaluation Criteria .................................................................... 23  
4.2  Design and Implementation ......................................................................................... 24  
4.2.1  Design Pattern Research ....................................................................................... 25  
4.2.2  Assessment of Potential ....................................................................................... 26  
4.2.3  Design Method ..................................................................................................... 27  
4.2.4  Implementation Method ....................................................................................... 27  
4.3  Evaluation .................................................................................................................. 27  
4.3.1  Evaluation Metrics ............................................................................................... 27  
4.3.2  Methods for Gathering Evaluation Data ................................................................ 28  

Chapter 5  Design and Implementation ............................................................................. 33  
5.1  Finding and Evaluating Possible Solutions .................................................................. 33  
5.1.1  Found Solutions ................................................................................................. 33  
5.1.2  Estimation of Pattern Potential ........................................................................... 33  
5.2  Adapting Found Solutions to the Modeling Environment ........................................ 35  
5.2.1  Incarnate Injection: An Adapted Design of Constructor Injection ....................... 36  
5.2.2  Port Injection: An Adapted Design of Setter Injection ....................................... 37  
5.2.3  Parameterize Transition: An Adapted Design of Parameterize Method ............... 38  
5.2.4  Adapting Factory Method .................................................................................... 38  
5.2.5  Adapting Abstract Factory ................................................................................... 39  
5.2.6  Adapting Interface Injection ............................................................................... 40
5.3 Implementing the Solutions........................................................................................................40
  5.3.1 Incarnate Injection..................................................................................................................41
  5.3.2 Port Injection.........................................................................................................................42
  5.3.3 Parameterize Transition........................................................................................................43
  5.3.4 Factory Method.....................................................................................................................43
  5.3.5 Abstract Factory....................................................................................................................44
  5.3.1 Impact on Performance..........................................................................................................45

Chapter 6 Evaluation ..........................................................................................................................47
  6.1 Usability ......................................................................................................................................47
  6.2 Performance...............................................................................................................................49

Chapter 7 Discussion ..........................................................................................................................51
  7.1 Evaluation Results .....................................................................................................................51
  7.2 Method......................................................................................................................................52
  7.3 The Work in a Wider Context ....................................................................................................53

Chapter 8 Conclusion ..........................................................................................................................55

References 57
Introduction

Unit testing is an almost mandatory part of any software developing process today. The purpose of unit testing is to ensure that each delimited unit of code behaves as expected when the units are isolated from each other. In object-oriented programming, the most common – and arguably most natural – delimitation to make is the one between classes. In order to facilitate the isolation between classes it is often necessary to “break” dependencies that a class might have on other classes. This is quite logical: if the class under test is dependent on another class, it will be more difficult to determine which of the classes causes tests to fail. In order to get reliable output from unit tests, it is also important that the test environment is as similar to the “real” runtime environment as possible. In other words: breaking a dependency should not alter the overall behavior of the class under test. These demands combined create the need for simulating, or faking, dependencies. This can be done by replacing a dependency with a stub or a mock. With stubs or mocks there is no need to change the class under test, meaning that its’ behavior is kept as authentic as possible.

Mocks are different from stubs in the sense that they actively monitor the calling behavior of the class under test. A mock object can therefore act as an additional tool for providing information about the correctness of the code. Typically, a mock object is created in the test case and passed to the class under test. The class under test then uses the mock without knowing its’ runtime type, since it depends only on the interface of the mock object. In order for this to work, the object under test must not create the depended on object by itself – if this were the case it would know the runtime type. This is a case of Inversion of Control (IoC) in action. IoC is a part of the Dependency Inversion Principle (DIP), which is one of the core design principles for object-oriented programming. Most programmers are familiar with the concept of Dependency Injection, which is a powerful technique for enforcing IoC.

Significant parts of the software in the Ericsson Long Term Evolution (LTE) project are developed using a modeling language for real-time software, called RSA-RTE [1]. In RSA-RTE a special type of model elements, called capsules, are used to generate thread-safe C++ code. Capsules are different from classes in the way they communicate with other elements and the fact that their lifecycle is managed by an underlying library. Nevertheless, capsules have the same demands as classes when it comes to unit testing. But how does Dependency Injection work for capsules? Experience has shown that injection for capsules is not as straight-forward as it is for classes. This poses a problem when we want to test capsules with mock objects: How do we hand the mock object over to the capsule under test? While there are several well-documented and well-proven design patterns exercising IoC in object-oriented programming languages such as C++ and Java, very little work has been focused on IoC for capsules.

1.1 Aim

The aim of the study was to look at existing IoC-enforcing design patterns and investigate how they could be used for capsules, in order to simplify unit testing. The focus was to find solutions that were usable from the programmers’ point of view, meaning that they should promote high user effectiveness, efficiency and satisfaction when implementing them. Also, since the modeling language was designed
for real-time elements, so should the found solutions. It was therefore important that program efficiency wasn’t affected negatively.

1.2 Research Questions

The study strove to answer the following question:

Which design solution, exercising Inversion of Control, can be regarded as most usable in order to improve unit testing of real-time software models?

Here, the term “design solution” is cautiously used instead of “design pattern”. The reason is explained in section 1.4. The term “usable” denotes the effectiveness, efficiency and satisfaction with which the users can apply the solutions.

1.3 Limitations

There are several ways to create mock objects and many useful frameworks exist for that purpose. The solutions found in this study are completely independent of how the injected mock objects were created, and the study does not touch on the subject of creating mock objects.

1.4 Definitions

The meaning of some of the terms used throughout this report may differ slightly from how they are used in other published literature in the technological field. These terms are explained in this section.

The terms “solution”, “design pattern” and “pattern” are used interchangeably in this report when referring to design patterns which are adapted to solve the problem of injecting mocks into capsules. Strictly speaking, the solutions in this study should not be considered design patterns on their own, but rather instantiations of existing design patterns. However, the solutions are presented in a way that should make them easy to reuse in various situations, and in the micro-world of capsule programming the terms “pattern” or “design pattern” can be justified.

Central in this study was the concept of dependencies. A dependency can be defined as “A semantic relationship between two things in which a change to one thing may affect the semantics of the other thing” [2]. In this report, the element at the originating end of a dependency will be denoted by the word dependent, while an element at the receiving end of a dependency will be denoted by the word dependee.

It is easy to mix up or misunderstand the terms Dependency Injection, mock injection and dependency inversion. In this report, dependency inversion is the design principle introduced by Martin in [3]. Dependency Injection is a specific design pattern adhering to the Dependency Inversion Principle. Mock injection is the act of injecting a mock object into another object during unit testing. Dependency Injection is one way to facilitate mock injection. However, there are other solutions, as will be apparent to the reader of this report. The Dependency Inversion Principle, dependencies and other related concepts are explained more in depth in section 3.2.

1.5 Report Disposition

These chapters will follow in this report:

Chapter 2 Background provides information about the modeling environment used at Ericsson.
Chapter 3 Theory presents the theory that was relied on in order to reach the resulting solutions.
Chapter 4 Method explains and motivates the choices of approach that were made.
Chapter 5 Design and Implementation shows the results of the design- and implementation process, i.e. what design patterns were considered, how they were assessed and adapted to the modeling language and the resulting implementation.

Chapter 6 Evaluation presents the results of evaluating the found solutions.

Chapter 7 Discussion discusses the results and how they were influenced by choices of method.

Chapter 8 Conclusion discusses the overall outcome of the study.

Appendix A includes the material used for the usability tests.
Chapter 2

Background

2.1 The Modeling Environment

A significant part of the software in the Ericsson LTE project is developed using a modeling tool for real-time software called Rational Software Architect - RealTime Edition (RSA-RTE) [1]. RSA-RTE was developed by IBM and is an Eclipse-based modeling tool using UML2 elements. The purpose of RSA-RTE is to facilitate development of real-time software by using a real-time subset of UML elements and using a profile known as UML-RT, which incorporates elements such as capsules etc. The created RSA-RTE models are transformed into C++ code which is compiled into an executable using a normal C++ compiler.

2.1.1 Modeling Concepts

The main components used in the RSA-RTE modeling environment are capsules and classes. Classes are transformed into normal C++ classes and are modeled using common elements such as attributes and operations. Capsules are more complex. In addition to operations, their interfaces are described using ports, which are modeled as attributes of the capsule. Ports can be seen as instantiations of protocols which define the possible events that can be invoked on the port. Invoking an event on a port is conceptually the same thing as passing a message through the port. There are in-events and out-events. The rationale for using ports to communicate with a capsule is that the surrounding framework can take care of synchronization due to a concept called run-to-completion semantics. This means that messages, even when sent from another thread of execution, are dispatched one-at-a-time, from one in-queue. Thus race conditions are avoided.

The behavior of capsules, and sometimes also classes, is described using state machines. At any point during its’ lifetime, a capsule has an active state. How to react to an incoming event is decided by the detail-level code located in the transitions between the active state and another state. The detail-level code is written in C++. This is where the logic is defined. Often a transition is used to manipulate data, send messages to other capsules (using service functions of the RT Service Library), or call functions in classes.

Both state machines and the structure of a capsule can be hierarchical. A state can consist of sub-states which further refine the behavior of the capsule. Similarly, a capsule can contain any number of sub-capsules, called capsule parts, which deal with different responsibilities inside a containing capsule. The composition of a capsule is hidden from the outside, meaning that only the ports of the outermost capsule are exposed to the outside. These ports may be behavioral ports, meaning that they trigger a transition in the state-machine of the top capsule, or connector ports, which pass on incoming or outgoing messages to or from a capsule part. This, however, is all hidden from the users of a capsule.

Figure 1 and Code example 1 show the state machine and the detail-level code of a simple capsule. The transition “SayHello” is triggered by an incoming event on the port “pHello” and generates an out-event on the same port. In order for “SayHello” to be triggered, “Waiting” must be the active state. The new active state after executing “SayHello” would be “SaidHello”.


Ericsson uses a test framework for unit testing of capsules, called Tools Basic Test, which has been developed in-house. The framework is basically an executable program built from a capsule model. When the program is run, macros from a configuration file are read and executed. Among other things, the macros tell the program which test suites and test cases should be executed. Test suites are implemented as passive classes and test cases within a test suite are implemented as member functions.

The capsule under test is included in the model as a capsule part of the top capsule of the program. Test cases communicate with the capsule under test using macros for sending and receiving signals. Exactly how the framework works will be left out of this study. It suffices to know that there are a number of macros available to test cases which are used to communicate “indirectly” with the test framework. The most important ones are those for sending and receiving signals to and from the capsule under test as well as assertion (verify) macros which cause a test case to fail if an assertion fails.

A simple example of how those macros are used is demonstrated in Code example 2, which shows a simple test case for the “hello” capsule in the previous section.

Google Mock has been integrated with the Tools Basic Test framework, making it easy to create and use mock objects directly in test cases.

```
//Transition “SayHello”
pHello.Response("Hello!").send();
```

Code example 1. Detail-level code.

```
2.1.2 The Test Framework

Ericsson uses a test framework for unit testing of capsules, called Tools Basic Test, which has been developed in-house. The framework is basically an executable program built from a capsule model. When the program is run, macros from a configuration file are read and executed. Among other things, the macros tell the program which test suites and test cases should be executed. Test suites are implemented as passive classes and test cases within a test suite are implemented as member functions.

The capsule under test is included in the model as a capsule part of the top capsule of the program. Test cases communicate with the capsule under test using macros for sending and receiving signals. Exactly how the framework works will be left out of this study. It suffices to know that there are a number of macros available to test cases which are used to communicate “indirectly” with the test framework. The most important ones are those for sending and receiving signals to and from the capsule under test as well as assertion (verify) macros which cause a test case to fail if an assertion fails.

A simple example of how those macros are used is demonstrated in Code example 2, which shows a simple test case for the “hello” capsule in the previous section.

Google Mock has been integrated with the Tools Basic Test framework, making it easy to create and use mock objects directly in test cases.

