Design Patterns in Action: Development of Java Applications for Configuration of Telecommunications Network Models

Kevin Coghlan

National University of Ireland, Maynooth

Department of Computer Science

A dissertation submitted in partial fulfilment of the requirements for the M.Sc. in Computer Science (Software Engineering)

January 2011

Supervisor: Dr. Diarmuid O'Donoghue

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of M.Sc. in Computer Science (Software Engineering), is *entirely* my own work and has not been taken from the work of others - save and the extent that such work has been cited and acknowledged within the text of my work.

Signed: ______________________ Date: ______________________

Abstract

During my work placement in a software development role at Ericsson, I have worked on a diverse range of software applications written in the Java programming language. These applications are utilised both within Ericsson, and by telecommunications network providers around the world, as part of a large software system responsible for the management of telecommunications network models. In this dissertation, I describe my work on two substantial software applications comprising part of this larger system, with a focus upon the software engineering techniques that were most relevant to my work.

Numerous design patterns were implemented during my work on each of these projects. Accordingly, both theoretical and practical aspects of relevant design patterns are discussed in depth. Several other software engineering techniques - code refactoring, software testing, and code reviews - collectively facilitated the delivery of high quality code. These techniques are each discussed in the context of my project work.

Having completed the two software projects described in this dissertation and witnessed their successful deployment, it is clear that each of the aforementioned software engineering techniques played an important role. Knowledge and application of these techniques accelerated the development of these projects, and minimised faults in the final delivered product.

Contents

List of Figures

Chapter 1

Introduction

On May 31st 2010, I commenced my work placement with LMI Ericsson Ltd., at the Ericsson Software Campus in Athlone, Co. Westmeath. Throughout my work placement, I have been a member of the *Configuration Service* (CS) Design Team, working on applications that form part of the large CS software system that Ericsson develops, maintains, and delivers to customers worldwide.

In this dissertation, I will focus specifically upon *two* of the most substantial projects that I have undertaken at Ericsson. The software engineering issues most relevant to my work on these two projects will be discussed in detail. These issues include:

- Identification and application of *design patterns* during software development constitutes the main focus of this dissertation. During my work placement, I have applied design patterns on many occasions to implement effective solutions to design problems that I encountered.
- I have frequently carried out *code refactoring* to improve the design of pre-existing code during my placement. I will discuss the usefulness of code refactoring techniques with reference to my own work.
- Exposing source code to the scrutiny of other software engineers through a *code review* process helps to ensure that the code is of high quality. I will describe the benefits of code review in the context of my own projects.
- *Automated software testing* is heavily used by the CS Design Team to ensure that modifications and additions do not break existing code. During my placement, I created a suite of automated tests to accompany a new application that I had developed. Details of this test suite and its usefulness during development will be discussed in depth.

1.1 Background & Company Details

Ericsson is a leading provider of telecommunications equipment and related services to mobile and fixed telecommunications network operators around the world. The company is currently the largest mobile telecommunications equipment vendor in the world, possessing a market share of 35% in this sector.[28] Two Ericsson facilities are currently present in Ireland; in addition to the Ericsson Software Campus in Athlone at which I am employed, another Ericsson facility resides in Clonskeagh, Co. Dublin. Some key statistics[5] about Ericsson include:

- More than 40% of all mobile telecommunications traffic worldwide passes through an Ericsson network.
- Ericsson employs 88,060 people globally, across 175 countries, where more than 10,000 Ericsson telecommunications networks reside.
- Over 25,000 patents have been awarded to the company, whose inventions include such influential technologies as Bluetooth.

My work has at Ericsson has revolved around a single large software system, which is part of a larger collection of network management tools provided by Ericsson. To provide some wider context for my own work, I will briefly describe the purpose of this collection as a whole.

1.1.1 Operations Support System - Radio & Core

Ericsson provides a collection of wireless network management tools which is collectively referred to as the *Operations Support System - Radio & Core* (OSS-RC). The powerful assembly of tools comprising the OSS-RC enables centralised management of large, complex telecommunications networks. Network administrators can utilise these tools to perform a wide variety of tasks, including:

- Monitoring of activities across an entire network.
- Configuration of individual network elements.
- Analysis of faults in the network.
- Performance measurement.

Numerous large telecommunications companies around the world, as well as Ericsson itself, rely upon OSS-RC for performing an array of network management tasks. However, during my placement, all of my project work involved one component of OSS-RC in particular. I will now describe this specific component.

1.1.2 Configuration Service

The two projects described in this dissertation involve applications that are each part of the *Configuration Service* (CS), an important component of the OSS-RC. This component is responsible for management of *telecommunications network models*. These models serve as finely-detailed records of the real-world configuration of telecommunications networks, and can be used to plan physical changes to individual networks prior to enacting them.

Each network model resides in its own *object database*, which the CS connects to in order to manipulate the model.[1] As depicted in Figure 1.1, object databases are utilised by telecommunications network providers around the world to store large numbers of objects, referred to as *Managed Objects* (MOs). Each of these MOs serves as a representation of a different type of real-world telecommunications network element (such as a radio broadcast tower, for example). Collectively, MOs contained within these object databases represent comprehensive, highly detailed models of real telecommunications networks.

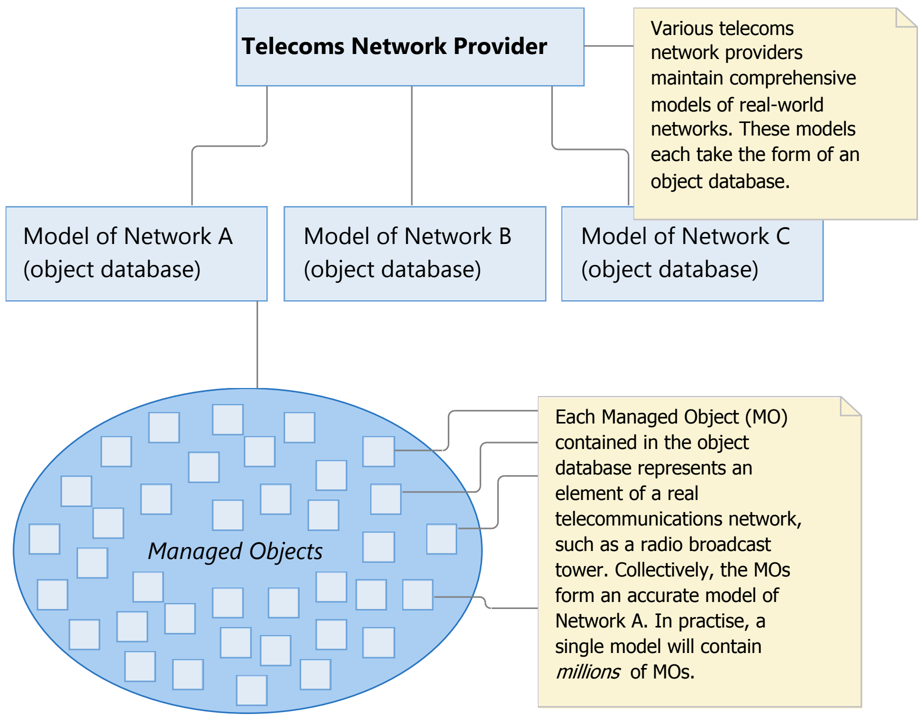

Telecoms Network Provider
Various telecoms network providers maintain comprehensive models of real-world networks. These models each take the form of an object database.
Model of Network A (object database)
Model of Network B (object database)
Model of Network C (object database)
Managed Objects
Each Managed Object (MO) contained in the object database represents an element of a real telecommunications network, such as a radio broadcast tower. Collectively, the MOs form an accurate model of Network A. In practise, a single model will contain *millions* of MOs.

Figure 1.1: Network models, comprised of *Managed Objects*

In order for these models to be useful, a system must exist that can examine

[1]For confidentiality reasons, I cannot provide details about the specific object database technology that the CS employs.

and manipulate the contents of any particular model. The CS fulfils this role; by use of CS functionality, it is possible to carry out a wide range of tasks upon telecommunications network models. Such tasks include, but are not limited to:[2]

- Creation of new MOs, which represent real-world network elements.
- Creation of various types of association between existing MOs.
- Deletion of existing MOs from the model.
- Modification of MOs which currently reside in the model.
- Retrieval of information about a defined range of MOs in the model.

The CS is a large, complex, and heavily-used software system. Due to the sheer size of the telecommunications network models that the CS maintains, efficiency of execution is considered a *crucial* factor of any code written during development work upon the CS.

All of my software development work at Ericsson has involved applications that comprise part of the CS. On account of this, during the subsequent chapters I will make frequent reference to the CS and the MOs that the CS manipulates.

1.1.3 The CS Design Team

The CS Design Team, of which I am a member, is responsible for all development and maintenance work pertaining to CS functionality.

There are nine software developers on the team, all situated at the Ericsson Software Campus in Athlone. The activities of the team as a whole are coordinated by a project manager, who assigns developers to each task and keeps track of progress made. I was the only member of the team on a work placement, with all other members having been part of the team for several years. I have benefitted from the experience of my colleagues in terms of improving my own skills as a developer, as well as reaching a high level of familiarity with many aspects of the CS codebase.[3]

The development model used by the CS Design Team is not fixed, with different development processes utilised for different projects. Some aspects of agile methodology have been employed from time to time, such as sprint backlogs to keep track of individual tasks that collectively make up a project.

[2]For confidentiality reasons, I cannot delve into too much detail regarding CS functionality. However, the explanation provided in this section should be more than sufficient for understanding the context of my own work.

[3]Knowledge transfer through *code review* was particularly useful in this regard.

Some members of the team are currently working on a detailed new development model to guide future CS Design Team projects.

Often, tasks are subdivided to allow individual developers to work independently on separate areas of code. This has been the case with the two projects described in my dissertation. In both cases, most of the code was developed by me, while my colleagues simultaneously worked on other areas of the CS codebase.

However, the work of individual team members is often strongly interrelated[4], and consultation between team members is frequent during development. For instance, on many occasions while working on my projects, I have presented ideas to my colleagues and solicited their opinions before implementing these ideas in code.[5]

Furthermore, all work carried out by individual developers is thoroughly reviewed by other developers on the team before being delivered to Ericsson customers. Throughout my placement, I reviewed the work of my colleagues in addition to having my own work reviewed.[6]

1.2 Relevant Modules

During my Computer Science (Software Engineering) M.Sc. degree course at NUI Maynooth, I studied eight separate modules prior to my work placement, which focused upon techniques with high relevance to industrial software engineering practise. During my work placement, two of the modules in particular that I studied have proven to be the most relevant to my work:

1. *Requirements Engineering & System Design (CS607).* Among other software engineering topics, this module placed a strong emphasis upon the application of design patterns as a key aspect of software engineering best practises, with numerous design patterns covered in depth throughout the module. During my work placement, I have encountered many situations in which a particular design pattern was useful in solving a design problem encountered during software development. As my use of design patterns constitutes the main focus of this dissertation, the CS607 module proved to be the most relevant module to my work placement.

   In addition to design patterns, various aspects of the Unified Modelling Language (UML) were also encompassed by this module. During my work

[4]For instance, as I worked on the application described in Chapter 3, some of my colleagues were working on CS features that my own application would utilise.

[5]Specific examples will be brought up in Chapters 2 & 3.

[6]As this dissertation pertains to my own projects, all code reviews discussed will involve my own work.

placement, I have utilised UML diagrams on numerous occasions as a design aid. I will provide relevant UML class diagrams throughout this dissertation in order to illustrate design aspects of the projects that I completed during my placement. I will also use class diagrams to depict the templates for design patterns that I applied during these projects.

2. *Software Testing (CS608).* This module provided comprehensive coverage of software testing principles and techniques, many of which proved relevant during my work at Ericsson. During my work placement, I created an automated test suite to accompany a newly-developed application. To build this test suite, I utilised a testing framework that was covered during CS608. This test suite embodied key software testing techniques such as *black box testing*, which received a thorough treatment during the course.

1.3 Relevant Languages & Tools

I will now provide some background information regarding the programming language that I used for all of my development projects at Ericsson (Java), in addition to a selection of tools which were heavily utilised during my work on these projects.[7]

1.3.1 Java

Many software projects at Ericsson have been developed in the Java programming language. In particular, the CS is comprised predominantly of Java code.

Various aspects of Java were covered in the CS613 (*Object-Oriented Programming*) module during my Masters degree course, which was useful in terms of revising some of my Java knowledge. As I had also written many Java applications over the course of my B.Sc. degree[8], I began my work placement at Ericsson with a reasonably strong knowledge of the language.

The projects that I have undertaken over the course of my work placement have provided me with many opportunities to improve upon my existing Java knowledge. These projects involved use of some features of the language that were less familiar to me. In particular, Java's extensive support for multiple threads of execution, which I had not worked with extensively in the past, would play a pivotal role during my work on various multithreaded applications[9] at Ericsson.

[7]Each of the tools described here will be discussed further in Chapters 2 & 3, with specific examples of their use.

[8]Computer Science & Software Engineering, NUI Maynooth.

[9]Chapter 3 describes a heavily multithreaded application that was developed from scratch during my placement.

1.3.2 JUnit Framework

JUnit is an open-source unit testing framework for the Java programming language. The tool is intended to facilitate the development and execution of repeatable, fully-automated test suites. JUnit test suites, executed upon Java applications, are comprised of test cases that are themselves written in the Java programming language.

During my work placement, I made liberal use of the JUnit framework, along with all other members of the CS Design Team. Since JUnit was covered extensively during the CS608 module described previously in Section 1.2, I was immediately comfortable with the framework upon commencement of my work placement. All tests for the CS itself that I have written and executed while at Ericsson have been based upon the JUnit framework. Additionally, the new automated test suite that I created[10] alongside a newly-developed application was based upon JUnit functionality.

1.3.3 Apache Ant

Apache Ant is a software tool built around an XML-based scripting language, which is heavily used by Java software developers to automate software build processes. This functionality can be particularly useful when working with large Java applications (such as the CS), which are composed of an enormous number of class files that must be *enhanced* during the compilation process in order to facilitate their interaction with object databases.

Ant configuration files are XML-based, and are easy to adjust and extend based upon the unique needs of a particular project. While Ant is most commonly used to automate build processes, the capabilities of the tool extend far beyond simple compilation of code; the tool can also be used to automate diverse activities such as the execution of automated test suites.

In the context of my software development work at Ericsson, Ant has proven to be an indispensable tool. Many tasks are routinely performed by the CS Design team using custom-written XML configuration files for Ant, such as:

- Compilation of large Java projects, including the CS itself, into numerous .jar files.
- Enhancement of Java source code to facilitate interaction with object databases.
- Execution of large JUnit test suites containing many individual classes.

[10]This test suite will be described in Section 3.6 (page 84).

During my work placement, I have frequently made use of existing XML files that were written by my colleagues, in addition to performing numerous modifications and additions to these files to suit my own projects. In one particular case involving an automated test suite, I wrote an Apache Ant script[11] of my own in order to facilitate fully automated testing of a newly-developed Java application.

1.3.4 Unified Modelling Language

The Unified Modelling Language (UML) facilitates comprehensive visual modelling of software systems.[2] Using the visual syntax provided by UML, software engineers can prepare detailed diagrams which can depict many different aspects of a software system's architecture. The latest version of UML (2.2) specifies 14 different types of diagrams.

UML was frequently used by myself and other team members during my work placement, commonly for the purpose of clarifying software designs prior to implementation, and communicating design ideas among individual software developers on the team. I often found UML diagrams very useful for visualising a design and experimenting with different configurations of classes, prior to implementing new source code. Thus, during my work placement, both for my own benefit and to clarify my designs to others, I composed numerous UML diagrams.

In this dissertation, I will frequently use UML class diagrams to:

- Illustrate architectural details of the software that I have written.
- Depict templates for design patterns that I implemented.

1.3.4.1 Notes on UML Notation

The diagrams depicted in this dissertation were created using IBM Rational Software Architect (RSA). In class diagrams, RSA depicts the protection level of attributes and methods using distinct icons to the left of the attribute or method name. These icons are distinguishable both by their shape, and their colour. The meaning of these icons is clarified in Figure 1.2.

[11]This will be discussed further in Section 3.6.2 (page 85).

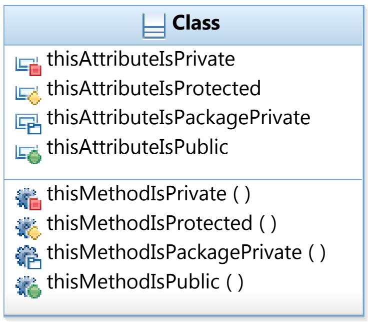


Figure 1.2: Protection level notation in Rational Software Architect

1.3.5 IBM Rational ClearCase

ClearCase is a software configuration management (SCM) tool which is used extensively at Ericsson. While working with the CS source code, all members of the CS Design Team deposit source code changes into a *Versioned Object Base* (VOB) within ClearCase. The VOB is accessible to all team members. Source code can be *checked out* at any time by individual team members. Developers can then *check in* modified source code to the VOB when they have completed their changes.

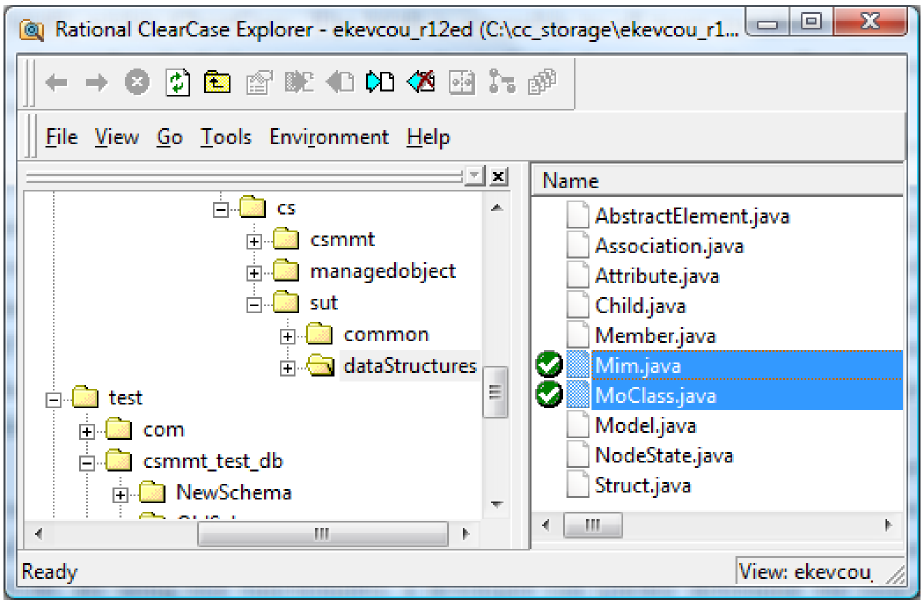


Figure 1.3: Rational ClearCase Explorer, depicting two *checked out* files

A descriptive comment can be provided at the time of each check-in to the VOB. This practise, although optional, is encouraged and frequently employed among the members of the CS Design Team. Typically, these comments will contain a succinct description of the changes that have been made (i.e. "Fixed issue #124 by repairing method X", "Removed redundant code from method Y"). These descriptions allow developers to determine at a glance which changes have been made to particular files, rather than having to look at the source code directly.

Since all developers on the team share access to the same VOB, any developer can review the work of other developers at any time, provided that it is checked in. Furthermore, ClearCase allows any given file to be compared with the previous version stored on the VOB. By use of this functionality, a developer can quickly determine the exact changes that another developer has made, and step through the changes in sequence. In this regard, ClearCase facilitates the practise of frequent code review among CS Design Team members.

1.4 Relevant Software Engineering Techniques

In this section, I will provide an overview of some software engineering techniques relevant to the projects that I worked on at Ericsson. Specific applications of each of these techniques will be discussed in detail in the subsequent chapters.

1.4.1 Design Patterns

A design pattern represents a general solution to a commonly occurring problem in software design.[7] For a particular problem encountered while designing an application, there will often exist an applicable design pattern that can be utilised to implement an effective solution to the design problem. Over the course of the two subsequent chapters relating to specific projects that I worked on at Ericsson, I will describe in detail eight separate design problems, and the design patterns that I applied in order to solve them.

1.4.1.1 Origin of Design Patterns

The concept of design patterns was first originated by Christopher Alexander, a civil engineer. Alexander recognised that certain design issues relating to the architecture of buildings and towns tended to arise again and again in similar forms. Recognising elements that were common to these design issues, Alexander realised that a set of patterns could be devised, with each pattern presenting a solution to a particular design issue.[1] These patterns represented tried-and-tested solutions to these recurring design issues. As these known design issues arose during a particular architectural project, an architect familiar with Alexander's patterns would be able to determine the appropriate pattern to apply in order to resolve the issue effectively.

Inspired by Alexander's concept of patterns, a group of computer scientists assembled a collection of design patterns that would serve as a library of reusable solutions to commonly-occurring problems encountered during software design. This collection of design patterns was published in a book in 1994, entitled *Design Patterns: Elements of Reusable Object-Oriented Software.*[7] This book has been highly influential to the field of software engineering, as the design patterns that it describes have been applied to a great number of software projects since the book was first published.

The principle of *reuse* of known solutions is fundamental to design patterns. Rather than solving every encountered design problem from scratch, a software developer can make use of modular, repeatable solutions in the form of design patterns that have been shown to provide effective solutions when applied to similar problems in the past. As the use of design patterns represents a considerably more efficient approach than 'reinventing the wheel' to solve every design problem encountered, they can dramatically accelerate the software development process when correctly applied to design problems that they are suited for solving.

1.4.1.2 Design Pattern Categories

Design patterns are divided into three categories:

- **Creational patterns.** These patterns define specific approaches for *instantiation* of objects. The *factory method* design pattern, which will be first discussed in Section 2.6.2.1, and again in Section 3.5.5.1, represents an example of a creational design pattern.

- **Structural patterns.** These patterns define particular ways in which relationships between classes and objects can be defined. They are concerned with how classes and objects are composed together to form larger structures. The *facade pattern* discussed in Section 3.5.1.1 and the *decorator pattern* discussed in Section 3.5.4.4 represent examples of structural design patterns.

- **Behavioural patterns.** These patterns define particular ways in which classes and objects can communicate with each other. I utilised several behavioural patterns while working on the two applications described during this dissertation. Behavioural design patterns that I have applied during software development at Ericsson include: the *command pattern* (discussed in Section 2.6.1.1), the *observer pattern* (discussed in Section

2.7.2.1), the *strategy pattern* (discussed in Section 3.5.2.1), and the *template method pattern* (discussed in Section 3.5.3.1).

1.4.2 Code Refactoring

Code refactoring is the process of modifying an application's source code in order to improve the internal structure of the application, making it easier to understand and maintain, without affecting the application's observable behaviour.[6] Refactoring of code can be necessary for a number of reasons, which include:

- As multiple developers make changes to an application's source code in order to realise short-term goals (such as the addition of new features to an existing system), the code can gradually lose its structure. As a consequence, the code becomes more difficult to read, understand, and maintain.

- A first attempt at implementing a particular feature will often not yield optimal code. On a few occasions during my work placement, I have experienced this myself: sometimes my first attempt at implementing some particular functionality, regardless of whether it *worked* as intended, turned out not to be optimal at the source code level. In such circumstances, I would then refactor my own code to optimise its internal structure and complete my work.[12]

The need for code refactoring can often be indicated by the presence of *code smells*. A code smell represents some identifiable characteristic of source code that indicates the possible presence of a problem. These problems can then be eliminated by refactoring the code to remove the code smell.

For example, the most prominent and well-known code smell is *duplicate code*. If the same code structure exists in more than one place, this indicates the need for refactoring in order to unify the duplicate structures. This allows for easier maintenance of the source code, since any subsequent modification to this unified structure will only have to be performed in one place, rather than in multiple places across the codebase. Furthermore, it makes the source code less bug-prone; when multiple instances of duplicate code exist throughout a codebase, some of these instances can be forgotten when modifications are made, leading to them becoming inconsistent with each other. Such incidents can give rise to severe bugs, further emphasising that duplicate code is best avoided, and eliminated wherever it is found.

[12]A specific example of this will be described later, in Section 3.5.3 (page 68).

1.4.3 Software Testing

Software testing is an important software engineering subdiscipline, for which numerous definitions exist. In 1979, Glenford J. Myers defined software testing as the process of executing a program or system with the intent of finding errors. William C. Hetzel presented an alternative definition in 1988: software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Broadly speaking, software testing processes are crucial for ensuring that software functions as its developers intended it to.

Numerous testing techniques and tools (such as the JUnit Framework described previously) were covered during the CS608 module. I will now briefly discuss some testing techniques that were applicable during my work placement at Ericsson, in particular during my development of the automated test suite described in Section 3.6 (page 84).

1.4.3.1 Black Box Testing

The specification of an application determines how it is intended to function. The purpose of black box testing is to ensure that an application functions according to its specifications. As the name of this testing technique suggests, the source code of the application under test is treated as a 'black box', as black box testing is not concerned with the inner workings of the application at the source code level. Rather, black box testing checks whether the *output or result* of an application's execution matches the *expected output or result* for a particular set of inputs.

Black box tests are commonly used among the extensive suite of JUnit tests that are regularly executed upon the CS, in addition to the automated test suite of JUnit tests that I developed to accompany a newly-developed application during one of my projects.

1.4.3.2 Regression Testing

Often when changes are made to an area of source code, these changes will cause unforeseen problems elsewhere in the source code. To detect these problems early and facilitate their repair, *regression testing* should be performed regularly.[19] This involves executing a series of tests after a change has been made to source code, which were known to have been passing *prior* to the change being made. If the tests now fail, this confirms that the recent change to the source code is responsible for the failure. This information has proven to be invaluable on numerous occasions, enabling the underlying problem to be found and fixed quickly.

Throughout my work placement at Ericsson, this testing technique has been employed extensively. During work upon the CS itself and applications which interact with the CS, thorough regression testing has been performed each time any significant changes have been made to the source code. This testing has often exposed the presence of problems, which may have gone undetected for some time if the testing had not been performed.

The automated test suite that I developed during my work placement was regularly executed for the purpose of carrying out regression testing upon my code as changes were made. Regression tests exposed newly-introduced problems in my source code on several occasions, which I was then able to immediately fix based upon the information I had gleaned from the tests.

