UiO : **Department of Informatics**
University of Oslo

A Possible Composite Design Pattern for Immature REST APIs

A Study of Maintainability, Reusability, and Testability

Martin Røed Jacobsen

Master's Thesis Autumn 2016

Abstract

Software design patterns are formalized best practices, which provide reusable solutions to commonly occurring design problems in a given context in software engineering.

The aim of this study was to investigate how software design patterns can be combined to provide a wait-free abstraction layer over REST APIs on the low end of service maturity.

We present an implementation of a possible “composite pattern” in C$^\sharp$ and perform static analysis of the example source code to measure its maintainability, reusability, and testability. Thus, we collect five software quality metrics: Coupling Between Objects (CBO), Cyclomatic Complexity (CC), Depth of Inheritance (DIT), Lines of Code (NLOC) and Maintainability Index (MI).

All of the classes involved in the implementation receive a very good score on all metrics.

Our research suggests that this particular combination of design patterns is very useful, and that it may be a design pattern by itself. However, it has to be identified in other independent and unrelated systems for that to be true. Note that it may be challenging to integrate this solution into existing code bases due to its considerable structural requirements.

Acknowledgements

I would like to extend my sincerest gratitude to Wan Thai Foods Industry for manufacturing the dirt cheap, yet superb instant noodles, YumYum (chicken flavor), thus contributing to my continued existence for the duration of this thesis.

Thanks to Orona Norway for providing Ole Johan Dahl's house (OJD) with the slowest elevators in the combined history of mankind. They have been an inexhaustible and invaluable source of conversation, and has been an inspiration for future research into elevator firmware.

Thanks to the entire 10th floor of OJD at the University of Oslo, the Programming & Software Engineering (PSE) research group, for their encouragement and for being a great bunch of people. A special thanks to Stein Krogdahl for keeping the coffee machine in mint condition, and for teaching me about its mysterious ways and secrets.

Finally, and most importantly, I would like to thank my ~~life coach~~ supervisor, Eric Jul, for being a guiding beacon through these uncertain times.

Blindern, August 3, 2016
Martin Røed Jacobsen

Contents

Abstract **i**

Acknowledgements **iii**

1 Introduction **1**
- 1.1 Motivation 2
- 1.2 Goals 3
- 1.3 Approach 3
- 1.4 Criteria for Solution 3
- 1.5 Work Done 3
- 1.6 Evaluation 4
- 1.7 Results 4
- 1.8 Contributions 4
- 1.9 Conclusion 4
- 1.10 Outline 5

I Background **7**

2 Introduction **9**
- 2.1 Outline 9

3 Representational State Transfer **11**
- 3.1 Introduction 11
- 3.2 The Richardson Maturity Model 12
- 3.3 Hypermedia 13
- 3.4 Constraints 14
  - 3.4.1 Client-Server 14
  - 3.4.2 Stateless 14
  - 3.4.3 Cache 14
  - 3.4.4 Uniform Interface 15
  - 3.4.5 Layered System 15
  - 3.4.6 Code-On-Demand 15
- 3.5 Create Read Update Delete 16
- 3.6 Summary 16

4 Software Design Patterns **17**

4.1 Introduction 17
4.2 History 17
4.3 Definition 18
4.4 Creational 19
4.5 Structural 19
4.6 Behavioural 19
4.7 Used Patterns 20
4.7.1 Command 20
4.7.2 Factory Method 20
4.7.3 Observer 21
4.8 Summary 21

5 ReisAPI from Ruter **25**
5.1 Introduction 25
5.2 ReisAPI 26
5.3 Summary 27

6 Windows Communication Foundation **29**
6.1 Introduction 29
6.2 Sample Code 30
6.3 Summary 30

7 Summary **33**

II A Potential Composite Design Pattern **35**

8 Introduction **37**
8.1 Outline 37

9 Performer **39**
9.1 Intent 39
9.2 Basis 39
9.3 Motivation 39
9.4 Applicability 40
9.5 Structure 40
9.6 Participants 42
9.6.1 Command and Factory Method 42
9.6.2 Observer 43
9.7 Collaborations 44
9.8 Consequences 44
9.9 Implementation 45
9.10 Sample Code 46
9.10.1 Consumer 47
9.10.2 Performer 47
9.10.3 Request 47
9.10.4 Response 48

9.10.5 SimplePerformer 50

10 Summary 55

III Evaluation 57

11 Introduction 59
11.1 Windows Communication Foundation 60
11.2 Outline 60

12 Software Quality Metrics 61
12.1 Introduction 61
12.2 Functional Quality 62
12.3 Structural Quality 62
12.4 Complexity Metrics 62
12.4.1 Lines of Code 63
12.4.2 Halstead Complexity Measures 64
12.4.3 McCabe’s Cyclomatic Complexity 65
12.4.4 Coupling Between Objects 66
12.4.5 Depth of Inheritance 66
12.5 Measuring Maintainability 67
12.5.1 Oman’s Maintainability Index 67
12.6 Criticism 68
12.7 Summary 69

13 Measurements 71
13.1 Introduction 71
13.2 Base Classes and Interfaces 77
13.2.1 Interfaces 77
13.2.2 Base Classes 77
13.2.3 Concrete Classes 78
13.3 Subclasses of BaseRequest 79
13.4 Subclasses of BaseResponse 80
13.5 Summary 80

14 Summary 81
14.1 Maintainability 81
14.2 Reusability 82
14.3 Testability 82
14.4 Size 82

15 Conclusion 83
15.1 Summary 83
15.2 Main Contributions 84
15.3 Further Work 85
15.3.1 Generic Base Classes 85

15.3.2 Identify Performer in Other Systems 85
15.3.3 Pluggable Representation Formats 85
15.3.4 Reduced Dependence on Type Checking 86
15.3.5 Transparent Authentication Mechanism 86
15.3.6 Use in Server Applications 86

List of Figures

3.1 The REST technology stack 13

4.1 UML diagram of Command 20
4.2 UML diagram of Factory Method 21
4.3 UML diagram of Observer 22
4.4 Diagram of Observer 22

5.1 Facsimile from the `GetTravels` documentation 27

6.1 Using WCF with REST 30

9.1 UML diagram of Performer 41
9.2 `IConsumer` and `AbstractConsumer` 47
9.3 `IPerformer` and `AbstractPerformer` 48

15.1 Performer in the context of a server 87

List of Listings

6.1 A data contract in C♯ using WCF 31
6.2 Parsing an object in C♯ using WCF 31
9.1 `IConsumer` in C♯ 47
9.2 `AbstractConsumer` in Python 47
9.3 `IPerformer` in C♯ 48
9.4 `AbstractPerformer` in Python 48
9.5 `Request` in C♯ 49
9.6 `Request` in Python 49
9.7 `Response` in C♯ 50
9.8 `Response` in Python 50
9.9 `SimplePerformer` in C♯ 51
9.10 `SimplePerformer` (`Start`) in C♯ 52
9.11 `SimplePerformer` (`ThreadMain`) in C♯ 53
9.12 `SimplePerformer` in Python 54
13.1 The `GetTravels` constructor 79
15.1 The `GetStop` class using generics 85

List of Tables

3.1 Richardson Maturity Model (RMM) 12
3.2 Create Read Update Delete (CRUD) 16

4.1 Classification table for software design patterns 19

5.1 Modules in ReisAPI 26

12.1 Overview of Software Quality Metrics 63
12.2 Maintainability Index in Visual Studio® 68

13.1 Basic class level measurements 72
13.2 Basic method level measurements 72
13.3 Request classes measured on the class level 73
13.4 Request classes measured on the method level 74
13.5 Response classes measured on the class level 75
13.6 Response classes measured on the method level 76

1 Introduction

Computer programs, or “software”, are in many ways similar to buildings and vehicles in that they are a composition of different parts from different manufacturers and places. Houses vary widely in both size and design, but they all have a lot in common. They need doors, lights and windows, among other things. An architect may decide that a particular window should have a round shape, or that of an arch, but the architect cannot spend valuable time inventing new windows with every new house. Instead, the architect has a book with thousands of patterns, to pick and choose from. In a sense, one could say that the house is a patchwork of ideas borrowed from others. What makes the house unique is the way in which it combines these ideas, or “design patterns”, that it is comprised of.

An architectural design pattern for a particular style of windows may describe how square shaped bricks are laid out with only their corners touching, in the style of a diamond, to form an arch. Variations of this design pattern can be reused in the context of different buildings, again and again, wherever a window in the shape of an arch is desired. It is a timeless solution to a timeless problem.

Software engineering is a discipline concerned with the design and construction of not buildings, but computer programs. Whereas an architect carefully selects doors and windows to achieve a harmonious look and feel, software engineers make abstract decisions regarding the representation of “things” in a program, and how these things interact with each other, like the gears of a clock. A car is typically comprised of four wheels, a steering wheel and an engine. Those are isolated components that, in principle, can be replaced entirely without changing anything else in the vehicle. One could switch from a diesel engine to an electric one, but the car would still be a car, retaining its intended functions. This is the kind of modularity that software engineers try to reproduce in computer programs, using collections of design patterns.

A design pattern is not something that is “invented”, rather, it is discovered by observing the past work of others. It is a way of recording experience so that ideas can be named, discussed and reused in other places, without having to be rediscovered. In this thesis, we take a closer look at one combination of three “traditional” software design patterns. We discovered this particular combination while developing a program that needed to issue HTTP requests without blocking or using callback methods. We believe that we may have come across a design pattern, and here we describe how it works with a concrete example.