```
//Test case
//Send a signal, invoking event Hello on the port pHello.
SEND_VOID(pHello, Hello);
//Receive a signal invoking event Response on port pHello. The response will
//be assigned to the variable “answer” of type char[].
RECEIVE(pHello, Response, char[], answer);
//Verify that the capsule under test behaved as expected.
VERIFY(strcmp(answer, “Hello!”) == 0);
return true;
```

Code example 2. A simple test case.
Chapter 3

Theory

3.1 Design Patterns

Riehle et al. describes patterns as mental concepts derived from our experience, which act as guidelines for our way of interacting with the world. Their definition of a pattern is *an abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts* [4]. A pattern can only materialize as an instance in a certain context. But the notion of a pattern does not need to be bound to a specific instantiation. The same pattern can be applied in different settings.

The Gang-of-Four\(^1\) define design patterns as “*descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*” [5]. However, they also state that “a pattern is like a template that can be applied in many different situations”. Thus, a design pattern, even when described in a particular programming language, should be applicable to different situations as long as the general design problem is the same.

Alexander describes a pattern as a rule which describes the relation between a context, a problem and a solution [6]. But once again, the environment in which a pattern might occur is not stated by the description of that pattern.

[4] and [6] seem to agree that a solution to a general problem can only become a pattern once it has been used to solve that problem a certain amount of times. This implies that a pattern cannot be consciously created, but is rather abstracted from concrete instances in contexts where the pattern has recurred. [5] and [7], on the other hand, seem to be of the opinion that a pattern can in fact be created by will, although it is a time-consuming process and empirical evidence of the usefulness of the pattern would be needed in order to establish the pattern.

3.2 The Dependency Inversion Principle and Related Design Patterns

3.2.1 The Dependency Inversion Principle

The Dependency Inversion Principle (DIP) was coined by Martin in the mid-90s and is one of five object-oriented design principles gathered in [3]. The principle can be summed up in the following two guidelines:

\(a\) High-level modules should not depend on low-level modules. Both should depend on abstractions.

\(b\) Abstractions should not depend on details. Details should depend on abstractions.

---

\(^{\text{1}}\) The Gang-of-Four (GoF) is a popular reference to the four authors of the book *Design Patterns: Elements of Reusable Object-Oriented Software*: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.
In [3], Martin talks about how a layered architecture benefits from DIP. In a naïve layered architecture, where high-level modules depend on low-level modules, dependencies are transitive. This means that a dependent is not only depending on its’ dependees, but also on all of their dependees, and so on down the layers. If one module changes, all modules in the dependency chain might have to change as well. However, by giving ownership of the low-level interface to the high-level module and letting the high-level module depend on that interface, the high-level module is no longer coupled to the implementation of the low-level module. This is illustrated in Figure 2. The inversion is complete since the low-level module now derives from an interface declared in the above layer. This is what Sweet referred to as the Hollywood principle: *Don’t call us, we’ll call you* [8]. The client of a service dictates the design of the service interface. Another common term for this is Inversion of Control (IoC).

![Figure 2. The dependency inversion principle in a layered system](image)

A dependent should only keep references to an interface or abstract class that the dependee derives from. The rationale for this is simple: The implementation of a concrete class is much more likely to change than its’ interface. If a new implementation is needed, a new subclass of the same interface can be created. This promotes the open-closed principle, which is another of Martin’s design principles. Now, by depending on that interface, the dependent does not need to change following a substitution of a dependee implementation. This is very useful for a number of reasons. For example, software developed for different platforms might require slightly different implementations, which means that implementation classes would need to be substituted statically before compiling the software. DIP makes this much easier, as the substitution can be done at one central place in the program. In fact, this substitution does not have to be done statically, but could as well be done at runtime thanks to dynamic polymorphism [3].

The central place where the substitution is done has to be where a concrete instance of the implementation class is created, since instantiation of abstract classes or interfaces is not possible. In many cases there is a lot to be gained from taking the responsibility of creating dependee objects away from the dependent. This is another example of IoC. One of the biggest gains from doing this can be seen during unit testing, since it allows for dependees of a class to easily be substituted for mocks or stubs (see section 3.4). There are several design
patterns that can be used to achieve this kind of IoC. Sections 3.2.3, 3.2.4, 3.2.5 and 3.2.6 describe the most common ones.

3.2.2 More on Dependencies

Feathers divides dependencies into two categories: external and internal. External dependencies arise when the dependee is passed into the dependent by some outside entity; usually as a parameter to a constructor or a function. Internal dependencies arise when the dependee is not passed into the dependent. These include global variables/singletons, free functions, object creation in constructors and object creation in methods [9].

In terms of dependency inversion, external dependencies only cause problems when the dependee parameter is a concrete class, as this prevents from using polymorphism to alter the dependee implementation. Fowler gives one useful pattern, which is the foundation of DIP: Extract interface: If not already existing, create an interface for the dependee and make sure that the dependent is only depending on that interface instead of the concrete dependee class. This enables polymorphism to be used to alter the concrete class of the dependee.

The problem with internal dependencies is that the dependent is responsible for the creation and use of the dependee, making it impossible to control what dependee implementation is used. An internal dependency to a singleton can be broken by introducing a static setter method [9] which can be used to change the implementation of the singleton from the outside. Other than that, breaking internal dependencies is the main objective of many patterns adhering to the DIP.

3.2.3 The Dependency Injection Pattern

The act of injecting dependees from outside of the dependent had already been practiced for many years when Fowler brought the concepts together and introduced the term Dependency Injection (DI) in [10]. Fowler mentioned three forms of DI: Constructor Injection, Setter Injection and Interface Injection. The common theme between the three forms is that there is a dependent depending on an abstraction of a dependee through a member attribute, and another class (referred to as an Assembler in [10]) populating that attribute with an appropriate concrete dependee implementation. This is shown in Figure 3. Notice how the relation dependent - abstract dependee – concrete dependee is the same as the inter-layer relation between modules in Figure 2.

The assembler can take many forms which are not specified by the pattern. There is a number of frameworks, often referred to as IoC containers, taking care of the assemblage of dependent classes. Many of these frameworks allow for assemblage configurations for the program to be set up in XML files. A simpler occurrence of an assembler is during unit testing, when the test case can act as assembler for the class under test.

The difference between DI forms is the way the dependee attribute is populated. With Constructor Injection, a dependee object is simply passed into the constructor of the dependent, and is used to set the member attribute in the dependent. Constructor Injection has the advantage that it clearly states what dependencies need to be provided in order for a class to be working. This has the consequence that objects, once created, are always guaranteed to be in a fully initialized state. However, there is the downside that once a dependent object is created, its’ dependencies cannot be changed dynamically. In order to achieve that dynamism, another dependent object would need to be created, which could be costly in some cases.

Setter Injection works very similarly as Constructor Injection, but with the member attribute being set in a setter method instead of in the constructor. Having the injection taking place at any time during the lifecycle of the dependent object has the consequence that the object could end up in an uninitialized state. This has to be considered by clients of the
dependent. On the other hand, setter injection allows for dynamic substitution of dependee implementations during the lifetime of a dependent object.

![Figure 3. The roles of the Dependency Injection pattern](image)

The third form of DI, Interface Injection, takes slightly more work to implement than the other two alternatives. For each dependee in the program, an interface is declared, e.g. IInjectDependee, with a function, e.g. injectDependee, having an abstract dependee as a parameter. All dependents needing that dependee derives from the interface and implements the injectDependee function, setting their dependee attribute. The assembler injects a concrete dependee by calling the injectDependee function on the dependent object. Since additional interfaces need to be created in order to realize the interface injection pattern, it is more invasive than constructor and setter injection [10]. However, it is a clean way of declaring what dependencies a class has, since each dependency causes the derivation of a new interface.

Besides from the three forms mentioned by Fowler, there is one more way to realize the DI pattern. It performs injection through templates or generics. Instead of using dynamic polymorphism, the concrete dependee is given by a template parameter. This way, the wiring of components is done at compile time rather than at runtime. Of course, this lacks the dynamism that setter injection has, but on the other hand, it ensures fully initialized objects and it doesn’t compromise program efficiency.

### 3.2.4 Factory Patterns

In literature touching on design patterns, several varieties of factory patterns have been described. This section will briefly explain the following three types of factory patterns: Factory, Abstract Factory and Factory Method. [5] introduces two of these types: the Abstract Factory pattern and the Factory Method pattern. [3] describes the Factory pattern, which is most general, but also mentions substitutable factories, corresponding to the Abstract Factory pattern.

The Factory pattern allows dependents to create concrete dependees while only depending on abstract interfaces [3]. This is done by providing a single location of object creation; a factory object. By delegating object creation to a function in the factory which returns the abstract type of the dependee, the dependent will never depend directly on a concrete dependee. This concrete dependency has been moved out of the dependent, to the factory
object. This makes it easier to vary dependencies across a program by assembling the factory in a desired way, or by substituting implementation of the factory interface.

The substitution of a factory implementation is what [3] refers to as substitutable factories, and what [5] calls the Abstract Factory pattern. It requires a factory interface specifying what types of products can be created. The factory is acquired by the dependent through a global variable, or a static variable in a global class (an application of the Singleton pattern). Once again, the dependent does not know, or care, about which factory implementation it is using. This allows for the factory implementation to be varied from somewhere else in the program. Abstract Factory allows for families of products to be created without relying on their concrete classes. These product families can easily be switched - at runtime if desired. This feature is often useful during unit testing, as mentioned in section 3.2.1.

Both Factory and Abstract Factory suffer from a problem recognized by both [5] and [3]: When creating a new derivative of a product interface, a new function needs to be added to the factory interface and all its’ implementations. This is against the open-closed principle and affects all clients of the factory interface. One solution to this problem is to provide one single parameterized factory method which can create all kinds of products in the product derivation tree. The parameterization of this function could be done using a string parameter or by using generics. However, this comes at the cost of compile-time type safety.

The purpose of the Factory Method pattern is to defer creation of dependee objects to subclasses of the dependent [5]. This is done by encapsulating the creation of a dependee in a factory method on the dependent. This factory method is overridden by subclasses of the dependent in order to create the desired dependee implementation. By substituting subclasses of the dependent the dependee implementation is varied, while the implementation of the dependent is only bound to the abstract interface of the dependee.

One of the strengths of factory patterns – the fact that they enforce the dependency inversion principle by letting dependents depend only on interfaces – can sometimes also be seen as a liability [5], [11]. In cases when the dependent needs subclass-specific behavior from a dependee returned from a factory, a downcast is needed. Strictly speaking this could be avoided if the DIP were always followed, but in practice many frameworks are implemented in a way that requires downcasting due to the use of factory patterns.

The most obvious liability of factory patterns is that they introduce more complexity in a program, with class hierarchies being created just for the purpose of instantiating objects of a certain kind. DI is therefore often to prefer if the DIP is to be followed, but in cases when whole families of products are to be varied factories can provide a powerful solution.

3.2.5 The Service Locator Pattern

The Service Locator pattern is often considered an architectural pattern, in contrast to factory patterns which are more pure design patterns. However, the Service Locator pattern can be implemented on the object level, and will therefore be regarded as a design pattern in this context. The implementation of Service Locator is very similar to that of a Factory. There are, however, two important differences.

Firstly, the (possibly abstract) factory class is replaced by a (possibly abstract) service locator class. The difference from the point of view of the client is not obvious. However, the role of this class is different: A factory creates a new instance of a service, while a service locator locates a service which could possibly have been registered in the service locator by an outside entity. Secondly, because the service object is not necessarily created by the service locator, ownership of the service object is not given to the client of the service locator. This ownership stays with the service locator or the outside entity that registered it. Any component in a program can register services in the service locator, meaning that there is typically not one, but many assembler roles in the pattern.
It is possible to vary the implementation of the service locator class in order to vary the set of services provided by it, much in the same way as Abstract Factory allows varying the set of products it produces. In practice, though, service locators are usually expected to locate any kind of service in a system, regardless of abstract type. In order to be able to do this without extending the interface every time a service class is added, generics or templates are often used. A generic service locator is implemented by creating an interface consisting of generic methods for registering and locating a dependee, and a concrete class implementing that interface. The concrete class needs some kind of map implementation mapping service names to dependee implementations. Only one service locator needs to be implemented for the whole system. This solution, however, has been criticized because it violates the interface segregation principle, which states that clients should not be forced to depend on methods they do not use. There is even more critique directed towards the Service Locator pattern. Since the dependent can request services on demand using a generic interface, dependencies are not made explicit. Clients of a dependent need to know which dependees to register with the service locator before calling the dependent. However, there is no way to tell from the interface of a dependent what dependees are needed. This means that code without compile errors might cause hard-to-trace runtime errors. One way to see it is that Service Locator deals with conceptual “services” instead of typed “objects”, which is dangerous in a typed OOP language. Therefore, many regard Service Locator as an anti-pattern which should be avoided.

3.2.6 The Parameterize Method Refactoring Pattern

Parameterize method is a refactoring strategy mentioned by Fowler et al. in [13]. However, it was Feathers who pointed out that parameterize method can be used to break an internal dependency caused by object creation in a method [9]. The simple solution is to have the object passed in as a parameter, making it an external dependency instead. The drawback of this pattern is that it propagates the dependency to all the clients of the dependent. This is different from DI where the assembler, rather than the client, has the role of assembling the dependee. This problem of exposing clients to implementation issues is similar to that of the Strategy pattern explained in [5].

The fact that the dependee is passed to the dependent every time a service function is called has the consequence that the scope of the dependency is, or at least should be, limited to that function. Having the dependee as an attribute of the dependent does not make any sense, since the dependee is made available when it is needed anyway. In fact, having a dependee attribute with the intention of using it elsewhere in the dependent could even lead to inconsistent behavior due to an indirect dependency between clients.

Despite this drawback, in the case when a mock is needed in order to test a class whose clients are not yet implemented, Parameterize method is an easy-to-use refactoring solution.

3.3 Differences and Similarities between C++ and RSA-RTE

There are a few differences between capsules and classes that are relevant when adapting C++ design patterns to RSA-RTE [1]. These differences originate from the fact that capsules are managed by the RT Service Library [14], which handles synchronization between capsule objects and manages their lifecycle. Table 1 summarizes the differences and similarities discussed in this section.

One difference is that capsules are not created by calling a constructor as is the case with normal classes, but via a call to a library function called incarnate. Thus, the actual call to the constructor is abstracted away. Technically capsules have constructors, but they cannot be used directly by the developer. Instead, initializing code can be placed in a transition
originating from the initial (pseudo) state. Data sent as one of the parameters to the incarnate function, corresponding to constructor parameters in C++, are passed along to the initial transaction.

Capsules are incarnated into capsule parts. This means that an explicit incarnation of a capsule has to be done from the containing capsule. This also has to do with the fact that the incarnate function is a member of a service protocol called “Frame”, which requires a port on the containing capsule in order to be accessed. However, an explicit call to incarnate is only needed when a capsule part is set to be “optional”. If instead a capsule part is set to be “fixed”, the capsule part is incarnated implicitly at program start-up. In practice, therefore, it is often not needed to deal with capsule creation in detail-level code.

A transition between two states is triggered by an incoming event. This is the most common way of invoking capsule behavior. Thus, transitions in RSA-RTE can be considered to correspond to method bodies in C++, with the interface of a capsule being described by its’ set of external ports. C++ function calls are therefore replaced by message passing when communicating with capsules in RSA-RTE. It should be noted that it is possible to declare public methods on capsules. However, the use of public methods nullifies the benefits of the run-to-completion semantics associated with message passing. Therefore references to capsules are usually not passed around and capsule operations are not used to communicate with a capsule. In some cases it is a possibility to declare static methods on a capsule and communicate with it using static method calls. This avoids the need for capsule references, but suffers from the same synchronization problem as normal methods.

This difference in behavior invocation is also affecting performance. C++ class behavior is of course invoked by calling a function. At run-time, all it takes to transfer control to the called function is a lookup in the vtable, which is relatively efficient. Capsule behavior, on the other hand, requires much more overhead in order to be invoked. In order to transfer control to a capsule, a call is made to a send function in the RT Service Library, which handles memory allocation for the message and puts it in a message queue. The job of the message queue is to make sure that no two messages are handled simultaneously on the same thread. The sent message therefore has to wait for messages sent from other threads to be dispatched. Depending on how busy the program is in terms of waiting messages, this could be time costly. Therefore, any extra messages sent could make the system considerably slower compared to calling a function in the case of classes.

Invoking class behavior from a capsule is simple. It is the matter of calling a function as usual. Of course, this does not guarantee synchronization between threads. The state of a class object which is being used by capsules running on different threads might be the object of race conditions. Invoking capsule behavior from outside the capsule using message passing is synchronized, but more expensive in terms of program efficiency.

C++ and RSA-RTE also differ slightly in the concept of inheritance. C++ only has one type of class. It can be abstract if one or more of its’ functions are declared as pure virtual. If all its’ functions are pure virtual, it is conceptually an interface. Interfaces can be modeled in RSA-RTE using the UML interface element. RSA-RTE capsules can realize the operations declared in an interface using a feature called interface realization. It is possible for a capsule to realize multiple interfaces. When it comes to inheritance and overriding behavior, RSA-RTE is stricter. A capsule can extend another capsule using the UML relationship “generalization”. Capsules can only specify this relationship to one other element, and the other element has to be a capsule. When a capsule specifies the “generalization” relationship to another capsule, it inherits attributes, operations (except private ones), ports, capsule structure (such as capsule parts) and the state machine of the other capsule. Overriding of these elements is called redefinition.

Even though parallels can be drawn between classes and capsules, it is important to understand that the typical capsule is much more complex than a class. This is mainly due to the hierarchical structure of state-machines, which makes it possible to implement a capsule’s
behavior on different abstraction levels. This results in capsules often differing from classes when it comes to the “Single Responsibility Principle”\(^2\), with capsules breaking the principle due to the inner complexity they are often enclosing.

Another important difference between classes and capsules becomes apparent during unit testing. Since transitions can be triggered only when they originate from an active state of a capsule, the test case must know how to set up the capsule in the right state before testing a transition.

<table>
<thead>
<tr>
<th>C++ class concepts</th>
<th>RSA-RTE capsule concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor call</td>
<td>Call to framework function incarnate</td>
</tr>
<tr>
<td>Constructor body</td>
<td>Capsule transition “initial”</td>
</tr>
<tr>
<td>Multiple constructor parameters</td>
<td>Incarnate parameter “data”</td>
</tr>
<tr>
<td>Classes can be instantiated by anyone</td>
<td>Capsule parts are instantiated by containing capsule</td>