1.4.4 Code Review

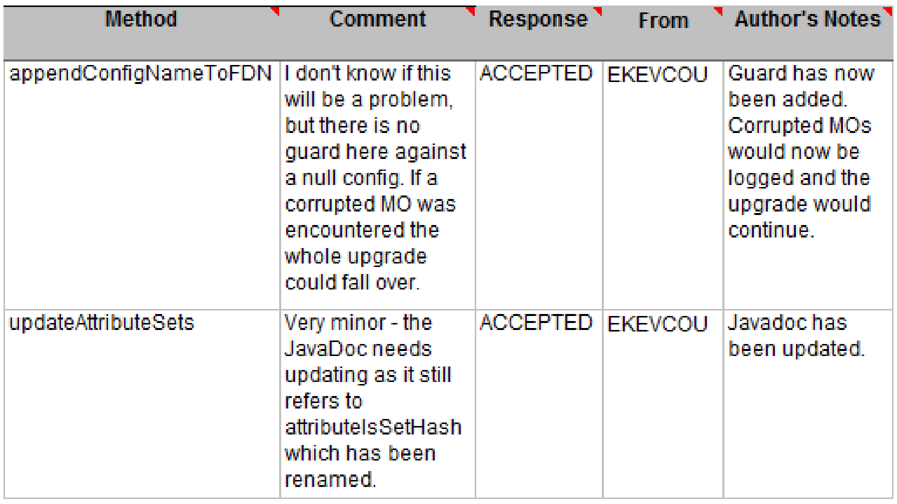
Code review is a thorough examination of source code, performed by developers who did not write the code under review. This practise is extremely useful during software development, as reviewers may detect issues with the source code that were missed by the original developers. Studies have demonstrated that code review can substantially reduce defects in delivered source code, resulting in 65% fewer code defects on average.[17]

In addition to improving the quality of delivered code, code review promotes *knowledge transfer* among team members in a number of ways:

- Reviewers develop familiarity with the source code being reviewed. By performing code reviews, CS Design Team members improve their familiarity with the CS as a whole.
- The developer having their source code reviewed will often gain new knowledge from the review, as reviewers present ideas that the original developer had not considered.

At Ericsson, each code review performed is accompanied by a *Code Review Record* document. This document stores all observations made by reviewers, and the response to these comments from the designer of the reviewed code. A very small snippet[13] of such a document is depicted in Figure 1.4.

[13]Many other columns exist in a typical code review document; these could not be depicted here due to page width constraints.



Method	Comment	Response	From	Author's Notes
appendConfigNameToFDN	I don't know if this will be a problem, but there is no guard here against a null config. If a corrupted MO was encountered the whole upgrade could fall over.	ACCEPTED	EKEVCOU	Guard has now been added. Corrupted MOs would now be logged and the upgrade would continue.
updateAttributeSets	Very minor - the JavaDoc needs updating as it still refers to attributeIsSetHash which has been renamed.	ACCEPTED	EKEVCOU	Javadoc has been updated.

Figure 1.4: Small section of a *Code Review Record* document

The purpose of each column is explained below:

- **Method**: This column refers to the Java method[14] that the reviewer is commenting upon.
- **Comment**: This column contains comments made by a particular[15] code reviewer.
- **Response**: This column contains my own response to each comment. In this case I have entered "ACCEPTED" for both comments, indicating that I agree with the observations of the reviewer.[16]
- **From**: This column refers to the original developer of the code being reviewed. "EKEVCOU" is my own identification code at Ericsson; this indicates that the code being reviewed was written by me.
- **Author's Notes**: This column contains my own written response to the reviewer's original comment. In this case, my responses state the actions I have taken to resolve each issue identified by the code reviewer.

Code reviews will be further discussed in the context of a specific project in Section 2.8 (page 41).

[14]Another column that is not depicted in this image contains fully qualified Java class names, to indicate which *class* the named method resides in.
[15]The specific reviewer who made the comment is identified in another column.
[16]If I disagreed with a comment, I could enter "DECLINED" instead. This might occur if a code reviewer made an inaccurate observation in error.

1.5 Dissertation Structure

The software development work performed over the course of my work placement has encompassed multiple distinct projects, ranging from minor modifications of existing code to development of an entirely new application from scratch. During this dissertation, I will discuss *two* of the more substantial projects that I have been involved with. These projects are related in the sense that both are part of the CS system, and share some software engineering techniques and themes in common, such as use of design patterns[17] and code refactoring. In all other respects, these two projects are independent of each other. In order to discuss these projects individually, they each receive a chapter of their own.

In **Chapter 2**, I will discuss my work on the *CS Library Test Tool* (Cslibtest). I carried out work on this pre-existing tool as part of one of the earlier projects that I was involved with at Ericsson. This project involved a substantial amount of code refactoring, which is discussed in detail during this chapter. As part of the code refactoring process, and in order to introduce entirely new functionality to the application, several design patterns were applied to the Cslibtest codebase as well.

In **Chapter 3**, I will discuss my work on the *CS Metadata Migration Tool* (CSMMT). This new application, developed entirely from scratch, represents the larger of the two projects covered in this dissertation. This chapter contains the initial analysis and early design decisions relating to the application, followed by comprehensive details of the implementation of CSMMT, including selection and implementation of applicable design patterns. The chapter concludes with details of the automated test suite that I constructed to accompany this new application.

In **Chapter 4**, I will briefly discuss the conclusions that I have drawn based upon my experiences with the software engineering techniques described over the course of Chapters 2 and 3.

[17]One design pattern in particular would prove useful during *both* projects, as will be discussed later.

Chapter 2

CS Library Test Tool

This chapter details my work on the *CS Library Test Tool*, which was originally developed by Ericsson in 2007. This tool is typically referred to in shortened form as *Cslibtest*.

2.1 Chapter Structure

In **Section 2.2 (*Overview of the Cslibtest Application*)**, I will describe the purpose and operation of the Cslibtest application.

In **Section 2.3 (*Motivation*)**, I will briefly discuss the motivation behind the modifications and additions that I made to Cslibtest.

In **Section 2.4 (*Requirements*)**, I will discuss each of the requirements that were determined for the work to be performed on Cslibtest. All of these requirements were to be satisfied prior to the release of the next Cslibtest version.

In **Section 2.5 (*Code Refactoring of the `CstestCLI` class*)**, I will describe the refactoring work performed upon the `CstestCLI` class, which was the main class of Cslibtest when I began working on the application. This refactoring entailed the creation of several new classes, in addition to considerable restructuring of the `CstestCLI` class itself.

In **Section 2.6 (*Addition of Command Queue Functionality*)**, the new additions made to the Cslibtest source code in order to support queuing of commands will each be explored in turn.

In **Section 2.7 (*Addition of Multiple Model Support*)**, I will discuss the work performed in order to implement support for multiple simultaneous network model connections, an improvement upon the single connection support that Cslibtest had provided previously.

In **Section 2.8** (***Code Review***), I will briefly describe the code reviews performed upon the Cslibtest source code over the course of my work on this application.

Finally, in **Section 2.9** (***Project Outcome***), I will describe the outcome of the project, which culminated in the release of a new version of Cslibtest.

2.2 Overview of the Cslibtest Application

Through a command-line interface, the Cslibtest application provides comprehensive access to the functionality provided by the Configuration Service. By entering commands into this interface, users of this tool can manually execute CS commands, which are used to manipulate managed objects within a network model. The purpose of the Cslibtest application is to serve as a *backdoor* into the CS for testing purposes. The operation of Cslibtest is depicted in Figure 2.1.

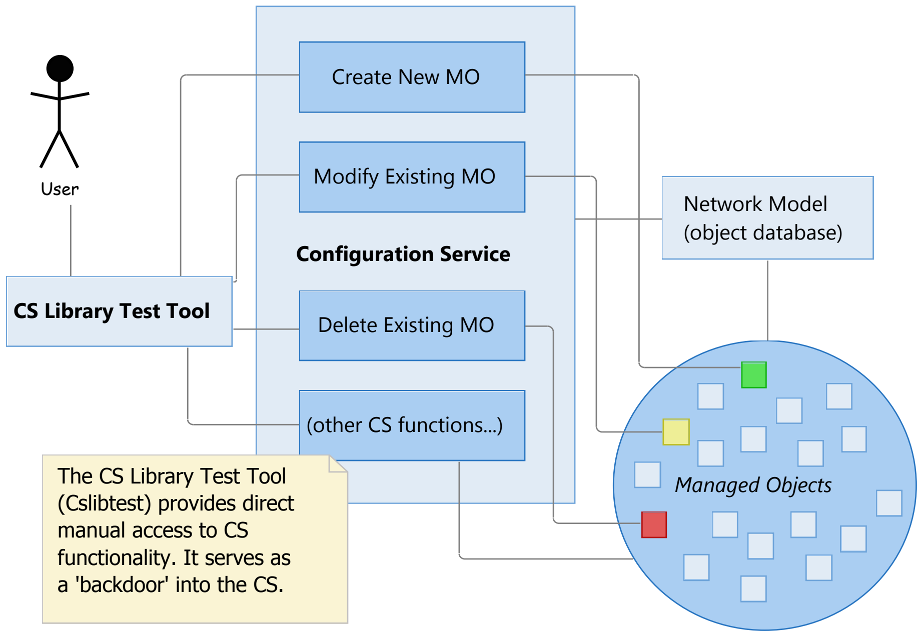


Figure 2.1: Cslibtest provides direct access to core CS functionality

By allowing such direct access to the internal functionality of the CS, Cslibtest has proven invaluable for preparation and execution of test scenarios which exercise the functionality of the CS, ensuring that each aspect of its functionality is working correctly. When changes are made to the CS, Cslibtest is commonly used afterwards to determine whether the changes made are functioning as the developers intended.

2.2.1 Modes of Operation

At the start of its execution, Cslibtest takes in a reference to specific network model as a command-line parameter. This network model will be connected to by Cslibtest; all CS commands subsequently executed by Cslibtest will be carried out upon MOs contained within this particular network model.[1] A *mode* parameter is also taken in, which determines the *mode of operation* that it will follow during the current session. These three modes are each described below:

1. **Interactive mode.** This is the only mode that involves user interaction with the application during its execution. When Cslibtest is executed in this mode, a command-line interface is presented to the user. Using this interface, the user can enter individual Cslibtest commands. As these Cslibtest commands are entered, CS functionality is executed upon the network model to which Cslibtest has connected. Results of these commands are output to the screen as they are executed.

2. **Single command mode.** If a single Cslibtest command is supplied as a command-line argument, the CS functionality corresponding with this command will be executed upon the network model to which Cslibtest is connected. The output from the single command will be displayed, and Cslibtest will then terminate.

3. **Command file mode.** If the location of a Cslibtest *command file* is supplied to Cslibtest as a command-line argument, Cslibtest will then execute in command file mode. Command files contain one or more Cslibtest commands, to be executed in linear sequence by Cslibtest, from first to last. After the CS functionality corresponding with *all* of the commands from the command file has been executed, Cslibtest will then terminate.

As these modes of operation are central to the operation of Cslibtest, I will refer back to them in the coming sections as I describe the modifications and additions that I have implemented.

[1]Since Cslibtest is a *testing* tool for CS functionality, usually a 'dummy' network model will be used as a testing sandbox, containing a relatively small number of MOs. By contrast, real network models utilised by telecommunications network providers often contain millions of MOs.

2.3 Motivation

When I arrived at Ericsson, the *Cslibtest* application had already been developed some years previously, and was in use throughout the company. However, over time, Cslibtest users had pointed out a number of perceived shortcomings. Based upon this feedback, it was clear that there was potential for improvement across numerous aspects of the application.

As one of my earlier projects at Ericsson, I was assigned by the CS Design Team project manager to carry out improvements to Cslibtest that were deemed to be required. I will now explain each of these requirements.

2.4 Requirements

The requirements for this project work were provided to me by other CS Design Team members who had been made aware of the shortcomings of the application in its current iteration, prior to my arrival at Ericsson. The project work for Cslibtest was separated into three separate tasks, each of which necessitated substantial modifications to the existing Cslibtest codebase.

2.4.1 Code Refactoring

Before I began working on Cslibtest, some substantial sections of the Cslibtest source code were in a disorganised state. Lack of documentation and a large number of complicated methods made the source code difficult to read and understand, which made the task of extending and maintaining the application more difficult for developers. Refactoring of these large sections of source code would be required, in order to improve the design of Cslibtest and render the code easier to understand and maintain by developers who would work on the application in the future.

Among the 111 classes present in the Cslibtest application, the class that required by far the most refactoring was the main class, named `CstestCLI`. During my work on Cslibtest, I performed some degree of refactoring upon many classes in the Cslibtest codebase. During my discussion of refactoring work done over the course of this project, I will focus predominantly upon my work on `CstestCLI`, as it represents one of the classes that experienced the most dramatic changes as a result of code refactoring.[2]

[2]Several *new* classes were created as a result of the code refactoring of `CstestCLI`; these will be discussed as well.

2.4.2 Command Queuing Support

Prior to my work on the application, Cslibtest users executing the application in *interactive mode* were unable to enter any input until a connection to the chosen network model had been established. As the CS functionality corresponding with each Cslibtest command could not be executed upon the network model until a connection with it was established, this approach appeared to make sense initially. However, users reported that this delay in accepting input made the application seem sluggish and unresponsive. These users expressed that they would prefer to be able to begin entering commands while the connection process was taking place.

In order to address this issue, I was asked to improve the *usability* of Cslibtest by enabling users to begin entering commands as soon as the application had started, even before a connection to the network model was first established. Any commands entered at this time must be *queued*, and then executed upon the network model immediately as soon as the connection process was finished. After any queued commands have finished executing and the results had been displayed, control must then be returned to the user.

Whereas the first requirement necessitated the modification of existing code, this requirement called for the addition of entirely new functionality to the Cslibtest application. The third and final requirement also necessitated the addition of new functionality that had not existed in Cslibtest before.

2.4.3 Multiple Model Support

Previously, Cslibtest was capable of connecting to only one network model during a particular session. Furthermore, there was no facility for switching between different network models after Cslibtest had been executed. If the user wanted to connect to a different network model, they had no choice but perform one of the following alternatives:

- Shut down Cslibtest and specify a different network model upon starting Cslibtest again.
- Execute several parallel instances of the Cslibtest application, each connected to a different network model.

These measures were proving to be an inconvenience for Cslibtest users; lack of support for simultaneous connections to multiple network models was deemed to be a significant weakness of Cslibtest in its current iteration.[3]

[3]Having access to several different network models at the same time during an execution of Cslibtest is useful for various test scenarios. For instance, each of the network models may contain MO types of different configurations, allowing for a diverse range of test scenarios.

To address this shortcoming, I was responsible for implementing support for simultaneous Cslibtest connections to an arbitrary number of network models. Several new classes were be required to implement this functionality, along with various smaller adjustments across the Cslibtest codebase.

Having described each of the requirements, I will now describe the work performed on Cslibtest to satisfy the first requirement.

2.5 Code Refactoring of the `CstestCLI` class

As specified in the requirements, based upon earlier code inspections carried out upon Cslibtest[4], the `CstestCLI` class represented the class most in need of refactoring of the 111 classes that comprised the Cslibtest source code. In order to decide upon a course of action, I analysed the existing source code for this class to determine why my colleagues considered it to be a prime candidate for refactoring.

2.5.1 Excessive Functionality of `CstestCLI`

The name of the `CstestCLI` class contains the CLI acronym, standing for 'command-line interface'. Intuitively, this would appear to suggest that the purpose of this class is to provide a command-line interface, for use when Cslibtest is executed in *interactive mode*. As I expected from the class name, this functionality was present in the class.

However, upon further examination of the class, I realised that it contained other functionality as well. As was described in Section 2.2.1, Cslibtest supports three distinct modes of operation. Although the name of the `CstestCLI` class suggests that it contains functionality relevant to *interactive mode* only, I found that all code relevant to *command file mode* and *single command mode* were present in the class as well. I realised that the class had too many responsibilities.

In order to improve the design of this class and Cslibtest as a whole, some of this functionality would have to be *refactored* into more suitable locations. As illustrated in Figure 2.2, the crowded nature of the `CstestCLI` class in its original form is clearly apparent upon examination. There were simply too many methods in this class, performing too many unrelated tasks.

[4]These code inspections took place prior to the start of my work placement.

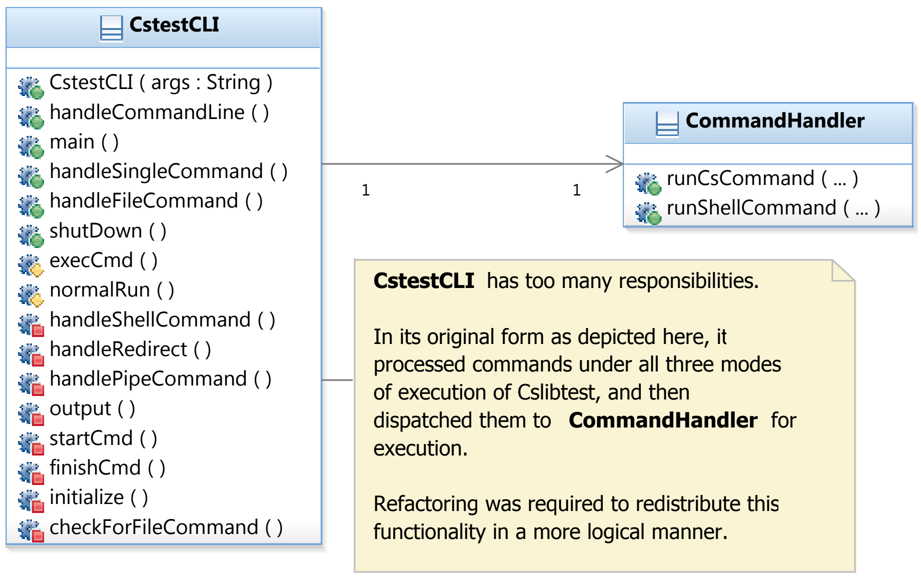

CstestCLI
CstestCLI (args : String)
handleCommandLine ( )
main ( )
handleSingleCommand ( )
handleFileCommand ( )
shutDown ( )
execCmd ( )
normalRun ( )
handleShellCommand ( )
handleRedirect ( )
handlePipeCommand ( )
output ( )
startCmd ( )
finishCmd ( )
initialize ( )
checkForFileCommand ( )
1
1
CommandHandler
runCsCommand (...)
runShellCommand (...)
CstestCLI has too many responsibilities.
In its original form as depicted here, it processed commands under all three modes of execution of Cslibtest, and then dispatched them to CommandHandler for execution.
Refactoring was required to redistribute this functionality in a more logical manner.

Figure 2.2: `CstestCLI` prior to refactoring

2.5.1.1 Aside: The `CommandHandler` class

The `CommandHandler` class, depicted in Figure 2.2, is responsible for receiving Cslibtest commands and coordinating their execution.

- Most Cslibtest commands specify some CS functionality to be executed on the network model; these commands are ultimately passed into the `runCsCommand` method for execution.
- Cslibtest also supports the execution of some shell commands, which are passed directly to the underlying operating system for execution. These commands are passed into the `runShellCommand` method. This additional functionality is useful for some Cslibtest users, who may want to execute such commands without having to quit the Cslibtest application.

Although I carried out minor modifications on the `CommandHandler` class during my work on Cslibtest, I did not refactor it extensively. However, since several of the classes that I will describe do interact heavily with `CommandHandler`, it will be depicted on several class diagrams for completeness.

I will now return to the refactoring of `CstestCLI`. As I examined this class further, I realised that its problems were not confined to a mere excess of functionality.

2.5.2 Layout Issues & 'Spaghetti Code'

As I examined the `CstestCLI` source code, I realised that the layout of the class was very convoluted. Disorganised calls from method to method created a confusing web of execution paths through the class. The protection levels for each method appeared to have been set almost at random, with several `public`, `private`, and *package-private* methods present with no apparent reason for the differences in protection levels between them.

Unfortunately, the aforementioned problems were all compounded by an almost total absence of documentation. No Javadoc was present to explain the purpose of any of the methods. Code comments, though present in some places, were few and far between.

This combination of issues indicated the presence of the most famous *anti-pattern* in computer science, which has existed in various forms since the emergence of the first programming languages. *Spaghetti code*, named after the twisted and tangled appearance of a bowl of spaghetti, refers to convoluted application code, often undocumented, which has an intricate and apparently arbitrary control structure that is difficult for software developers to understand, extend and maintain.[4] Code exhibiting this antipattern is characterised by unpredictable 'jumps' from one part of the source code to another.

In older programming languages that supported 'goto' statements, liberal use of these statements tended to be responsible for the unpredictable jumps across the source code that spaghetti code embodies. Java does not support 'goto' statements, but this does not eliminate the susceptibility of this programming language to the spaghetti code antipattern. Instead, the antipattern tends to manifest itself in Java as a confusing sequence of calls from one method to another, with no readily discernible structure.

It was apparent that extensive refactoring was necessary in order to render `CstestCLI` free of spaghetti code. My goal was to repopulate this class with clear, succinct, well-documented code that would be considerably easier to understand and maintain. These changes would also allow easier addition of command queuing support and multiple model support to Cslibtest. Owing to the importance of the functionality that resided within the original `CstestCLI` class, the application as a whole would be considerably easier to extend if this functionality was well-factored.[5]

Having examined the `CstestCLI` class, I next considered the exact changes to be made in order to improve the situation.

[5] As intuition would suggest, addition of new functionality to *any* application is easier if the source code of that application is well-factored; such an application is more readily *extensible* by developers.[6]

2.5.3 Outline of Major Refactoring Steps

I determined that several sweeping changes would have to be made to the `CstestCLI` class, culminating in the creation of some new classes and a significantly smaller `CstestCLI` class with many of its current roles stripped away. Before delving into the details of how these changes were carried out, I will briefly outline the planned series of changes as a whole.[6]

- As was shown in Figure 2.2, `CstestCLI` contained the `main()` method for Cslibtest. However, I believed that `CstestCLI` should be associated with *interactive mode* only, and therefore should only be instantiated if *interactive mode* was in effect. In light of this, the `CstestCLI` class did not represent the most logical location for the `main()` method. This method would have to be situated elsewhere.

- I decided to create a **new class** called `CstestController`, which would contain the `main()` method instead of `CstestCLI`. This class would perform all processing of command-line arguments, and initiate the connection process to each network model that was specified. If a single command was supplied on the command-line, it would be dispatched directly to `CommandHandler` for execution, thereby fulfilling the responsibility associated with *single command mode*, the simplest mode of operation.[7] If either of the other two modes of operation were in effect, specialised classes would be instantiated to handle them. `CstestCLI` would serve as one of these classes.

- A second **new class** called `FileHandler` would be created. All functionality relevant to *command file mode*, which originally resided in the overcrowded `CstestCLI` class, would be moved into the newly-created `FileHandler` class.

- After these refactoring steps had been completed, the `CstestCLI` class itself would now be dedicated to provision of command-line interface functionality alone. Instead of being utilised during every execution of Cslibtest as had been the case until this point, the `CstestCLI` class would only ever be instantiated and used when *interactive mode* was in effect.

- After the factoring-out of excess functionality within `CstestCLI` had been completed, the remaining methods relating to *interactive mode* would be *rewritten*, to eliminate the last traces of spaghetti code from the class. The rewritten methods would feature a more readily comprehensible control

[6]The new classes named here will be discussed in greater detail in the coming sections.
[7]No distinct class is required for this mode of operation, due to its simplicity.

structure, with code comments and Javadoc added to clarify the functionality present. Furthermore, most methods would be made `private`[8] to ensure that `CstestCLI` was the only class that could use them, rendering the design of Cslibtest as a whole more robust and easier to understand.

This concludes the outline of refactoring steps that I decided upon carrying out on the functionality contained within the `CstestCLI` class. I will now describe each change in the implementation that resulted from carrying out these steps.

2.5.4 Refactored Functionality - the `CstestController` class

As determined previously, it was inappropriate for `CstestCLI`, which would provide command-line interface functionality only from now on, to serve as a point of entry for the Cslibtest application as well. On account of this, I factored the `main()` method out of the `CstestCLI` class and into an entirely new class which serves as a dedicated point of entry for Cslibtest. This new class is named `CstestController` in order to succinctly describe the purpose of the class, while conforming to existing naming conventions within the Cslibtest codebase.

`CstestController` is responsible for:

- Processing all command-line arguments passed into Cslibtest at execution time, in order to determine the mode of operation in effect for this particular execution of Cslibtest.
- Establishing a connection to the network model(s) that Cslibtest will execute CS functionality upon.[9]
- Instantiation of the classes *appropriate to the mode of operation* which was specified at the command-line. Rather than attempting to supply methods for *all* modes of operation (as had been the case with the original `CstestCLI` class), `CstestController` instead *delegates* this work to other classes that are dedicated to specific tasks.
- Termination of Cslibtest after all commands had been executed (if operating in *command file mode* or *single command mode*), or at the point at which the user decides to quit (if operating in *interactive mode*).

With the `CstestController` class in place as depicted in Figure 2.3, the `main()` method had been successfully factored out of the `CstestCLI` class. The next step was to factor out functionality that was specific to *command file mode*.

[8] *"After carefully designing your class's public API, your reflex should be to make all other members private."* - Joshua Bloch, *Effective Java*[3]

[9] These connections would be established by leveraging the functionality of newly-developed classes. These will be discussed in Section 2.7 (*Addition of Multiple Model Support*).

2.5.5 Refactored Functionality - the `FileHandler` class

As described previously, a *command file* is processed by Cslibtest when it is executed in *command file mode*. The original convoluted implementation of `CstestCLI` contained all code related to this mode of operation. It was clear that this functionality did not belong in the `CstestCLI` class.

To remedy the situation, I factored this functionality out into a new class named `FileHandler`. As the name of the new class implies, `FileHandler` is responsible for all processing of command files, which are utilised by Cslibtest when it is executed in *command file mode*. As such, this class is only ever instantiated and used by `CstestController` when this mode of operation is in effect.

2.5.6 Outcome of `CstestCLI` Code Refactoring

With the excessive functionality of the original `CstestCLI` factored out into newly-created classes, `CstestCLI` was now much shorter and more focused in purpose than it had been when I first began working on Cslibtest. I rewrote the command-line interface code that remained, detangling the original web of methods into a tighter, more readily comprehensible structure. Code comments and Javadoc were added, clarifying the functionality of each method. Together, these changes resulted in a revitalised `CstestCLI` class, containing source code that was succinct, well-documented, and easy to understand.