1.1 Motivation

Consider a computer program that can tell you how to travel between two geographical locations: a travel planner. The program displays a window that consists of four parts; a place to type in the destination, a list of possible routes, a map to visualise the selected route, and a list of interesting things to do at the selected destination.

The program does not possess knowledge of every destination in the world, or any destination at all, for that matter. What it does have, is the ability to obtain this information from another source, over the Internet, when it is needed. Due to many factors beyond your control, searching can be slow and you may have to wait for the answers to arrive. The program “locks up”, and ignores your attempts at using it. This behaviour is caused by the fact that the program is only capable of doing one thing at a time, and right now, it is waiting for information from some other computer on the Internet. One way to solve this problem is to change the program so that it can perform more than one task at the same time. This is called multithreading.

Multithreading is complicated, and hard to do right. A program with two “threads” is comparable to a human mind, that can maintain two unrelated lines of thought simultaneously. A program can run flawlessly for years, even if the programmer makes a mistake. Even so, no matter how unlikely an error is to manifest, some day it may happen for seemingly inexplicable reasons. Tracking down the source of such an error may be close to impossible if the program is very old. A failure in a travel planner is unlikely to cause economical damage, but data loss may occur, which if not a disaster, is an annoyance. Restricting the use of multiple threads to a single, generic, component in the program code can prevent some of these problems.

We wish to enjoy the benefits of multithreading in our travel planner, without cluttering our program code with code that supports multiple threads. To achieve this, we will make sure that the code in question is contained in one central place in the application, in a component that can be reused by other, similar, programs at any point in the future.

1.2 Goals

Our goal is to demonstrate that this software design pattern combination may be a pattern by itself. In other words, we describe a potential pattern for wait-free issuance of HTTP requests (“fire and forget”). We use nothing other than commonly available features in object-oriented languages, and well known design patterns.

1.3 Approach

We investigate how well a particular combination of software design patterns work together, and show how they can be used to design a wait-free abstraction layer for HTTP requests in the context of a basic REST API. We create a small travel planner program (using the example code in this thesis) that communicate with ReisAPI, a RESTful travel planner provided by Ruter, a common management company for public transport in Oslo and Akershus in Norway. Visual Studio® 2015 Enterprise is used for both development and for the collection of software quality measurements, of which we perform a systematic evaluation.

1.4 Criteria for Solution

The potential design pattern is tested in the field through our travel planner program. Our primary concerns are that the maintainability, reusability, and testability of the program code are within acceptable limits, but we also provide a subjective opinion on whether or not the pattern “feels” good to work with.

1.5 Work Done

We have documented and evaluated a possible design pattern that provides wait-free, HTTP-based communication by taking advantage of nothing other than commonly available features in object-oriented languages, and some existing and well known design patterns. Using the sample code included in this thesis, we developed a small travel planner program based on the possible design pattern. The program communicates with ReisAPI, a Norwegian travel planning system. We have measured the maintainability, reusability, and testability of our solution using development tools from Microsoft®, and we have given our a subjective opinion on how the potential pattern is to work with.

1.6 Evaluation

We have developed a computer program based on a possible design pattern, and we have performed an evaluation of the maintainability, reusability, and testability of the code by collecting five software quality metrics.

1.7 Results

The potential composite pattern introduces minimal complexity in program code, and as such it provides a simple interface to parallel programming (multithreading for "free"). The test program scores very well on all five software quality metrics, and looks to be fairly easy to extend (and reduce). There is, however, room for improvements that can make it even better. We have also identified another potential area of applicability for it on the server-side.

1.8 Contributions

Our contribution is a potential new design pattern called "Performer" that can be implemented in any programming language with support for object-orientation and multithreading. A client program was developed for ReisAPI, for the purpose of evaluating a concrete implementation of the potential pattern, but the program itself is not an important part of the thesis, or our contribution.

1.9 Conclusion

This thesis has shown that the potential design pattern works well, is relatively simple to implement, and is a "good" way to communicate with services on the Internet. However, the test program was designed from the ground up with the pattern in mind, and as such, we make no claim that it is easy to integrate it into an existing program.

1.10 Outline

This thesis is divided into three separate parts: Background, A Potential Composite Design Pattern, and Evaluation. Each part is divided into a number of chapters and sections, that may be read individually or in the order they are presented, of which we recommend the latter.

- **Part I: Background**
  In this part we offer a short introduction to the concept of design patterns in software engineering, and we look at how the World Wide Web (WWW) can be leveraged as an application platform using REST. Furthermore, we take a brief look at ReisAPI, a travel planner for public transportation in Norway, and WCF, which is an abstraction layer for communicating services.

- **Part II: A Potential Composite Design Pattern**
  This part introduces the potential Performer composite software design pattern, formatted in the style of GoF, with sample code in the C$^\sharp$ and Python programming languages.

- **Part III: Evaluation**
  Provides a brief presentation of various software quality metrics collected from the example program, and a discussion regarding the maintainability, reusability, and testability of the code. We end this thesis by reviewing everything we have learned, and provide some suggestions on future improvements, and an alternative use for the Performer pattern on servers.

Part I

Background

2

Part I: Background

Introduction

In this part, we introduce four topics that together form the basis of this thesis. We begin by taking a look at the architectural style REST, which is heavily influenced by the WWW. Then we define what a “design pattern” really is in the context of software engineering (and how it differs from an “algorithm”), followed by a narrow selection of three design patterns used later in this thesis. We conclude this part by providing a short summary of the capabilities of the ReisAPI travel planning API (used in our example application), and a technical chapter explaining WCF, the technology we use to process response data.

2.1 Outline

- **Chapter 3: Representational State Transfer**
  REST is an architectural style in software engineering, that uses the “Web” as highly scalable and flexible application platform.
- **Chapter 4: Software Design Patterns**
  Software design patterns in software engineering contribute to reusable object-oriented software design by providing solutions to some of the problems that frequently occur during the development of software applications.
- **Chapter 5: ReisAPI from Ruter**
  ReisAPI is a Norwegian travel planning service inspired by the REST architectural style, that provides information about public transportation in the capital of Norway.

- **Chapter 6: Windows Communication Foundation**
  WCF is an abstraction layer for services that need to communicate (typically) over a network.

3

Part I: Background

Representational State Transfer

3.1 Introduction

In this chapter, we introduce a way of organising the internal and external components in a very large (Internet scale) system, and a way of classifying such systems according to their usage of the features offered by the web.

REST, or Representational State Transfer, is described by many as the architectural style of the WWW. It was formally derived by Roy Fielding in 2000 as part of his doctoral dissertation ‘Architectural Styles and the Design of Network-based Software Architectures’. His research suggests that the architectural constraints provided by REST, when applied as a whole,

> “emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.” [Fie00]

What this means is that REST provides a set of tools that can be used to build highly scalable systems with generic interfaces that, in principle, can be taken advantage of with only a generic understanding of “hypermedia”. In this chapter, we provide a brief summary of the software engineering principles that guide the implementation of RESTful services, and the interaction constraints that must be applied to retain those principles.

3.2 The Richardson Maturity Model

Level	Description
3	Supports the notion of “hypermedia as the engine of application state” [Fie08]. Resource representations contain links to other resources, leading consumers through a sequence of resources which causes application state transitions.
2	Hosts numerous resources and supports several HTTP methods. This service level introduces CRUD services, which we cover in section 3.5 on page 16.
1	Employs many logical resources, yet supports only one HTTP method (typically `GET`). Operation names and parameters are inserted into a URI and then transmitted to a remote service.
0	Offers a single large and complex resource (URI) through which all interactions are tunnelled, and supports only one HTTP method.

Table 3.1
The three (four) levels of service maturity according to the Richardson Maturity Model (RMM). Level 0 *is not RESTful.*

During his talk ‘Justice Will Take Us Millions Of Intricate Moves’ at QCon San Francisco in 2008, freelance consultant and writer Leonard Richardson proposed a classification of services on the Web that promotes three levels of service maturity based on how much a service employs URIs, HTTP and hypermedia, recognised by many as the three core technologies of the Web.

The diagram in fig. 3.1 on the next page shows the maturity levels as a layered structure, in which each layer builds on the layers below. At the top of the diagram are the most mature services, which employ all of the technologies in the layers below, whereas the least mature services rest at the bottom [WPR10]. Table 3.1 presents the three levels of service maturity. The fourth level, level 0, includes services that use too few web features to be considered RESTful.

The Richardson Maturity Model (RMM) assumes that most of the constraints we covered in section 3.4 on page 14 are fulfilled because they are implied by the Web architecture. Level 0 services are not considered RESTful [WPR10].

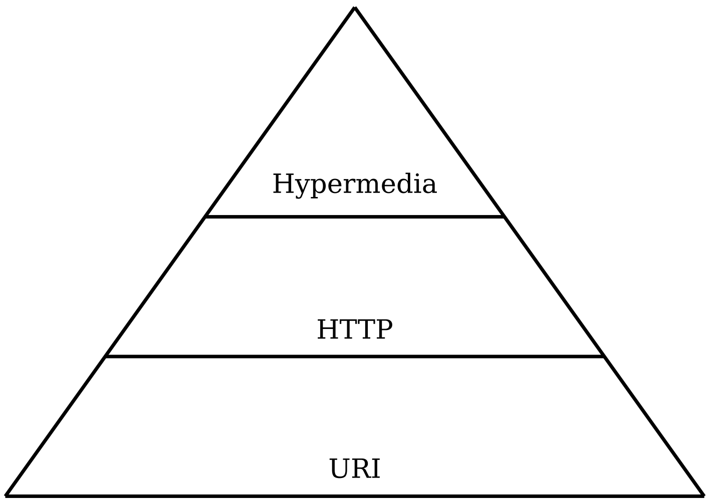

Figure 3.1
Each level in Richardson's model builds upon the levels below.