</tr>
<tr>
<td>Public method</td>
<td>State transition</td>
</tr>
<tr>
<td>Method argument</td>
<td>Trigger parameter, accessed through “rtdata”</td>
</tr>
<tr>
<td>Public method</td>
<td>Public static method</td>
</tr>
<tr>
<td>Method call</td>
<td>Message passing</td>
</tr>
<tr>
<td>Method call</td>
<td>Static method call</td>
</tr>
<tr>
<td>Class’ public interface</td>
<td>Set of external ports</td>
</tr>
<tr>
<td>Interface class</td>
<td>UML Interface element</td>
</tr>
<tr>
<td>Implementation of interface</td>
<td>Interface realization feature</td>
</tr>
<tr>
<td>Method overriding</td>
<td>Redefinition of methods or state transitions</td>
</tr>
<tr>
<td>Classes are instantiated in the code</td>
<td>Capsules are instantiated using the model (fixed capsule parts) or in the code (optional capsule parts)</td>
</tr>
</tbody>
</table>

Table 1. Differences and similarities between C++ and RSA-RTE.

\(^2\) A class should have only one reason to change. [3]
3.4 Mock Objects

In the literature, the term stub has been mentioned in the context of software testing since the late 70’s. The use of stub methods or stub objects is a well-known method for simplifying the testing process, both during unit testing and integration testing. Stub objects simulate the behavior of an object and can be used to verify object state during test runs. The notion of mock objects was introduced in the Extreme Programming community in the early 00’s. A mock object is a substitute implementation to emulate or instrument other domain code [15]. This is an example of sensing, which increases controllability of an object under test [9]. In other words: mock objects, similarly to stub objects, can be used to replace a dependee of an object under test. Mock objects behave very similarly to stub objects with respect to the calling code, i.e. the object under test. However, as pointed out by Fowler in [16], there are differences in the way the tester works with mocks versus stubs. The most obvious difference is the way test cases are constructed. When testing with mock objects, behavior for each mock instance is defined inside the test case, along with expectations on the calling code. This way behavior of the mock can be tailored to fit the need of the specific test case. A stub’s behavior is defined on the class level. What to verify is also decided on the class level, for example by adding attributes and verification functions that can be called from the test cases. Following from this, stubs normally verify object state while mocks verify object behavior. Mock objects changes the focus of TDD from thinking about the changes in state of an object to thinking about its interactions with other objects [17]. Fowler also points out that the usage of mock objects forces the tester to think about the behavior of the object under test rather than it’s interface. Test cases become coupled to the implementation of the object under test. This has the effect that if the implementation of an object changes, so must its’ test case. Tests are no longer black-box.

Systematic use of mock objects in TDD can lead to an approach where each layer of a system is mocked and then implemented, starting from the upper layer and continuing downwards. When an object is tested, its’ dependees are mocked. In the next iteration, the mocks are replaced by real implementations, whose dependees are mocked, and so on. With this testing approach, narrow service interfaces are designed based on what the clients actually need, and the services are implemented based on that [17]. This is in fact what the dependency inversion principle wants to achieve (see section 3.2.1, in particular Figure 2).

Many have pointed out [16], [17], [18] that mocks are not always the best option for unit testing. In order to answer the question about when to use mocks, Langr [18] mentioned the following reasons: to reduce test runtime, to improve run consistency of tests, to simulate something that does not yet exist, to generate events that are hard to trigger or to write tests for large dependency chains.

Using mocks for testing capsules has not been discussed in any published literature that I could find. However, there is no reason why mock testing should not be feasible for capsules if it is for classes. In fact, Fowler’s remark that mock testing makes black-box testing impossible is not as much of an issue for capsules, since testers need to be aware of and set up the capsule state before the execution of a test case. Thus, capsules are always white-box during testing anyway.
Chapter 4

Method

4.1 Pre-study: Choosing Evaluation Criteria

The choice of evaluation criteria guided the rest of the study to a large extent, since it provided a clear overall goal for the design and implementation. Therefore, a pre-study was made with the intent of finding concrete criteria to measure the solutions against. The criteria had to take into account the needs of testers and developers at Ericsson and the organization as a whole. The pre-study consisted of interviews with Ericsson employees, as well as literature research in the area of software usability.

The purpose of the interviews was to get a clearer understanding of the development process at Ericsson, the role of unit testing in that process, and what problems and improvement areas existed for the unit testing process. Interviews were chosen over a survey since, even though a survey could have increased the sample size, hearing the thoughts of someone working with the test framework was considered more valuable than the statistical significance of the results. Interviews were also considered a better alternative than observations because of the technical nature of the problem: Asking a developer how he wanted to improve the unit testing process would elicit his opinion and make use of his expertise. This seemed more to-the-point than observing said developer write a unit test. The interviews could have been done in another format, such as a focus group. In the end, though, it was decided that a one-on-one interview would go deeper and make the interviewee answer more comprehensively to asked questions.

The interviews were what Robson would describe as semi-structured, meaning that they aimed to answer a set of questions while letting the flow of the conversation decide the order of the questions [19]. Two interviews were made; one with a product guardian responsible for technical issues with the test framework; the other with a developer. The data gathered was considered enough to fulfill the purpose of the interviews. After the interviews it was clear that the proposed patterns should aim to be useful to work with for the developers, more so than focusing on quality measures of the code.

Thus, the goal of the literature research was to find a definition of usability and a method for evaluating software solutions with regard to usability. The criteria was that the definition should be accepted and applied in the software development field, i.e. employed in case studies and used in the literature. Furthermore, the definition had to be applicable not only to software products, but for design solutions as well. The usability evaluation method needed to be able to measure usability qualitatively as described by the definition.

There have been many definitions of usability in software literature and the term has had different meaning depending on the context it has appeared in. Some effort has been made to measure API usability [11], [20], [21], [22]. This was of interest for this study, since API usability is a quality of a software programming tool rather than a quality of an interactive software product. These papers employed the Cognitive Dimensions Framework [23]. The Cognitive Dimensions Framework was developed as a tool for evaluating programming language usability. It was used during development of the first release of .NET API:s, and was adapted to evaluation of API usability during this process [21]. Although the Cognitive
Dimensions Framework could be used for this study, it does not give a clear definition of the term usability. Furthermore, it seemed that a some adaptation would be needed for the sake of evaluating design solution usability rather than usability of language features. That adaptation process would introduce a new validity threat, which is why the Cognitive Dimensions Framework was disregarded for this study.

The IEEE standard defines usability as “a measure of an executable software unit’s or system's functionality, ease of use, and efficiency” or “the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component” [24]. These definitions are restricted to usability of software systems and thus fail to meet our criteria of being applicable to design solutions.

According to [25], Old ISO standards such as ISO/IEC 9126 from 1991 defined usability as “a set of attributes that bear on the effort needed for use and on the individual assessment of such use, by a stated or implied set of users”. This definition viewed usability as a set of inherent attributes of a product, making it a measurable quality. However, it made the measurement dependent on the type of product. Also, it was unclear whether this “set of attributes” would be easy to find in a product as intangible as a software design solution. In 1995, Bevan introduced the notion of usability as quality of use. This changed the view of usability from being an inherent attribute of a product to a quality perceived by the users. Bevan stated that quality of use measures “the effectiveness, efficiency and satisfaction with which specified users can achieve specified goals in specified environments” [26]. This definition of usability was soon absorbed by the ISO standard, e.g. [27]. ISO/IEC 25010 from 2010 states that “usability can either be specified or measured as a product quality characteristic in terms of its subcharacteristics, or specified or measured directly by measures that are a subset of quality in use” [28], sharing Bevan’s view of usability as quality of use. This definition fit the criteria of being accepted in the software field and applied in case studies, as well as being measurable, using metrics for quality in use. And the view of usability as a quality perceived by the user made it applicable for any kind of product, including design solutions.

Following Bevan’s definition, the evaluation needed to focus on measuring the effectiveness, efficiency and satisfaction with which Ericsson developers could inject mock objects while working in their programming environment, using the proposed patterns.

Since the modeling language is used for developing real-time software, it was also necessary to test the runtime efficiency of the solutions. Methods for defining and measuring runtime efficiency, as well as the process of breaking down effectiveness, efficiency and satisfaction into measurable qualities and finding an evaluation method for measuring those qualities, are described in section 4.3.1 and 4.3.1.

More technical evaluation criteria, such as to what extent found solutions solved different problem scenarios, were also considered. These criteria helped to decide which solutions to implement and which not to. They are discussed in Chapter 5.

### 4.2 Design and Implementation

The design- and implementation approach resembled the waterfall model. However, it had some iterative elements, since implementation was an integrated part of the design phase to some extent. This had to do with the fact that a big part of the design problem was to figure out what was plausible to implement. Since the volume of code needed for each found solution was relatively small, trial and error was practiced at times. There was, however, a clear thread of execution of the design- and implementation phase, illustrated in Figure 4.
The main goal of the study was to find out which solution for injecting mock objects is most useful with regard to simplifying unit testing of capsules. In order to reach this goal, an initial set of candidate solutions was thinned out at various stages of the design- and implementation phase. The final set of solutions were tested according to the evaluation criteria found in the pre-study.

The design and implementation stage consisted of three activities. The first activity was to list possible solutions through research and to assess their potential to solve the problem. The second activity was to create designs for the solutions which were applicable to the modeling language and also took into account the challenges posed by the testing framework. The third activity was to implement the solutions.

### 4.2.1 Design Pattern Research

The problem, from a technical standpoint, was the following: we wanted to be able to inject a dependency to another element rather than having the dependent control the creation of the dependee. Existing widespread OOP design patterns provide several solutions to this problem. It was therefore natural to look for solutions in design pattern literature.

In order to know what kind of design patterns to look for, some sort of search criteria was needed. The search criteria was based on technical aspects of the problem. Thus, the study first concentrated on finding technical solutions and only later assessed those solutions using the evaluation criteria determined in the pre-study.

In order to find solutions that actually solved the technical aspects of the problem, a careful breakdown of the problem was needed. The following question was used as starting-point: *When do we have the need to test a class using mock objects?* This question has two interpretations; one on a strictly technical level and one on a more general level. Answering both interpretations of the question gave good hints as to what kind of solutions the research should be looking for.

On the technical level the question has a straight-forward answer: We might want to mock out any class that is depended on by the class under test, i.e. there is a dependency from the class under test to some other class. Implementation-wise there are different types of dependencies. Using Feathers’ notion of dependency types, discussed in section 3.2.2, the following research criteria could be established: The found patterns should make it possible to inject mock objects in situations where the dependee is
passed into a constructor (external dependency),
- passed into a function (external dependency),
- a singleton/global variable (internal dependency),
- created in a constructor (internal dependency), or
- created in a function (internal dependency).

In order to make mock injection possible for any of these situations, an “inversion of dependencies” was needed. Therefore the research focused on patterns adhering to the dependency inversion principle.

The question, when asked in a more general sense, can be reformulated as: *When should mocks be used in the first place?* Using Langr’s answer to this question, mentioned in section 3.4, it was determined that the following effects needed to be taken into consideration when looking for solutions, i.e. found solutions should not counteract the following:

- Reduced test runtime
- Improved run consistency of tests
- Possibility to write tests for large dependency chains

Furthermore, there were two other possible ways of seeing the problem: As a pure design problem or as a problem of refactoring existing code. A lot of material on both design patterns as well as refactoring patterns can be found in published literature. Refactoring patterns focus on actions that can be applied to existing code in order to change the program design. Often, the new design results in a design pattern or part of one, but not always. During the interviews in the pre-study, Ericsson staff mentioned that most of the work was directed at working with existing code, but of course, sometimes new code was added. Therefore, both refactoring patterns and pure design patterns were considered. The resulting solutions, however, are presented as designs rather than actions.

With the above criteria in mind, the research focused on patterns adhering to the dependency inversion principle. Patterns mentioned by Martin [3], the first author to formulate the dependency inversion principle, were considered. Fowler [10], [13] and Feathers [9] also mention the dependency inversion principle in their work, and also give suggestions as to how to refactor code in order to use mocks. The work of Gang-of-Four [5] was also studied because of the widespread use of the patterns presented in their famous pattern catalogue.

### 4.2.2 Assessment of Potential

The assessment of each pattern’s potential to solve the problem was based on an estimation of how well they would perform with regard to the usability evaluation criteria. Therefore the same metrics as shown in Table 2 in section 4.3.1 were used, but using nominal values such as low, medium and high instead of exact numerical values to denote the expected result on each measure point. User expectations were not considered at this stage as this is a user specific criteria. The same goes for runtime efficiency, which is dependent on modeling language features and thus could not be assessed at this stage. Note that this assessment was done without looking into specifics of the modeling language and thus focused on theoretically known features of the patterns.

For the purpose of this assessment, user efficiency was interpreted as the efficiency with which Ericsson developers can inject mock objects using the found solutions. Since the technical features of the patterns were in focus at this stage, the estimation of user efficiency was mainly based on the amount of code that needs to be added in order to apply the solutions.

Perceived ease of use was interpreted as the perceived ease with which Ericsson developers can implement the found solutions. This assessment was mainly based on an estimation of added complexity to the code.
While ease of use focuses on the actual use of the solutions, usefulness focuses on the results of using the solutions. Perceived usefulness was interpreted as the perceived degree to which found solutions actually facilitates injection of mock objects. For this sake, dependency explicitness was an important factor, i.e. to what degree a dependent makes its’ dependencies visible to clients and test cases. Dependencies displayed as attributes or function/constructor parameters was considered good dependency explicitness and therefore indicated high usefulness, while dependencies not displayed as attributes or function/constructor parameters was considered less good.

### 4.2.3 Design Method

In order to be able to implement the found solutions, a structured comparison between C++ concepts and modeling language concepts was made. This was necessary because many object-oriented language constructs are hidden behind an abstraction layer in the modeling framework, making it difficult to apply a design pattern out-of-the-box. This was also important because it provided concrete motivation as to why certain design decisions were made. The comparison was made after consulting documentation of the modeling language, as well as testing out solutions using trial-and-error in order to get a feel for what was possible to do with the modeling language. Here the testing framework was also a factor to consider. The comparison process resulted in a table of comparisons, where C++ concepts are compared to RSA-RTE concepts. The comparisons made the design process relatively straight-forward.

### 4.2.4 Implementation Method

The third activity was to implement the found solutions. This was done using the same modeling tool and test framework as those used at Ericsson, which was considered an advantage as it increased the understanding of the implementation issues Ericsson developers face regularly.

The actual implementation was done in the following way: first, a dependee interface and a concrete implementation of that interface were created. A mock implementation was created using an integrated tool created by Braaf in [29]. These models could be reused for all solutions. The general working process for implementing a pattern was the following:

1. Create a simple dependent delegating its’ only responsibility to the concrete dependee implementation.
2. Refactor the dependent according to the pattern, in order to support injection of a mock object.
3. Create a test case testing the dependent by injecting the real implementation of the dependee.
4. Create a test case setting up a mock object and injecting it into the dependent.

More test cases were created when needed, e.g. for testing side-features of some of the patterns.

### 4.3 Evaluation

#### 4.3.1 Evaluation Metrics

The criteria to evaluate against were decided in the pre-study described in section 4.1. The pre-study resulted in the following measurable criteria: User effectiveness, user efficiency, user satisfaction and runtime efficiency.
According to [26] and [30], temporal efficiency, in the context of usability testing, can be calculated as:

\[ \text{Temporal efficiency} = \frac{\text{effectiveness}}{\text{task time}} \]

Here effectiveness denotes the percentage of successful task completions. [26] also states that task time itself can be used as a usability measure if all users complete the task satisfactory (corresponding to giving effectiveness the value 1 in the formula). Thus, the user tests could have been conducted in a way that disregarded effectiveness as a measuring point, by providing detailed user instructions and giving hints during the test sessions in order to guarantee that tasks were completed. This would have made some sense considering the fact that the solutions were designed to solve the tasks, especially when in the hands of experienced developers. One could argue that what was interesting was how long they took to implement. However, since usability was in focus the decision was made to also measure effectiveness, since this would hopefully say something about the complexity – and actual ease of use - of applying the patterns.

Effectiveness, in this study, was measured as the number of builds (including failed builds) that were needed before the task was actually completed in order to complete each task, with 1 of course being the ideal number.

In 2000, Mahmood et al. made a literature study investigating variables that had been found to correlate with information technology user satisfaction. It was found that there are three categories of factors that affect user satisfaction: Perceived benefits and convenience, user background and involvement and organizational attitude and support [31]. For this study, perceived benefits and convenience was most interesting for the sake of comparing different design solutions, as the other two categories are user- or context dependent and will not vary when the same user tests different solutions in a constant setting.

There were three variables falling into the category of perceived benefits and convenience: user expectations, ease of use and perceived usefulness. Those three were therefore used as quality metrics when comparing design solutions in this study. The assumption had to be made that perceived satisfaction when using a software design solution as a tool is approximately the same as perceived satisfaction when using an information system as a tool.

In order to measure the runtime efficiency of the solutions, it was necessary to decide what would be an appropriate and relevant metric. The resulting patterns had the following two attributes with regard to runtime efficiency: Time taken to inject a dependee into a dependent capsule and time taken for the dependent capsule to serve requests using the dependees. Since dependees are normally only injected once while requests are often handled continuously during the lifetime of a dependent, time taken to serve requests using dependees was considered far more important. Thus, the metric “usage time”, denoting time taken to serve a request from a test case, was introduced.

The creation of test cases and the runtime test procedure is explained in section 4.3.1. The metrics used are summarized in Table 2.

### 4.3.1 Methods for Gathering Evaluation Data

The final qualitative comparison of the found solutions was made using two sources of information: user tests and runtime data from a set of test cases.

The user tests were conducted as talk-aloud sessions, where the test subjects were asked to perform a set of tasks using the different solutions while being encouraged to talk about what they were thinking and doing. In 2007 Ellis et al. made a usability study on API design choices, using talk-aloud sessions for those user tests [11]. The study made use of both quantitative data and user remarks to support its’ findings. That study resembled this one in the sense that it tested usability of design choices rather than a finished software product, and was used as inspiration for the design of the user tests in this study.
For the user tests, 5 test subjects were selected out of the developer staff at Ericsson. The only requirement for the test subjects was that they had been working with RSA-RTE at some point during their employment. Developers were asked by going from top to bottom on a list of names given by the supervisor at Ericsson. The resulting sample of developers had varying experience with RSA-RTE, ranging between 15 months and 6 years. The estimated proportion of work time devoted to RSA-RTE ranged between 6 hours and 20 hours per week.