The refactoring of `CstestCLI` had resulted in the creation of two new classes that had not existed previously. This new class structure is depicted in Figure 2.3, with *three classes* now capable of dispatching commands to `CommandHandler`, depending upon the mode of operation in effect. When this new diagram is contrasted with Figure 2.2 (page 23), it is readily apparent that the responsibilities that were initially concentrated within `CstestCLI` are now better distributed, with each class serving as a modular part of the whole, performing *one specific role*.

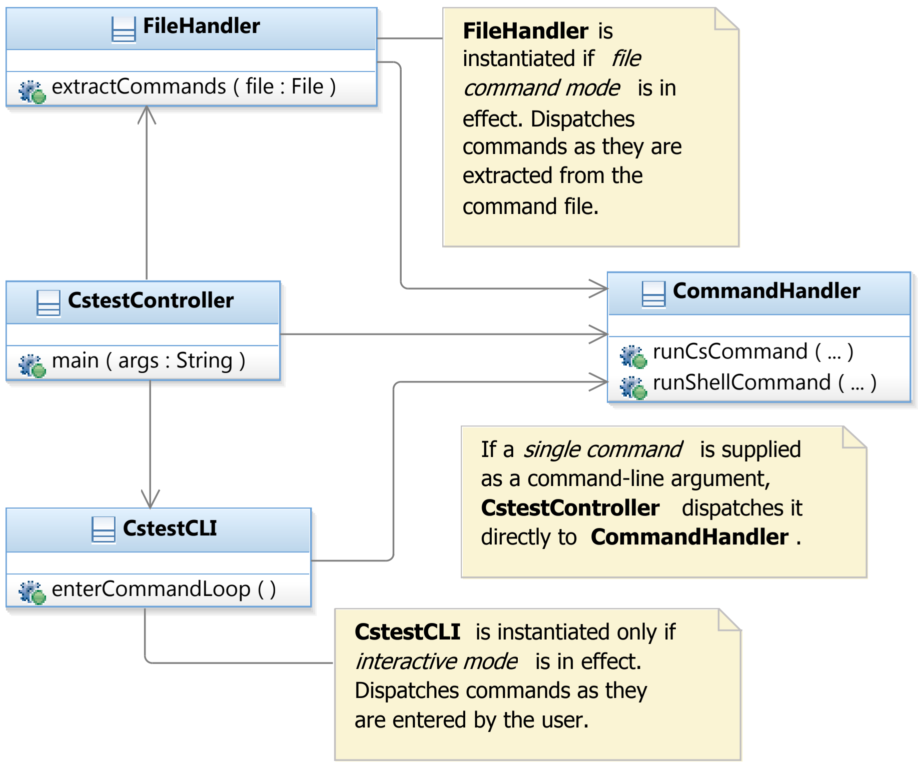

FileHandler
extractCommands (file : File)
FileHandler is instantiated if *file command mode* is in effect. Dispatches commands as they are extracted from the command file.
CstestController
main (args : String)
CommandHandler
runCsCommand (...)
runShellCommand (...)
If a *single command* is supplied as a command-line argument, CstestController dispatches it directly to CommandHandler .
CstestCLI
enterCommandLoop ( )
CstestCLI is instantiated only if *interactive mode* is in effect. Dispatches commands as they are entered by the user.

Figure 2.3: `CstestCLI` after refactoring; two new classes present

With refactoring of the `CstestCLI` class completed, I was ready to move on to implementation of command queue functionality to Cslibtest, as specified by the second requirement for my work on the application.

2.6 Addition of Command Queue Functionality

The refactoring of the `CstestCLI` class paved the way for my subsequent work on *new* Cslibtest functionality, by facilitating easier extension of the Cslibtest codebase. My next set of modifications involved the implementation of command queuing functionality, so that *interactive mode* users could begin entering commands while the network model connection was still being established, rather than having to wait until the connection process had completed.

As a first step towards carrying out these modifications, I examined the current internal representation of Cslibtest commands. I realised that there was room for improvement, and considered implementing a *new* internal command representation for the Cslibtest application.

2.6.1 Design Problem - How to Represent Commands?

In the original state of the application, Cslibtest commands were represented internally as one-dimensional *string arrays*. Each string array collectively represented one Cslibtest command, with each index of the array containing the individual arguments for the command. Although this approach was *functionally* adequate, I didn't consider this to be the most elegant possible representation of commands within Cslibtest from a *design* point of view.

I realised that a design pattern that I had studied during the CS607 module, the *command pattern*, could serve as an ideal template for a new, more object-oriented representation of commands in Cslibtest. Individual Cslibtest commands would now be represented as *objects* rather than strings, encapsulating the fine details associated with each command as easily-accessible fields. The command pattern provided a proven, pre-existing template for accomplishing this.

2.6.1.1 Solution: The Command Pattern

The command pattern[13] is classified as a *behavioural* design pattern, as it defines a manner in which communication between classes or entities can be controlled. This pattern enables all of the information required to carry out a request to be encapsulated within a single object. These command objects can then be passed around between methods and classes, and executed at an appropriate time. This structure is particularly useful when supporting activities that involve the execution of a *series* of commands. Since each command is represented as an object, it is straightforward to place these command objects within data structures, such as command *queues*.

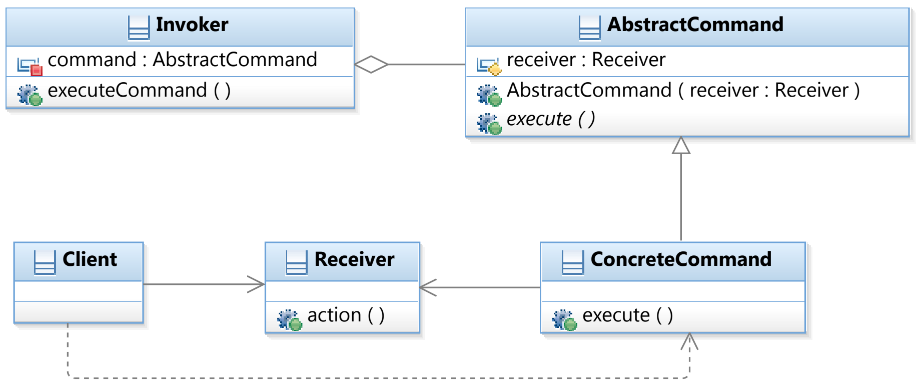


Figure 2.4: Command Pattern

Brief summaries of the role of each class in the command pattern template depicted in Figure 2.4 are as follows:

- The **Client** is responsible for creating individual command objects, and linking them to one or more **Receiver** objects.
- The **Receiver** object contains the methods that will be executed when command objects are invoked. All functionality required to carry out each command resides within this object. This promotes *loose coupling*[10] by keeping the functionality triggered by each command *separate* from the command definitions.
- The **AbstractCommand** class serves as an abstract superclass for all command objects, although it can also be implemented as an interface. It contains a reference to the receiver object, in addition to an abstract method, **execute()**, which will be called when commands are invoked.
- The **ConcreteCommand** classes are subclasses of **AbstractCommand**, which provide implementations for the **execute()** method. They contain all information that is required in order to execute the appropriate method contained within the **Receiver** object.
- The **Invoker** object invokes the command objects by calling their **execute()** methods.

The template provided by the command pattern represented an ideal solution to the design problem I was considering. Encapsulation of commands within dedicated objects would constitute an improvement to the design of Cslibtest as a whole, since it would remove the need for passing raw string data representative of Cslibtest commands between methods and classes. Furthermore, this design pattern also laid out a structure that was perfectly suited to the command queueing functionality that was to be implemented.

I will now describe each of the classes that collectively constitute my implementation of the command pattern within Cslibtest.

2.6.1.2 `CommandHandler` - The *Receiver* Class

In the context of my command pattern implementation for Cslibtest, the pre-existing `CommandHandler` class would function as the *receiver* class, as this class already contained the methods required to execute Cslibtest commands. No changes to the `CommandHandler` class were necessary in order for it to fulfil this role. However, all of the other classes involved in this implementation of the command pattern were created from scratch.

[10] *"Try to create modules that depend little on other modules. Make them detached, as business associates are, rather than attached, as Siamese twins are."* - Steve McConnell, *Code Complete: A Practical Handbook of Software Construction.*[18]

2.6.1.3 The `Command` Interface

This new interface defines a type that all concrete `Command` implementations belong to. An `execute()` method is defined by this interface, obligating all concrete `Command` types to implement this method.

2.6.1.4 The Concrete Command Classes

With the `Command` interface in place, the next step was to create some *concrete classes*. These classes include `CsCommand` and `ShellCommand`, differently named to signify the different execution procedure corresponding with the different command types.[11] These new classes are depicted in Figure 2.5.

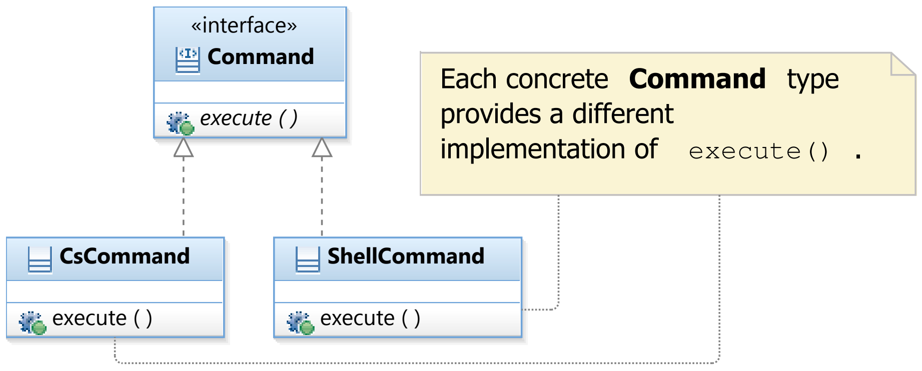

«interface»
Command
execute ( )
Each concrete Command type provides a different implementation of execute() .
CsCommand
execute ( )
ShellCommand
execute ( )

Figure 2.5: `Command` interface and concrete classes

2.6.1.5 `CommandQueue` - The *Invoker* Class

Finally, I created the `CommandQueue` class, as depicted in Figure 2.6. An instance of this class is now used to hold incoming Cslibtest commands in a queue at the beginning of Cslibtest's execution, when connection to a network model had not yet been fully established. Instead of `Command` objects being passed directly to `CommandHandler` to be executed, they are now routed through the new `CommandQueue`, within which they are stored in a `Queue` data structure.

When Cslibtest has finished connectiong to the network model(s) that have been specified at the command-line, `CstestController` makes contact with the `CommandQueue` to inform it that commands may now be executed. The `CommandQueue` then satisfies its role as the *invoker* class by calling the `execute()` method of each `Command` object held in the command queue. After all commands

[11]As was described in Section 2.5.1.1 (page 23), in addition to its main purpose of executing CS functionality upon a network model, Cslibtest also supports the execution of shell commands upon the underlying operating system. These two command types are now more intuitively distinguished from each other by the specific *implementation type* of the `Command` objects that encapsulate them.

had been executed, the `CommandQueue` is *disabled*[12], causing commands to be routed directly to the `CommandHandler` thereafter.

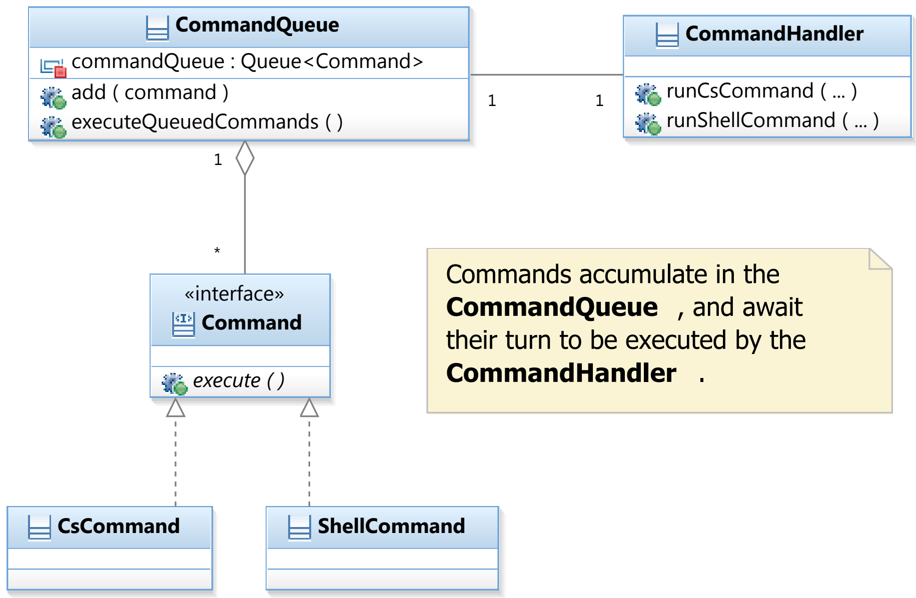

CommandQueue
commandQueue : Queue<Command>
add (command)
executeQueuedCommands ( )
CommandHandler
runCsCommand (...)
runShellCommand (...)
1
1
1
*
«interface»
Command
execute ( )
Commands accumulate in the CommandQueue , and await their turn to be executed by the CommandHandler .
CsCommand
ShellCommand

Figure 2.6: The new `CommandQueue` class, with concrete commands

With the implementation of the `CommandQueue` class as depicted in Figure 2.6, the basic infrastructure required for queuing commands within Cslibtest was now in place.

2.6.1.6 Conclusion & Next Steps

With command pattern implementations, all information required to execute a command is encapsulated within individual command objects. A key advantage of the command pattern is that these command objects can then easily be added to data structures such as queues - an advantage that my own implementation leveraged to good effect. Of course, the broad applicability of the command pattern is not limited to the particular scenario in which I put it to use.

Commands could also be placed into a stack data structure to implement undo functionality, for example. Utilising this data structure, the last command could then be undone simply by 'popping' the stack and reverting the actions of the command object that emerges. Numerous other situations involving the placement of commands within data structures exist that would stand to benefit from implementation of the command pattern; the situation encountered during

[12]When the connection process has completed, Cslibtest has no further need to delay commands from executing by placing them in the `CommandQueue`, as the network model(s) are now ready to have commands executed upon them.

my work on Cslibtest is just one of many that the command pattern would be well-suited to, due to the flexibility and wide applicability that this design pattern embodies.

Having integrated the command pattern into the design of Cslibtest, much of the required class infrastructure was now in place to carry out command queuing within Cslibtest. However, one final class was required in order to complete the command pattern implementation: the *client* class referred to by the command pattern template, which provides a mechanism for the *creation* of `Command` objects upon demand. This represented the final design problem to be solved in order to complete my work on Cslibtest's new command queuing support.

2.6.2 Design Problem - How to Create Commands?

It was apparent that depending upon the specific mode of operation in effect, different classes in Cslibtest would be required to set the creation of command objects in motion:

- When *interactive mode* is in effect, the `CstestCLI` class must trigger the creation of `Command` objects as the user typed them in.
- When *command file mode* is in effect, the `FileHandler` class must trigger the creation of `Command` objects for each individual command extracted from the command file.
- When *single command mode* is in effect, the `CstestController` class must trigger the creation of a `Command` object to encapsulate the single Cslibtest command received.

In order to trigger the creation of a `Command`, the above classes must access the *client* class.[13] At this point, the *client* class represented the only aspect of the command pattern implementation that had not yet been put in place. Upon consideration, I recognised that in this particular situation, the *factory method pattern* provided an ideal structure for this new class.

2.6.2.1 Solution: The Factory Method Pattern

The factory method pattern[9] is classified as a *creational* design pattern, as it defines a manner in which class instantiation can be controlled. This pattern constitutes a replacement for class constructors, abstracting the creation process so that the specific *type* of object created can be determined at run-time, depending upon the parameters passed into the factory method.

[13]To briefly recap, the *client* class in a command pattern implementation is responsible for the creation of `Command` objects.

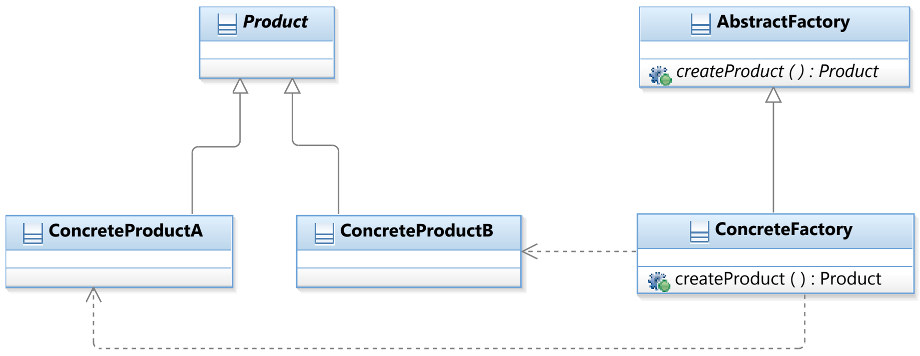


Figure 2.7: Factory Method Pattern

The roles of the classes and methods depicted in Figure 2.7 are as follows:

- The **AbstractFactory** class holds functionality common among all concrete factory classes.[14]
- The **ConcreteFactory** class implements the logic which is utilised to create and return new products.
- The **Product** class can be either an interface or abstract class which is implemented or inherited by all of the concrete products that the factory class can create. It defines aspects of functionality that are common to all of the concrete objects.
- The **ConcreteProductA** and **ConcreteProductB** classes, each with their own distinct functionality, represent the actual objects that are created and returned by the concrete factory class.

Upon consideration of the design problem at hand, I decided that only one concrete factory class was required in this case; for this reason, I did not need to create my own version of the **AbstractFactory** class specified above. Instead, I created a single concrete factory class: the `CommandFactory`.

2.6.2.2 The `CommandFactory` class

The new `CommandFactory` class, as depicted in Figure 2.8, is responsible for manufacturing `Command` objects for use within Cslibtest.

[14]For situations in which only one concrete factory class is required, this class may be omitted altogether.

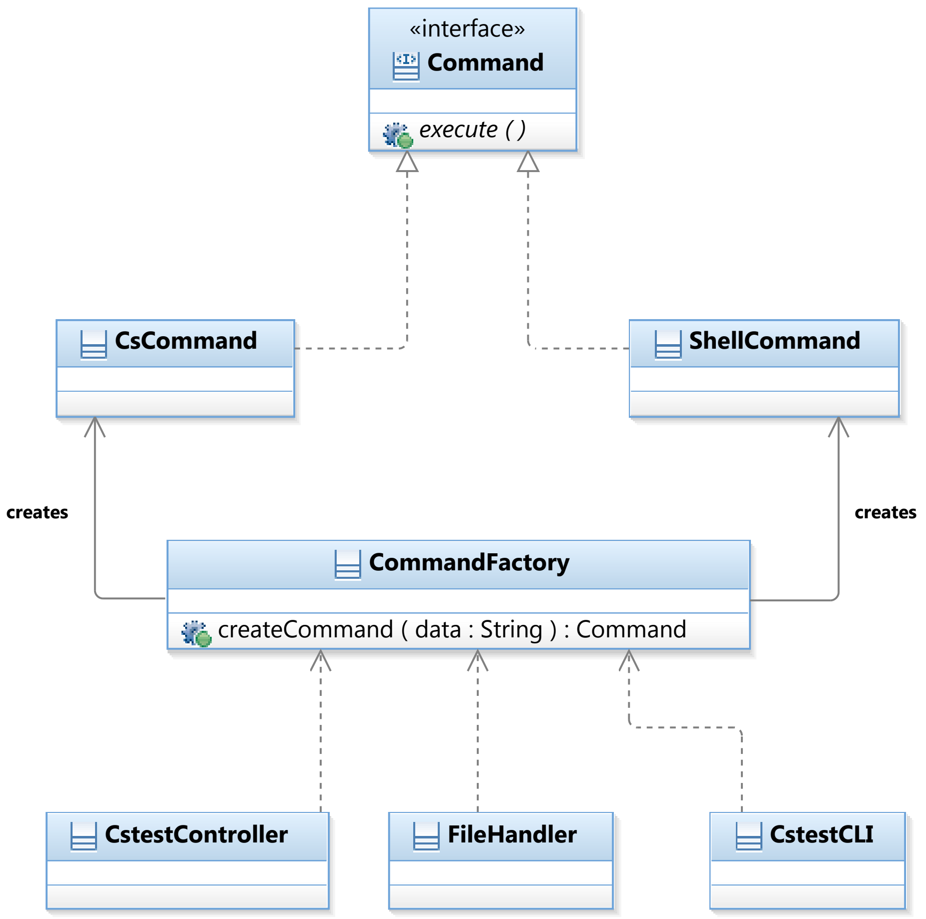


Figure 2.8: The new `CommandFactory` class

Raw command information that has been retrieved by `CstestController` (when operating in *single command mode*), `FileHandler` (when operating in *file command mode*), or `CstestCLI` (when operating in *interactive mode*) is now passed into the `createCommand` method supplied by the `CommandFactory`. This method determines the appropriate concrete command type to create (`CsCommand` or `ShellCommand`), based upon the contents of the raw command data which has been passed into the method.

Finally, a `Command` object of the appropriate type is returned, which neatly encapsulates all information associated with the Cslibtest command.

2.6.2.3 Conclusion

Application of the factory method pattern entailed the addition of a new class, the `CommandFactory`, dedicated to the creation of `Command` objects. This new class integrated smoothly into the Cslibtest implementation, providing functionality that is utilised by all three of the classes that were involved in my earlier refactoring work. Furthermore, as all code relating to the creation of `Command` objects is concentrated in the `CommandFactory` class, unnecessary code duplication is avoided.

The factory method pattern allows for a great deal of flexibility when creating objects. Rather than returning one class type only, a factory method can return a command object of a dynamically chosen type, depending upon the contents of the request passed into the factory method; in this regard, Cslibtest clearly benefits from the flexibility that this design pattern affords, as the two types of command that Cslibtest supports (CS commands, and shell commands which are less frequently used) may now be clearly distinguished from each other. This approach is modular, intuitive, and takes full advantage of the inheritance capabilities offered by an object-oriented language such as Java.

2.6.3 Command Queue Support - Outcome

With the *command* and *factory method* design patterns having both been applied to the Cslibtest source code, the application now possessed the command queuing support that had been specified as one of the requirements for my work on the application. As well as adding new functionality to Cslibtest, use of these patterns contributed new classes to the Cslibtest codebase that are modular, loosely coupled, and easy to understand. These characteristics will contribute favourably to the future maintainability and extensibility of the application.

With the first two requirements having been satisfied, one task remained to be completed: addition of support for multiple simultaneous network model connections. I will now describe the work done to complete this final task.

2.7 Addition of Multiple Model Support

As specified in the final requirement given to me for my work on Cslibtest, the application was required to support simultaneous connections to multiple network models at once, in order to improve upon the single connection support that had existed in Cslibtest up to this point. In order to implement this support, I had to make numerous modifications to several classes across the Cslibtest source code, in addition to developing some entirely new classes.

Upon examination of the existing source code, I found that the code responsible for establishing a connection to a network model at the beginning of Cslibtest's execution was buried inside a class which performed numerous other duties as well. This situation was reminiscent of the previous structure of the formerly problematic `CstestCLI` class; the network model connection functionality did not really belong where it was currently situated.

As my first step towards implementing multiple network model support, I decided to refactor the source code responsible for establishing individual network model connections, by moving this code into a **new class** dedicated solely to the task of establishing network model connections. I will now describe the operation of this new class.

2.7.1 The `Connection` class

The new `Connection` class serves as the only location in Cslibtest in which connections to network models are established. Each instance of `Connection` is dedicated to carrying out a connection process between the Cslibtest application and *one* specific network model. Collectively, *multiple* instances of `Connection` can establish multiple network model connections.

In practise, the connection process to each network model can take a varying length of time to complete. I realised that utilisation of multiple threads to execute all of the `Connection` processes in parallel would allow the connection process to proceed as efficiently as possible; the entire process would take considerably longer if these processes executed sequentially, rather than concurrently as multithreaded execution would enable. Therefore, I decided that each `Connection` instance would operate upon a single thread. Implementing Java's `Runnable` interface, each `Connection` instance is instantiated as a new `Thread` of execution at runtime.

After the `Connection` instances corresponding with each network model have been instantiated, connections with each model are established one by one, until *all* connections have been established. While this process is taking place, any Cslibtest commands entered by the user (if Cslibtest is operating in *interactive mode*) are placed in the `CommandQueue`, taking advantage of the command queuing support that I described previously.

Eventually, when *all* connection processes have completed, the `CommandQueue` is signalled by `CstestController` to execute all waiting commands.

However, a design problem became apparent. With several `Connection` threads executing at once, `CstestController` needs to be kept informed of their progress in order to determine when all `Connection` processes had resolved, since the `Command` objects held within the `CommandQueue` cannot be safely executed until all network model connections have been established.

2.7.2 Design Problem - How to Inform `CstestController`?

In order for `CstestController` to signal `CommandQueue` to execute the commands contained within its command queue, it needs to be kept *informed* of the current status of each `Connection` instance. As each single-threaded `Connection` instance completes its work, it must signal `CstestController` somehow, to inform it of the newly-established connection. In essence, the `CstestController` has to *observe* the progress of the `Connection` instances.

After some consideration, I realised that this situation seemed familiar. I had studied a design pattern during CS607, the *observer pattern*, which was intended for addressing design problems such as the one that I had just encountered. I will now describe the template for this pattern.

2.7.2.1 Solution: The Observer Pattern

The observer pattern[14] is classified as a *behavioural* design pattern, as it defines a manner in which communication between classes or entities can be controlled. This pattern enables objects, known as *subjects*, to publish changes to their state as these changes occur. Other objects, referred to as *observers*, can subscribe to the subjects so that they are immediately made aware of any changes to the state of the subjects.