3.3 Hypermedia

Hypermedia as the Engine of Application State (HATEOAS) is a distinguishing constraint of the Representational State Transfer (REST) architectural style, and is what differentiates it from other architectures. HATEOAS corresponds to level three in the RMM (section 3.2 on page 12). In our limited experience, most services seem to be at level one or two in the RMM.

Whereas many architectural networking patterns communicate using static interfaces, a REST client receives hypermedia dynamically from the server, and uses its embedded hyperlinks to discover other parts of the service. Only the protocol and the entry point of the service are fixed (just like web pages visited by humans). As the client explores the service, it selects links from the hypermedia it receives and transitions through states directed entirely by the application on the server, in the same way a state machine does. Thus, the "engine" (E) in HATEOAS is hypermedia.

Similarly, to use the "Example" service with a browser, all we (a human user) need to know is that it is available using HTTP at example.com. Visiting the service presents us with buttons and menus that we can use to discover its functionality, without having to look up specific URIs in the documentation, just as HATEOAS requires. Hypermedia was originally designed to be used by humans, though it turns out that machines are just as good at following state machine protocols.

A truly RESTful service should be usable both by generic REST clients, but also by human readers entering the service through the same URI. A human user may receive executable code (see section 3.4.6

on the next page) from the service to aid in the presentation of the page in a web browser (e.g. JavaScript), but that is not likely to be a useful feature for a computer program.

3.4 Constraints

When people refer to an API as being “RESTful”, it appears to us that in many cases, what they really mean is an API where JSON or XML messages are delivered to some predefined URL, with the HTTP protocol as its transport. However, this is far from what REST was intended to be [WPR10]. This section is a summary of Fielding’s six RESTful constraints, as he defined them.[1]

3.4.1 Client-Server

Business logic and data is kept separate from the user interface. This separation of concerns improves the **portability** of the user interface across multiple platforms and the **scalability** of the server by simplifying its components. The user interface is the “client”, whereas business logic and data resides on the “server”.

3.4.2 Stateless

Session state is kept entirely on the client and the server is stateless. Each request includes all state information necessary to understand and fulfil it. This constraint improves **visibility** by allowing monitoring systems to understand the entire nature of a request without having to keep track of earlier or future requests. Furthermore, the **reliability** is improved because recovery from partial failure becomes easier. Resources are quickly freed because the server does not have to store state information, thus improving **scalability**.

3.4.3 Cache

Response data must be labelled as to whether or not it can be cached. A cacheable response can be reused by a client for later, equivalent requests. Unlabelled retrieval responses are cacheable by default, while other responses are non-cacheable unless explicitly marked as such. The cache constraint can contribute to improved **efficiency**, **scalability** and perceived **performance** by reducing the average latency. However, **reliability** may suffer, if a cache contains stale data.

[1]Refer to Chapter 5 in his dissertation [Fie00] for additional information.

3.4.4 Uniform Interface

Interfaces between components must be uniform and general in nature. Implementations are decoupled from the services they provide. Data is transferred using standardised formats. This constraint simplifies the overall system architecture and the **visibility** of interactions between components is improved. Furthermore, in order to obtain a uniform interface, Fielding suggests that four interface constraints are needed:

1. Uniform identification of resources (URIs).

2. Manipulation of resources by altering their representations.

3. Messages are comprehensive and independent (stateless).

4. Application state is driven by hypermedia.

3.4.5 Layered System

An architecture can be composed of any number of hierarchical layers. Any number of intermediary layers can be added between the client and the server without the knowledge of either. No component has any knowledge about any layer beyond the one with which they exchange data. This layering can be used to improve system **scalability** by introducing load balancers and caches. However, layering introduces overhead and latency, which may reduce the perceived performance of the system. Then again, this disadvantage can be reduced by appropriate use of caches. Additionally, layering allows data to be filtered through firewalls and other monitoring systems, thus allowing organisations to enforce **security** policies.

3.4.6 Code-On-Demand

The functionality of the client can be extended by downloading code. This is an optional constraint within REST, because it reduces **visibility**. Code in the form of applets or scripts can be downloaded and executed, thus improving the functionality of the client and overall **system extensibility**. For example, an application can deliver JavaScript to aid browsers in the presentation of data. This constraint — if implemented — should be designed with the understanding that it may be disabled because of external factors beyond the control of the system.

Operation	HTTP	Description
Create	`POST`	Create a new resource and receive its new URI in the HTTP response.
Read	`GET`	Request the state of a resource identified by URI.
Update	`PUT`	Replace a resource identified by URI with the representation included in the request.
Delete	`DELETE`	Logically remove a resource identified by URI.

Table 3.2
The Create Read Update Delete (CRUD) pattern treats resources as relational databases.

3.5 Create Read Update Delete

CRUD is a pattern for treating resources with the same operations available in relational database applications: `POST`, `GET`, `PUT`, and `DELETE`. See table 3.2 for a short description of the four available operations. Because of their lack of support for hypermedia, CRUD services reach level two on the RMM [WPR10] covered in section 3.2 on page 12.

3.6 Summary

REST is an architectural style that can be used to build highly scalable and generic systems. Originally developed to support the WWW, though in recent years its popularity has increased as a way to deliver services, not only to human consumers, but to computer programs.

The RMM is a way of classifying the maturity of services based on REST. There are four levels of service maturity, of which level 0 indicates that a service is using the web merely as a transport, and level 3 indicates that a service uses most of the features offered by the web. We believe that “immature” is a suitable term for describing any REST Application Programming Interface (API) below level 2.

4

Part I: Background

Software Design Patterns

4.1 Introduction

An important feature of software design patterns is that they have names that allows for discussion on a higher level. Instead of describing the interactions of a set of objects, developers can simply refer to that specific design pattern, and other developers will know the interactions from experience.

This chapter offers a brief look at the historical origin of design patterns as a concept, and outlines the basic structure of design patterns in the context of software engineering. We conclude this chapter by presenting the three main categories of software design patterns, and three concrete patterns: Command, Factory Method, and Observer.

4.2 History

The concept of design patterns first appeared in architecture following the publication of *The Timeless Way of Building* [Ale79] in 1979. The book was written by the architect Christopher Alexander, which at the time held the position Professor of Architecture at the University of California, Berkeley. Alexander proposed a new philosophical theory that architecture and design relies on quality through the deliberate configuration of design patterns, which has had an enormous influence in the software engineering industry [Sal].

The usage of patterns gained popularity in the software industry when the Gang of Four (GoF) published their book *Design Patterns: Elements of Reusable Object-Oriented Software* [Gam+94] in 1994, as a collection of recorded experience in object-oriented software engineering. It is intended to be used as a catalogue with its 23 fundamental software design patterns that are named, explained and evaluated systematically.

4.3 Definition

Whereas an “algorithm” is a finite collection of unambiguous instructions that uses an input to perform a specific task (resulting in an output), a “design pattern”, like a blueprint, is a general way of structuring program code to express the relationship between components and their roles. In *A Pattern Language* (1977), Christopher Alexander wrote that a pattern

> “describes a problem that occurs over and over again”, and that it “describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” [Ale77]

According to the GoF, a design pattern is a design that has been successfully applied more than once in unrelated systems. What this means is that it is impossible to wilfully create a design and call it a pattern. The design has to be discovered and used independently in more than one place. In other words, you can not “create” a design and claim that it is a pattern.

Whereas design patterns in architecture concerns quality regarding buildings and cities, a software design pattern addresses commonly occurring problems in software engineering, describing the interaction among customized objects and classes in a particular context [Gam+94]. Applying a software design pattern typically carries several advantages, such as decreased coupling between objects, improved extensibility and encapsulation of implementation details.

In this thesis, we concern ourselves with only a small subset of the fundamental software design patterns recorded by GoF in [Gam+94]. However, other families of more specialised patterns concerning, among others, game design, computational and enterprise applications do exist. The design patterns we concern ourselves with are classified by *purpose* and *scope*. Patterns can have either a *creational*, *structural* or *behavioural* purpose. The scope indicates whether the pattern applies primarily to classes, or to instances of classes (objects). See table 4.1 on the facing page for a classification table.

		Purpose		
		Creational	**Structural**	**Behavioural**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Table 4.1
Classification table for software design patterns [Gam+94].

4.4 Creational

Design patterns concerning the way objects are instantiated and configured at runtime (i.e. Factory Method) are referred to as creational patterns. Creational *class* patterns delegate the responsibility of object instantiation to subclasses, whereas creational *object* patterns involve another otherwise unrelated object (i.e. Abstract Factory) [Gam+94].

4.5 Structural

Design patterns concerning the way that classes are composed and assembled are referred to as structural patterns. The structural *class* patterns compose classes by using inheritance to promote reusability, while structural *object* patterns assemble objects by having them refer to instances of each other, instead of relying on inherited functionality [Gam+94].

4.6 Behavioural

Design patterns that affect the control flow and interaction between objects in complex algorithms are referred to as behavioural patterns. Behavioural *class* patterns describe how inheritance can be used to implement algorithms and desired flow of control. Behavioural *object* patterns perform tasks that require more than one object, by describing the way the objects cooperate [Gam+94].