For the talk-aloud sessions, two tasks were given to the test subjects. In the first task, the test subjects were given a text file containing test code where a mock was created and an imaginary capsule was being tested. The task was to complete the test code by writing pseudo-code for injecting the mock into the imaginary capsule. The goal was to elicit the test subjects’ expectations regarding injection of mocks from the client’s (in this case the test case’s) perspective. The text file is attached in appendix A.1. User efficiency and effectiveness were not measured during this task.

The second task was carried out in the following way: The test subjects were given a code example containing a simple capsule (a dependent) being dependent on a passive class (a dependee). The dependee was created at the place in the code where it was being used. Also given was a test case testing the dependent by invoking its’ primitive behavior. The test subjects were asked to build and run the test case to make sure that the code example was indeed working. The task was to refactor the dependent so that it could be tested using a mock of the dependee. This also required changing the test case to inject a mock object. Code for creating the mock object was already given. The task was considered completed when the test case passed. This task was repeated for every design solution that was to be tested. A description of each design solution was given. Time taken to apply each solution was measured, as well as the number of builds needed before the test case passed. The order in which the solutions were tried was varied between the test sessions in order to eliminate the influence of learning effects on the results.

The test subjects were given a few survey questions immediately after reading the instructions for a design solution. These questions were designed to elicit the test subjects’ first impression of the solutions and their expectations regarding ease of use. Similar survey questions were given immediately after solving the assignment, to investigate whether implementing the solutions was easier or harder than expected. After the talk-aloud sessions, the test subjects were orally asked debriefing questions related to the perceived usefulness of the solutions. The full set of questions, as well as task descriptions and instructions, are given in appendix A.2.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Evaluation criteria</th>
<th>Measured using</th>
<th>Data gathered from</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build effectiveness</td>
<td>User effectiveness</td>
<td>Number of builds needed before successful test run</td>
<td>User tests</td>
</tr>
<tr>
<td>Task time</td>
<td>User efficiency</td>
<td>Average minutes [min]</td>
<td>User tests</td>
</tr>
<tr>
<td>User expectations</td>
<td>User satisfaction</td>
<td>User pseudo code, survey answers</td>
<td>User tests, pretest questionnaire</td>
</tr>
<tr>
<td>Perceived ease of use</td>
<td>User satisfaction</td>
<td>User quotes, observations</td>
<td>User tests, debriefing interview</td>
</tr>
<tr>
<td>Perceived usefulness</td>
<td>User satisfaction</td>
<td>User quotes</td>
<td>Debriefing interview</td>
</tr>
<tr>
<td>Usage time</td>
<td>Runtime efficiency</td>
<td>Time per request [μS]</td>
<td>Test cases</td>
</tr>
</tbody>
</table>

Table 2. Metrics used to compare found solutions
In order to gather measurement data for estimating how the found solutions affected runtime efficiency, the solutions were first examined in order to assess if there was any logical reason as to whether they would compromise performance. For those solutions where this was found to be the case, a special set of test cases were created. The purpose of those test cases was to estimate the runtime that the found solutions were adding to the program in its’ intended environment of usage, relative to the case when no dependency inversion was performed. Thus, the following two cases were compared:

- The case where the dependee is kept as an attribute and used directly by the dependent.
- The case where the test case creates the dependee and injects/configures it with the dependent.

In order to simulate the resource heaviness that dependees might possess in an actual application, the dependee was implemented holding a reference to a class which was used in one of Ericsson’s products. This was considered needed, since it would be more difficult to notice any difference in runtime if the dependee was a dummy class with initialization time close to 0.

Over time, the overall efficiency is affected by the efficiency with which capsules can serve their clients. In order to simulate this runtime behavior, the test cases injected the dependee implementation once and then called the capsule under test 10000 times. Test cases measured the time taken to serve the requests from the test case. An example of such a test case is shown in Code example 3.

The test binary was configured to run each test case 20 times and the average time was measured. In order to limit the variance in execution time between runs caused by the OS, the binary was run 10 times and the average was measured. Thus, in total, the average from 200 test runs was calculated, meaning that an average from 2 million requests was measured for each solution.
//Setup Test
int i = 9; //Data to be sent
RequestHandlerD handler; //The dependee
RTTimespec t1, t2, t3;

//Warmup
for(int x = 0; x < 20000; ++x) {
  //Send request and receive result
  SEND(pRequestService, Request, i);
  RECEIVE(pRequestService, Response, int, notUsed);
}

RTTimespec::getclock(t1); //Time 1

//Inject the dependee (Port injection):
RequestHandlerDataD sig;
sig.data = &handler;
SEND(pInjectRequestHandler, Inject, sig);

RTTimespec::getclock(t2); //Time 2

//Usage
for(int x = 0; x < 10000; ++x) {
  //Send request and receive result
  SEND(pRequestService, Request, i);
  RECEIVE(pRequestService, Response, int, notUsed);
}

RTTimespec::getclock(t3); //Time 3

//Code to log the time intervals t2-t1 and t3-t2
...

return true;

Code example 3. A test case measuring injection time and usage time.
Chapter 5

Design and Implementation

5.1 Finding and Evaluating Possible Solutions

The findings presented in this section are based on the theoretical potential of the solutions to solve our problem without in detail considering applicability to the RSA-RTE context. Design- and implementation issues are discussed in sections 5.2 and 5.3.

5.1.1 Found Solutions

Literature research found the following 7 solutions: Constructor Injection, Setter Injection, Interface Injection, Abstract Factory, Factory Method, Service Locator and Parameterize Method. These patterns are described in section 3.2.3 through 3.2.6. Template Injection is another variation for languages supporting generics. RSA-RTE is not such a language, which is why Template Injection is not included among the solutions.

It was found that the solutions can be divided into two categories based on their way of dealing with internal dependencies: they either change internal dependencies into external dependencies or they create a new, more easily manageable internal dependency and move the original dependency into the new one. It should be pointed out that changing an internal dependency into an external one merely moves the internal dependency to clients or assemblers of the dependent. Internal dependencies always have to exist somewhere in the program, in order to instantiate a concrete class.

Solutions falling into the first category are the Dependency Injection patterns and Parameterize Method. Solutions falling into the second category are Abstract Factory, Service Locator and Factory Method.

5.1.2 Estimation of Pattern Potential

Definitions and interpretations made for this estimation are presented in section 4.2.2. The assessments are summed up in Table 3.

User efficiency

Constructor Injection and Parameterize Method are applied by adding extra parameters to a constructor or a function, one new parameter per mock type. The test case injects the mock object by simply passing it to the constructor or function. Setter Injection is applied by adding a Setter Method on the dependent, one per mock type. The test case injects the mock by calling the setter method with the mock as parameter. These solutions require the least code to implement and should therefore promote high user efficiency.

Factory Method is applied by encapsulating the creation of a dependee in a method returning the abstract type of the dependee. The default implementation of the factory method returns the real dependee implementation. A subclass of the dependent needs to be created
and configured with a mock object which is returned by the overridden factory method. Interface Injection requires a new interface to be implemented for each dependee. Besides from that, it works similarly to Setter Injection. Service Locator requires some work to be implemented but only one service locator implementation is needed regardless of how many mock types that are needed to test the system. The test case has to register the mock object with the service locator before testing the dependent. These solutions should lead to medium user efficiency.

Abstract Factory requires a new interface to be created and implemented for each new mock type. This requires a lot of new code to be written every time a mock is to be added and should therefore lead to low user efficiency.

**Perceived ease of use**

Constructor Injection, Setter Injection and Parameterize Method all declare dependencies in parameters which are visible in the interface of the dependent. Parameter passing is a natural element of almost any type of programming, so the ease of use for these patterns were regarded as high.

Abstract Factory requires a class hierarchy for each group of dependees. However, the complexity of the abstract factory interface is relatively low, mainly consisting of create-functions. Factory Method introduces a certain degree of complexity when a create-function is added for the purpose of being overridden by subclasses. Interface Injection introduces a similar type of complexity by requiring the dependent to implement one or more interfaces in order to support injection. This indicated medium ease of use for those patterns.

Service Locator introduces a lot of complexity since a new class hierarchy needs to be introduced. Service Locator might not introduce as many new classes as abstract factory, but a generic implementation with some implementation for dealing with service registering forces the service locator to hide a lot of complexity. It was therefore regarded as difficult to use.

Interface Injection introduces the same kind of complexity when interfaces have to be created for every new dependee. Factory Method makes its’ dependees visible in the signature of the factory method. However, complexity is introduced when the dependent has to be subclassed. These solutions were considered to be moderately easy to use.

**Perceived usefulness**

All Dependency Injection patterns, along with Factory Method, clearly make dependees visible to the programmer. They facilitate mock injection from a test case since it usually suffices to inject the mock object once during the setup of the test case, and this is done in a straight-forward manner. This indicates high usefulness.

Abstract factories explicitly state, through their interface and usually also through class naming conventions, which dependees can be required from a factory. However, the direct connection between a dependent and a dependee is not visible by looking at the dependent interface. Furthermore, in order to inject a mock, the program has to be configured with a factory that has been set up with the mock object. Parameterize Method might require several injections in one test case since an injection only lasts for the execution of the method. Also, Parameterize Method has the ugly side-effect of transferring dependencies to clients of the dependent, who should ideally not need any knowledge about implementation details of the service they are calling. These two patterns are therefore considered to be moderately useful.

Service Locator does not give any hint as to which dependencies a dependent has without looking at the implementation of the dependent. Furthermore, failing to recognize what dependencies exist causes compiling software to crash at runtime. This solution was therefore considered to have low usefulness.
<table>
<thead>
<tr>
<th>Design pattern</th>
<th>Type of dependency in dependent</th>
<th>User efficiency</th>
<th>Ease of use</th>
<th>Usefulness</th>
<th>Score</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor Injection</td>
<td>Dependee is passed into a constructor</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Setter Injection</td>
<td>Dependee is passed into a function</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Parameterize Method</td>
<td>Dependee is passed into a function</td>
<td>High</td>
<td>High</td>
<td>Medium</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Factory Method</td>
<td>(Abstract) dependee is created in a function</td>
<td>Medium</td>
<td>Medium</td>
<td>High</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Interface Injection</td>
<td>Dependee is passed into a function</td>
<td>Medium</td>
<td>Medium</td>
<td>High</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>Dependee is a singleton</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Service Locator</td>
<td>Dependee is a singleton</td>
<td>Medium</td>
<td>Low</td>
<td>Low</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 3. The found patterns and their estimated potential to meet the evaluation criteria.

### 5.2 Adapting Found Solutions to the Modeling Environment

Based on the estimation of potential with regard to the evaluation criteria presented in the previous section, it was decided that Service Locator was not a good fit for the problem of injecting mock objects. Thus, the design phase focused on trying to adapt the other six solutions to the modeling environment. Table 4 shows the mappings that were needed in order to create the adapted designs. These mappings originate from the differences and similarities that were discussed in section 3.3 and shown in Table 1.

<table>
<thead>
<tr>
<th>Mapping</th>
<th>C++ class concepts</th>
<th>RSA-RTE capsule concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>Constructor call</td>
<td>Call to framework function incarnate</td>
</tr>
<tr>
<td>M2</td>
<td>Constructor body</td>
<td>Capsule transition “initial”</td>
</tr>
<tr>
<td>M3</td>
<td>Multiple constructor parameters</td>
<td>Incarnate parameter “data”</td>
</tr>
<tr>
<td>M4</td>
<td>Classes can be instantiated by anyone</td>
<td>Capsule parts are instantiated by containing capsule</td>
</tr>
</tbody>
</table>
5.2.1 Incarnate Injection: An Adapted Design of Constructor Injection

Mappings M1, M2, M3, M4 and M14 were applied in order for constructor injection to be adapted to the modeling environment. The following steps need to be taken in order to apply an adapted version of constructor injection to the modeling environment:

- Use the data passed to the “initial” transition of the dependent in order to set dependee attribute(s).
- Pass the dependee(s) into incarnate when the dependent capsule part is created, utilizing the parameter named data which is untyped\(^3\). Support for multiple dependees requires a user-defined struct encapsulating the dependees.

Mapping M4 impacts the way in which the test case injects mocks. If the test case is not itself implemented as a capsule containing the test object as a capsule part, the test framework must provide functionality to create the test object with initialization data. The test framework used at Ericsson required such a feature, and a solution is presented in section 5.3.1.

According to mapping M14, capsules are instantiated automatically with the model as base when capsule parts are fixed. In this case, no instantiation data is passed to the initial transition. When the test framework is set up, however, the capsule under test is always an optional capsule part, which makes it possible to explicitly pass initialization data to it. Capsules can always be implemented to handle both these cases by checking if the data argument is NULL and create a “default” dependee in that case. With such an implementation, injection is not needed in production code but can still be utilized in the test environment.

Naturally enough, I decided to call this solution “Incarnate Injection”.

\(^3\) The parameter has the type const void *, “pointer to const void".
Incarnate Injection

<table>
<thead>
<tr>
<th>Design of dependent capsule</th>
<th>Injector/Assembler</th>
<th>Dependent is configured with a dependee using</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependees are kept as attributes which are set in the initial transaction using a passed-in object.</td>
<td>The injector is the containing capsule. The dependent part has to be set to optional.</td>
<td>A call to incarnate, using the parameter data.</td>
</tr>
</tbody>
</table>

Table 5. The design of incarnate injection.

---

**5.2.2 Port Injection: An Adapted Design of Setter Injection**

There are several ways to adapt setter injection to the modeling environment. One way, which is at least possible in theory, would be to use normal setter methods on the dependent. However, since a public operation can only be called on an object reference, and the use of capsule references opposes the purpose of using capsules in the first place, this is not a recommended solution. Using mapping $M7$ and making the setter method static is another possible design. However, this is far from a clean solution which could complicate unit testing and potentially make test results unpredictable, as pointed out by Stephens in [32].

Therefore, the most natural adaptation was to use mappings $M5$, $M8$ and $M10$ and inject the dependee through a message triggering a “setter” transition. This type of injection had already been used at Ericsson, so this study should not be credited for the adapted design. The fact that capsule behavior is described in terms of a (hierarchical) state machine makes it possible to tailor the design depending on when dependees are needed. For example, an internal transition on the top state would make the setter transition available during the full lifetime of the dependent, the same way a setter method would be. Having an initial state waiting for the dependency to be injected would eliminate one drawback of setter injection, i.e. that a class might be in an uninitialized state when its’ behavior is invoked. These two solutions can easily be combined.

Since this design uses the port interface of a dependent to inject dependees, I gave it the name “Port Injection”.

---

Figure 5. Incarnate Injection. The dependent is a capsule part on the assembler.
### 5.2.3 Parameterize Transition: An Adapted Design of Parameterize Method

The adapted design of Parameterize Method, which here goes under the name Parameterize Transition, uses the same mappings as Setter Injection: M5, M8 and M10. The main difference between Parameterize Method and Dependency Injection is transferred to the adapted versions: with Parameterize Transition the dependency should not be held as a member attribute, meaning that the scope of the dependency is limited to the scope of the transition that the dependee is sent to.

<table>
<thead>
<tr>
<th>Design of dependent capsule</th>
<th>Injector/Assembler</th>
<th>Dependent is configured with a dependee using</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependees are kept as attributes which are set in a “setter” transition using a passed-in object.</td>
<td>The injector is a client (usually a capsule) that holds a reference to a port which is connected to the injection port on the dependent.</td>
<td>A message with the dependee(s) as message data. A protocol for injection of dependees is needed.</td>
</tr>
</tbody>
</table>

Table 7. The design of Parameterize Transition.

### 5.2.4 Adapting Factory Method

Factory Method relies on encapsulating and overriding the class behavior where the concrete dependees are created. According to mapping M13, overriding of behavior in RSA-RTE can either be done using method overriding or redefinition of state transitions. Encapsulating parts of a state transition in RSA-RTE is normally done using non-public methods. Since factory method only requires overriding of non-public functions which are called from within the capsule, no capsule references are involved. Therefore, it is possible to implement factory method in RSA-RTE without any modifications.

Mapping M14 states that the model can specify which dependent type should be used. Unit tests can use a subclass which has been configured so that the factory method returns a mock dependee instead of creating a new dependee. However, still there is the problem of injecting the mock object into the subclass, since we want to be able to create and set up the mock object in the test case. Both incantate injection and port injection can accomplish this, but since the subclass is isolated to the test environment where it only needs to exist in one
instance, setter injection with a static setter method is another option that can be considered safe. An example of such a solution can be found in section 5.3.4.

<table>
<thead>
<tr>
<th>Factory Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design of dependent capsule</strong></td>
</tr>
<tr>
<td>The creation of dependees is encapsulated in factory methods, which are overridden by subclasses of the dependent.</td>
</tr>
</tbody>
</table>

Table 8. The design of Factory Method.

### 5.2.5 Adapting Abstract Factory

Abstract Factory does not involve classes communicating with capsules like the other solutions. Instead, the design of Abstract Factory revolves around a factory entity which is called by the dependent capsule. This entity can be implemented as a passive class in RSA-RTE. Therefore, Abstract Factory can be implemented in RSA-RTE without modifying the existing design of the pattern. In fact, Abstract Factory was one of the existing solutions that had been tried at Ericsson before this study.

During testing, a mock factory can be used instead of the “default” factory. The mock factory needs to be configured with a mock object somehow. Since the mock factory is a normal class, any type of dependency injection can accomplish this for us.

With the current test framework, in order to be able to vary the factory implementation from the test case, capsules should not keep dependees as attributes that are set upon initialization of the dependent. This is because the dependent will be initialized before the test case gets a chance to change the factory to a mock factory.

<table>
<thead>
<tr>
<th>Abstract Factory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design of dependent capsule</strong></td>
</tr>
<tr>
<td>The creation of dependees is delegated to a global static factory instance.</td>
</tr>
</tbody>
</table>

Table 9. The design of Abstract Factory.
5.2.6 Adapting Interface Injection

Interface Injection requires the dependent to implement public setter methods declared in interfaces. According to mappings $M11$ and $M12$, interface implementation can be done in RSA-RTE by declaring an interface element containing public methods which are implemented by capsules using the interface realization feature. However, this does not help us with the actual injection of dependees, since mappings $M5$ and $M7$ tell us that the use of public methods in C++ corresponds to using either state transitions or static methods for capsules – not normal public methods. A design which actually utilizes the interface realization feature would be to encapsulate injection code in the interface functions and call them from the transitions that set up the dependencies. This could possibly be used to make dependencies more explicit, since each dependency is explicitly shown by an implemented interface. Indeed, this is one of the ideas behind interface injection. Still, however, the actual injection would have to be made using another pattern (such as Port Injection). Thus, Interface Injection, as such, cannot be adapted into a pattern that solves the problem of injecting dependencies by itself.

5.3 Implementing the Solutions

This section displays the results of implementing the designs. The implementations shown here are proof that the designs presented in the previous section are realizable for RSA-RTE. However, they are only examples of possible implementations. Other implementations cohering to the same designs might be possible as well.

Subsection 5.3.1 discusses how the solutions could potentially impact runtime performance.
5.3.1 Incarnate Injection

For Incarnate Injection, ownership of the dependee should be kept by the entity creating it. Thus, no deletion of the dependee pointer should be performed in the dependent if the dependee was passed as initialization data. If the capsule should also handle the case when it is a fixed capsule part, a boolean value can be used to indicate whether the dependee attribute should be deleted by the dependent. This is illustrated in Code example 4. Unfortunately, some ugly typecasting is required in order to set the dependee attribute correctly.

In this example, the test case communicates the initialization data (the mocked dependee) to the capsule containing the dependent using a macro, telling the containing capsule to incarnate the dependent with the mock object.