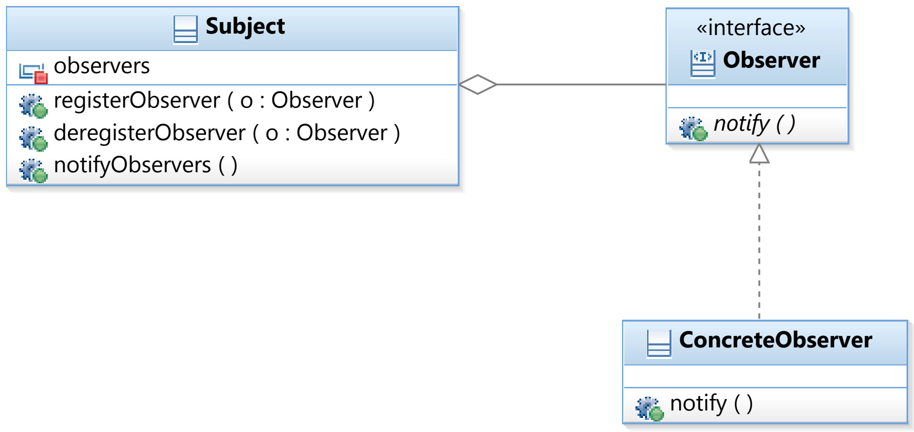


Figure 2.9: Observer Pattern

The roles of each class in the observer pattern template depicted in Figure 2.9 are as follows:

- The **Subject** represents a class which can be *observed*. When the **Subject** changes state, it uses the **notifyObservers()** method to inform all registered observers of the state change. To achieve this, the method loops through all registered observers, calling their **notify()** methods.

- The **Observer** interface obligates all classes that inherit this interface to provide a **notify()** method. This allows **Subject** instances to which the observers have registered to notify the observers when state changes occur.
- The **ConcreteObserver** objects are the *observers* that depend upon knowledge of the current state of the **Subject**. They provide their own implementations of the **notify()** method; as this method will be called when subjects change their state, the contents of this method will enable the observer to take appropriate action at that time.

Given the nature of the design problem that I was facing, an implementation of the observer pattern appeared to represent an ideal solution. I will now provide a detailed description of my solution to the design problem at hand, which was implemented using the observer design pattern as a template.

2.7.2.2 Application of the Observer Pattern

In the context of my observer pattern implementation, each `Connection` instance is considered a *subject*, while the single `CstestController` instance present within Cslibtest throughout its execution is subscribed as an *observer* to each of the `Connection` *subjects*.

One of the responsibilities of the `CstestController` class, as was previously discussed in Section 2.5.4 (page 26), is setting the network model connection process in motion when Cslibtest is executed. In order to carry out this responsibility, it creates `Connection` objects corresponding with the network models that have been specified at the command-line.

Upon creation of these `Connection` objects, `CstestController` also stores a list of the network models that have `Connection` objects allocated to them. This list of *pending connections* is altered by `CstestController` as the `Connection` instances announce changes in their state.

As was depicted in Figure 2.9, all *observers* must implement an interface containing a **notify()** method, so that *subjects* can call this method in order to announce changes to their state. To this end, I created my own version of this interface named `ConnectionEventListener`. This interface provides a method named `handleConnectionEvent`, which fills the role of the **notify()** method depicted in the observer pattern template.

As `CstestController` is an *observer* class in the context of the observer pattern, it now implements the `ConnectionEventListener` interface, which obligates it to provide an implementation for the `handleConnectionEvent()` method. The full class layout is depicted in Figure 2.10.

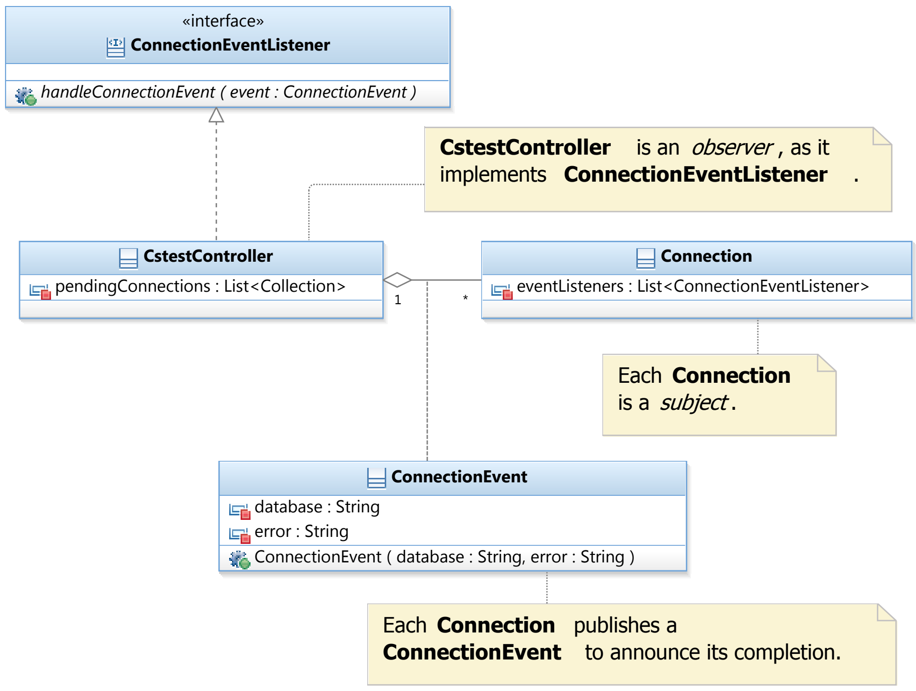

«interface»
ConnectionEventListener
handleConnectionEvent (event : ConnectionEvent)
CstestController is an observer, as it implements ConnectionEventListener.
CstestController
pendingConnections : List<Collection>
1
*
Connection
eventListeners : List<ConnectionEventListener>
Each Connection is a subject.
ConnectionEvent
database : String
error : String
ConnectionEvent (database : String, error : String)
Each Connection publishes a ConnectionEvent to announce its completion.

Figure 2.10: Observer pattern in action

I also created a simple container class called `ConnectionEvent`, which is used to contain data sent from each *subject* to the *observer*. Thus, instances of `ConnectionEvent` are utilised to provide the fine details of the `Connection` state changes to the `CstestController` observer. The two fields within each `ConnectionEvent` allow them to convey:

- The name of the network model that a particular `Connection` *subject* attempted to connect to.
- An error string. If this string is empty, this indicates that the `Connection` established a network model connection successfully. If the string is not empty, this indicates that the `Connection` instance failed to establish a network model connection, and provides a descriptive reason for the failure.

As each `Connection` instance finishes connecting to the network model associated with it, it *notifies* `CstestController` by passing a `ConnectionEvent` object into the `handleConnectionEvent()` method which is implemented by the `CstestController` class. If the network model connection was successfully established, the network model name contained within the `ConnectionEvent` is then removed from the list of *pending connections* that is stored within `CstestController`.

Eventually, when the last remaining `Connection` instance has finished connecting to its allocated network model and has dispatched a `ConnectionEvent` object to announce this, `CstestController` removes the network model mentioned in the `ConnectionEvent` from its list of *pending connections*. The list of *pending connections* is now empty, which is confirmed by a quick check at the end of the `handleConnectionEvent()` method. After this confirmation, `CstestController` signals the `CommandQueue` to inform it that all queued commands can now be executed.

With this new class structure in place which embodies the observer design pattern, Cslibtest is now capable of connecting to multiple network models simultaneously.

2.7.2.3 Conclusion

My implementation of the observer pattern provided an elegant solution to the final substantial design problem that I encountered during my work on the Cslibtest application. As this pattern allows a *one-to-many dependency* to be defined between objects, it proved to be excellently suited to 'scaling up' the functionality of Cslibtest to handle multiple simultaneous network model connections instead of just one.

Of course, this pattern would be equally applicable in many other situations in which a similar 'scaling up' of functionality was desired for a particular application. This pattern is also commonly utilised in conjunction with graphical user interfaces, with the GUI serving as the *observer* and the underlying state of the application serving as the *subject*.

2.7.3 Multiple Model Support - Outcome

With multiple network model support for Cslibtest now in place, all requirements that had been given to me at the start of my work on the Cslibtest application had been satisfied. With support for command queuing integrated into the Cslibtest codebase as well, and considerable refactoring having been performed upon the original source code, my development work on Cslibtest was complete.

2.8 Code Review

During my work on Cslibtest, my code was reviewed on several occasions during development by other members of the team. These code reviews were among the first that I experienced at Ericsson, as Cslibtest was one of my earlier projects.

These reviews ranged from quick 'deskchecks'[15] to longer reviews which took place in a meeting room, with a projector used to provide several developers in the room with a common view of the source code being examined.

The features of Cslibtest that I had written at the point of review were typically fully functional; the review process did not catch any major bugs. However, the developers performing the code reviews suggested several *structural* improvements to the code. Examples include:

- In a very early iteration of my additions to the Cslibtest codebase, the `Connection` class existed as a *nested class* of `CstestController`. This arrangement was fully functional, but upon examination, it was suggested by another team member that the source code would be more comprehensible if `Connection` was made into a fully separate class instead. I agreed, and the change was made.
- One of my colleagues noticed that a single variable in the `CommandQueue` class was redundant, and could be removed completely if I made a small change to the logic of the class.[16]
- An early version of the `CstestController` class only checked the validity of the command-line arguments at the point at which they were used. Another developer suggested that if the validity of these arguments was checked in the constructor, this would simplify the structure of the class by ensuring that the constructor would only finish successfully if all arguments were valid.

Since Cslibtest was my first substantial project at Ericsson, I was relatively inexperienced when these first code reviews were performed. Through these code reviews, I gradually accumulated knowledge from my more experienced colleagues. On account of this, subsequent code reviews tended to prompt fewer suggestions from my colleagues, as the quality of my code was steadily improving.

After implementing the various minor suggestions that my colleagues gave me, they were satisfied with my implementation and approved the product for delivery.

[15]A deskcheck simply involves a software developer examining another developer's source code from their own computer. Since ClearCase allows shared access to source code files as described earlier (page 9), deskchecks may be carried out conveniently at any time, provided that the latest source code changes have been checked in.

[16]Precise observations such as this are a testament to the rigour employed by my colleagues during code reviews!

2.9 Project Outcome

Cslibtest represents one of the larger applications that I have worked on at Ericsson to date. My work on the application included a significant amount of code refactoring, with functionality being relocated from an oversized class into multiple new classes, with each new class purpose-built to carry out a single set of tightly related functions. I *refactored* Cslibtest's internal treatment of commands to a series of classes that together formed an implementation of the *command pattern.* Other new classes that I created were integrated into implementations of the *factory method* and *observer* design patterns.

This project represented the first time in which I had made substantial use of design patterns during development of a commercial software product. Prior study of these patterns during the CS607 module proved to be very useful; since I had been exposed to these patterns previously, it didn't take long to recognise their applicability to design problems that I faced. My experience of utilising design patterns during my work on Cslibtest confirmed the assertions made during the CS607 module (and in software engineering literature) that design patterns represent *broadly applicable* solutions to many design problems which are frequently encountered during software development.

With the requirements having been satisfied and my work on Cslibtest now complete, a new version of the application was deployed soon afterwards. This version, containing the new functionality that I described throughout this chapter, is now in use throughout Ericsson. In the future, developers who wish to extend or maintain the Cslibtest source code will benefit from the clearer, more focused class structure which was introduced to the Cslibtest codebase through refactoring. In the present, end-users will benefit both from the greater usability facilitated by pre-connection command queuing, and the greater productivity enabled by the newly-implemented support for multiple simultaneous network model connections.

Chapter 3

CS Metadata Migration Tool

Some time after I finished my work on Cslibtest, I began working on a new software project, the *CS Metadata Migration Tool* (CSMMT). This project proved to be larger in scope than Cslibtest, offering a wide range of new design problems to be solved. As this new software application was to be developed from scratch, code refactoring of pre-existing 'legacy' code was not be required as in Cslibtest; however, use of design patterns was even more prominent. Furthermore, my knowledge of software testing techniques was exercised, as my responsibilities included construction of an automated test suite to accompany the new application. In this chapter, I will describe in detail my work on CSMMT.

3.1 Chapter Structure

In **Section 3.2 (*Motivation*)**, I will provide some background information about the events that led to the development of CSMMT.

In **Section 3.3 (*Requirements*)**, I will list and explain the requirements that were specified for the CSMMT application.

In **Section 3.4 (*Analysis of CSMMT Main Task*)**, I will step through my high-level analysis of the requirements and the functionality required to implement them. This coverage of high-level design decisions undertaken early in the project will pave the way for subsequent discussion of the CSMMT implementation in the following section.

In **Section 3.5 (*Implementation & Use of Design Patterns*)**, I will discuss the development of CSMMT in detail, with extensive reference to specific design patterns that were utilised during the development process in order to solve numerous design problems that were encountered.

In **Section 3.6 (*Automated Testing of CSMMT*)**, having previously described the key aspects of the CSMMT implementation, I will describe the automated test suite that I developed to accompany it, which resulted from my practical application of software testing techniques covered during the CS608 module.

Finally, in **Section 3.7 (*Project Outcome*)**, I will briefly discuss the outcome of the CSMMT project.

3.2 Motivation

As was previously described in Section 1.1.1 (page 2), the CS facilitates interaction with MOs contained within telecommunications network models, which reside in object databases. Several hundred distinct MO types will be typically be supported by each of these models, with up to several million MO instances held within each model. Each different MO type is represented by a distinct Java class, each of which inherits from an abstract superclass in the CS (hereafter referred to as `ManagedObject`) common to all MO types.

Crucially, some important *CS metadata*[1] is specified within this abstract superclass, in the form of several `private` fields. These fields are *always* populated in *all* instances of *all* MO types.

In the weeks prior to the commencement of my work on CSMMT, various modifications had been implemented by the CS Design Team across the CS codebase, with the goal of improving overall performance of the system.[2] As part of these modifications, the structure of the `ManagedObject` class within the CS was significantly overhauled. This involved *extensive changes* to the data structures within `ManagedObject`, which contain the crucial metadata required for correct functionality of the CS.

The altered data structures would yield *faster performance* during normal

[1]Specific details pertaining to this CS metadata and related data structures have been omitted for confidentiality reasons. In terms of understanding the operation of CSMMT, it is sufficient to know that this metadata *must* be accurately specified within each MO, in order for the CS to function properly.

[2]Between my work on Cslibtest and CSMMT, I was involved in carrying out some of this work upon the CS in conjunction with other CS Design Team members. However, as the Cslibtest and CSMMT projects were more substantial as a whole and presented more challenging design problems, they represented more interesting dissertation topics.

CS operation. However, in order for these performance benefits to be realised, the new data structures would have to be propagated to *all network models* in use, both within Ericsson and by Ericsson customers. This substantial task called for the creation of a new tool.

3.2.1 New Application Required: The CSMMT

In order for the network models to benefit from the performance improvements that the new data structures within `ManagedObject` would provide, a large update would have to be performed upon each network model individually. For *every* MO instance in a particular network model:

1. The new data structures (initially empty) must be put in place in the `ManagedObject` class, as depicted in Figure 3.1.

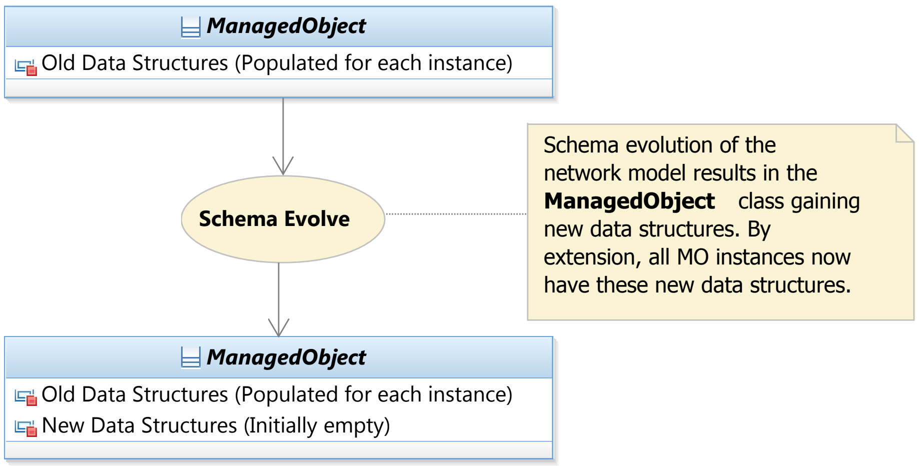

ManagedObject
Old Data Structures (Populated for each instance)
Schema Evolve
Schema evolution of the network model results in the ManagedObject class gaining new data structures. By extension, all MO instances now have these new data structures.
ManagedObject
Old Data Structures (Populated for each instance)
New Data Structures (Initially empty)

Figure 3.1: Schema evolution introduces new data structures to all MOs

2. CS metadata must then be *migrated*[3] from the old data structures to the new. This process must be performed on every MO in the network model, as depicted in Figure 3.2.

[3]This migration is not a simple 'copy and paste' affair, as the format of the new data structures is considerably different to that of the old ones.

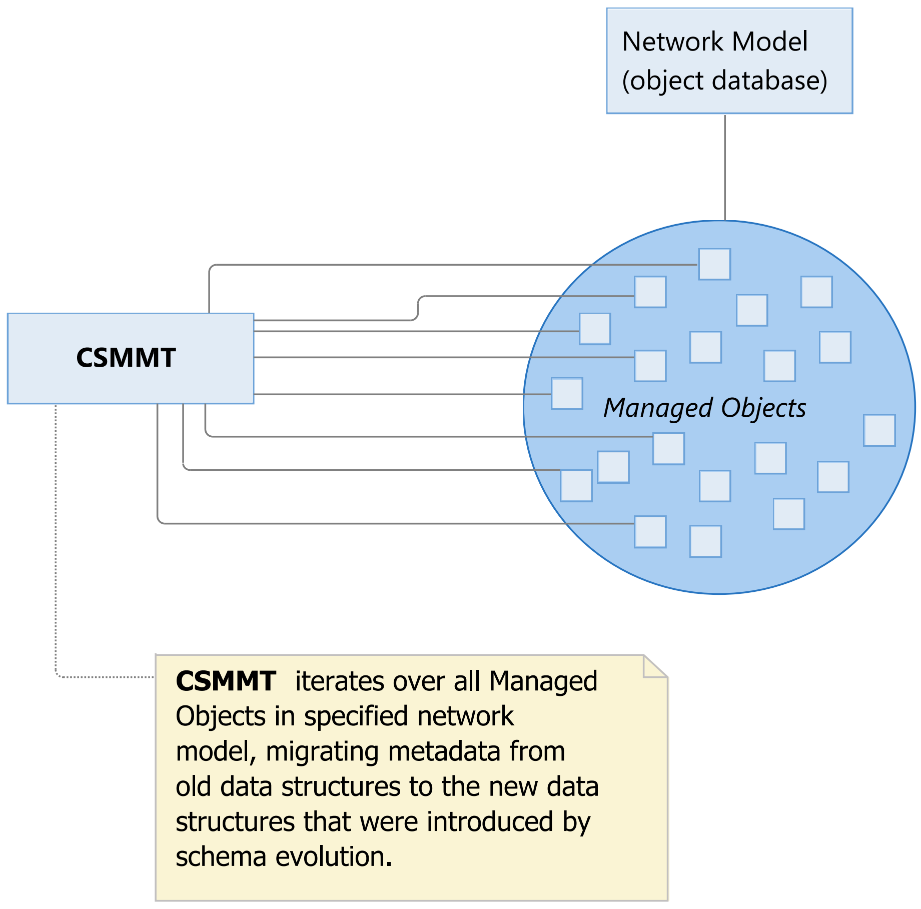

Network Model
(object database)
CSMMT
Managed Objects
CSMMT iterates over all Managed Objects in specified network model, migrating metadata from old data structures to the new data structures that were introduced by schema evolution.

Figure 3.2: CSMMT migrates metadata for all MO instances in the model

To enable delivery of these updates to individual network models, an entirely new software tool was required. This new application, the CSMMT, must perform a one-time update[4] of *all* of the managed objects residing within each network model.

3.3 Requirements

The requirements for the new tool were identified by two CS Design Team members who performed a preliminary analysis.[5] My colleagues discussed these

[4]The work carried out by CSMMT only needs to be carried out *once* on any particular network model, since the old data structures are left empty following the metadata migration, and will never be populated again.

[5]At this very early stage, I was not involved with the CSMMT project. My involvement began when the requirements were identified and passed on to me for further analysis and implementation.

requirements with me, and I assumed responsibility for development of CSMMT from that point forward.[6]

The key requirements can be summarised as follows:

1. The *schema* of the network model to be updated by CSMMT must be *evolved* prior to any other action taking place. This *schema evolution* updates the class structure, introducing new data structures into the `ManagedObject` abstract superclass, and by extension, to every MO instance in the network model.[7]

2. As its *main task*, accounting for the vast bulk of code written and most of CSMMT's execution time, the application is required to perform four distinct updates[8] upon *every* individual MO instance in a given network model. Each of these updates must convert and migrate the metadata contained within a particular set of old data structures into the new data structures which were introduced by the previous *schema evolution*.

3. In practise, customer network models are only taken down for limited periods of time for maintenance tasks to be carried out. Consequently, a *time constraint* was imposed upon the time required for CSMMT to execute upon a typical customer network model. The application must operate efficiently in order to finish executing on time.

This concludes my outline of the CSMMT requirements that were provided to me at the beginning of the project. I will now describe the analysis that I performed after receiving these requirements, with a view to determining the broad structure of the CSMMT application and the exact approach to be taken towards tackling the large-scale network model updates that CSMMT was required to perform.

3.4 Analysis of the CSMMT Main Task

After being given the requirements for the new CSMMT application, the next step was to determine how CSMMT's main task would be carried out - the mass update of a large number of MOs within a manageable time span. While

[6]As I was writing this application, other team members worked in parallel upon areas of core CS functionality that were impacted by the new data structures placed in `ManagedObject`.

[7]This requirement was trivial to satisfy, as existing tools offered this schema evolution functionality already. The focus of this chapter will be upon CSMMT's main functionality; migration of the metadata from the old data structures to the new ones.

[8]For confidentiality reasons I cannot delve into detail regarding the exact nature of each of these updates, but for the purpose of understanding CSMMT's operation, it is sufficient to know that *four* sets of metadata must be migrated by CSMMT within each MO instance - hence the need for four separate updates.

performing my analysis, I frequently consulted with my colleagues to explain my ideas and solicit their opinions. The decisions made during analysis would play a major role in shaping the implementation of CSMMT. I will now explain these decisions, and the process of reasoning that led to them.

3.4.1 Multithreaded Execution & MO Updater Objects

Since a tight time limit per execution of the application was explicitly specified in the requirements, it was clear that performance of the application would be a major factor to be borne in mind during the design process.

Computer hardware has improved greatly in its capacity for true parallel operations in recent years, with an increasing trend towards increasing numbers of cores rather than increasing single core speeds.[26] As a result, multiple threads of execution have become more commonly utilised in applications with high performance requirements, in order to take full advantage of the power that multiple CPU cores offer. As typical customer machines would possess several CPU cores that could be fully utilised during the downtime imposed for network model updates, I recognised that efficient use of multiple threads would be a critical performance factor for this application.

Accordingly, I decided to design CSMMT around multiple threads of execution, each of which would update only a subset of the MO instances present in the model. As I would be developing CSMMT in Java, an object-oriented language, it seemed logical that individual objects occupy each thread of execution. Each of these *MO updater objects* would be responsible for updating a particular subset of the MO instances contained within the network model. Since each MO updater would be tackling only a subset of the MO instances, and would be executing upon its own thread, this would enable the work of updating *all* MO instances to be paralleled across multiple threads of execution.

By splitting up the work into multiple threads of execution which could then execute in parallel, CSMMT would be able to take full advantage of all available processor cores. Multiple MO updaters would access the network model concurrently, performing their updates at the same time as other active MO updaters.

With millions of MO instances residing in a typical network model, the work of updating these MO instances would have to be cleanly divided among the individual MO updater threads in order for this multithreaded approach to work. With the basic details of the approach set out, I moved on to consider how best to divide CSMMT's work among multiple MO updaters.

3.4.2 Division of Work Among MO Updaters

In order to divide the work among multiple threads, it seemed reasonable to make each MO updater object responsible for updating the instances of *one specific MO type* only. Each individual collection of MOs comprised of a single type could then be updated concurrently by separate MO updaters, without any interference between them, since they would be operating on entirely different collections of MOs within the network model.

However, as stated during the introductory chapter (page 2), the MO instances in a typical customer network model are collectively comprised of hundreds of different MO types. On account of this, it was apparent that a limitation upon the number of MO updater objects which could execute at any one time would have to be imposed. This would serve to restrict the number of simultaneously active threads to the maximum number that could be efficiently handled by the CPU cores available on a given machine, while not placing too great a burden on memory.

After consideration and discussion with other members of the CS Design Team, it was apparent that a *queue* data structure would be a good solution. A limited number of thread 'slots' would be made available, and all MO updaters would be placed in a queue to acquire one. The first few MO updaters in the queue would occupy the available threads of execution, and the remainder would have to wait for their turn to execute. When a thread of execution freed up as an MO updater finished updating the MO instances assigned to it, the next MO updater in the queue could then step in, occupying the thread and carrying out its own updates. Eventually, all MO updaters would complete their work, the queue would be empty, and CSMMT's main task would be complete.

In order to implement this system of allowing multithreaded execution up to a clearly defined point, a limit would have to be defined[9] upon the number of MO updater objects which would be allowed to execute at any one time.

3.4.3 Manual Thread Management vs. Threadpools

From a design perspective, there were numerous approaches available which would enable to me to implement this system of 'queuable' threads. One option would be to perform extensive manual management of `Thread` objects, in order to implement the functionality previously described. However, as of Java version 1.5, the `java.util.concurrent` package has been a part of the language. Among other things, this package allows the use of `ExecutorServices` which can be used to create thread pools. As it is generally considered better

[9]To allow for different hardware configurations, this limit would be configurable via a Java system property, which could be specified on the command-line upon execution of CSMMT.

programming practise to utilise thread pools instead of performing extensive manual management of individual threads of execution[3], I opted to implement an `ExecutorService` thread pool[23] of fixed size.

The thread pool would allow a fixed number of MO updaters to execute in parallel at any one time. The remaining threads would wait in a queue until one of the active threads in the threadpool completed, at which point the next thread in the queue would become active. All of this work would be performed 'under the hood' by the `ExecutorService`. This hidden complexity facilitated by the `java.util.concurrent` package would provide the functionality that CSMMT required, while helping to keep the source code succinct and comprehensible by freeing it of manual thread management.[10]

I moved on to another consideration: how would CSMMT know which MO types resided within a particular network model? Across different network models, the MO types utilised to simulate each network element can vary. This information would have to be known in order for MO updaters to be properly allocated to *specific* MO types.