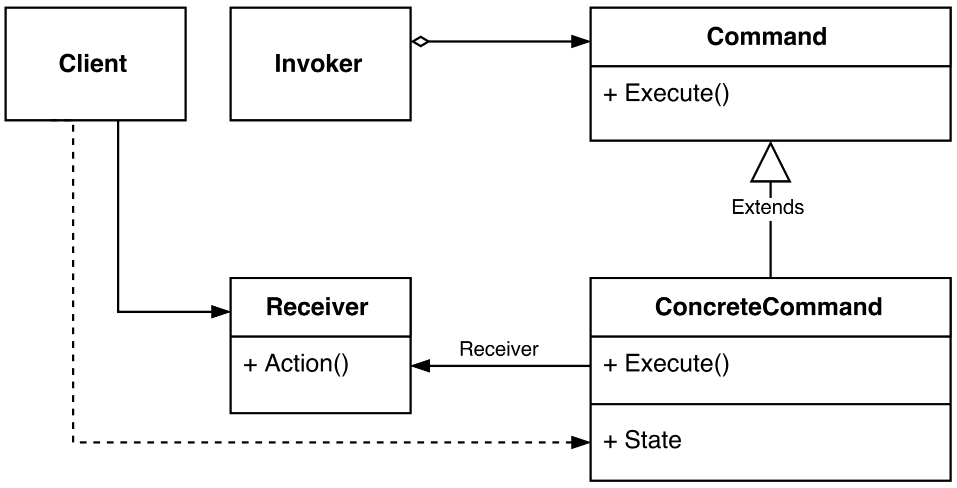

Figure 4.1
UML diagram of Command, which allows the creator of a request to delegate the responsibility of executing it to another class [Gam+94].

4.7 Used Patterns

In this thesis, we describe a potential composite software design pattern that is based on the Command, Factory Method and Observer patterns. Command is used to encapsulate requests to API URIs, while Observer invokes the Factory Method provided by the Command to parse the response appropriately. It then proceeds to propagate the response around the application.

In the following sections, we provide a very brief explanation of the aforementioned design patterns (see [Gam+94] for extensive information on them).

4.7.1 Command

Command allows us to encapsulate a method as an object. Arguments are passed to the constructor of the Command object, which stores them and forwards them to the appropriate methods when executed by an invoker. Refer to the Unified Modeling Language (UML) diagram in fig. 4.1 for an illustration of the structure required to implement Command.

4.7.2 Factory Method

Factory Method is a pattern in which subclasses of a “creator” class define the type of object they instantiate and return to the caller. In a strongly typed language, such as C$^\sharp$, this return type must conform

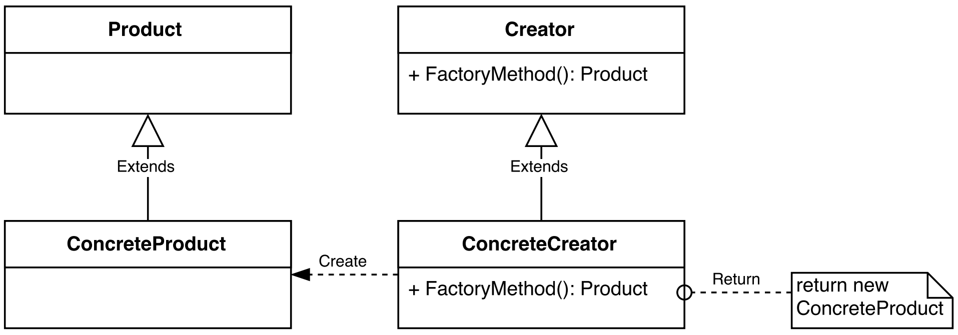
Product
Creator
+ FactoryMethod(): Product
Extends
Extends
ConcreteProduct
Create
ConcreteCreator
+ FactoryMethod(): Product
Return
return new ConcreteProduct

Figure 4.2
UML diagram of Factory Method, a pattern that forwards the responsibility of choosing which class to instantiate, to subclasses [Gam+94].

to an interface or a common base class, which can be subclassed and specialised. A UML diagram that illustrates the structure of Factory Method can be found in fig. 4.2. In this thesis we use the pattern to construct response objects from the data that is generated by issuing requests to an API, by having each request class implementing a factory method to handle replies.

4.7.3 Observer

Observer defines a one-to-many dependency between objects so that when one object (subject) performs a state change, all dependents (observers) are updated automatically. Observer is often applied when different presentations can depict the same information, such as a spreadsheet or different kinds of charts. They have no knowledge of each other, yet their behaviour suggest that they do. When information is altered in the spreadsheet, a bar chart may reflect those changes immediately, and vice versa [Gam+94]. This is exemplified in fig. 4.4 on the next page. See fig. 4.3 on the following page for a UML diagram.

4.8 Summary

A design pattern is a named record of experience that presents a problem and describes a way to solve it by proposing class design, structure, and interaction, like a blueprint. The concept originated in architecture long before it became common in software engineering.

Patterns are discovered, and cannot be created wilfully. Once a design has been applied successfully in a number of systems, it may be that it is a pattern.

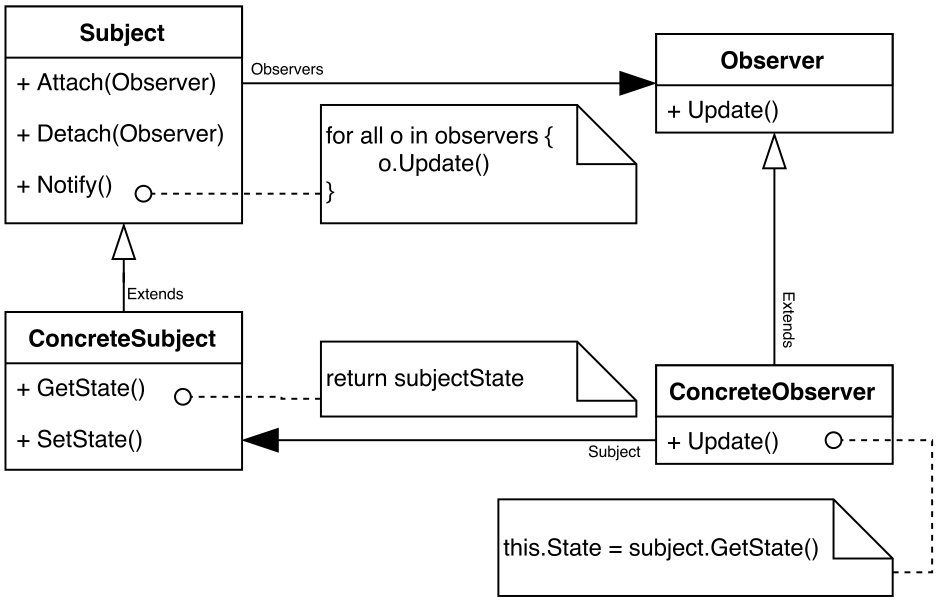

Figure 4.3
UML diagram of Observer, a pattern that is used to define a one-to-many dependency between objects [Gam+94].

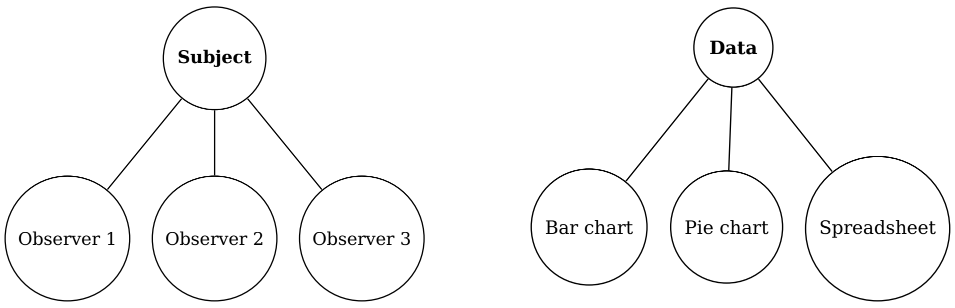

Figure 4.4
Diagram of Observer, a pattern that is used to update any number of objects that depend on the state of one, common, subject [Gam+94].

In this chapter we have described three software design patterns that are used in subsequent chapters: Command, Factory Method, and Observer.

5

Part I: Background

ReisAPI from Ruter

5.1 Introduction

The purpose of this chapter is to document a concrete example of an immature REST API, and as such ReisAPI could be replaced with any other similarly immature API. ReisAPI is a Norwegian travel planning service that uses some elements of the REST architectural style, and that provides information about public transportation in the capital of Norway.

Ruter is a common management company for public transport in Oslo and Akershus in Norway [Ruta]. Through the Labs initiative, they have made two REST APIs freely available [Rutc] for both commercial and private use (“ReisAPI” and “DeviAPI”) under the permissive Norwegian Licence for Open Government Data (NLOD).

ReisAPI provides an interface for access to a travel planner and various information retrieval resources. DeviAPI offers public transportation deviation information through the Service Interface for Real Time Information (SIRI) protocol based on the Web Services Description Language (WSDL) format, which we will not cover in this thesis. The remainder of this chapter will provide a brief overview of ReisAPI and its capabilities.

Module	Description	Resources
Favourites	Simpler interface to the StopVisit module.	1
Heartbeat	API availability checks.	1
Line	Detailed line information.	6
Meta	Information about other modules.	1
Place	Detailed stop information.	7
StopVisit	Departures in real- and fixed-time.	1
Street	Geographical data about houses and streets.	1
Travel	Travel planner.	3
Trip	Detailed information about trips.	1

Table 5.1
Summary of the modules in ReisAPI Modules and their resources.