```cpp
//Transition “initial” on the dependent capsule
if(rtdata) { //rtdata is the initialization data passed to “initial”
    this->initDataWasNull = false;
    this->dependeePtr =
        const_cast<DependeeIfD*>(static_cast<const DependeeIfD*>(rtdata));
} else { //Will happen if the dependent is a fixed capsule part
    this->initDataWasNull = true;
    this->dependeePtr = new DependeeD;
}

//Transition “teardown” on the dependent capsule
if(this->initDataWasNull) {
    //Will only happen if the dependee was created by the dependent.
    delete this->dependeePtr;
    this->dependeePtr = NULL;
}

//Test case
DependeeMockD* mock = new DependeeMockD;
INCARNATE_TEST_OBJECT(mock);
.
.
//Code for setting expectations on the mock and testing the dependent capsule
.
.
delete mock; //The test case is responsible for deleting the dependee.
mock = NULL;
CLEAR_TEST_OBJECT();
return true;
```

Code example 4. An example implementation of Incarnate Injection.
5.3.2 Port Injection

For Port Injection, the entity passing the dependee pointer to the dependent keeps ownership and should be responsible for deleting it. This is required since there is no way for the dependent to know the (concrete) runtime type of the injected dependee. It is therefore important that the injecting entity exists for as long as the dependent exists. Otherwise, the dependent might try to dereference a NULL pointer at some point.

Figure 7 shows an example of how the state machine can be used to support port injection. In this example, a waiting state is added before anything else can happen, to ensure dependees are injected initially. In the figure, setDependencies and changeDependencies are both triggered by an incoming event, typically on a port dedicated to injection of dependee. The top state is hierarchical and contains logic which is hidden at this level. Dependees can be changed at any point while the top state is active, corresponding to setter methods on a class.

There are many ways to implement port injection. An alternative could be to first have the dependent send a message to another capsule, requesting the dependee from it. Interestingly, this can be seen as a capsule implementation of the Factory pattern.

Figure 7. Example state machine for port injection.

//Transition “setDependencies” on the dependent capsule.
this->dependeePtr = rtdata; //The type of rtdata can be specified in the model

//Test case
DependeeMockD* mock = new DependeeMockD;
//Inject the mock by sending a message. SEND is part of the test framework.
SEND(portInjectDependees, Inject, mock);
.
.
//Code for setting expectations on the mock and testing the dependent capsule
.
delete mock; //The test case is responsible for deleting the dependee.
mock = NULL;
return true;

Code example 5. An example implementation of Port Injection.
5.3.3 Parameterize Transition

Parameterize Transition requires the creator of the dependee to keep ownership and also be responsible for deleting it. This is normally not a problem since the dependee is not kept by the dependent after the injection. Code example 6 shows an example implementation of Parameterize Transition.