3.4.4 Schema Processing

I was initially unsure of how to obtain a definitive list of MO types residing within a particular network model. I pointed out this issue to a colleague, who suggested that I look into utilising the *schema definition file*. Every network model is associated with a file of this type, which specifies a list of MO types in XML notation[11] that are allowed to reside within the network model.

Through some processing upon this schema definition file, a definitive list of MO types present in the associated network model could be obtained for use by CSMMT. Since this processing would be an essential prerequisite for dividing up work between MO updater threads, it was apparent that this schema processing would have to occur *at the very start* of CSMMT's execution, before any creation of MO updater objects.

At this point, the beginnings of a conceptual approach for CSMMT's main task had been mapped out. Processing of the schema definition file would yield a list of MO types. One single-threaded MO updater would be allocated to each MO type, allowing CSMMT to update several MO types in parallel by placing the MO updater objects in a thread pool. After all MO updaters had finished their work, CSMMT's main task would be done.

[10]As described previously in Section 2.7.1, I performed some manual management of `Thread` objects during my work on Cslibtest. The multithreaded aspect of Cslibtest was considerably smaller and simpler than the multithreaded architecture that CSMMT was designed around. For this reason, manual thread management was a viable option for that project.

[11]Reading the contents of these files involved some substantial processing, which prompted the use of a design pattern. This will be discussed on page 58.

However, I soon received some data analysis results which had been derived from a customer network model.[12] Upon examination of this data, I realised that the conceptual approach as it stood could suffer from decreased efficiency under certain circumstances. Given the importance of maintaining a high level of efficiency throughout CSMMT's execution, this issue would have to be addressed before I could proceed further.

3.4.5 Problem Identified: 'Outlier' MO Types

The data analysis results that I received consisted of a detailed breakdown of all MO types which were present within the customer network model, including the exact number of instances for each MO type. Upon examination of the data, I realised that a handful of MO types had *disproportionately large numbers of instances* in the model by comparison to the other MO types. One MO type in particular had an enormous number of instances relative to the others.

My conceptual approach in its current state could run into difficulty if any such *outlier MO types*[13] existed in the network model being updated. After all MO updaters associated with non-outlier MO types had completed their work, a single MO updater could still be working on the enormous number of instances associated with the outlier MO type. This would mean that CSMMT could be operating on a single thread of execution for a significant portion of its execution time, while this last MO updater finished its work.

This 'worst case scenario' would result in several CPU cores being left idle, while a single CPU core did all the work of updating the instances of the outlier MO type. Of course, this would be a very undesirable situation in terms of efficiency. I began considering how I could adapt my initial approach to maintain efficient *concurrent* execution throughout, even with the presence of outlier MO types.

3.4.6 Evaluation of Possible Solutions

Despite the efficiency concern that I had identified, my initial conceptual approach remained perfectly adequate for *almost* all MO types. Problems would only occur when outlier MO types were subjected to this same approach, as well. I realised that perhaps the best course of action would be to stick with the previously described solution for most MO types, with some special treatment

[12]This network model could be considered a representative sample of the type of network model that CSMMT would typically be executed upon.

[13]Technically, a MO type with relatively *few* instances could also by definition be considered an 'outlier'. However, such an MO type would not pose any obstacle to efficient execution of CSMMT, and is therefore not of special interest. During this chapter, I will use the word 'outlier' to refer *exclusively* to MO types that have considerably *more* instances than average.

added for outlier MO types only.

I realised that there were multiple different refinements to the original approach that could potentially be implemented, in order to apply this special treatment to outlier MO types. I will now discuss *two* of the refinements that were considered.

3.4.6.1 Solution #1: Outlier MO Types Prioritised

Initially, I considered the possibility of manipulating the queue of MO updaters waiting to enter an available thread slot in the threadpool; perhaps by changing the order in which they were queued, thread utilisation could be increased. I will describe this concept with the aid of a simple example, depicted in diagrams. Figure 3.3 illustrates a highly simplified example situation: CSMMT is being executed upon a network model containing *four* MO types[14], with the maximum number of active threads in the threadpool set to *two*[15]. Of the 4 MO types depicted, one of them is an outlier MO type, clearly identifiable in the diagram by its greater number of instances. Each 'block' in the stacked bar chart represents the execution of a distinct MO updater object. Note that queued MO updaters can only begin their work when a space frees up in the threadpool, allowing a different MO updater to begin updating MO instances.

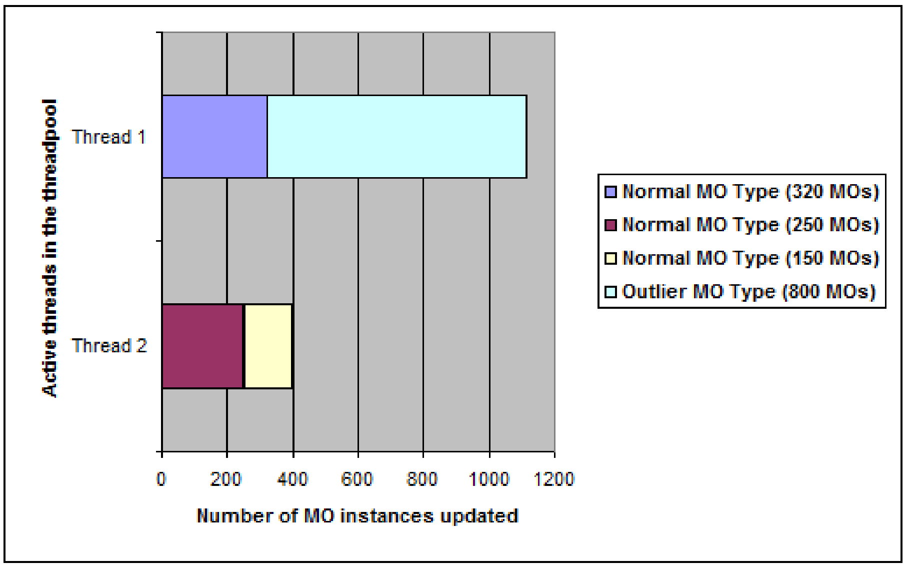


Figure 3.3: Ineffective use of multiple threads

In this example, the single MO updaters which have been allocated to each

[14]In practise, hundreds of MO types will be present within a given network model.
[15]In more realistic usage scenarios, CSMMT would typically execute with at least six active threads of execution.

of the four MO types have been added to the threadpool queue in ’natural’ order - no measures have been taken to fine-tune which MO updaters were permitted to join the queue first. As is apparent in the diagram, Thread 1 ends up having to update far more instances than Thread 2 does; and as soon as Thread 2 is finished and Thread 1 is left working, CSMMT becomes a single-threaded application. This is not a very effective use of the two available threads.

However, what if MO updaters assigned to outlier MO types were *prioritised* and moved to the front of the queue, so that they would be the first to enter an available thread ’slot’ in the threadpool? With all other elements identical to the first example, Figure 3.4 depicts the result of this queue manipulation. By allowing the MO updater associated with an outlier MO type to begin its work first, utilisation of the two available threads has been greatly improved. This strategy would clearly improve performance in many situations.

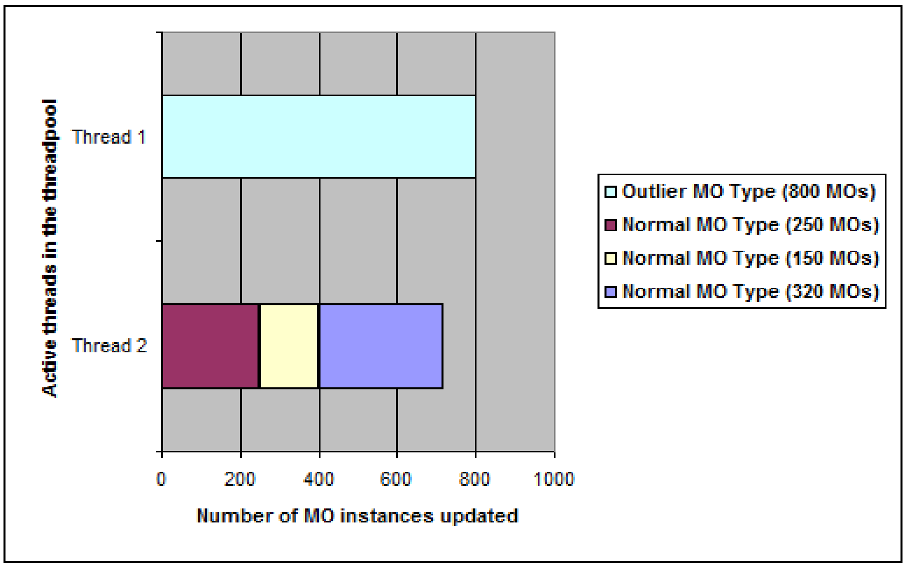


Figure 3.4: Manipulation of the MO updater queue improves performance

However, it quickly became apparent that this solution was inherently limited in terms of flexibility. Although it often presented an improvement over arbitrary queuing of MO updaters, a lot of CPU time could still be wasted in certain situations. For instance, the situation depicted in Figure 3.4 changes greatly when a single extra thread of execution is made available, as is shown in Figure 3.5.

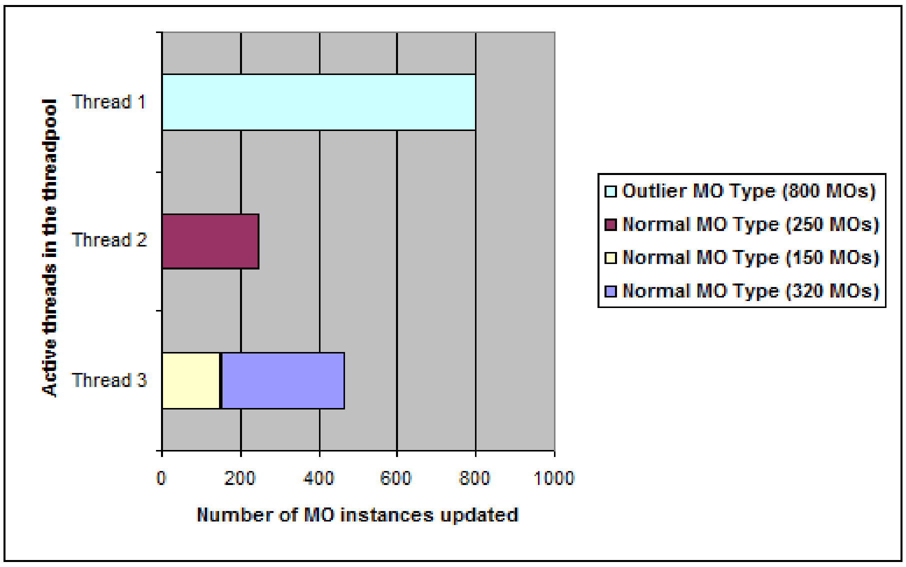


Figure 3.5: Very ineffective use of multiple threads

As depicted above, the queue manipulation strategy has fallen short. Regardless of how the MO updater queue is manipulated in this situation, two of the threads are under-utilised for a significant proportion of the application's execution.

Upon discussion with my colleagues, we agreed that this type of approach was not feasible, and it was dropped from consideration. I began considering another more sophisticated approach. It was apparent that the work of updating the outlier MO types would have to be *broken up*, if effective thread utilisation was to be attained.

3.4.6.2 Solution #2: Multiple MO Updaters for Outlier MO Types

After consideration as to how to break up the work of updating the instances of outlier MO types, I arrived upon a possible solution. With this solution, the outlier MO types would still be updated with MO updater objects; however, the nature of these MO updaters would be *different* to those that would be applied to 'regular' MO types. Instead of one MO updater being assigned to each outlier MO type, *several* MO updaters would be assigned instead. Each of these specialised MO updaters would update only a *limited subset* of the outlier MO type's instances, instead of all of them as with the standard approach applied to all other MO types.

These specialised MO updaters would each operate on a separate thread, just as the standard ones would. However, with multiple MO updaters applied to a single outlier MO type, multithreaded execution of instances *within a single*

MO type would be enabled. This further breakdown of work would combat the problems associated with the sheer number of instances that accompany outlier MO types.

The example depicted in Figure 3.5 depicted the shortcomings of the 'queue manipulation' approach. The same situation depicted in that diagram is shown again in Figure 3.6, with one difference: the work of updating the outlier MO type's instances has now been broken up among *six* specialised MO updaters.

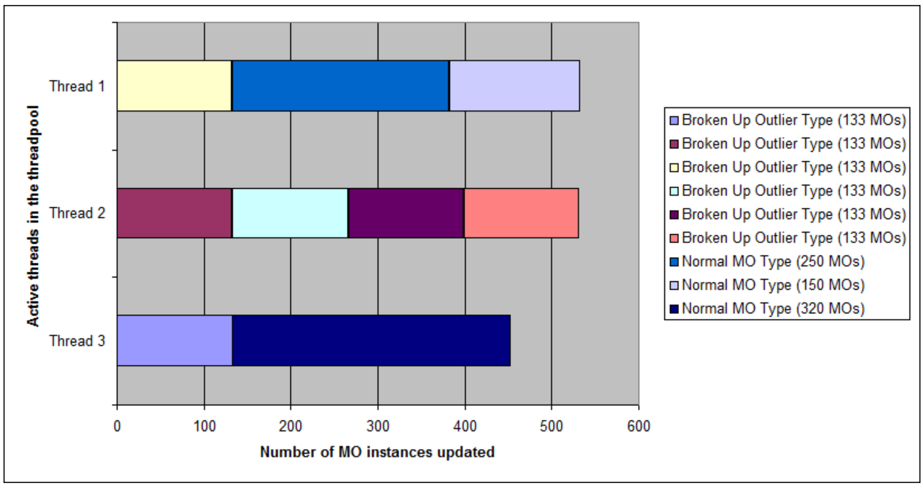


Figure 3.6: Outlier MO type is 'broken up' for superior thread utilisation

As shown in the diagram above, thread utilisation has greatly improved due to this division of work.[16] In practise, the number of specialised MO updaters applied to outlier MO types could be *configurable*, in order to achieve optimal thread utilisation based upon the number of threads available and the number of instances associated with a particular outlier MO type.

Ultimately, among the possible solutions that were considered for the problem of outlier MO types, I felt that this solution represented the best choice. I discussed this idea with other CS Design Team members, and we agreed that it was promising. The high-level algorithmic question of how I would split CSMMT's work into multiple threads of execution had been answered. With this aspect of the analysis completed, it was time to move from concepts to implementation details. Before I describe the CSMMT implementation, I will outline the key decisions made during analysis.

[16]Even with this optimisation, thread utilisation is not at 100% throughout the entire execution of the application. This is impractical in practise. The goal in this case is to maximise performance by maintaining 100% thread utilisation for the vast majority of CSMMT's execution time.

3.4.7 Outcome of Analysis

Following analysis, numerous key design decisions were made which would heavily influence the CSMMT implementation. These decisions can be summarised as follows:

- CSMMT would execute on multiple threads for greater performance. Each thread of execution would target one MO type.
- In order to retrieve a list of MO types that exist in a given network model, a schema file would be processed at the start of CSMMT's execution run.
- Some MO types may have a disproportionately large number of instances. To ensure maximum thread utilisation through CSMMT's execution run, this handful of outlier MO types would each have their work split among more than one thread of execution. This would be achieved by allocating specially-configured multiple MO updaters to these outlier MO types, instead of the single MO updaters that would be allocated to all other MO types.

Having discussed the key aspects of the analysis performed, I will now describe how the approach decided upon during analysis was translated into working implementation code.

3.5 Implementation & Use of Design Patterns

In this section, I will move through each key aspect of the CSMMT implementation step-by-step, explaining the rationale behind each major decision made during development of the application's core components, with a focus upon the *design patterns* that I deemed to be suitable for each design problem encountered.

I will begin by describing the first design problem that I encountered after moving on from initial analysis and beginning the process of determining the fine implementation details of CSMMT.

3.5.1 Design Problem - How to Process Schema?

As determined during analysis, CSMMT would have to process a schema definition file at the very beginning of its execution run, in order to determine which MO types resided within the network model to be updated. This preliminary step would enable the work of updating different MO types to be divided neatly among different threads of execution.

While considering how to go about implementing the required schema processing functionality, I examined the CS codebase and discovered that a library existed which supplied much of the functionality required to manipulate the schema definition files. This library will hereafter be referred to as the *schema file library* (SFL).

In this instance, it made more sense to reuse the functionality of the existing SFL, rather than 'reinventing the wheel' by developing my own alternative code. However, although the SFL provided useful functionality that would save me the time and effort of developing my own analogous functionality, it did not represent a completely ready-made solution. The SFL provided a large number of methods through its API. In order to compile a list of MO types present in the schema file, numerous method calls would be required to the SFL in a particular sequence.

Upon consideration of this situation, I soon realised that a design pattern that I had studied during CS607, the *facade pattern*, was a good fit for this design problem. Prior to explaining my use of the pattern, I will provide some basic details about the pattern itself.

3.5.1.1 Solution: The Facade Pattern

The facade pattern[12] is classified as a *structural* design pattern, as it defines a manner in which relationships between classes and entities can be created. The purpose of the facade pattern is to create a simplified point of access to a more complex underlying subsystem. This point of access could be described as 'wrapping around' the functionality of the complex subsystem beneath the facade.

Use of the facade pattern is appropriate when the underlying system is sufficiently complex that several method calls are required to complete a particular operation. An implementation of the facade pattern can offer simpler, more readable code, by encapsulating several method calls to the underlying subsystem within one easy-to-use method. This single method can then be called throughout the program, presenting a more convenient alternative than having to perform multiple method calls each time.

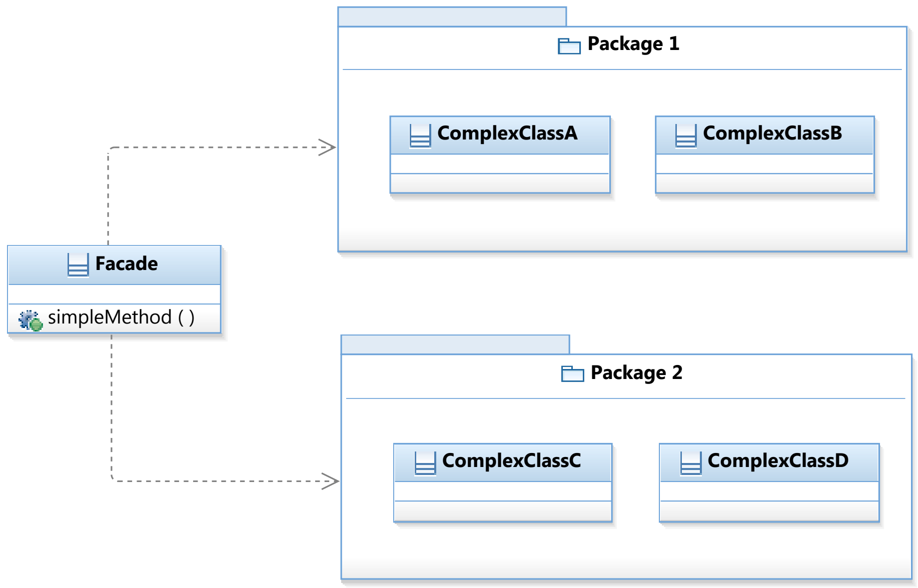


Figure 3.7: Facade Pattern

Brief summaries of the role of each class in the facade pattern template depicted in Figure 3.7 are as follows:

- The **Facade** class contains *simple*, publicly-accessible methods that access the underlying subsystem in order to perform a complex task, while *hiding* this complexity from classes that call the simple method(s).
- The **ComplexClass** classes contain the complex functionality that is 'wrapped' by the **Facade**.

Having identified this design pattern as applicable to my own situation, I will now elaborate upon my implementation of this pattern to solve the design problem at hand. This entailed the creation of a new class, which serves as a *facade*.

3.5.1.2 The `SchemaProcessor` class

In order to retrieve the list of MO types that CSMMT requires, a multitude of method calls must be performed upon the existing SFL. I realised that concentrating all of these method calls together in one place, in order to hide their complexity and provide a *single point of access* to the functionality that was required, would serve to *simplify* the CSMMT codebase. Classes that required a list of MO types would only need to use the simple methods presented by the *facade*, rather than performing a complex series of method calls directly upon the SFL themselves.

To fulfil the role of the facade, I created a new class named `SchemaProcessor`, which wrapped the functionality of the SFL to perform CSMMT-specific functions.

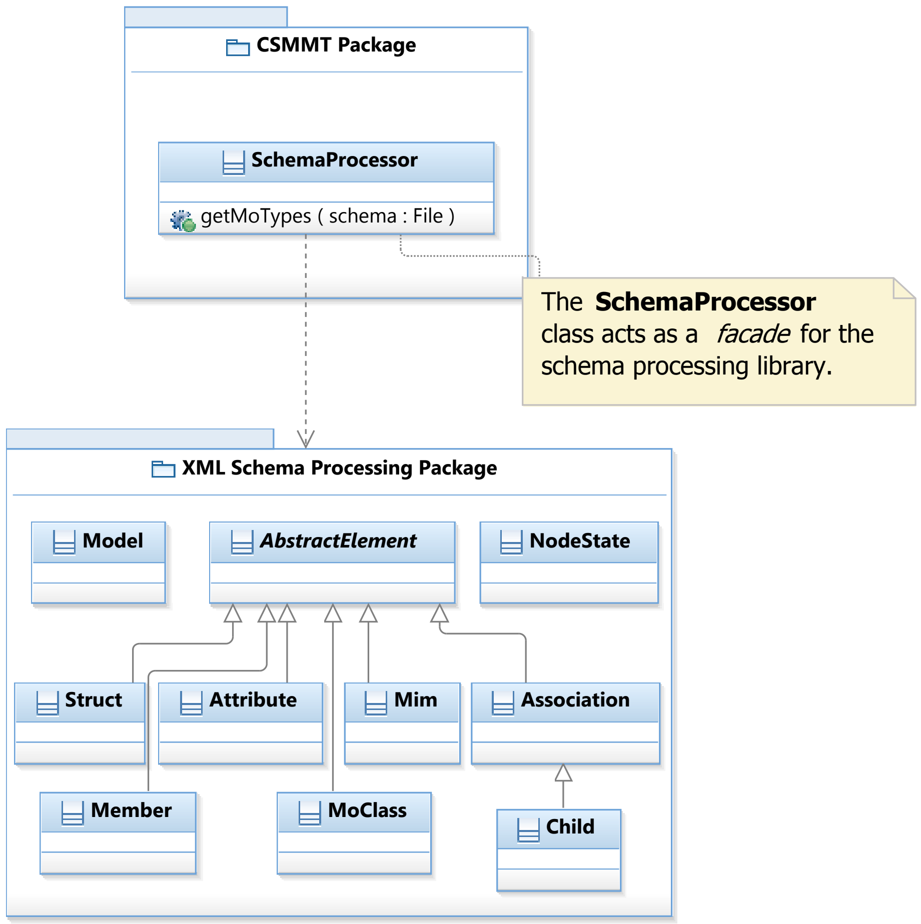

CSMMT Package
SchemaProcessor
getMoTypes (schema : File)
The **SchemaProcessor** class acts as a *facade* for the schema processing library.
XML Schema Processing Package
Model
AbstractElement
NodeState
Struct
Attribute
Mim
Association
Member
MoClass
Child

Figure 3.8: The `SchemaProcessor` facade class

As depicted in Figure 3.8, the `SchemaProcessor` class provides a public `getMoTypes()` method, which takes in the location of a schema definition file as a parameter. When called, this method performs various method calls of its own upon the underlying SFL, in order to perform all operations required to compile a complete list of MO types that reside within the network model. This complex sequence of operations is *completely obscured* from any client that utilises the `getMoTypes()` method.

3.5.1.3 Conclusion & Next Steps

With the `SchemaProcessor` class in place serving as a *facade* for the complicated functionality of the SFL, a design problem had been solved. If I had not implemented this pattern, I would have had to place many direct calls to the SFL subsystem in the middle of other source code devoted to different tasks. Although this would have been perfectly acceptable from a *functional* point of view, from a *design* point of view it would have resulted in more convoluted, less readable source code. It made far more sense to factor out this functionality to a dedicated *facade* class.

I will now discuss a more substantial design problem that was encountered, relating to the implementation details of the MO updater objects that were described earlier.

3.5.2 Design Problem - The MO Updater Objects

As detailed at the beginning of the chapter, the main task of the CSMMT is to deliver a series of four updates to every individual MO instance in the network model. As determined during initial analysis, this was to be achieved by MO updater objects. Most MO types would be worked on by a single MO updater object, which would be responsible for iterating over all MO instances of that type and updating them.

However, some outlier MO types with disproportionately large numbers of instances in the network model would represent an exception to this rule, as they would have *several* MO updater objects assigned to them, to facilitate efficient multi-threaded execution.

Given that different behaviour is required in these two separate cases, it was apparent that *two distinct classes* should be created. One variety of MO updater would iterate over all instances of a given MO type, whereas the other variety would target only a defined *range* of a MO type's instances. These approaches would be quite different programmatically, as the network model would be interacted with in two different ways.

However, despite the differences between these two classes, they would also share substantial *common* functionality: they would each carry out the same set of updates upon individual MO instances in the network model. The difference between the two classes lies in *how* they would carry out these updates. These two classes represent *interchangeable algorithms* - although they carry out their tasks differently, the end result is identical.

After taking the above considerations into account, it was apparent that a particular design pattern covered during CS607 was applicable to this design problem. I will now describe the theory behind the pattern.

3.5.2.1 Solution: The Strategy Pattern

The strategy pattern[15] is classified as a *behavioural* design pattern, as it defines a manner in which communication between classes or entities can be controlled. Use of the strategy pattern enables creation of an *interchangeable family of algorithms*, allowing any of the algorithms to be chosen for use at runtime.