5.2 ReisAPI

ReisAPI offers nine modules containing a total of 22 resources (methods) regarding travel planning and live traffic data. These modules are listed in table 5.1. The official documentation [Rutb] does not specify expected HTTP status codes, however, according to our observations anything other than `200 OK` should be treated as an error (keep in mind that the API is read-only, and that it supports only the `GET` method). This is included to provide context for the Evaluation chapter, and to serve as an example of how a level 1 API is commonly structured. It will not be referenced later.

reisapi.ruter.no/Favourites/ Contains only a single resource, which appears to be an alternative interface to the `GetDepartures` resource provided by the `StopVisit` module: `GetFavourites`.

reisapi.ruter.no/Heartbeat/ Provides the “Index” resource which returns the string “Pong” to indicate that the service is available. We recommend treating all other responses as an error.

reisapi.ruter.no/Line/ `Line` is one of the larger modules in ReisAPI, providing information about routes in public transport where a vehicle departures at a fixed interval (“lines”). The module includes six resources: `GetLines`, `GetLinesByStopID`, `GetDataByLineID`, `GetStops-ByLineId` (duplicate of the resource with the same name in the `Place` module), `GetLinesRuterExtended` and `GetLinesRuter`.

reisapi.ruter.no/Meta/ The `Meta` module provides the `Get-Validities` resource which always returns the earliest and latest point in time that ReisAPI has information about.

reisapi.ruter.no/Place/ The biggest module in ReisAPI is `Place`,

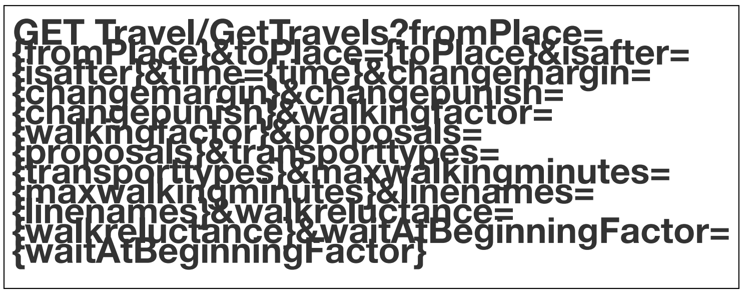
```
GET Travel/GetTravels?fromPlace=
{fromPlace}&toPlace={toPlace}&isafter=
{isafter}&time={time}&changemargin=
{changemargin}&changepunish=
{changepunish}&walkingfactor=
{walkingfactor}&proposals=
{proposals}&transporttypes=
{transporttypes}&maxwalkingminutes=
{maxwalkingminutes}&linenames=
{linenames}&walkreluctance=
{walkreluctance}&waitAtBeginningFactor=
{waitAtBeginningFactor}
```

Figure 5.1
The title of the `GetTravels` documentation page [Rutb] as rendered in the Google Chrome browser. The resource accepts 13 query parameters. Other ReisAPI resource are documented similarly.

with 13 resources. Six of the resources are, documented as “unofficial” and subject to change. In practice, the module contains 7 “stable” resources: `GetStop`, `GetPlaces`, `GetStopsByLineID` (duplicate of the resource with the same name in the `Line` module), `GetClosestStops`, `GetStopsByArea`, `GetSalePointsByArea`, and `GetStopsRuter`.

reisapi.ruter.no/StopVisit/ `StopVisit` is the name of a “departure” object in ReisAPI. It is also the name of the module that contains the `GetDepartures` resource, which provide information about departing transportation from a place.

reisapi.ruter.no/Street/ Provides the `GetStreet` resource, which returns all houses that are considered to be a part of a “street”.

reisapi.ruter.no/Travel/ Contains three resources, though only one which is documented: `GetTravels`. This resource may be considered the heart of ReisAPI, as this is the only resource that provides complete travel information given a departure location and a destination.

reisapi.ruter.no/Trip/ According to the ReisAPI documentation, the `GetTrip` resource provided by the `Trip` module returns information about a trip.

5.3 Summary

In this chapter we discussed a concrete example of an immature REST API. ReisAPI provides access to nine modules over HTTP at static URIs, of which only one provides the actual travel planning functionality (`Travel`). The API fulfils the requirements for level 1 on the RMM.

6

Part I: Background

Windows Communication Foundation

6.1 Introduction

In this chapter, we briefly present framework for building applications that exchange Simple Object Access Protocol (SOAP) messages over a network: Windows Communication Foundation (WCF). WCF is a very large and complex part of the .NET framework, and in this thesis we use only a tiny fraction of its features to process the JavaScript Object Notation (JSON) responses we receive from ReisAPI in our example program. The remainder of this chapter will focus on the WCF functionality we use explicitly.

All resources in ReisAPI can be represented in both JSON and Extensible Markup Language (XML), depending on the `Accept` HTTP request header provided by the client. The header can be set to either `application/json` or `application/xml`, and the server will respond accordingly.[1] What this means for us is that we need to parse JSON or XML to use the service, however, the code required to do this would impact the measurements that we perform in part III on page 59. Additionally, because parsing of representation formats is highly dependent on the libraries available in the environment one is working, we would like to avoid including this when measuring software quality metrics. For this reason, we use WCF.

[1]From our observations it seems that the default behaviour of the service is to respond with XML, if the specified value of the `Accept` contains an invalid representation format.

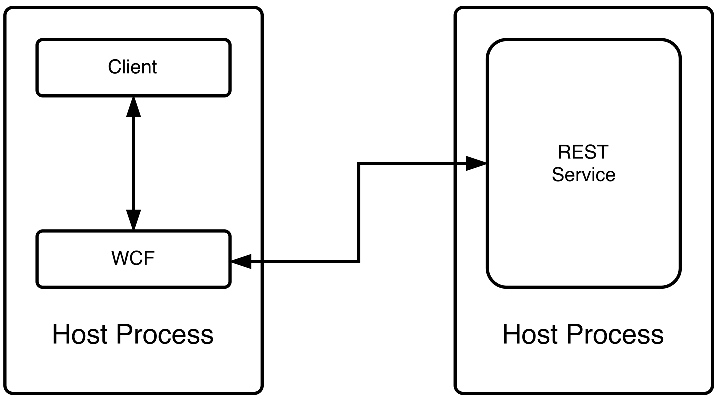

Figure 6.1
UML diagram of a basic REST client based on WCF, which provides interoperability with applications built on several other technologies, such as ASMX, Remoting and REST.

WCF was introduced by Microsoft® in 2007 and was known as "ADO.NET Data Services" until 2009, when it was renamed. WCF is a programming model for building service-oriented applications using the .NET Framework, and it has the ability to bypass any member variable protection level in .NET classes. As such, one can define a class entirely with private member variables or properties, but still use it with WCF. WCF, however, does not execute the object constructor upon instantiation of an object.

The advantage of using WCF with our ReisAPI client implementation is that we get most of the required parsing functionality for free and with minimal impact on the size of the code base. Consider section 6.2 for an example of how a data structure can be constructed to allow this form of parsing, and for a demonstration of the actual code that is required to create a C♯ object from a JSON source.

6.2 Sample Code

Listing 6.1 shows a data contract for the `Ruter.House` type in C♯ using WCF, and listing 6.2 shows how the response is parsed using a small amount of code.

6.3 Summary

WCF is an abstraction layer for communicating services. In this thesis, we use it for the purpose of decoding incoming JSON messages.

```
using System.Runtime.Serialization;

namespace Ruter
{
    [DataContract]
    public class House
    {
        [DataMember]
        public string Name { get; set; }

        [DataMember]
        public int X { get; set; }

        [DataMember]
        public int Y { get; set; }
    }
}

```

Listing 6.1
This C♯ example shows how a data contract for the `Ruter.House` type is defined by applying the `DataContractAttribute` and `DataMember-Attribute` attributes to the class and its members.

```
// Required: using System.IO;
//           using System.Net;
//           using System.Runtime.Serialization.Json;

DataContractJsonSerializer js = new DataContractJsonSerializer(typeof(House));

using (MemoryStream stream = new MemoryStream(bytes))
{
    return (House)js.ReadObject(stream);
}

```

Listing 6.2
This C♯ example shows how an instance of the `Ruter.House` type can be created from a JSON source to an object, using the data contract facilities available in WCF.

7

Part I: Background

Summary

In this part we have introduced four subjects that together form the basis of this thesis.

REST is an architectural style in software engineering that takes advantage of the “Web” as a highly scalable and flexible application platform. The degree of “RESTfulness” can be classified with the RMM, which indicates to what extent an application takes advantage of features offered by the web. A REST API is more than simple messages transported on top of HTTP, which is evident in the six constraints we discussed in section 3.4 on page 14: client-server, stateless, cache, uniform interface, layered system, and code-on-demand.

Software design patterns are blueprints that provide reusable solutions to commonly occurring problems in object-oriented software engineering. In this thesis, we take advantage of three such patterns: Command, Factory Method, and Observer.

ReisAPI is a Norwegian travel planning service that uses some elements of the REST architectural style, and that provides information about public transportation in the capital of Norway.

WCF is an abstraction layer for services that communicate (typically) over a network.