```c
//Transition "doWork" on the dependent capsule. rtdata might contain some //"normal" parameters along with the dependee.
DependeeIfD dependeePtr = rtdata->dependeePtr;
dependeePtr->doWork(rtdata->requestData, this->someData);
```

//Test case
DependeeMockD* mock = new DependeeMockD;
int request = 1;
DoWorkParametersD sig;
sig.dependeePtr = mock;
sig.requestData = request;
 .
 .
//Code for setting expectations on the mock.
 .
 .
//Actual trigger code.
SEND(portRequest, DoWork, sig); // SEND is part of the test framework.
 .
 .
delete mock; //The test case is responsible for deleting the dependee.
mock = NULL;
return true;
```

Code example 6. An example implementation of Parameterize Transition.

5.3.4 Factory Method

Just as Factory Method encapsulates creation of dependees in a method, the same should be done for the deletion of dependees. Delete methods should be overridden by sub capsules since they know what concrete dependee types they are creating. Thus, ownership of dependees stays with dependents. For tests, however, the test case should have the responsibility to delete mocked dependees. Fortunately, it is possible to “trick” the dependent that it is controlling the deletion of the dependee by overriding the delete method and make it do nothing. This should not be considered bad design since the sub capsule is specific to the test environment anyway. On the other hand, using a static attribute to inject the dependee into the test sub capsule could be considered questionable.

With this design, it is fine to keep dependees as attributes which are deleted during capsule teardown.
//Transition “initial” on the dependent capsule. The concrete type created is hidden to the implementation since it depends on the capsule type of the dependent.
this->dependeePtr = this->createDependee();

//Function createDependee on the “default” dependent capsule return new DependeeD;

//Function createDependee on the test sub capsule. dependeePtr is a static attribute which is set from the test case.
return dependeePtr;

//Function deleteDependee on the test sub capsule. It is typically called from a teardown method.
//Do nothing. Test cases are responsible for deleting mock objects.

//Test case DependeeMockD* mock = new DependeeMockD;
//Inject the mock object using a static reference to the test sub capsule.
DependentCapsuleMockC_Actor::dependeePtr = mock;
.
.
//Code for setting expectations on the mock and testing the dependent capsule
.
.
delete mock; //The test case is responsible for deleting the dependee.
mock = NULL;
return true;

Code example 7. An example implementation of Factory Method.

5.3.5 Abstract Factory

Abstract Factory should rely on delete methods on the factory instance since dependents do not know the runtime type of the dependees. Unlike Factory Method, however, dependees should not be kept as attributes, but should instead be fetched from the factory when needed, due to reasons explained in section 5.2.5. It is the responsibility of a dependent that has fetched a dependee from a factory to also call the corresponding delete method on the factory.

Create methods and delete methods are implemented in the same way as for Factory Method, with the difference being that the methods are members of a separate class hierarchy. Such a hierarchy is shown in Figure 6.

Notice the static implementation pointer which can be substituted using the method loadFactory. The public methods createRequestHandler and deleteRequestHandler use the implementation pointer to delegate their work to hook methods which are overridden by concrete classes. This can be seen in Code example 8.