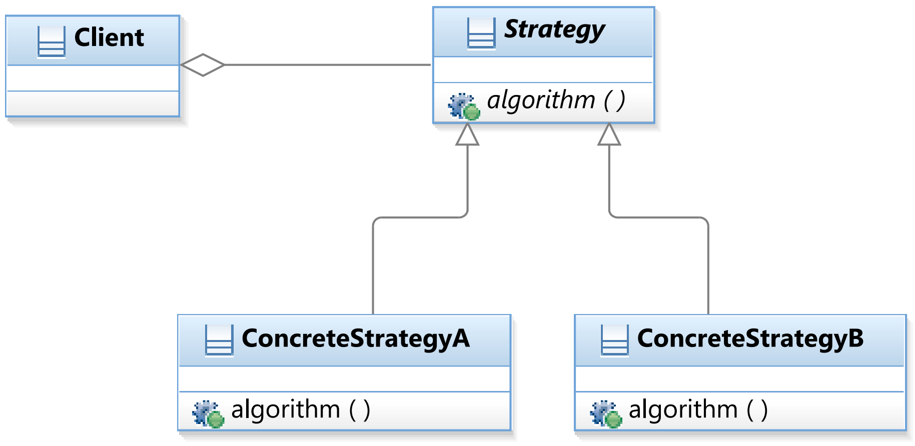


Figure 3.9: Strategy Pattern

I will now summarise the role of the classes from the pattern template depicted in Figure 3.9:

- The **Client** is a class that wishes to use an interchangeable algorithm. In the template above, a field is present within the Client for holding a strategy class instance. This field can be set at runtime depending upon which algorithm the Client requires.

- The **Strategy** is an abstract class that contains all aspects of functionality which are *common* among all of the interchangeable algorithms.

- The two **ConcreteStrategy** classes represent the actual interchangeable algorithms. Common functionality among the algorithms is inherited from the Strategy class, but the two ConcreteStrategy classes will *carry out their tasks in different ways*. In practise, there may be two or more concrete strategies present.

A logical way to begin implementing this pattern was to first implement the **Strategy** class from the above pattern template. I will now describe this new class, as I implemented it initially.[17]

[17]My first implementation was not perfect, and left room for refinement. This would ultimately lead to a new design problem, which will be discussed in Section 3.5.3.

3.5.2.2 The `AbstractMoUpdater` class

I created the `AbstractMoUpdater` class to correspond with the **Strategy** class depicted in Figure 3.9. This abstract class contains all functionality that is *common* to all MO updater objects. In particular, the calls to each of the four update methods[18] that carry out updates upon individual MOs are situated in this class.

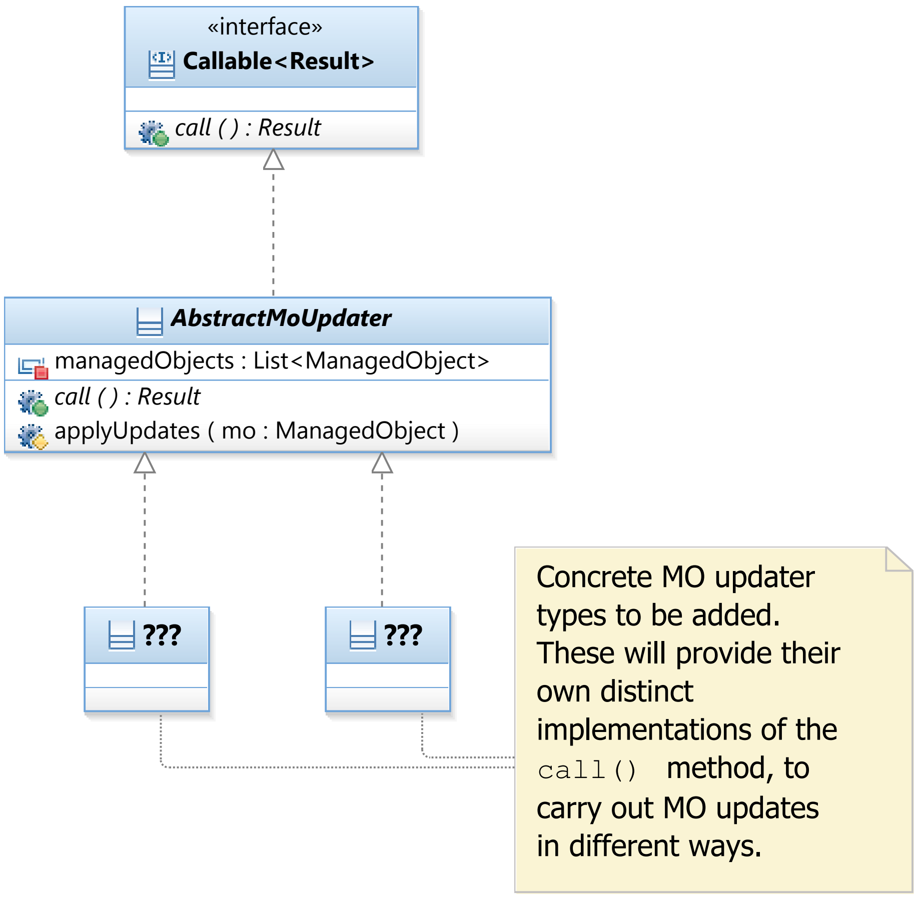

«interface»
Callable<Result>
call ( ) : Result
AbstractMoUpdater
managedObjects : List<ManagedObject>
call ( ) : Result
applyUpdates (mo : ManagedObject)
???
???
Concrete MO updater types to be added. These will provide their own distinct implementations of the call() method, to carry out MO updates in different ways.

Figure 3.10: `AbstractMoUpdater` class, first implementation

As depicted in Figure 3.10, the class contains a `call()` method which has been made `abstract`. This method, as the only public method in the class, serves as the ignition key for each MO updater. When a MO updater reaches the front of the threadpool queue and becomes active, this method is called, signalling the MO updater to begin its work. Since this `call()` method is declared `abstract` in the `AbstractMoUpdater` class, the concrete MO updater

[18]The exact location of these four update methods will be discussed later, in Section 3.5.4 (page 72).

classes are obligated to *provide their own* implementations. This allows them to distinguish themselves with their own unique approaches to carrying out the MO updates required.

With the `AbstractMoUpdater` class in place serving the role of the **Strategy** class from Figure 3.9, the next step was to implement the two concrete MO updaters which would each constitute a **ConcreteStrategy** class in the context of the strategy pattern.

3.5.2.3 Strategy #1: The `StandardMoUpdater` class

During execution of CSMMT, most MO types will have *one* MO updater assigned to updating their instances. Since this represents the more common, *standard* approach to updating MOs within CSMMT, I chose the name of `StandardMoUpdater` for the concrete strategy class that would perform this work.

As this class inherits from `AbstractMoUpdater`, it is obliged to provide its own implementation of the `call()` method, which is triggered automatically by the threadpool when the MO updater reaches the front of the queue. I populated this method with functionality that characterised the approach of `StandardMoUpdaters` for carrying out updates upon MO instances:

1. Some querying is done upon the network model to access the MO type that this `StandardMoUpdater` will work on.

2. An iterator is obtained for the collection of MO instances that are of the MO type associated with this `StandardMoUpdater`.[19]

3. The MO updater iterates over every MO in the collection. For each MO, it calls the `applyUpdates` method, which resides within the common `AbstractMoUpdater` class as depicted in Figure 3.10.

4. When finished updating MOs, the class returns a simple `Result` object to indicate success or failure.

[19]Since this type of MO updater simply updates all instances of an MO type from start to end, it does not need to refer to specific indexes in the collection of MO instances; an iterator alone is sufficient.

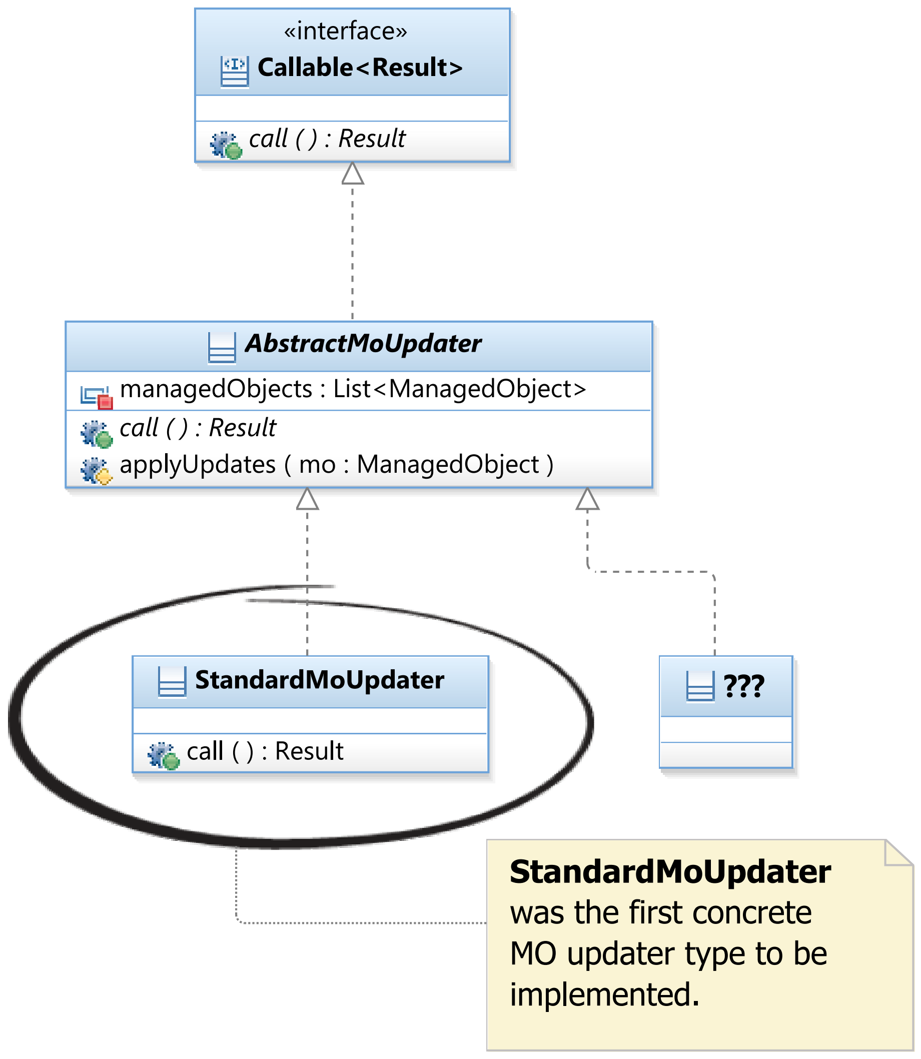

«interface»
Callable<Result>
call ( ) : Result
AbstractMoUpdater
managedObjects : List<ManagedObject>
call ( ) : Result
applyUpdates (mo : ManagedObject)
StandardMoUpdater
call ( ) : Result
???
StandardMoUpdater was the first concrete MO updater type to be implemented.

Figure 3.11: `StandardMoUpdater` class, first implementation

Having implemented my first version of the `StandardMoUpdater` class as depicted in Figure 3.11, I now had one more concrete strategy class to implement: the specialised MO updater type that would be targeted at outlier MO types only.

3.5.2.4 Strategy #2: The `RangedMoUpdater` class

I created the `RangedMoUpdater` class to serve as the second and final MO updater concrete strategy class. This type of MO updater would only be instantiated for use upon outlier MO types, which would have several `RangedMoUpdaters` dedicated to updating their large number of MO instances. The name for this class was chosen to indicate that each instance of this type of MO updater would only update a *limited range* of MOs of a particular type, rather than updating *all* of them as a `StandardMoUpdater` object would.

As with the `StandardMoUpdater` class, the `RangedMoUpdater` class is also obligated to provide its own implementation of the `call()` method, which has been declared `abstract` in `AbstractMoUpdater`. The following tasks are carried out:

1. Some object database querying is done upon the network model to access the MO type that this `RangedMoUpdater` will work on.
2. As well as having a MO type allocated to it, each `RangedMoUpdater` also has a *start* and *end* index allocated to it, indicating the *range* of MOs that this MO updater must update. Rather than using an iterator like a `StandardMoUpdater`, a `RangedMoUpdater` instead targets specific indexes within the range defined by the `startIndex` and `endIndex` integer values.
3. The MO updater traverses the range of MOs to be updated, passing each of them into the `applyUpdates` method which resides in the common `AbstractMoUpdater` class.
4. When finished updating MOs, the class returns a simple `Result` object to indicate success or failure.[20]

[20]Notably, the first and last steps laid out here are *identical* to those which were described for the `StandardMoUpdater` class. This suggested that some further work on these classes was required, which I will elaborate upon shortly.

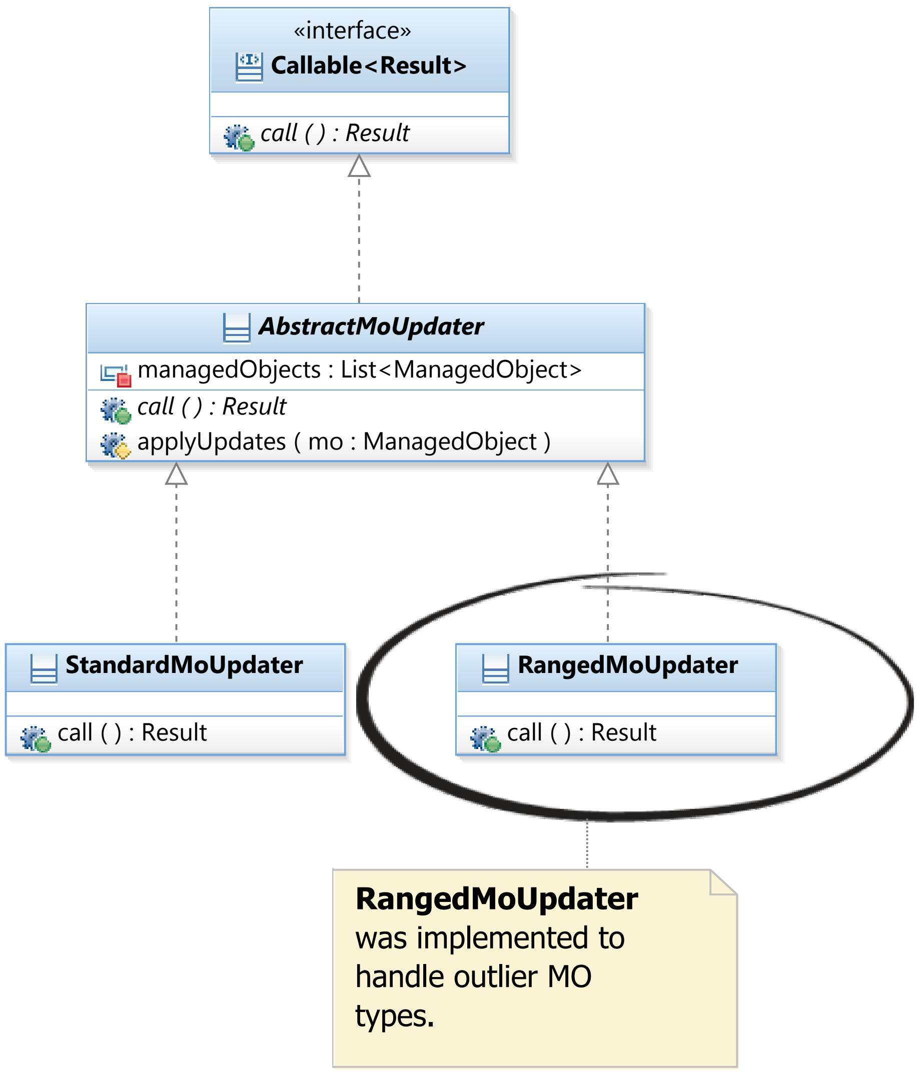


Figure 3.12: Initial implementation of the MO updater classes

With the `RangedMoUpdater` class in place as shown in Figure 3.12, my first implementation of the three crucial MO updater classes was in place, following the template laid out by the strategy pattern.

3.5.2.5 Conclusion & Next Steps

My use of the strategy pattern enabled implementation of an elegant solution for updating individual MOs within an network model. All common functionality for updating MOs resided in the `AbstractMoUpdater` class; this prevented unnecessary code duplication, allowing both concrete classes to utilise this functionality without having to reproduce it in their own classes. The two concrete classes which have been described, `StandardMoUpdater` and `RangedMoUpdater`, serve as interchangeable algorithms which can be chosen by CSMMT at runtime, as MO types are determined to be outliers or otherwise.

Furthermore, the strategy pattern allows for a great deal of future *flexibility*. Although I have implemented only two concrete MO updater strategies, I (or another developer) could easily add more in the future, without modifying the code of the *any* of the existing classes that were depicted in Figure 3.12. This flexibility serves to emphasise the *modularity* that the strategy pattern embodies.

My initial implementation of the the MO updater strategy classes was functionally sound. *However*, I realised that some code was being duplicated at the beginning and end of the `call()` method of each of the two MO updater strategies. Although the amount of duplicated code was small, this was still a *code smell* that signalled the need for refactoring. I will now describe the problem in more detail, and the process of reasoning that led to a solution.

3.5.3 Design Problem - How to Avoid Code Duplication?

As described previously, each of the two MO updater strategies was obligated to provide an implementation for the `call()` method, which was declared `abstract` in the common `AbstractMoUpdater` class. This approach enabled each of the two strategies to distinguish themselves from the other, by use of different logic within the `call()` method. However, a shortcoming of this approach became apparent:

- At the **beginning** of the `call()` method for both strategies, code was present for querying the network model to access a particular MO type.
- At the **end** of the `call()` method for both strategies, some code was present to return a `Result` object.

In both cases, the code involved was *identical* across the two strategies. I realised that in order to perfect the design of these strategy classes, I would have to eliminate the duplicate code at the beginning and end of the two `call()` methods. This code would have to be factored out to a single location that both MO updater types could access. As an abstract class extended by both concrete MO updater types, the `AbstractMoUpdater` class was the obvious candidate.

During the execution of CSMMT, when a MO updater reaches the front of the threadpool queue and it is time for it to be executed, the `call()` method of that MO updater is activated automatically, to signal the MO updater to begin working. I had previously written this method into `AbstractMoUpdater` as an `abstract` method, allowing the concrete MO updater types 'free reign' to provide their own implementations for the method. Although this approach provided the functionality required, it also resulted in code being duplicated across both MO updater types. I realised that *a more structured approach* would be necessary, in order to eliminate this duplicate code from the design.

After consideration, I realised that by implementing another design pattern, the *template method pattern*, I could achieve this more structured approach and factor out the duplicate code. Crucially, this pattern could be applied *without any change* to the current class structure. The class structure put in place by my use of the strategy pattern suited my needs perfectly, so I did not want to change it. It was the *internal structure* of the classes that I did want to change, in order to facilitate reuse of code that would otherwise be duplicated.

3.5.3.1 Solution: The Template Method Pattern

The template method pattern[8], like the strategy pattern, is also classified as a *behavioural* design pattern, since it defines a manner in which communication between classes or entities can be controlled. The pattern lays out an approach for defining basic algorithmic steps which are *common* among several concrete classes. Not all concrete classes will share the same approach for all algorithmic steps, though. For algorithmic steps where differentiation is necessary, the concrete types will be permitted free reign to provide their own unique implementations of these steps.

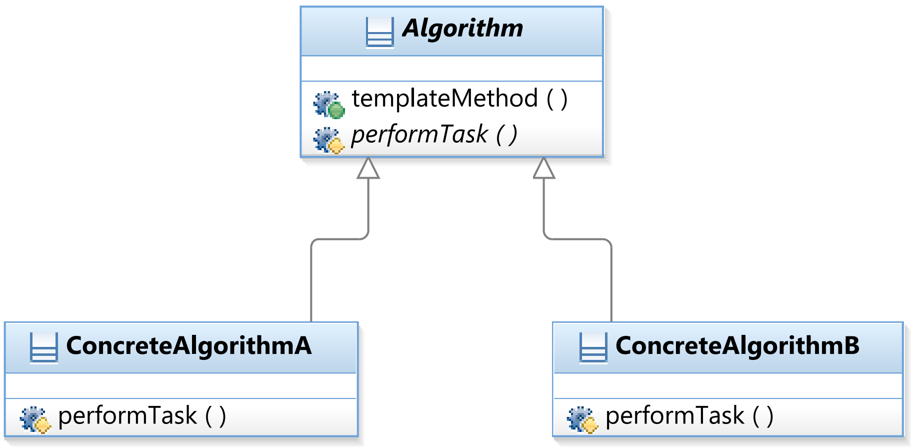


Figure 3.13: Template Method Pattern

I will now describe the classes and methods present in the pattern template depicted in Figure 3.13:

- The **Algorithm** class is the abstract superclass of all concrete implementations of a particular algorithm. Within this class, **templateMethod()** represents the publicly accessible method called by other classes, in order to set the algorithm in motion. The algorithmic steps which are *common* regardless of which subclass is executing them are *fully implemented* in this method.

- However, the algorithmic steps which vary by subclasses will not have any implementation in this method at all. The **performTask()** method is an example of an algorithmic step which *varies* by subclass. To allow the concrete classes to implement their own functionality for this method, it is declared `abstract`.
- At the appropriate point during the algorithm which is laid out in **templateMethod()**, the `abstract` **performTask()** method will be called. This will enable one of the concrete classes to step in and perform its own unique functionality, before returning control to the common algorithmic steps defined in the **templateMethod()**.
- Finally, the **ConcreteAlgorithm** classes are the concrete classes that provide their own implementations for the **performTask()** method. This enables them to provide their own distinct functionality *as part* of the common algorithm defined in the `templateMethod()`.

Next, I will describe how I applied the template method pattern as described above, in order to solve the design problem at hand and eliminate the duplicate code from my design.

3.5.3.2 Application of Template Method Pattern

During the execution of CSMMT, when a MO updater is activated within the threadpool, the `call()` method is triggered. In my initial implementation, I had left this method abstract, allowing the concrete MO updater strategies to implement it themselves. I now realised that it would be better to have the `call()` method serve an identical role to the **templateMethod()** from the template method pattern. To accomplish this, I altered `AbstractMoUpdater` so that the common algorithmic steps at the beginning and end of each MO updater's execution are now fully implemented in this class, instead of being duplicated across the two concrete subclasses.

However, apart from these common steps, the `StandardMoUpdater` and `RangedMoUpdater` MO updater types operate in different ways. I had to give these concrete classes the opportunity to step in and provide their own unique implementations for this non-common section of the algorithm. I provided this opportunity with the new abstract `updateMOs()` method in `AbstractMoUpdater`.

After the common initial algorithmic steps have been carried out by the `call()` method, the `updateMOs()` method is then called. Since this method has no implementation in the abstract superclass, the implementation provided by the applicable concrete type - `StandardMoUpdater` or `RangedMoUpdater` - is used. Once the MOs have been updated and this non-common part of the algorithm is over, control returns to the `call()` method situated in the `AbstractMoUpdater` class, which contains the common conclusion of the algorithm (the return of a `Result` object).

The final implementation of these three classes, with the template method pattern applied in addition to the strategy pattern, is depicted in Figure 3.14.

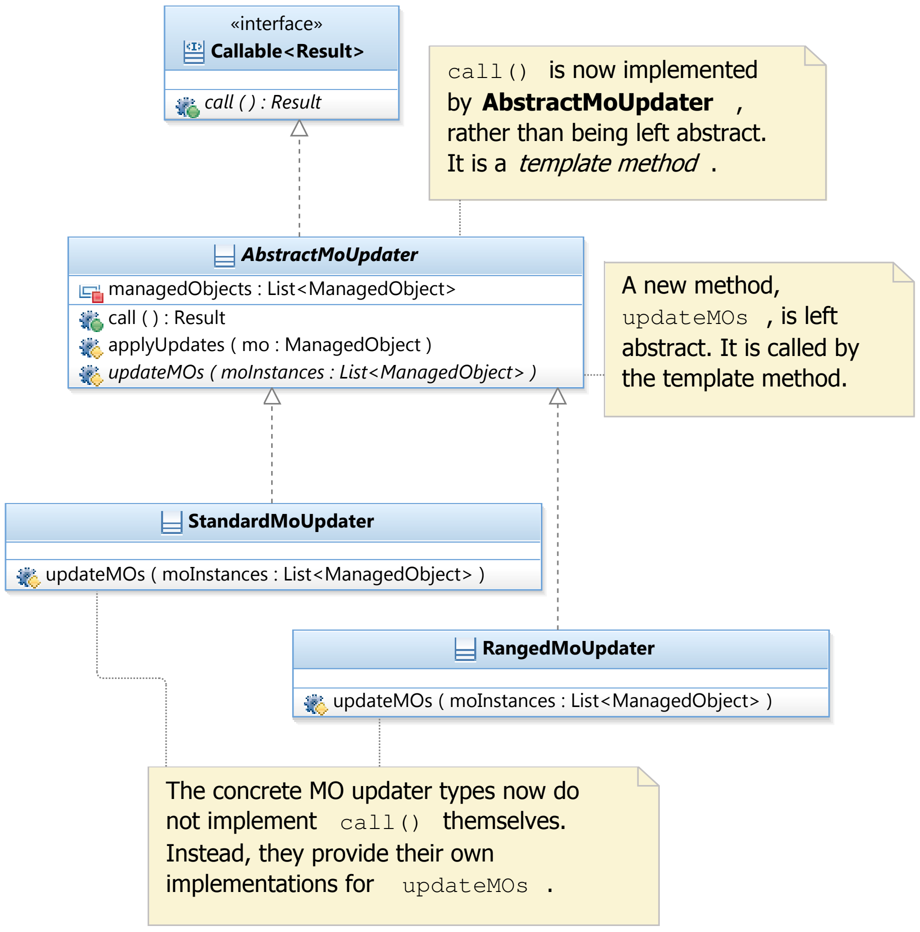

«interface»
Callable<Result>
call ( ) : Result
call() is now implemented by AbstractMoUpdater, rather than being left abstract. It is a template method.
AbstractMoUpdater
managedObjects : List<ManagedObject>
call ( ) : Result
applyUpdates (mo : ManagedObject)
updateMOs (moInstances : List<ManagedObject>)
A new method, updateMOs, is left abstract. It is called by the template method.
StandardMoUpdater
updateMOs (moInstances : List<ManagedObject>)
RangedMoUpdater
updateMOs (moInstances : List<ManagedObject>)
The concrete MO updater types now do not implement call() themselves. Instead, they provide their own implementations for updateMOs.

Figure 3.14: Template method pattern and strategy pattern used together

3.5.3.3 Conclusion & Next Steps

My initial implementation of the three MO updater classes with the strategy pattern had provided the required functionality, with a considerable amount of common functionality residing in the `AbstractMoUpdater` class (such as the `applyUpdates()` method, which was called by each of the concrete MO updater types during their update process). However, with some duplicate code residing within each of the two concrete types, I realised that there was room for improvement. Subsequent application of the template method design pattern served to eliminate duplicate code from this aspect of the CSMMT design.