Part II

A Potential Composite Design Pattern

8

Part II: A Potential Composite Design Pattern

Introduction

In the following chapter, we explore a composite (potential) design pattern that we have named “Performer”. The pattern describes how most of the complexity related to multithreading and HTTP can be hidden inside a single object, thus keeping the code concise and simple in other parts of an application.

We cannot stress enough, however, that it is impossible to *invent* a new design pattern. It must be *observed* in multiple independent systems. Hence, this chapter is a discussion of something that may have the *potential* to become a design pattern, even though we do refer to it as a “pattern” from time to time.

8.1 Outline

The format of the chapter is based on the style used by GoF in [Gam+94]. Plenty of sample code written in C$^\sharp$ and Python 3 is provided, for which a basic understanding of object oriented programming in general should be sufficient.

- **Chapter 9: Performer**
  Performer combines the Command, Factory Method and Observer patterns to provide a multithreaded execution “engine” that accepts and executes encapsulated requests to REST services on behalf of other objects in an application.

9

Part II: A Potential Composite Design Pattern

Performer

9.1 Intent

Exploit the stateless nature of REST by encapsulating requests as independent, parameterised objects containing the smallest amount of data possible, and by performing requests in an object that knows how to exchange data with the relevant API. Allow any object to update its internal state based on the received data.

9.2 Basis

- Command
- Factory Method
- Observer

9.3 Motivation

Multithreading is often used to offload the thread that powers a user interface (usually the main thread). In simple programs, one thread is usually enough; however, if an action performed by the user can cause a long running operation to execute, then threading is a useful mechanism. If the main thread starts reading a large file from a spinning hard drive, or downloads data from the Internet using a slow connection, the user interface becomes unresponsive and the user may perceive the application as being slow. We can keep the user interface responsive by offloading the heavy lifting to a separate thread of execution. However, threading (and synchronisation) is hard to get right and can, if applied incorrectly, introduce numerous issues in systems that may be very hard to debug and understand.

In some contexts, more than one part of a system may be interested in the response from a certain kind of request, without knowing anything about the issuer of the request, or other interested objects. For example, a travel planner application may download data about all departures from a given geographical location. This data may be used in several parts of the application: displayed in a graphical timetable, logged to a file, and perhaps aggregated in a database and used for statistical analysis later. Perhaps a calendar event is created and distributed to a mailing list, or uploaded to a CalDAV service.

Much like a thread pool, a performer accepts work from others and executes it on their behalf. What differentiates the performer is that it implements domain knowledge of a specific REST API, and instead of invoking a callback function, it broadcasts the result to anyone subscribed to it.[1] Implementing threading and synchronisation in one place, instead of in any context where a long running operation may be performed, reduces unnecessary complexity and future maintenance efforts, and improves the potential for reuse. Hence the solution is to devise one object that takes responsibility for the execution of requests.

9.4 Applicability

Use Performer when multiple (potentially unrelated) objects depend on the outcome of a REST operation initiated by an arbitrary object.

9.5 Structure

See the UML diagram in fig. 9.1 on the next page.

[1]A callback function is a function that is passed as an argument to some other code with the assumption that it will be called some time in the future. Callback functions can be implemented in many different ways, among which are blocks (JavaScript), function pointers (C) and lambda expressions (C$^\sharp$, Java).

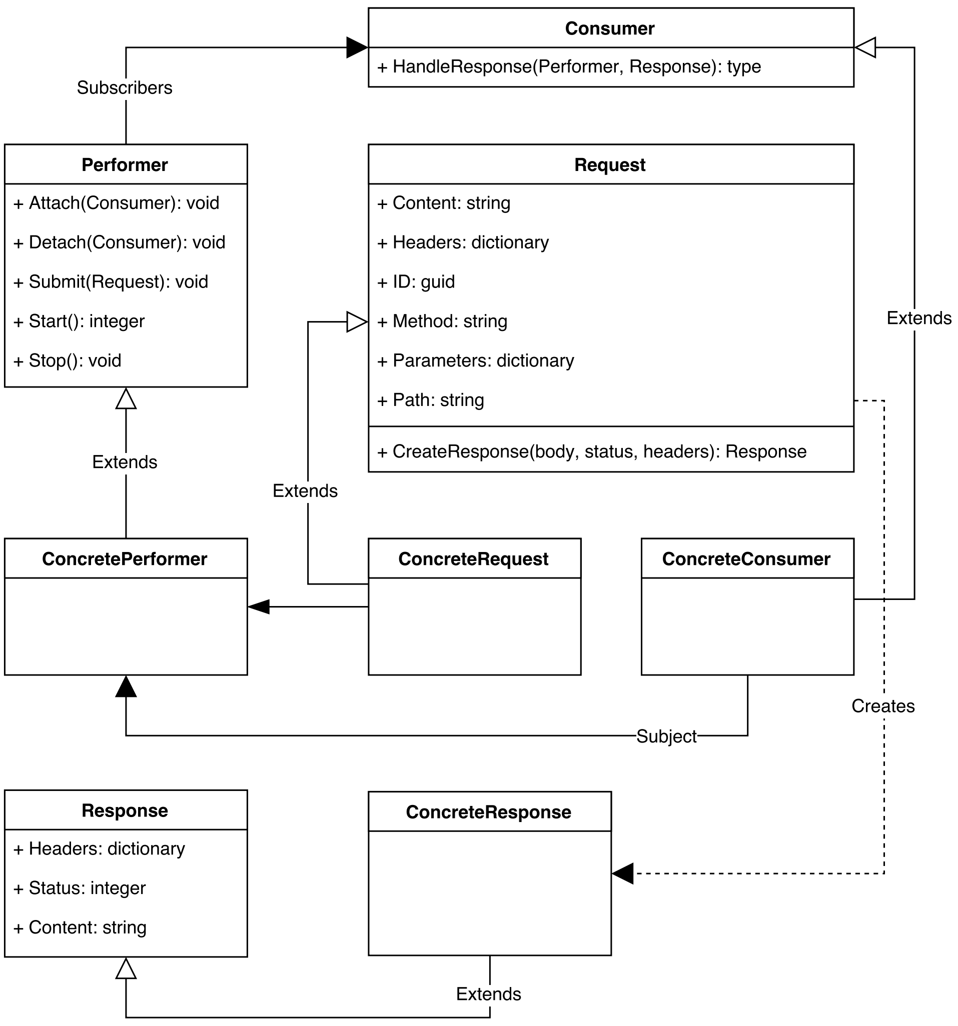

Figure 9.1
A UML diagram depicting the structure of Performer.

9.6 Participants

Performer is a composition of the Command, Factory Method, and Observer patterns, which we briefly present in section 4.7 on page 20. Command encapsulates requests, while Observer delivers the results after they have been parsed appropriately by the Factory Method implemented by each request (Command).

9.6.1 Command and Factory Method

Command Used to encapsulate HTTP requests as objects, thereby letting you parameterize clients with different requests. Requests can be queued, logged, discarded, or retried by a performer. New request types can be added without changing existing classes. Each request object implements a Factory Method that knows how to parse the expected response into a suitable response object.

Factory Method When an arbitrary object issues a request to an API service, the only one that really knows what the response looks like and how to interpret it (typically in JSON or XML) is the request object itself. In Performer, each type of request is responsible for instantiating the appropriate response object when a reply is received from the service. The method that holds this responsibility, CreateResponse, is defined in the Request base class and is overridden by subclasses, and as such it is a "Factory Method" by definition that manufactures responses according to specification.

- **Request**
  - A concrete object that can be subclassed or used directly.
  - Encapsulates the information required to perform a request (Command).
  - Provides a method to parse the response received from the server (Factory Method).
- **Response**
  - Defines an interface for querying the data contained in a response.
  - Is the common base class upon which all response types are built.
- **ConcreteRequest** (GetStopRequest, GetTravelsRequest)

    - Instantiates the ConcreteResponse it needs to represent the response representation received from the server (Factory Method).
    - Knows its arguments (query parameters), body (e.g., JSON, XML), headers, HTTP method and URI (path).
    - Represents a specific request to an API.
- **ConcreteResponse** (PlaceResponse, TravelListResponse)
    - Defines a response type which corresponds to a resource representation from the REST service.
    - Knows how to parse itself from JSON or XML.
    - Knows its body (e.g., JSON, XML), headers, and HTTP status code.

9.6.2 Observer

Used to define a one-to-many dependency between objects so that when one object issues a request, all dependents receive the response.

- **Performer**
    - Maintains a reference to each of its consumers.
    - Provides an interface for attaching and detaching Consumer objects, submitting Request objects and for starting and stopping execution of requests.
    - Knows how to connect to and communicate with a REST API.
- **Consumer**
    - Defines an interface for notifying an object about a received response, allowing it to query the state of the response object (i.e. its return value).
- **ConcretePerformer** (SimplePerformer)
    - Queues and invokes requests submitted to it.
    - May retry failing requests (either instantly or deferred) and may or may not pause execution (e.g. due to loss of network connectivity).
    - Broadcasts the responses it receives from the server to all consumers (Observer).
- **ConcreteConsumer** (LogConsumer)

  - Keeps its own state and can act upon being notified about received responses from the server.
  - Logs events to standard out.

9.7 Collaborations

- An object creates an instance of a ConcreteRequest and submits it to an instance of ConcretePerformer.
- The ConcretePerformer queues the ConcreteRequest.
- The ConcretePerformer executes the request as soon as a thread is available. If the request fails, the performer can either
  1. ignore the request and continue execution, or
  2. retry the request immediately, n number of times, or
  3. add the request back to the queue and try again later.
- Upon successfully receiving a response from the server, the performer uses the CreateResponse factory method on the request object and adds the response object to its response queue.
- The main thread, which is blocking on the response queue, pops the response and sequentially notifies all consumers subscribed to the performer.