Similarly to Factory Method, the mock factory overrides the delete method to do nothing, since test cases own the mock object and are responsible for their deletion. Since we are dealing with normal classes, the test case can use any form of dependency injection to inject the mock object into the mock factory.
//Function “createRequestHandler” in UehUeRequestHandlerFactoryIfD.
return implPtr->_createRequestHandler();

//Transition “doWork” on the dependent capsule.
RequestHandlerIfD handlerPtr =
    UehUeRequestHandlerFactoryIfD::createRequestHandler();
handlerPtr->handleRequest(rtdata);
UehUeRequestHandlerFactoryIfD::deleteRequestHandler(handlerPtr);

//Test case
RequestHandlerMockD* mock = new RequestHandlerMockD;
//Inject the mock into the mock factory and load it into the abstract
//factory.
UehUeRequestHandlerFactoryMockD mockFactory(mock);
UehUeRequestHandlerFactoryIfD::loadFactory(mockFactory);

//Code for setting expectations on the mock and testing the dependent
//capsule

delete mock;  //The test case is responsible for deleting the dependee.
mock = NULL;
return true;

Code example 8. An example implementation of Abstract Factory.

5.3.1 Impact on Performance

Now that we know how to implement the solutions in RSA-RTE, we should be able to say something about how the patterns will impact runtime performance of the code. Here there are two aspects of the patterns that could possibly be of interest: time added for injecting dependees and time added when the dependent is using the dependee to handle requests from clients. When injection is only done once during an initialization phase, the overall performance, over time, is hardly affected at all, so we will disregard that case.

Intuitively it is the case when dependencies are injected during usage, i.e. when a dependee is given to or required by the dependent every time it is serving a request, that might impact performance. For Incarnate Injection, the dependee is kept as an attribute, which is set upon initialization of the dependent. This is almost identical to the case when no injection is used. Usage of the dependee should not be affected by the pattern. Port Injection requires one extra signal to be sent to the dependent in order to set the dependee attribute, but usage of the dependent should not be affected after that. Parameterize Transition requires a pointer to the dependee to be sent with each service request. As long as the message size is kept within the buffer size this should not be a big issue. However, when the buffer size is exceeded performance should be affected slightly. Also, some extra work is required to assemble the message and extract the dependee from the message every time the service is requested, which could also impact performance negatively. For Factory Method, the factory method can be called once to set a dependee attribute on the dependent. Thus, usage performance should not be affected. In the case of Abstract Factory, the factory is called every time a dependee is
needed. This could potentially affect performance negatively, since a new dependee object is created each time the factory is called. In order to avoid this, an alternative implementation of Abstract Factory would be to have the factory keep an instance of the dependee which is returned every time the create method is called. This is exactly what the mock factory does, but having the real factory do the same thing could be good with regard to performance. Of course, this implementation requires that dependents across a program are fine with sharing the same dependee object, and is therefore not always a possible solution.

To summarize, Parameterize Transition and Abstract Factory were considered to potentially affect performance negatively. They were therefore tested with regard to runtime performance during usage. The alternative implementation of Abstract Factory was tested as well. The results of the performance tests are presented in section 6.2.
Chapter 6

Evaluation

6.1 Usability

User effectiveness and efficiency were measured during the user tests in numbers of failed builds and task time, respectively. The results are shown in Table 10. In order to make it fair, the order in which the patterns were tested was varied between test sessions.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>User effectiveness - Number of failed builds</th>
<th>User efficiency – Task time [min]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average</td>
<td>Highest</td>
</tr>
<tr>
<td>Incarnate Injection</td>
<td>2,4</td>
<td>5</td>
</tr>
<tr>
<td>Port Injection</td>
<td>3,8</td>
<td>10</td>
</tr>
<tr>
<td>Parameterize Transition</td>
<td>2,8</td>
<td>4</td>
</tr>
<tr>
<td>Factory Method</td>
<td>5,2</td>
<td>11</td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>3,4</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 10. Effectiveness and efficiency as measured during user tests.

Answers to the survey questions are presented in Figure 8. The dark points mark the answers immediately after reading the instructions on how to implement each pattern, and the light points mark the answers immediately after solving the task using each pattern. For Incarnate Injection, 3 test subjects increased the score for perceived ease of use after solving the task and no test subjects lowered the score, while for perceived usefulness, 1 test subject increased the score and 1 test subject lowered the score. 2 test subjects claimed to have used Incarnate Injection or something similar for capsules before. Port Injection saw 3 increases and 2 decreases for perceived ease of use, with no increases and 1 decrease for perceived usefulness. 2 test subjects claimed to have used Port Injection or something similar for capsules before. Parameterize Transition saw 2 increases and no decreases for perceived ease of use, with no increases and 1 decrease for perceived usefulness. 1 test subjects claimed to have used Parameterize Transition or something similar for capsules before. Factory Method saw 1 increase and 3 decreases for perceived ease of use, with 1 increase and 2 decreases for perceived usefulness. 3 test subjects claimed to have used Factory Method or something similar for capsules before. Abstract Factory saw 1 increase and 2 decreases for perceived ease of use, with no increases and no decreases for perceived usefulness. 2 test subjects claimed to have used Abstract Factory or something similar for capsules before.
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Ease of use</th>
<th>Usefulness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The pattern seems easy to implement (1 – completely disagree, 5 – completely agree).</td>
<td>I believe that the pattern is a useful solution for testing with mock objects (1 – completely disagree, 5 – completely agree).</td>
</tr>
<tr>
<td>Incarnate Injection</td>
<td><img src="image" alt="Ease of use graph" /></td>
<td><img src="image" alt="Usefulness graph" /></td>
</tr>
<tr>
<td>Port Injection</td>
<td><img src="image" alt="Ease of use graph" /></td>
<td><img src="image" alt="Usefulness graph" /></td>
</tr>
<tr>
<td>Parameterize Transition</td>
<td><img src="image" alt="Ease of use graph" /></td>
<td><img src="image" alt="Usefulness graph" /></td>
</tr>
<tr>
<td>Factory Method</td>
<td><img src="image" alt="Ease of use graph" /></td>
<td><img src="image" alt="Usefulness graph" /></td>
</tr>
<tr>
<td>Abstract Factory</td>
<td><img src="image" alt="Ease of use graph" /></td>
<td><img src="image" alt="Usefulness graph" /></td>
</tr>
</tbody>
</table>

**Figure 8.** Average points from answers to survey questions (dark markers – before, light markers – after).

In the debriefing interview after the user tests, 4 of the test subjects mentioned Incarnate Injection among the patterns that they would prefer to use in the future. 2 test subjects mentioned Incarnate Injection among the simplest patterns to implement. 2 test subjects were concerned about the amount of code that Incarnate Injection might add to the initial transition.

1 test subject mentioned Port Injection among the simplest patterns to implement. 3 test subjects mentioned Port Injection as an ugly solution implementation-wise. 1 test subject called Port Injection “intrusive” and another expressed concern that it might “pollute the code”.

1 test subject mentioned Parameterize Transition among the patterns that they would prefer to use in the future. The same test subject mentioned Parameterize Transition as the simplest pattern to implement. 2 test subjects called Parameterize Transition an ugly solution implementation-wise. 3 test subjects were concerned that Parameterize Transition, although it simplified mock testing, would be difficult to use in production code.

1 test subject considered Factory Method the ugliest solution implementation-wise. 1 test subject was concerned that factory method would be intrusive in production code and another said that “Abstract Factory has an advantage over Factory Method since no inheritance is needed”, referring to overriding of the capsule under test.
All 5 test subjects mentioned Abstract Factory among the patterns that they would prefer to use in the future. 1 test subject said they liked it because it is a well-known pattern. 1 test subject liked it because it is “non-intrusive”. 3 test subjects mentioned Abstract Factory among the cleanest solutions implementation-wise, while 1 mentioned it as “not so clean, but flexible”. 1 test subject mentioned Abstract Factory among the simplest patterns to implement “once the factory class is in place”. 1 test subject called it “tricky to implement” while another test subject called it “abstract to understand, but pretty good”. 1 of the test subjects pointed out that creating new dependees might be inefficient and would prefer a getter-solution like the alternative implementation of Abstract Factory mentioned in section 5.3.1.

### 6.2 Performance

Performance was tested for 4 different designs: the “normal” case, where the dependee is held as an attribute and the dependent creates the dependee by itself, Parameterize Transition, the “normal” implementation of Abstract Factory and the alternative implementation of Abstract Factory mentioned in section 5.3.1.

The results of the test runs are presented in Table 11.

<table>
<thead>
<tr>
<th></th>
<th>Normal performance</th>
<th>Parameterize Transition</th>
<th>Abstract Factory, normal implementation</th>
<th>Abstract Factory, alternative implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average [s]</td>
<td>0.8362</td>
<td>0.8112</td>
<td>0.9793</td>
<td>0.8098</td>
</tr>
<tr>
<td>Average per request [µs]</td>
<td>83,62</td>
<td>81,12</td>
<td>97,93</td>
<td>80,98</td>
</tr>
<tr>
<td>Median [s]</td>
<td>0.8263</td>
<td>0.7988</td>
<td>0.9868</td>
<td>0.7986</td>
</tr>
<tr>
<td>Standard deviation [s]</td>
<td>0.1285</td>
<td>0.1224</td>
<td>0.1273</td>
<td>0.1195</td>
</tr>
<tr>
<td>Max [s]</td>
<td>1.531</td>
<td>1.135</td>
<td>1.366</td>
<td>1.170</td>
</tr>
<tr>
<td>Min [s]</td>
<td>0.3701</td>
<td>0.3627</td>
<td>0.4776</td>
<td>0.3567</td>
</tr>
</tbody>
</table>

Table 11. Results from performance tests. The time for 10000 requests was measured. The sample size was 200 test runs spread over 10 launches of the executable.
Chapter 7

Discussion

7.1 Evaluation Results

Since the patterns in this study are new in the sense that they have not been discussed in the context of RSA-RTE and capsules before, the evaluation results can be seen as a first indication of how they will fare in the future. It should be pointed out, though, that the best indication of pattern usefulness will likely be the experience that comes from using them. Thus, I am not writing any of the patterns off just yet. That being said, the user tests did give some interesting results, which will be discussed next.

The scores given to the design patterns in Table 3 gave a few hypotheses as to how the patterns would perform in the user tests. These hypotheses did not affect the conduction of the user tests in any way, but are interesting for the sake of discussing the results. It should be noted that the hypotheses were based on design pattern potential before even designing them for RSA-RTE, while the solutions evaluated in the user tests were adapted versions of the design patterns. However, in most cases the resemblance between the original design patterns and their RSA-RTE counterpart was significant enough to make the comparison.

According to Table 3, Constructor Injection, Setter Injection and Parameterize Method would promote high user efficiency, while Abstract Factory would lead to low user efficiency, with Factory Method promoting medium user efficiency. Judging by the task times during the user tests, it was clear that Setter Injection performed worse than expected when adapted to Port Injection. Abstract Factory and Factory Method performed on a similar level, which can either be interpreted as Factory Method performing worse than expected or Abstract Factory performing better than expected. However, Constructor (Incarnate) Injection and Parameterize Method went fast to implement, as expected.

For perceived ease of use, Constructor Injection, Setter Injection and Parameterize Method were predicted to score better than Factory Method and Abstract Factory. This turned out to be a correct prediction, judging by the answers to the survey question regarding ease of use after solving the task (the light markers in Figure 8). From these answers it was clear that Factory Method was perceived as the most difficult pattern. It was somewhat surprising that Factory Method scored that much lower than Abstract Factory in this category. Factory Method was also given a considerably lower score after solving the task than before, which might indicate a disappointing user experience. Also notable was that Incarnate Injection scored clearly best in this category, and had the best before-after ratio. This result could likely be credited to the incarnate macro which probably made the injection step more straightforward than the test subjects had expected beforehand.

In terms of perceived usefulness Constructor Injection, Setter Injection and Factory Method were expected to score best, while parameterize method and abstract factory were expected to score slightly lower. Judging by the answers to the survey questions regarding usefulness, Incarnate Injection scored high as expected and Port Injection scored slightly lower than expected. The biggest letdown was Factory Method, which once again was at the bottom of the scoring, while Abstract Factory scored higher than expected. Parameterize Transition, as expected, was not perceived as one of the more useful patterns.
The performance tests signified one thing: if the dependent needs to acquire new dependees each time one is needed, performance could be affected negatively over time. Out of the patterns tested, only the “normal” implementation of Abstract Factory has this property. For the alternative implementation of Abstract Factory, where a single dependee was managed by the factory, no negative impact on performance was discovered. However, this is a downside of Abstract Factory, since the alternative implementation is not applicable in cases when several dependents can’t share the same instance of a dependee.

No other issues were found in the performance tests. Of course, this does not mean that there aren’t differences between the patterns in terms of performance, but at least they were small enough to go under the radar in the performance tests conducted in this study. A more thorough investigation of pattern performance could be a whole new study of its own.