From a *design* perspective, the three classes as they are depicted in Figure 3.14 represent a clear improvement over the implementation depicted in Figure 3.12 (page 67). The new `updateMOs` method allows extension of the `AbstractMoUpdater` class to be performed in a very controlled way: the concrete MO updaters are only permitted to implement functionality which is *unique* to them.

Furthermore, this final approach is easier to understand than the initial approach, and therefore easier to maintain by future developers. By looking at the `call()` method within `AbstractMoUpdater`, a developer can see a high-level overview of the MO updater algorithm. Since the `call()` method embodies the *template method* that gives the template method pattern its name, it lays out a complete algorithmic skeleton from start to finish, deferring implementation details to concrete types *only when necessary.*

With this work completed, the infrastructure for accessing every MO instance in the network model through use of MO updaters was now in place. However, one final mechanism was required for performing the MO update work triggered by the MO updaters - the mechanism that would be triggered each time a MO updater called the `applyUpdates()` method upon a particular MO. Access to private MO data would be required to carry out each of the four individual updates that CSMMT is responsible for. While looking into this design problem, I realised that this would be trickier to achieve than it first appeared.

3.5.4 Design Problem - How to Access Internal MO Data?

In order to apply updates to each individual MO instance, the MO updaters would require access to *private* data fields stored within each MO instance. No 'getter' and 'setter' methods existed for these private fields, since external access to these fields from other Java classes has never before been required in the history of the Configuration Service.

This posed a critical design dilemma, because the MO updater objects would have to access this internal data *somehow*, in order to migrate the metadata within each MO from one data structure to another. It was apparent that in order to carry out this work, I would have to make careful modifications to the design of the CS itself, since the `ManagedObject` class resides within the CS.

This class is particularly critical to the operations of the CS; I would have to modify it with great care.

I realised that there were several approaches that I could take in order to carry out these needed modifications, providing CSMMT with access to internal MO data.

3.5.4.1 Approach #1: Updating 'In-Situ'

The CSMMT update methods could be placed directly within the `ManagedObject` class. These update methods, triggered by the `applyUpdates()` method within each MO updater, would be able to access the `private` data held within each MO by virtue of being situated inside the same class. This would remove the need for the addition of any 'getter' or 'setter' methods.

However, it was immediately obvious that this option would constitute a very poor design decision. The update methods supply CSMMT functionality, not CS functionality. Therefore, they belong somewhere within the jurisdiction of the CSMMT application, not directly within a core CS class as depicted in Figure 3.15. Only fundamental MO functionality belongs within the `ManagedObject` class. CSMMT-specific functionality, in addition to being out of place, would also serve to 'bloat' the class, rendering it more difficult to understand and maintain.

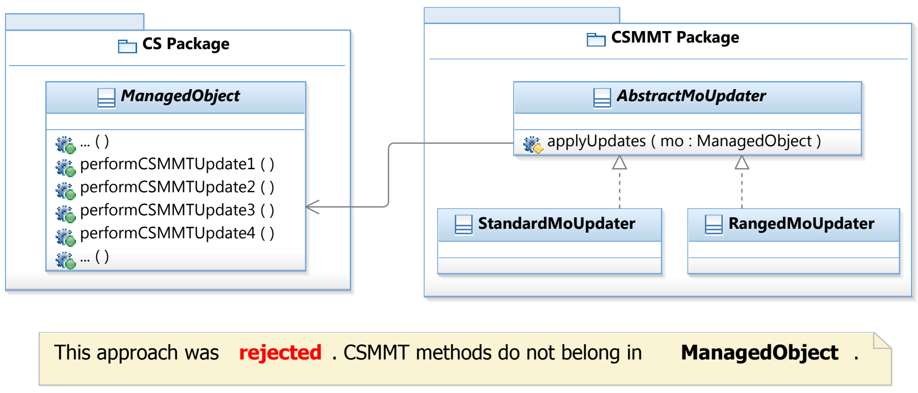


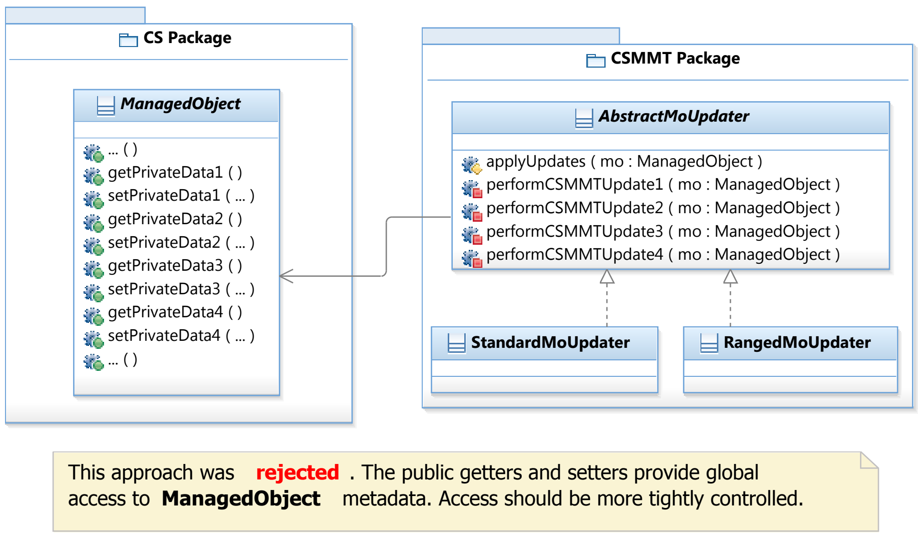
Figure 3.15: Approach #1 - This design idea was rejected.

With this approach quickly ruled out, I moved on to considering an alternative approach.

3.5.4.2 Approach #2: Public Getters and Setters

I considered the possibility of adding `public` getter and setter methods to the `ManagedObject` class, which could then be utilised by CSMMT to access the private data structures that had to be updated. This represented an improvement upon the previously considered approach in that the functionality for carrying out the updates would reside entirely within CSMMT classes; the only modification to the `ManagedObject` class would be the addition of simple getter and setter methods.

However, this approach still did not seem ideal. The aforementioned data structures, up to this point, have been entirely internal to each MO instance, with no external access available whatsoever. Providing *universal* access to them to any other Java class through `public` methods, as depicted in Figure 3.16, did not seem like a good idea. Although I wanted CSMMT to be able to read and set these data structures, I also wanted to limit this access to the greatest degree possible, as *minimisation of access* to members of a class is considered a key tenet of effective object-oriented programming.[21]



This approach was **rejected** . The public getters and setters provide global access to **ManagedObject** metadata. Access should be more tightly controlled.

Figure 3.16: Approach #2 - This design idea was rejected.

With this concept of *minimisation of access* in mind, I moved on to considering another approach.

3.5.4.3 Approach #3: Package-Level Access

The previous approach would have been ideal, were it not for the fact that it would provide excessive access to internal MO data by making the getter and setter methods `public`. To resolve this problem of excessive access, I con-

[21] *"The rule of thumb is simple: make each class or member as inaccessible as possible. In other words, use the lowest possible access level consistent with the proper functioning of the software that you are writing."* - Joshua Bloch, *Effective Java*[3]

sidered the possibility of making these methods *package-private* instead. This would allow access to the internal MO data that CSMMT had to update, but it would only permit access to classes that resided *within the same package* as `ManagedObject`.

In terms of minimisation of access, this approach seemed to be **the best choice** among those that I had considered. Of course, for this approach to work, I would need to create an entirely new class in the same package as `ManagedObject`. This class would contain functionality for utilising the getter and setter methods which had been added to `ManagedObject`, in order to perform updates upon individual MO instances. The `applyUpdates()` method within each MO updater would access the functionality of this new class, using it as an *intermediary* between the core CSMMT classes (which reside in their own package) and the `ManagedObject` class. This concept is illustrated in Figure 3.17, with the to-be-implemented intermediary class shown as "???".

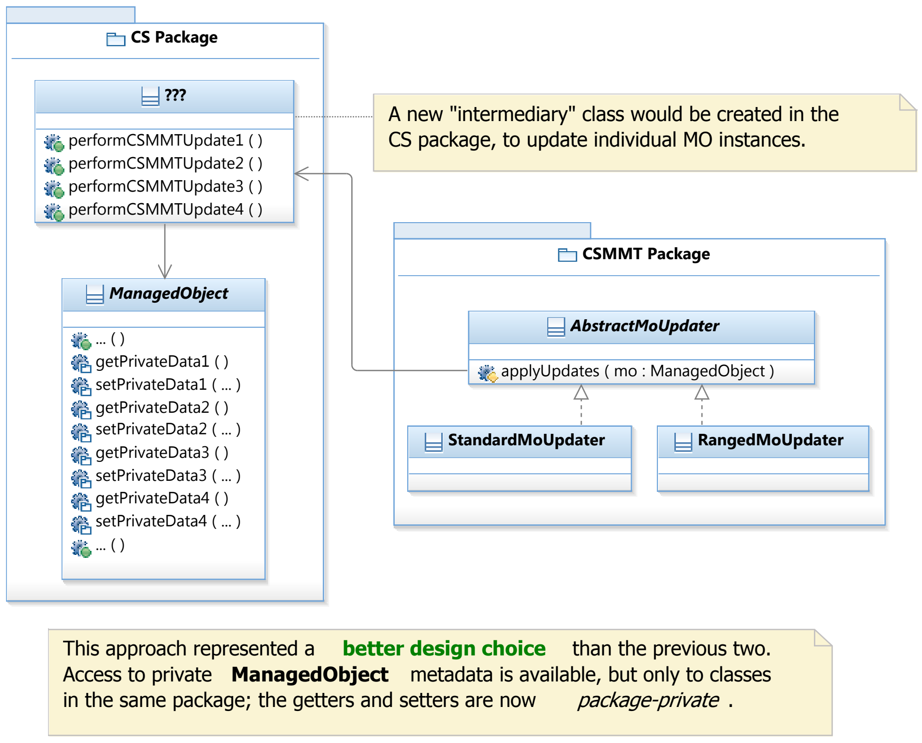


Figure 3.17: Approach #3 - This design idea was chosen for implementation.

I considered how the intermediary class would work. In my discarded first approach, I had considered the possibility of placing CSMMT update methods directly within the `ManagedObject` class. This idea was not feasible; however, there was no reason why these same update methods could not reside within the intermediary class instead. In fact, this idea could be taken further: the intermediary class could serve as an *embellished* version of the original `ManagedObject` class, which would supply some *extra* functionality that `ManagedObject` itself did not. After all, my true goal was to add some CSMMT-specific functionality to the `ManagedObject` class, without placing it *directly* inside that class.

I realised that this apparent contradiction could be resolved through application of a powerful design pattern - the *decorator pattern*. I will now describe the theory behind the pattern, and my subsequent implementation of it to solve the design problem.

3.5.4.4 Solution: The Decorator Pattern

The decorator pattern[11] is classified as a *structural* design pattern, as it defines a manner in which relationships can be created between classes or entities. This pattern allows the functionality of existing classes to be altered or *embellished* at runtime. This is achieved by wrapping the class with a new decorator class, which 'decorates' the wrapped class with differing or new functionality. A template for this pattern is depicted in Figure 3.18.

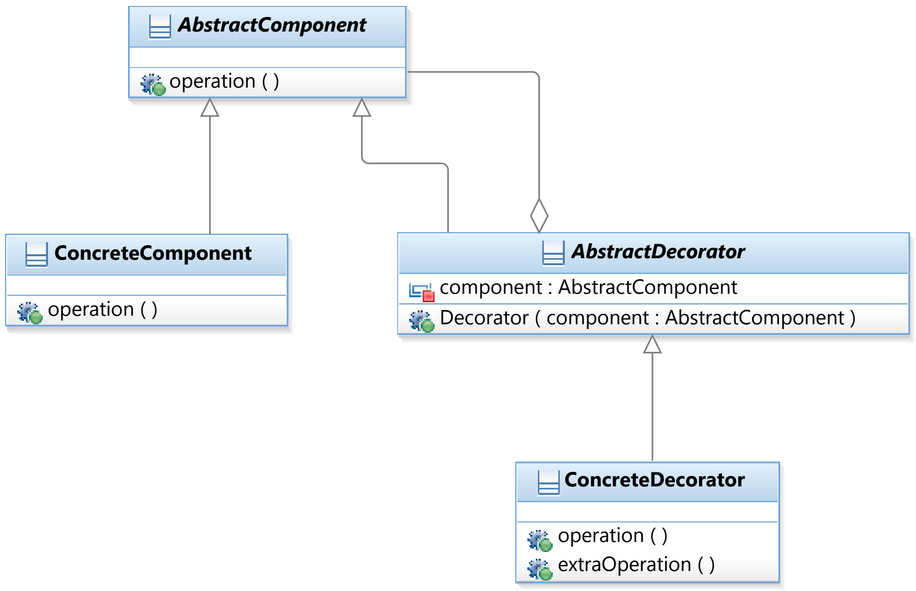


Figure 3.18: Decorator Pattern

The roles of the classes and methods depicted in the pattern template are as follows:

- The **AbstractComponent** class is the superclass of the component that is being decorated. This may be take the form of either an abstract class or an interface.

- The **ConcreteComponent** class represents the specific class that is being decorated.
- The **AbstractDecorator** class is *extended* by the concrete decorator. Since it inherits from **AbstractComponent**, it can be used in place of the component that is being decorated. As a result, the concrete decorator is rendered *interchangeable* with the decorated object itself. Within this class, a constructor is present that takes in an **AbstractComponent** as its parameter. This parameter represents the **AbstractComponent** instance that will be wrapped by the decorator.
- The **ConcreteDecorator** class provides the new or altered functionality that is being applied to the wrapped class. Within this class, the **additionalOperation()** method represents an extra operation that the decorator 'adds' to the wrapped object.

Since my goal was to embellish the `ManagedObject` class with CSMMT update functionality without adding CSMMT methods directly into `ManagedObject` itself, I realised that the decorator pattern as described above represented an ideal approach for tackling this design problem. The next step was to implement the new classes that were required.

3.5.4.5 The `ManagedObjectDecorator` and `CSMMTDecorator` classes

The new `ManagedObjectDecorator` class, depicted in Figure 3.19, represents an analogue of the **AbstractDecorator** class shown in the design pattern template. The constructor allows a MO instance to be taken in, which is then wrapped by a decorator.

The key class to my implementation of this design pattern is the concrete decorator class, `CSMMTDecorator`. Each instance of this decorator class wraps around a MO instance, and enables the four update methods contained within the decorator to be executed upon that MO instance. The `applyUpdates()` method contained within each MO updater calls these four update methods residing in `CSMMTDecorator` in sequence. Since `CSMMTDecorator` resides in the same package as the `ManagedObject` class, it is capable of utilising the newly-added package-level access methods present in the `ManagedObject` class, in order to retrieve and set internal MO metadata.

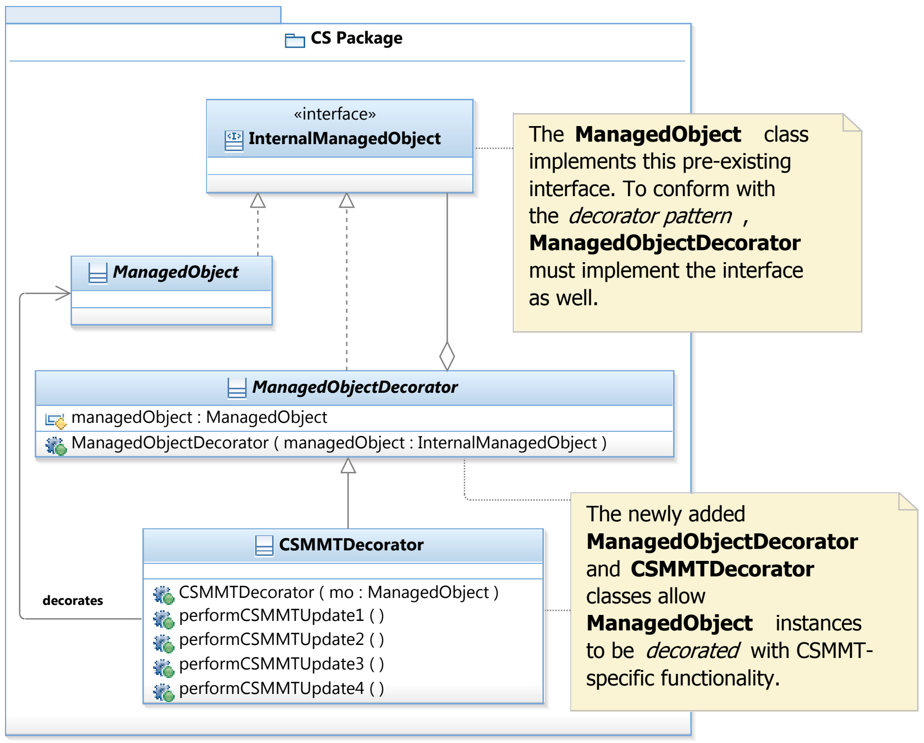

CS Package
«interface»
InternalManagedObject
ManagedObject
ManagedObjectDecorator
managedObject : ManagedObject
ManagedObjectDecorator (managedObject : InternalManagedObject)
CSMMTDecorator
CSMMTDecorator (mo : ManagedObject)
performCSMMTUpdate1 ( )
performCSMMTUpdate2 ( )
performCSMMTUpdate3 ( )
performCSMMTUpdate4 ( )
decorates
The ManagedObject class implements this pre-existing interface. To conform with the decorator pattern, ManagedObjectDecorator must implement the interface as well.
The newly added ManagedObjectDecorator and CSMMTDecorator classes allow ManagedObject instances to be decorated with CSMMT-specific functionality.

Figure 3.19: `CSMMTDecorator` embellishes the `ManagedObject` class

The new `CSMMTDecorator` class as depicted in Figure 3.19 represents an intermediary between MO updaters (`StandardMoUpdater` and `RangedMoUpdater`), and each MO instance. With this critical class in place, all necessary infrastructure was now in place for delivering updates to individual MO instances. This use of `CSMMTDecorator` by MO updater types is depicted in Figure 3.20.

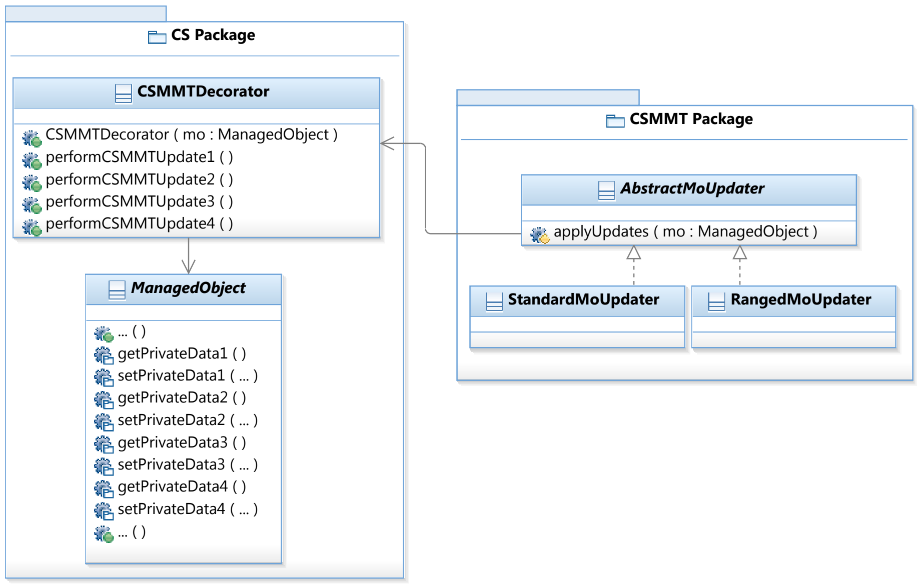


Figure 3.20: `CSMMTDecorator` is utilised by all MO updater types

3.5.4.6 Conclusion & Next Steps

Use of the decorator pattern enables the dynamic addition of behaviour to objects at runtime. In this instance, it constituted an ideal solution to the design problem that I had encountered. By *wrapping* individual MO instances with the `CSMMTDecorator`, the CSMMT application is able to treat these wrapped objects as *embellished MO instances*, which contain additional functionality required by CSMMT to carry out its updates, but also contain all methods than a MO instance would normally contain.

The application of the decorator pattern described above represented a landmark in the development of CSMMT. With the MO updater classes present, and the `CSMMTDecorator` in place to enable full access by CSMMT to internal MO metadata, a complete delivery system was now in place for the updates that CSMMT would have to deliver as its main task.

However, one last design problem remains to be described. In order for CSMMT to take advantage of the functionality made available by the MO updater objects, it would first need to *create* them. I will now discuss this final design problem, of which the solution would share some similarities to one which I had implemented during my previous work on Cslibtest.

3.5.5 Design Problem - Creating the Updater Objects

After processing the schema to glean a list of MO types present in the network model, and instantiating an `ExecutorService` threadpool to accommodate the MO updater objects, CSMMT has to carry out one final step before the MO updaters can carry out their work: the required MO updaters have to be created.

This creation process must be capable of creating objects of two different concrete MO updater types. It was apparent that creation of a `StandardMoUpdater` would be relatively simple: a MO type would be chosen from the list of MO types which had not yet been assigned an MO updater, and a `StandardMoUpdater` would then be created, with the unassigned MO type passed into its constructor. When this MO updater is later activated by the `call()` method, it will update all instances of its assigned MO type.

However, creation of `RangedMoUpdaters` would be more complicated. To properly handle an outlier MO type, *multiple* `RangedMoUpdaters` would have to be assigned to that MO type. Each would have to be assigned a *specific range* of MO instances to update, so that between all of the `RangedMoUpdaters` assigned to an outlier MO type, *all* of the instances of that type would be updated between them.

After considering the above, I concluded that a particular design pattern that I had utilised previously in the Cslibtest project - the *factory method pattern* - was applicable to this design problem. I will now describe my implementation of this final design pattern.

3.5.5.1 Revisited Solution: The Factory Method Pattern

The factory method pattern previously provided me with a solution for manufacturing different types of objects while I was working on the Cslibtest application. In the case of Cslibtest, these objects were Cslibtest command objects. With CSMMT, the objects to be manufactured were MO updater objects. Despite these differences, the basic principles behind the creation process remain the same for both applications; therefore, the *same* design pattern is equally applicable to both of them.

Since I have previously discussed the template for the factory method pattern in section 2.6.2.1 (page 33), I will not reiterate it here. Instead, I will move directly to explaining my application of this design pattern to the CSMMT application, with the creation of the new `MoUpdaterFactory` class.

3.5.5.2 The `MoUpdaterFactory` class

To implement the factory method pattern, I created the new `MoUpdaterFactory` class, which contains the `createMoUpdaters` factory method. This method takes in *two* parameters:

1. **An MO type.** This MO type is assigned to the MO updater(s) returned by a single call of the `createMoUpdaters` method.

2. **An integer value, indicating the number of MO updaters required for the given MO type.** This value is set to 1 for most MO types, indicating that only a single `StandardMoUpdater` is required. A value *greater than 1* indicates that an outlier MO type had been passed into the method, to which multiple `RangedMoUpdaters` must be assigned.

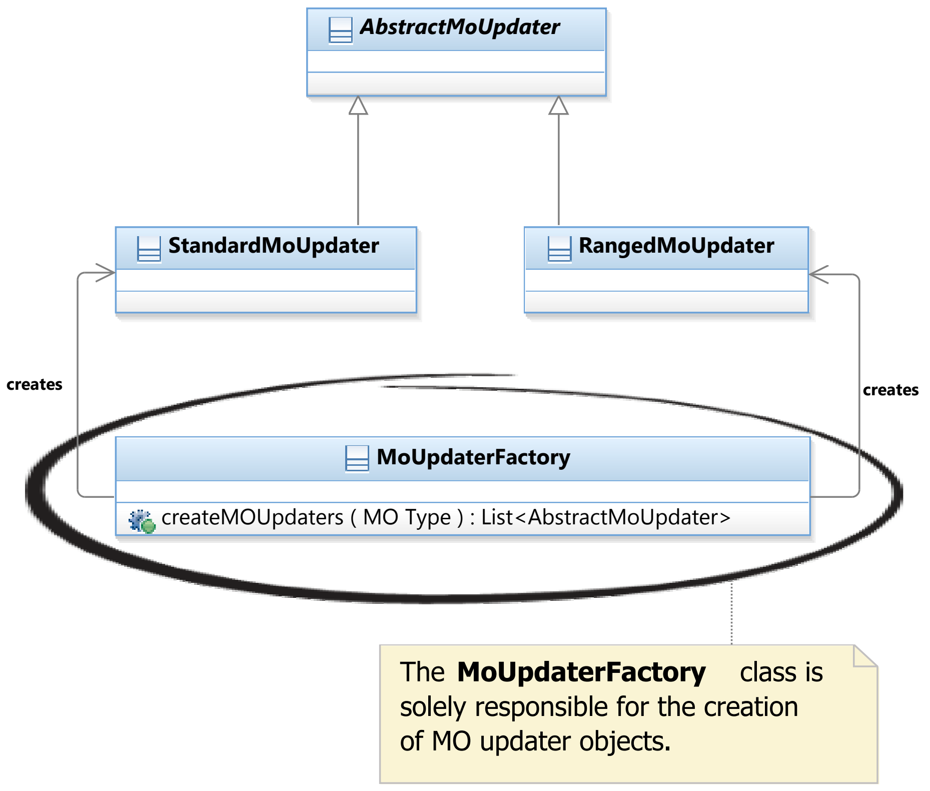

AbstractMoUpdater
StandardMoUpdater
RangedMoUpdater
creates
creates
MoUpdaterFactory
createMOUpdaters (MO Type) : List<AbstractMoUpdater>
The MoUpdaterFactory class is solely responsible for the creation of MO updater objects.

Figure 3.21: The `MoUpdaterFactory` class

If the integer value is set to 1, a `StandardMoUpdater` is instantiated for the supplied MO type, and returned. However, if the integer value is greater than 1, several steps are followed by the factory method to ensure that an even distribution of work is achieved among the `RangedMoUpdaters` to be created. The following steps enumerate the process of creating `RangedMoUpdaters` for a given MO type:

1. The exact number of instances of the outlier MO type is determined, by use of a specialised object database query to the network model.[22]

2. Computations are performed to determine a *start and end index* for each of the `RangedMoUpdaters` to be created, which results in each MO updater being responsible for the *same number* of instances.