9.8 Consequences

Most of the benefits and drawbacks of implementing Performer are inherited from the basis patterns.

1. Requests decouple the objects that initiate them from the ones that knows how to perform them, and from the ones that care about them.
   - All a performer knows is that it has a list of consumers conforming to a common interface.
2. Requests can be manipulated and extended like any other objects.
3. Requests can be added and removed easily without changing any existing classes.
   - However, because some objects may need to know the exact type of the response object, some type checking may occur,

which in turn implies that some code will have to be updated if it needs to know about the new type.

4. Any number of objects may be updated when a response is received from the server.
5. Unexpected and costly requests: Because consumer objects have no knowledge of each other, they can be blind to the actual cost of issuing a request. A seemingly insignificant request may cause a cascade of other requests to be issued by other consumers.
    - Can be partially mitigated with caching of (common) GET requests. Note that this functionality may be enabled by default in various HTTP libraries.

9.9 Implementation

The following is a list of issues that should be considered during the implementation of a performer:

1. **Automatic retries**
   Because a request at its core is nothing more that a container for related information to be used in a request, it can be executed repeatedly until it completes successfully. If the CreateResponse method encounters an unexpected payload in the response, it is appropriate to return a response object indicating the error (such as “401 Unauthorized”).
    - **Invalid credentials**
      If a request fails to due to an expired API authentication token, it should throw an exception which the performer must catch. This event should result in an appropriate response object which the object responsible for authentication can react to (renewing the token).
    - **Loss of network connectivity**
      If a request fails due to loss of network connectivity, it may either be requeued and retried, or the performer could pause execution and restart when the network is back online.
2. **Broadcast audience**
   An implementation may benefit from allowing consumers to indicate the kind of responses that interest them, if the number of consumers is large.
3. **Graceful shutdown**
   Each task processing thread must recognise the special Stop-Request object, which when received, causes the thread to end its

processing loop and resubmit the task (so that other threads may process it) before shutting itself down. Other mechanisms may be available, such as CancellationToken in C♯.

4. **Infinite loops of failure**
   A request may fail repeatedly because of internal or external factors, such as expired access tokens or disconnected networks. An implementation may choose to keep track of the number of times a request has failed, and terminate it to avoid burning CPU time. One approach to this is to include a counter in each request which the performer can check and increment.

5. **Parallel execution**
   A performer uses threading to execute several requests in parallel, thus increasing the perceived performance of the system.

9.10 Sample Code

The example code in this section sketches the implementation of the base classes and interfaces involved in Performer. These are all based on the familiar patterns we have discussed, and are fairly simple. We define the `IConsumer` and `IPerformer` interfaces, as well as the `Request`, `Response`, and `SimplePerformer` classes. We conclude this section by providing some examples pertaining to an actual implementation of some ReisAPI operations.

A performer could be implemented using a `ThreadPool`[2] or an `ExecutorService`[3], but it would make the examples in this section harder to translate to other languages without such classes readily available. Thus, we continue this section using simpler mechanisms.

Whereas in statically typed languages, one is required to strictly obey the rules specified by interfaces, there is no need to even define them in dynamically typed languages. One simply trusts that everything works as long as no error is encountered. However, for the purpose of clarity and consistency among the examples, we simulate interfaces in Python by utilising the “abc” package [Pyt16] (part of the standard library).

The sample code in this section performs little to no error checking, and does not necessarily demonstrate the “best” style of programming. They are nothing more than a simple demonstration of concepts. We have had to make compromises particularly with regards to the C♯ coding style, as we have limited space available both horizontally and vertically.

[2]https://msdn.microsoft.com/en-us/library/system.threading.threadpool.aspx
[3]https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html

```csharp
public interface IConsumer
{
    void HandleResponse(
        IPerformer sender,
        Response response);
}
```

Listing 9.1
`IConsumer` in C♯.

```python
import abc

class AbstractConsumer(abc.ABC):
    @abc.abstractmethod
    def handle_response(
        self,
        sender,
        response):
        pass
```

Listing 9.2
`AbstractConsumer` in Python.

Figure 9.2
The `IConsumer` interface in C♯ and the `AbstractConsumer` class in Python both define a single method, with no return value, whose sole purpose is to allow an object to handle any `Response` delivered to it.

9.10.1 Consumer

`IConsumer` defines the interface through which the performer notifies its subscribers when a response is received from the server. We refer to a subscriber in this context as a “consumer”.

Listing 9.1 defines the `IConsumer` interface in C♯, while listing 9.2 defines the abstract class `AbstractConsumer` in Python.

9.10.2 Performer

`IPerformer` defines the interface through which requests are submitted and customers attach to, or detach (subscribe or unsubscribe to updates) from the performer. The performer is responsible for handling communication with REST APIs on behalf of the system as a whole.

The `Start` method may be executed by any thread, but in general it may be a good practice to use the main thread for this purpose. `Start` may be implemented in such a way that it indicates the reason for its termination to the host operating system by returning an exit code (optional).

Listing 9.3 defines the `IPerformer` interface in C♯, while listing 9.4 defines the abstract class `AbstractPerformer` in Python.

9.10.3 Request

`Request` encapsulates all the non-shareable information needed to perform API requests, including the request payload, headers, method, parameters and path. Each instance of `Request` is equipped with a Globally Unique Identifier (GUID) that uniquely identifies each pair of

```csharp
public interface IPerformer
{
    int Start();
    void Stop();
    void Attach(IConsumer consumer);
    void Detach(IConsumer consumer);
    void Submit(Request request);
}
```

Listing 9.3
`IPerformer` in C♯.

```python
import abc

class AbstractPerformer(abc.ABC):
    @abc.abstractmethod
    def attach(self, customer):
        pass
    @abc.abstractmethod
    def detach(self, customer):
        pass
    @abc.abstractmethod
    def start(self, threads=3):
        pass
    @abc.abstractmethod
    def stop(self, exit_code=None):
        pass
    @abc.abstractmethod
    def submit(self, task):
        pass
```

Listing 9.4
`AbstractPerformer` in Python.

Figure 9.3
The `IPerformer` interface in C♯ and the `AbstractPerformer` class in Python define methods for submitting requests, management of subscriptions and methods for starting and stopping the engine.

`Request` and `Response` (useful for troubleshooting). `Request` provides a Factory Method that the performer calls to translate a response from the server into an appropriate `Response` object, which is what the performer broadcasts to its consumers. Subclasses of `Request` should override this method and provide a corresponding subclass of `Response`.

Listing 9.5 defines the base class `Request` in C♯, while listing 9.6 provides the corresponding definition in Python.

9.10.4 Response

The `Response` class contains the smallest amount of information required for the client to make sense of it (a subset of the complete response). We believe that the content body, the response headers and the HTTP status code is sufficient for most purposes. As such, the `Response` C♯ class defined in listing 9.7 and the Python class defined in listing 9.8 contain only this information.

Ideally, a `Response` object would provide access to the parsed version of the server response (such as an object delivered in JSON or XML), but because of the strongly typed nature of C♯, we have not been able to identify a good way of achieving this. For example, a `GetStop` request would return a single `Place` object, but a `GetDepartures` request would return a list of Departure objects. Thus, a subclass of `Response` would have to provide its own public “Departures”, or “Place”

```csharp
public class Request
{
    public Request()
    {
        Content = new Body(string.Empty);
        Headers = new WebHeaderCollection();
        ID = Guid.NewGuid();
        Method = "GET";
        Parameters = HttpUtility.ParseQueryString(string.Empty);
        Path = "/";
    }

    public Body Content { get; set; }
    public Guid ID { get; private set; }
    public NameValueCollection Parameters { get; private set; }
    public string Method { get; private set; }
    public string Path { get; private set; }
    public WebHeaderCollection Headers { get; private set; }

    public virtual Response CreateResponse(
        string body,
        HttpStatusCode status,
        WebHeaderCollection headers)
    {
        return new Response(ID, body, status, headers);
    }
}
```

Listing 9.5

The C$^\sharp$ implementation of the `Request` class is nothing more than a container of various information needed by a concrete implementation of `IPerformer` to perform a request.

```python
import uuid

class Request(object):
    def __init__(self):
        self.content = None
        self.headers = dict()
        self.id = uuid.uuid4()
        self.method = "GET"
        self.parameters = dict()
        self.path = "/"
    def create_response(self, body, status, headers):
        return Response(self.id, self.body, self.status, self.headers)
```

Listing 9.6

The Python implementation of the `Request` class is nothing more than a container of various information needed by a concrete implementation of `AbstractPerformer` to perform a request.

```
public class Response
{
    public Response(
        Guid id,
        string body,
        HttpStatusCode status,
        WebHeaderCollection headers)
    {
        Content = new Body(body);
        Headers = headers;
        ID = id,
        Status = status;
    }

    public Guid ID { get; private set; }
    public Body Content { get; private set; }
    public HttpStatusCode Status { get; private set; }
    public WebHeaderCollection Headers { get; private set; }
}
```

Listing 9.7
The C♯ implementation of the `Response` class uses properties to provide read-only data fields.

```
class Response(object):
    def __init__(self, id, body, status, headers):
        self.body = body
        self.headers = headers
        self.id = id
        self.status = status
```

Listing 9.8
The Python implementation of the `Response` class uses class variables to provide response data. These fields can be read from and written to unrestricted, as Python does not have access modifiers like C♯.

field to give access to these parsed objects. This is not optimal, because it requires that `Consumer` objects perform a type check before they use the response, which implies that they have to know about the existence of the kind of objects they wish to support. This is, however, how we have decided to do it in our implementation of ReisAPI.