Overall, Abstract Factory and Incarnate Injection performed best according to the criteria chosen for the evaluation. It was surprising to see that Abstract Factory got high praise in terms of usefulness and — especially — ease of use, even though the test subjects struggled somewhat with implementing it (as indicated by the task times). One explanation for this could be that the test subjects realized that once the factory is in place it can be reused by all dependents. Thus, they might consider that it is worth the extra effort of implementing the factory in order to get a flexible solution. Interestingly, a study comparing the usability of a factory solution and a constructor solution in API design [11] found that most of the test subjects preferred the factory solution — even some of those who had struggled with it. When asked why, several test subjects answered something in the style of “even though I didn’t understand it, it must be good since it is used in the API”.

For this study, it is possible that Abstract Factory was favored because it is a well-known design pattern which means that it has been proven to work before. In fact, one of the test subjects said this out loud. Of course, this is a factor which should be taken into account when comparing it with solutions which are adapted to the modeling language — and thereby new to the developers. This is perhaps one of the most important merits of Abstract Factory when it comes to usability, since it means that new employees who are not accustomed to the modeling language will still be able to understand how to manage dependencies as long as they are familiar with the Factory pattern.

### 7.2 Method

An effort of enforcing the validity of the study was made during the pre-study, when relevant evaluation criteria were determined based on interviews with Ericsson employees. Based on what was found out during those interviews, usability seemed like a valid criteria to evaluate against. A thorough effort of defining usability was made using published literature and ISO-standards, and landed on a definition introduced by Bevan [26], which seems to be well accepted by the field.

One could argue that the talk-aloud sessions were too distracting for the test subjects and affected the variables that were to be measured: user effectiveness, efficiency and satisfaction. An alternative would have been to perform controlled beta tests where user satisfaction data was collected from user surveys rather than the actual test sessions, rather than intrusive test sessions. Indeed, for a purely quantitative study this might have been a better alternative. However, a quantitative study would have required a larger sample size, for the sake of statistical reliability, than was feasible in this study. Furthermore, practice has shown that talk-aloud sessions are an effective way of gathering information about usability [33].

The decision of only conducting 5 user tests is supported by Nielsen at al.’s mathematical formula [34] and conclusion that as long as the group of users is homogenous, testing with more than 5 users will mostly discover the same problems instead of new ones [35]. It cannot be denied, however, that conducting 5 user tests and partly basing the evaluation results on survey answers and task times is a threat to the reliability of the evaluation results. With more
resources a larger user study would likely have increased the reliability of the results, given the chosen method.

Because of this, the survey answers and task times should rather be seen as intuitive support for the qualitative findings during the user tests, i.e. what the users thought about the patterns while and after trying them out. Using a questionnaire served another useful purpose: it forced the test subjects to reflect on their experience with the patterns and made it easier to remember the patterns for the debriefing interview.

Another threat to the reliability of the usability test results should be mentioned: Port Injection and Abstract Factory had been used for mock injection at Ericsson before the start of this study. Even though only 2 of the test subjects claimed to have used Abstract Factory for capsules before, it is a fair assumption to make that all test subjects knew that it was already in use. It is possible that this gave them more confidence to praise Abstract Factory, and might be an explanation why all test subjects mentioned Abstract Factory among the patterns that they would prefer to use, even though some seemed to struggle with implementing it. Interestingly, though, Port Injection did not receive that much praise, which gives reason to believe that the test subjects were able to stay somewhat unbiased after all.

The performance tests measured the performance of code examples that tried to accentuate the theoretical characteristics and differences between the patterns. An alternative to testing code examples would have been to test actual production code modified using the patterns. This would likely have given a truer picture of the performance of the patterns when applied “in practice”. This option was disregarded because it was feared that it would be difficult to discover any measurable differences between the solutions when the execution of pattern specific code made a much smaller proportion of the total execution time for each transaction. Thus, in the end it boiled down to a lack of precise measurement, something that could have been solved with more time at disposal. This definitely poses a problem for the reliability of the performance results. Pattern performance, as mentioned in the previous subsection, is something that should be investigated more thoroughly in a future study.

In general, the literature research focused on finding published journal articles and books in the technological field. The general approach was to choose references with high relevance for the topic in question, as well as literature with a high degree of “acceptance” in the field. High relevance includes applicability of the content onto this study and also the age of the literature in question. It had to be considered whether the literature content is still valid today. Besides this, a high citation count was regarded as an indicator of acceptance by the field. For articles, the reputation of the journal where they had been published was also considered an indicator of acceptance.

Sometimes a conflict emerged when trying to satisfy both these conditions. It had to be considered whether it was more important to use new, up-to-date literature or older, well-cited literature which ideas had “stood the test of time”. Since a lot of the research in this study was focused on design patterns and design principles which were hot topics in the 90s and early 00s, much of the bibliography is leaning towards articles that were published more than 10 years ago. This was a conscious choice since, as far as I know, the theory of object-oriented programming and design patterns hasn’t changed drastically over the past 15 years.

### 7.3 The Work in a Wider Context

Society, now more than ever, is heavily reliant on mobile networks. The briefest downtime might cause societal services to go out of function. This could potentially be expensive and, in the case of an alarm service, even put people in danger. Therefore society’s demand on the quality of mobile network software, like the software produced at Ericsson, is high. In order to meet such high demands software testing is essential. Being able to perform mock testing helps to make the unit testing process more efficient. This means that developers can write more test cases, improving the quality of mobile network software in the long run.
Chapter 8

Conclusion

The aim of this study was to simplify unit testing of capsules with mock objects by looking at object-oriented design patterns performing dependency inversion and applying them to the capsule concept. More specifically, the applied patterns needed to be usable with regard to effectiveness, efficiency and satisfaction from the programmers’ point of view.

When tested by five programmers, Incarnate Injection (an application of Constructor Injection) and Abstract Factory were considered the most usable patterns. Incarnate Injection seemed to be easier to implement and was more efficient in that sense, while Abstract Factory was considered more flexible, and thereby more useful. Abstract Factory was also considered good because it is a well-known design pattern that does not require modification in order to be implemented for capsules, making it easier for programmers who are new to the capsule concept.

Before this study we knew that there are differences between capsules and classes which make it difficult to apply existing object-oriented design patterns to capsules. This study has shown that it is possible to adapt design patterns to capsules by looking at similarities on a conceptual level. However, there are cases when this adaptation is impossible or at least not advisable (e.g. Interface Injection). Furthermore, even when adaptation is possible, it is apparent that different patterns adapt differently well to capsules. There are clear indications that the usability of some patterns differs between the class version and the capsule version. One example is Setter Injection. It is one of the more straight-forward ways of performing dependency injection for classes, but its’ capsule counterpart, Port Injection, was not a favorite among the programmers.

Although the usability tests were designed for Ericsson developers, there is no reason to believe that the usability results should not apply to any industrial setting where RSA-RTE is used to develop software. Writing code with effectiveness, efficiency and satisfaction is something we all would like to do, and the goals of Ericsson programmers should be no different from other programmers in that regard. It should also be said that the capsule concept extends beyond the RSA-RTE modeling language. Therefore, it is not impossible that these RSA-RTE patterns are also applicable to related modeling languages that deal with capsules.

Even though mock testing was in focus in this study, dependency inversion in general is an important design principle in object-oriented design. Although capsules are different from classes in the sense that a capsule often contains more logic and responsibilities than a single class, it is safe to assume that there are cases where a capsule application would benefit from using dependency inversion. The five patterns explained in this study provide ways of doing that.

So after this study, what should future studies focus at? Well, as mentioned in Chapter 7, it would be interesting to see a more thorough investigation of how the patterns affect program performance. Here, there are many aspects of performance that could be of interest that
weren’t considered in this study: effects on program size, memory concerns such as heap allocation versus stack usage, startup time etc. This is of relevance since the concept of capsules was created with real-time software in mind, and real-time software is usually associated with high demands on performance.

One way to make use of the patterns in this study could be to develop IDE support for implementing them in an effort to streamline the developing process. This would certainly increase the return on investment for a pattern like Abstract Factory, which was found useful but somewhat time-consuming to implement. It could also be interesting to shift the focus from dependency inversion and see a more systematic adaptation of other object-oriented design patterns to capsules. The conceptual mappings used in this study could be a means to this end.
References


Appendix A

User Test Material

This appendix contains samples of the material prepared for the user tests.

A.1 Task 1: Pseudo-code

```c++
using ::testing::Return;
using ::testing::_;
SETUP_GMOCK();

//Create a mock object
PongHelperIfMockD * pongMockPtr = new PongHelperIfMockD;

//----------------------------------------------------------------------------
//Assignment: Pseudo code
//----------------------------------------------------------------------------
//Assume that this is a test case for a capsule named CapsuleUnderTestC, which has a dependency that you want to mock out.
//TODO: Write pseudocode that configures CapsuleUnderTestC with the mock object pointed to by pongMockPtr.
//Your code here

//----------------------------------------------------------------------------
//Set expectations on mock
//----------------------------------------------------------------------------
EXPECT_CALL(*pongMockPtr, getPongData(_)).Times(1).WillOnce(Return(3));
//Ping data
int i = 3;

//Trigger Behavior
SEND(pingPong, Ping, i);
RECEIVE(pingPong, Pong, int, response);

//Verify Behavior
VERIFY(response == 3);

//Cleanup (Only if needed)
delete pongMockPtr;
pongMockPtr = NULL;
return true;
```
A.2 Task 2: Refactoring

A.2.1 General Instructions

You will be asked to solve this task 5 times using different design solutions. Each solution has a code skeleton associated with it. The given code skeletons are almost identical. The code skeletons are divided between a software unit (SwU) project and an associated test project. The SwU project contains a capsule with one responsibility: it takes a “Ping” message with an integer and returns a “Pong” message with another integer. This capsule, which will be referred to as the dependent capsule, delegates the “Ping-Pong” work to a passive class, a dependee. This dependee has an interface called “PongHelperIfD” and a concrete implementation called “PongHelperD” located in the project UehUeInjectionCommonLU. A mock implementation, called PongHelperIfMockD, is located in the project UehUeInjectionCommonLUTestDouble.

The test project has been set up to test the given implementation of the dependent, which is currently depending on PongHelperD. Your task will be to refactor the dependent using the different solutions for mock injection, and to change the test case so that it tests the dependent by injecting a mock object instead.

A.2.2 Sample Task Description

Port Injection

Port injection is the RSA-RTE equivalent to using a setter method in order to set a dependee attribute. In order to implement setter injection, a new service port is added to the dependent capsule. This new port has an in-event with a data class containing a pointer to an abstract dependee as data. A new “initializing” state and a new “setter” transition are added to the state machine. The dependee should be kept as an attribute of the dependent. A dependee object can now be sent via a message to the dependent capsule.

Pretest questions

After reading the above instructions; to what extent do you agree with the following statements?

I understand how to implement Port Injection.

- Completely disagree
- Completely agree

Port Injection seems easy to implement.

- Completely disagree
- Completely agree

I believe that Port Injection is a useful solution for testing with mock objects.

- Completely disagree
- Completely agree
I have used Port Injection, or something similar, for capsules before.

☐ Yes
☐ No

Task

Use the project importer to import project “UehUePortInjectionSwUTest” (and its project references) into the workspace. Take a look at “exampleTestCase” in the test project. It is set up to test “PortInjectionC”, which is currently managing its’ dependency to “PongHelperIfD” by itself. Please build the test project and run the executable to make sure that everything is set up properly.

Your task is to test the capsule called “PortInjectionC” in project UehUePortInjectionSwU with a mock object by refactoring the capsule so that it uses Port Injection for its’ dependency to “PongHelperIfD”. A data class containing a pointer to PongHelperIfD (to be used for message passing) has been created in UehUeInjectionCommonLU. Changes to “exampleTestCase” and the test framework model will also be needed. The task will be considered completed when the test project is built successfully and the test case “exampleTestCase”, using a mock object, passes.

Posttest questions

After working with the task; to what extent do you agree with the following statements?

Port Injection was easy to implement.

Completely disagree ☐ ☐ ☐ ☐ ☐ Completely agree

I consider Port Injection to be a useful solution for testing with mock objects.

Completely disagree ☐ ☐ ☐ ☐ ☐ Completely agree

A.2.3 Debriefing Questions

1. Which one of these design solutions would you rather use for mock testing in the future? Why?
2. Which design solution or solutions seemed easiest to use?
3. Which design solution or solutions seemed most clean code-wise for solving the problem?
4. Which design solution or solutions seemed most “ugly” code-wise for solving the problem?
5. Which design solution seemed most complicated to implement?
   • Can you still see any advantage that this solution has over the other solutions?
6. Did you come to think of a way to improve any of the solutions?