3. The `RangedMoUpdaters` are each instantiated with a pair of start and end indexes, drawing from the computation performed in the last step.

4. Finally, the new `RangedMoUpdaters` are all returned in a collection.

With the new `MoUpdaterFactory` class in place, a comprehensive mechanism for creation of specialised MO updaters was now in place for CSMMT to access at the beginning of its execution.

3.5.5.3 Conclusion

The factory method pattern, already of use during my work on Cslibtest, proved to be useful for a second time during my work on CSMMT. By example, this served to emphasise the broad applicability of design patterns such as the factory method pattern; although Cslibtest and CSMMT are entirely different applications, the same design pattern proved readily applicable to both.

The individual assemblies of classes which work together to provide the functionality required for carrying out CSMMT's main task, and the design patterns that inspired the structure of these assemblies, have each been explored in isolation. I will now summarise the execution of the CSMMT as a whole, illustrating the role of each of these assemblies as modular components of the CSMMT application. To this end, I will describe the class I developed to coordinate the activities of CSMMT and provide a point of entry to the application: the aptly-named `CSMMT` class.

3.5.6 Overall Execution of CSMMT - the `CSMMT` class

The `CSMMT` class, named after the application in line with naming conventions within the CS, leverages all of the classes that have been described thus far in order to update a single network model at a time. To describe the operation of this class, I will provide an overview of each step of the algorithm that it executes.

The following series of steps takes place when CSMMT is executed upon a network model:

[22]This computationally expensive counting operation represents a downside of the `RangedMoUpdater` approach. However, the improved thread utilisation gleaned from their use makes up for this disadvantage.

1. A *schema definition file* is processed, in order to compile a list of MO types which have instances in the network model. To achieve this, the `CSMMT` class utilises the `SchemaProcessor` class. As an implementation of the **facade pattern**, the `SchemaProcessor` class facilitates this schema processing by presenting a simplified point of access to a complicated schema processing library that resides within the CS.

2. Analysis is performed upon the number of instances of each MO type, to determine whether any outlier MO types with a disproportionately large number of instances are present. If an outlier type *is* present, a number greater than 1 is associated with that MO type, indicating the exact number of MO updaters that have been deemed necessary for this MO type[23] based upon the number of instances present.

3. The list of MO types is traversed. For each MO type in the list, a call is made to the `createUpdaters()` factory method present within `MoUpdaterFactory`, which represents an instance of the **factory method pattern**. This class provides a centralised point of creation for all MO updaters used during CSMMT's execution.

4. As new MO updaters are obtained from the factory method, they are compiled into a single collection of `AbstractMoUpdaters`. This class and the two concrete MO updater types collectively represent a combination of the **strategy pattern** and **template method pattern**. The synergy between these two patterns provides the benefits of interchangeable algorithms which can be selected at runtime, coupled with greater code reuse and the elimination of duplicate code.

5. An `ExecutorService` threadpool is created, and the entire collection of `AbstractMoUpdaters` is submitted to the threadpool queue. A limit is defined as to how many of these MO updaters will be active at once, so as not to overwhelm system resources with too many threads. The remaining MO updaters will queue, becoming active within the threadpool as other MO updaters finish updating their assigned MO instances and terminate.

6. Within the threadpool, each MO updater accesses the MO instances that it is responsible for updating. It decorates these MO instances with the `CSMMTDecorator` class. This use of the **decorator pattern** allows the `ManagedObject` superclass held within the CS to be *embellished* with metadata update methods that CSMMT requires. These methods are

[23]The more instances that are present in the outlier MO type, the greater the number of MO updaters that will be allocated to it, in order to ensure efficient multithreaded execution of CSMMT.

called by the MO updaters in order to perform the required updates upon each individual MO instance.

7. As MO updaters finish their work, they return `Result` objects which are compiled into a collection within the `CSMMT` class. When all MO updaters have finished working, the collection of `Results` is iterated over and checked. If no failures are found, this indicates that CSMMT has performed its work successfully, and the network model is now fully up to date. If failures are found, CSMMT will terminate with a descriptive error message.

The above steps encompass all of the CSMMT implementation aspects that were discussed in detail throughout this chapter. By leveraging the capabilities of these classes, the CSMMT class carries out the main task of the application in its entirety.

Although it did account for the majority of my time spent on the project, development of the CSMMT application itself did not comprise the entirety of my work on the project. I was also responsible for putting together an extensive suite of automated black-box tests for CSMMT. I will now describe the structure and operation of this automated test suite, and the role that it played during the development of CSMMT.

3.6 Automated Testing of CSMMT

While working on the CSMMT implementation, I was also responsible for developing an automated test suite to accompany the application. This test suite functions as a comprehensive series of black box tests, which can be executed at any time in order to determine whether or not CSMMT is carrying out *all* of its metadata updates correctly. I will now describe the layout of this test suite in more detail.

3.6.1 Series of Steps

Early in the development of these tests, I realised that in order to test the functionality of CSMMT effectively, several steps would need to be carried out in order. The steps are as follows:

1. A new 'dummy' network model is created and initialised, in order to ensure that each test run is entirely isolated and independent of any other test runs.

2. The network model is populated with some MOs for CSMMT to update. This is achieved by a new Java class that I developed, named `PopulateDatabase`. The class makes use of existing CS functionality to create each MO individually within the network model. Special care is taken to create these MOs in 'pre-update' format, without any new data structures present in `ManagedObject`.

3. Some Java code is then used to iterate over every MO in the network model, in order to record the metadata contents of the old data structures. After CSMMT, this recorded data is checked against the contents of the new data structures, ensuring that no data has been lost or corrupted during the execution of CSMMT.

4. CSMMT is then executed upon the network model.

5. After a successful execution of CSMMT[24], an extensive suite of JUnit tests is then executed upon the *current* state of the CS metadata within each MO instance, to ascertain whether or not it has been migrated correctly.

6. Upon completion of its tests, the JUnit test suite records its test results to an XHTML document and terminates.

This series of steps encompasses a wide variety of actions. When I was designing the test suite, it was readily apparent to me that it would be undesirable (and entirely impractical) to have to execute each individual step manually upon each test run. In order to be useful, this test suite would have to be fully automated. Having made use of existing Apache Ant scripts during my work placement prior to this point, I knew that Apache Ant represented an ideal tool for this task.

3.6.2 Automated Execution via Apache Ant

Apache Ant is widely used for numerous code compilation and testing tasks within Ericsson. Upon examination of the XML scripts that Ant uses, and some of the scripts written by my colleagues for other tasks, I realised that such a script would be ideally suited to the series of steps that I wanted to carry out in order to test CSMMT.

Since Apache Ant supplies dedicated functionality for launching JUnit test suites, I leveraged this feature to compile the complete test sequence that I previously described into a new Ant script. In Figure 3.22, the high-level structure

[24]If CSMMT produces a non-zero exit code indicating *failure*, the test sequence aborts prior to any JUnit tests being executed.

of the series of operations that I implemented into an Apache Ant XML script is graphically depicted.

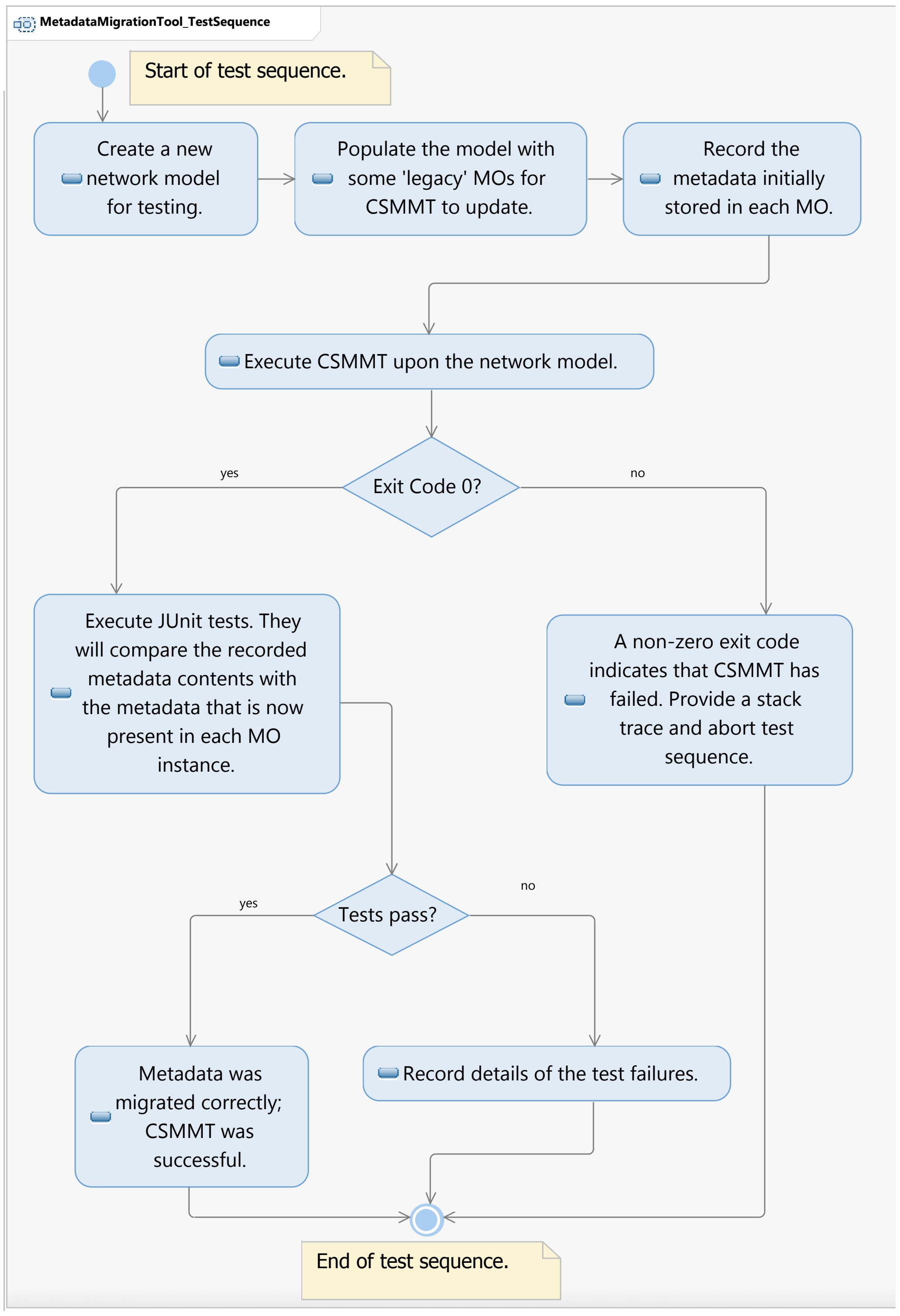

MetadataMigrationTool_TestSequence
Start of test sequence.
Create a new network model for testing.
Populate the model with some 'legacy' MOs for CSMMT to update.
Record the metadata initially stored in each MO.
Execute CSMMT upon the network model.
Exit Code 0?
yes
no
Execute JUnit tests. They will compare the recorded metadata contents with the metadata that is now present in each MO instance.
A non-zero exit code indicates that CSMMT has failed. Provide a stack trace and abort test sequence.
Tests pass?
yes
no
Metadata was migrated correctly; CSMMT was successful.
Record details of the test failures.
End of test sequence.

Figure 3.22: CSMMT automated test sequence

With this structure integrated into a reusable Apache Ant XML script, I was able to execute *all* of the steps described in Section 3.6.1 with a single command. This allowed for comprehensive, *repeatable* testing of the functionality of CSMMT throughout the development process, which proved to be extremely useful for pinpointing aspects of the application which were not functioning properly.

In particular, these tests proved to be very useful for *regression testing* of the CSMMT source code.

3.6.3 Regression Testing

On numerous occasions late in CSMMT's development, I made changes to the CSMMT source code to improve non-functional qualities of the code (such as code reuse and elimination of duplicate code) without any intention of affecting the actual *functionality* of CSMMT. However, on multiple occasions, I inadvertently disrupted the functionality of CSMMT while carrying out these changes, even though no errors were detected upon compilation of the source code.[25]

By rigorously executing my test suite after each change I made to the source code, I was able to immediately detect subtle glitches in the application that had been inadvertently introduced by my changes. The reports that the test suite generated allowed me to quickly track these problems down and rectify them. After fixing the problem in the source code that the test report had led me to, I would then compile a new version of the source code and execute the automated test suite again. This subsequent execution of the same set of tests would then verify whether or not I had solved the problem.

Without the presence of this automated test suite, the regular regression testing that I performed throughout the development of CSMMT would not have been possible.[26] As a facility for carrying out this regular regression testing quickly and easily, the automated test suite that I had developed alongside CSMMT proved to be indispensable on many occasions.

3.6.4 Conclusion

The JUnit Framework, covered during the CS608 module (Software Testing), proved to be very useful in carrying out individual tests upon the metadata stored within each MO instance after CSMMT had been executed. However, Apache Ant scripts made the automated execution of the test sequence depicted

[25] *Any* code refactoring, though beneficial in terms of the superior code that it produces, can also result in the accidental introduction of subtle bugs.[6] Through regular regression testing, these bugs can be found and fixed quickly.

[26] Without these regular regression tests, any run-time errors introduced to CSMMT through minor changes to the source code might have gone undetected for some time, becoming more and more difficult to track down and fix as additional changes were made to the source code thereafter.

in Figure 3.22 possible. The functionality offered by these scripts proved to be invaluable in enabling me to rapidly implement the automated series of testing steps that I wanted to execute regularly upon CSMMT.

My automated test suite was utilised heavily throughout the development of CSMMT. At several points during development of the application, *some* of the metadata migrations that CSMMT is responsible for were being carried out successfully, whereas others were not. Use of the automated test suite enabled me to pinpoint exactly which areas of the code were failing; this was of great assistance to me in tracking down problems within CSMMT's source code and resolving them quickly.

Eventually, *all* of the tests in my test suite passed for the first time, indicating that CSMMT was carrying out the entirety of its main task correctly. After this point, the automated test suite continued to be extremely useful as a facility for regression testing of the CSMMT source code as minor changes and tweaks were made.

3.7 Project Outcome

At the outset of the project, the requirements specified that CSMMT would have to be capable of updating millions of MOs residing inside a network model within a tightly constrained time window, if the project were to be deemed a success. Efficient multithreaded execution, and measures to ensure maximum thread utilisation throughout, proved to be instrumental factors in enabling CSMMT to reach this target.

As with my previous work on Cslibtest, design patterns played a key role in this project as well. Rather than having to re-invent the wheel each time I encountered a familiar design problem, I was able to draw upon existing knowledge, acknowledged and accepted throughout the software engineering field, of applicable solutions to these problems. The presence of these templates allowed me to quickly identify suitable class structures for CSMMT, which afforded me more time for developing and tweaking the code. Furthermore, use of these design patterns helped me to put together a readily understandable, highly modular, loosely coupled codebase. These characteristics should make life easier for other developers who wish to maintain or modify the CSMMT application in the future.

Prior to delivery of CSMMT, the code was reviewed in a similar manner to Cslibtest. The comments made by the reviewers did not suggest any serious problems; some very minor structural issues were identified in a handful of classes, in addition to a handful of methods that had no Javadoc information associated with them. These small issues were rapidly resolved, and the application was deemed ready for deployment.

With the successful conclusion of the CSMMT project, a new update package has since been delivered, both within Ericsson and to Ericsson customers around the world. This package includes the new CSMMT application, and a mechanism for automatically executing it upon network models that require a metadata migration update.

Chapter 4

Conclusions

4.1 Software Engineering Techniques

I will now briefly discuss, based upon my own experiences, my conclusions upon several aspects of software engineering that have been discussed during this dissertation in the context of two substantial software projects.

4.1.1 Conclusions on Design Patterns

As demonstrated by my project work at Ericsson, use of design patterns for appropriate design problems can dramatically accelerate the software development process. Many of the problems that I encountered, both with the Cslibtest and CSMMT projects, would have been considerably more time-consuming to solve if I did not have proven design patterns to refer to and utilise.

My prior exposure to the theory and practise of design patterns during my M.Sc. coursework enabled me to recognise the patterns applicable to particular design problems very quickly. This was of great benefit to me during my work placement, and would undoubtedly be of benefit to other software engineers as well. In addition to enabling me to implement design patterns in my own projects, my knowledge of patterns has allowed me to better understand areas of the Configuration Service that were written by other developers with heavy use of design patterns.

As I discovered during my work placement, it can often to be beneficial to 'think outside the box' in terms of the application of design patterns. On multiple occasions, I have found distinct benefits to be associated with applying more than one design pattern to the same set of Java classes. The most prominent example of this arose during my work on CSMMT, when I realised that code duplication could be eliminated by implementing the template method pattern *in combination* with the strategy pattern. It is clear that the *modularity* of design patterns enables them to be productive not only individually, but also when used together. Application of one design pattern to a particular set of classes does not necessarily preclude the application of additional patterns.

4.1.1.1 The Future of Design Patterns

Design patterns will continue to be useful to software engineers in the future. However, as newer programming languages become more widely used, new patterns may emerge, and the implementation of existing patterns may change or be replaced entirely with native language support. As is the case with many aspects of the software development industry, software engineers will have to learn and adapt in order to keep pace with changes that occur.

For instance, the multi-paradigm Scala programming language is gaining in popularity, having recently been adopted by Twitter to power the backend of their popular microblogging service.[27] It is necessary to re-evaluate several popular design patterns in the context of Scala. For example:

- Scala supports *pattern matching*[20], which provides a native equivalent of the *visitor design pattern*[16].
- Scala's *companion objects*[22] provide a native equivalent of the *singleton design pattern*[10].
- The *strategy design pattern* which I used in my own work (as discussed on page 62) is unnecessary in Scala, where functions are represented as first-class objects[21]. Since such functions can serve as strategies and are treated as objects, it is unnecessary to create entire classes to encapsulate individual strategies, as is the case in Java.

Although the aforementioned design patterns are highly relevant for Java designers as distinct constructs in source code (as has been the case in my own work at Ericsson), they have effectively been *subsumed* into the Scala programming language and are facilitated by native features of the language. As programming languages continue to evolve, design patterns will evolve as well.

4.1.2 Conclusions on Code Reviews

Any software project, irrespective of its size[1], will benefit greatly from regular code reviews by observant software developers. During both of the projects

[1]On the CS Design Team, even miniscule alterations to pre-existing code are reviewed by at least one team member prior to delivery. Problems have been detected with small modifications on a number of occasions.

discussed in this dissertation, code reviews performed by my colleagues resulted in detection of issues that may otherwise have been missed. When I performed code reviews upon the work of my colleagues, I also detected issues that otherwise may have been missed.

It is worth emphasising that code reviews do not constitute a 'magic bullet'; some faults may be missed by code reviews, as I observed on a couple of occasions during my placement.[2] However, the execution of thorough code reviews is virtually guaranteed to substantially *reduce* the number of faults in a software product. In combination with other techniques, such as software testing, code review represents a vital technique to be employed by any software development team in order to ensure that each finished product is of the highest quality possible.

4.1.3 Conclusions on Software Testing

A comprehensive suite of automated software tests is akin to a safety net for software engineers, catching faults and ensuring that they can be resolved prior to the delivery of a software product. During my placement, software testing was a particularly prominent aspect of the CSMMT project. Without the JUnit test suite that I produced, numerous faults in earlier versions of my code would surely have gone undetected for some time. If such faults had gone undetected during code review as well, it is conceivable that they could have remained present in the delivered product, resulting in the need for troublesome fixes later.[3]

As demonstrated during Section 3.6 (*Automated Testing of CSMMT*), the critical importance of this software engineering subdiscipline was validated by my own experiences at Ericsson. The detection and repair of problems in CSMMT by use of testing tools and techniques not only prevented severe issues from being present in the delivered product, but also accelerated the development process by enabling each problem to be tracked down and fixed rapidly.

Even the most comprehensive software testing can never guarantee the total absence of faults.[24] However, as is the case with code review, testing provides the assurance of *fewer* faults reaching the delivered product intact. Given the importance of software testing during the development process, my prior exposure to software testing techniques during my M.Sc. course proved advantageous on many occasions during my work placement. Based upon my own experience,

[2]On one occasion, a highly significant fault in some recently-altered CS code was missed, despite a thorough code review being performed. Thankfully, a comprehensive series of tests performed by a separate testing team detected the problem prior to delivery.

[3]As intuition would suggest, a problem detected earlier in the development process is considerably less costly to fix than a problem detected in the later stages.[25]

I believe that a solid grounding in the principles and practises of software testing is vital knowledge for any software engineer to possess.

4.1.4 Conclusions on Code Refactoring

Code refactoring proved to be necessary on both the Cslibtest and CSMMT projects, despite their differing nature. Cslibtest involved the modification of pre-existing code, and the need for code refactoring of this legacy codebase was apparent. However, habitual examination and refactoring of newly-written code is highly beneficial, as well, since it is very difficult to write perfectly structured source code on the first attempt. As a key example, my observation of code duplication in CSMMT led to significant refactoring of source code that I had written. This involved the introduction of a new design pattern, resulting in the combination of strategy and template method patterns that was described previously.

Although code refactoring is an immensely practical technique for software engineers, it was not covered in great depth during the degree courses that I have taken at university. I will now conclude with my thoughts on an important aspect of software development that, while of high relevance during my work placement at Ericsson, was not extensively covered during my degree courses at university.

4.2 Importance of Non-Functional Qualities

Not long after I first began my work placement, an early version of some code that I had written was reviewed by my colleagues. This code compiled and functioned exactly as required. However, there was room for improvement of the *non-functional qualities* of my code. For instance, my experienced colleagues pointed out instances of *high coupling* between classes in the code that I had written. Although the code functioned as intended, *maintainability* and *extensibility* of the software could be improved upon by decoupling the classes.

After these early code reviews, I implemented the suggestions of my colleagues, and the structure of my code improved rapidly. I could soon determine at a glance whether or not sections of source code were well structured as a whole. If they were not, I could identify the reasons why. As I reviewed the work of others and developed a greater familiarity with the CS codebase, I became more adept at spotting structural issues in the code that I was examining. With frequent exposure to code of varying levels of maintainability and extensibility, I developed a greater appreciation for these important subtleties, and the impact that they can have upon the software development process.

During my time at university (both during my B.Sc. and M.Sc. degree courses), taught modules have placed a strong emphasis upon external functionality where programming is concerned, but considerably less emphasis has been placed upon the structure of the underlying source code. This was reflected in how programming assignments were marked. Marks have generally been awarded based upon whether a programming assignment functioned as intended, with the underlying non-functional aspects of the code often treated as unimportant or even irrelevant.

Two programs with differing underlying structure but identical functionality will appear interchangeable from an external perspective. However, from the perspective of a software engineer, better structured code is *considerably easier* to understand, maintain and extend. In order to work effectively as part of a development team, with source code that remains in use for several years or longer, a software engineer cannot afford to ignore non-functional qualities of code.

I believe that software engineering students would benefit if greater academic emphasis was placed upon this important aspect of software development. As one example, a taught module with a significant focus on non-functional qualities of code could fill this gap, with the marking scheme for programming assignments heavily weighted upon the *structural attributes* of the code submitted rather than upon its external functionality alone. In addition to demonstrating qualities associated with well-factored code, such a module could also teach code refactoring techniques, to provide students with the necessary knowledge to improve the structure of *existing* code.

Of course, it is impossible to fully simulate an industrial environment in the realm of academia. Although university can provide a solid grounding in many fundamental areas of software development, it cannot prepare a student for every eventuality due to the enormous diversity of the field. Inevitably, software engineers are required to continue learning and improving their skills throughout their careers, long after their university education has ended. However, I do believe that a greater academic emphasis on non-functional aspects of software architecture such as maintainability and extensibility would provide a significant advantage to many graduates at the start of their careers, due to the great importance of these qualities in the field of commercial software development.

Bibliography

[1] Alexander, C (1977). *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press.

[2] Arlow, J & Neustadt, I (2009). *UML 2 and the Unified Process.* 2nd ed. Addison-Wesley Professional. p5.

[3] Bloch, J (2008). *Effective Java.* 2nd Edition. Prentice Hall.

[4] Brown, W.J. et al (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* Wiley. p119.

[5] Ericsson (2010). *Company information.* Available: `http://www.ericsson.com/thecompany`. Last accessed 5th Dec 2010.

[6] Fowler, M et al (1999). *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional.

[7] Gamma, E & Helm, R & Johnson, R & Vlissides, J. M (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional.

[8] Ibid., p63.

[9] Ibid., p107.

[10] Ibid., p127.

[11] Ibid., p175.

[12] Ibid., p185.

[13] Ibid., p233.

[14] Ibid., p293.

[15] Ibid., p315.

[16] Ibid., p331.

[17] Jones, C (June 2008). *Measuring Defect Potentials and Defect Removal Efficiency.* Crosstalk, The Journal of Defense Software Engineering.

[18] McConnell, S (2004). *Code Complete: A Practical Handbook of Software Construction.* 2nd Edition. Microsoft Press.

[19] Myers, G.J (2004). *The Art of Software Testing.* 2nd Edition. Wiley.

[20] Odersky, M & Spoon, L & Venners, B (2011). *Programming in Scala (eBook Version).* 2nd Edition. Artima Inc. p309.

[21] Ibid., p57.

[22] Ibid., p110.

[23] Oracle (2010). *ThreadPoolExecutor (Java 2 Platform SE 5.0).* Available: `http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ThreadPoolExecutor.html`. Last accessed 5th Dec 2010.

[24] Patton, R (2005). Software Testing. 2nd Edition. Sams. p24.

[25] Ibid., p9.

[26] Sutter, H (2005). *The free lunch is over: A fundamental turn toward concurrency in software.* Dr. Dobbs Journal, 30(3).

[27] Venners, B (2009). *Twitter on Scala.* Available: `http://www.artima.com/scalazine/articles/twitter_on_scala.html`. Last accessed 29th Jan 2011.

[28] Virki, T (2010). *Ericsson market share jumps in Q4: Dell'Oro.* Available: `http://www.reuters.com/article/idUSTRE61G0DS20100217`. Last accessed 5th Dec 2010.