9.10.5 SimplePerformer

`SimplePerformer` is a multithreaded “engine” that accepts and executes requests on behalf of other threads. It borrows elements from the Observer pattern and acts as a Subject. Because its definition in C♯ is relatively long, we have decided to use the “partial classes” feature that allows a class to be defined in more than one file. This feature is often used by Graphical User Interface (GUI) designers and other software that generate code automatically, to avoid mixing automatically generated and manually written code, but we use it in an attempt to improve the readability of the code (it has no impact on the collection

```csharp
partial class SimplePerformer : IPerformer
{
    readonly string m_base_uri;
    BlockingCollection<Request> m_requests = new BlockingCollection<Request>();
    BlockingCollection<Response> m_responses = new BlockingCollection<Response>();
    CancellationTokenSource m_cancel = new CancellationTokenSource();
    HashSet<IConsumer> m_consumers = new HashSet<IConsumer>();

    public SimplePerformer(string base_uri) {  m_base_uri = base_uri; }

    public void Submit(Request request) { m_requests.Add(request); }

    public void Attach(IConsumer consumer)
    {
        lock (m_consumers) { m_consumers.Add(consumer); }
    }

    public void Detach(IConsumer consumer)
    {
        lock (m_consumers) { m_consumers.Remove(consumer); }
    }

    public void Stop()
    {
        m_cancel.Cancel();
        m_cancel.Dispose();
        m_cancel = new CancellationTokenSource();
    }
}
```

Listing 9.9
The `SimplePerformer` class (C$\sharp$) works much in the same way as the Subject in the Observer pattern in that it allows other objects to subscribe and unsubscribe for notifications regarding updates to its internal state, except that in this case, state changes refer to requests that have received a response. This listing defines all internal member variables.

of metrics). The C$\sharp$ definition of `SimplePerformer` is provided in listings 9.9, 9.10 and 9.11, whereas the Python implementation can be found in listing 9.12.

```
 1 partial class SimplePerformer
 2 {
 3     public int Start() { return Start(4); }
 4 
 5     public int Start(uint threads)
 6     {
 7         for (uint i = 0; i < threads; i++)
 8         {
 9             new Thread(ThreadMain).Start();
10         }
11 
12         Response response = null;
13 
14         while (true)
15         {
16             try { response = m_responses.Take(m_cancel.Token); }
17             catch (OperationCanceledException) {  break; }
18 
19             lock (m_consumers)
20             {
21                 foreach (IConsumer consumer in m_consumers)
22                 {
23                     consumer.HandleResponse(this, response);
24                 }
25             }
26         }
27 
28         return null != response ? (int)response.Status : 0;
29     }
30 }
```

Listing 9.10

The `Start` method (C$^\sharp$) waits for the worker threads to receive and parse responses from the server. It delivers the responses it receive to all the consumer which is currently subscribed to the performer. `SimplePerformer` launches four threads by default. Notice how we break out of the loop at line 17.

```
partial class SimplePerformer
{
    private void ThreadMain()
    {
        Request request = null;
        HttpWebRequest http = null;
        UriBuilder builder = new UriBuilder(m_base_uri);

        while (true)
        {
            try { request = m_requests.Take(m_cancel.Token); }
            catch (OperationCanceledException) { break; }

            builder.Path = request.Path;
            builder.Query = request.Parameters.ToString();
            http = WebRequest.CreateHttp(builder.Uri);
            http.Accept = "application/json";
            http.Headers = request.Headers;
            http.Method = request.Method;

            try
            {
                using (var stream = http.GetRequestStream())
                {
                    byte[] body = request.Content.Bytes;
                    stream.Write(body, 0, body.Length);
                }
            }
            catch (ProtocolViolationException)
            {
                // The HTTP method does not allow a content body.
            }

            using (HttpWebResponse response = (HttpWebResponse)http.GetResponse())
            using (Stream stream = response.GetResponseStream())
            {
                Encoding encoding = Encoding.GetEncoding(response.CharacterSet);
                StreamReader reader = new StreamReader(stream, encoding);

                m_responses.Add(
                    request.CreateResponse(
                        reader.ReadToEnd(),  // Response content body.
                        response.StatusCode, // Response status code.
                        response.Headers));  // Response headers.
            }
        }
    }
}
```

Listing 9.11

`ThreadMain` (C♯) is the method in which all of the worker threads created by the `SimplePerformer` begins execution. Each thread waits for incoming requests, executes them and then queues the responses for delivery to subscribers.

```python
import queue
import threading

class SimplePerformer(AbstractPerformer):
    def __init__(self):
        self.__completed = queue.Queue()
        self.__customers = set()
        self.__pending = queue.Queue()
        self.__pool = set()

    def attach(self, customer):
        self.__customers.add(customer)

    def detach(self, customer):
        self.__customers.discard(customer)

    def start(self, threads=3):
        for i in range(0, threads):
            threading.Thread(
                args=(self.__pool, self.__pending, self.__completed),
                daemon=False,
                target=thread_main).start()
        while True:
            task = self.__completed.get(True)
            for customer in self.__customers:
                customer.update(task)
            if isinstance(task, StopRequest)
                break

    def stop(self, exit_code=None):
        self.submit(StopRequest(exit_code))

    def submit(self, task):
        self.__pending.put(task)
```

Listing 9.12
The Python implementation of the `SimplePerformer` class is farily short and concise, but it relies on a `StopRequest` sentinel class to end execution, as Python lacks the `CancellationToken` class we use in C♯ to abort blocking operations.

10

Part II: A Potential Composite Design Pattern

Summary

Performer is a composite software design pattern comprised of the three (widely used) Command, Factory Method and Observer patterns. As such, nothing about Performer is new or revolutionary. Its purpose is to provide applications with a multithreaded “engine” that takes care of the communication between application and service. Performer introduces a subscription model in which any object in the application can be notified every time a response is received from the server. This model means that functionality can be “plugged” in and out as desired during execution.

We have provided a comprehensive set of sample code that illustrate the concepts of Performer using the statically typed C$^\sharp$ programming language from Microsoft®, and the dynamically typed open source programming language, Python, which is maintained by the Python Software Foundation (PSF). The implementations are fairly similar in structure, though different in size with C$^\sharp$ having the shortest interface definitions and Python having the shortest implementations.

Part III

Evaluation

11

Part III: Evaluation

Introduction

In this chapter, we present a collection of software quality metrics that aim to capture various characteristics of program code as a single number. Then, we measure these characteristics in the ReisAPI example program to get an indication as to how the potential composite software design pattern affects various aspects of quality. Then, we provide a summary of the chapter and at the very end, we provide our conclusion.

The measurements have been collected with the “Calculate Code Metrics” functionality available in Visual Studio® Enterprise 2015 using its default settings with no modification [Micb]. This has allowed us to obtain various metrics with very little effort, but it has also limited us to the five metrics available in Visual Studio®: CBO, CC, DIT, NLOC, and MI (see chapter 12 on page 61). Given that measurements are best used as an indication of software quality aspects (not as scientific evidence), we interpret the data and discuss it with regards to the goal we established at the beginning of this thesis.

To ensure that the measurements measure only the maintainability, reusability, and testability directly related to the classes involved (due to the pattern being implemented), the measurements ignore the classes responsible for the parsing of resource representations received from the server. We could have achieved this goal by simply not parsing the responses we received from the service, dumping any data we received to standard out, but that would have made it much harder to verify that the client actually worked as advertised. Instead, we decided to use WCF, which is a part of the .NET Framework. For the purpose of evaluation, this makes little difference with regards to the results.

11.1 Windows Communication Foundation

As we take advantage of WCF to parse data from ReisAPI, we have been able to limit the amount of code not directly related to the composite pattern. Any data received from the service is passed directly to `DataContractJsonSerializer.ReadObject`, which uses the `DataContract` and `DataMember` attributes to instantiate the right object automatically. This is done in the constructors of `BaseResponse` subclasses, which are invoked from the `CreateResponse` method defined by the `BaseRequest` base class. Refer to chapter 6 or [Micc] for more information on WCF.

11.2 Outline

In the next chapters, we present the software quality metrics that we have made use of, and we take a look at the measurements collected from the ReisAPI client implementation.

- **Chapter 12: Software Quality Metrics**
  A summary of the measurements available in Visual Studio®, and some related measurements that we do not use directly.
- **Chapter 13: Measurements**
  We present and discuss the data we collected by measuring the source code of the ReisAPI C$^\sharp$ client. The result is favourable.
- **Chapter 14: Summary**
  This chapter provides a brief summary of what measurements we collect, and the conclusions we draw from it.
- **Chapter 15: Conclusion**
  This is the overall conclusion of this entire thesis. Our evaluation shows that Performer fulfills all of our requirements.

I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science, whatever the matter may be.

Lord Kelvin [Kel89]

12

Part III: Evaluation

Software Quality Metrics

12.1 Introduction

In this chapter, we present a summary of the measurements available in Visual Studio®, and some related measurements that we do not use directly (they provide the fundament for other metrics that we do use). The outcome from applying one of these measurements is a single number that gives an indication as to how "good" the program code is, with regards to what the metric aims to measure. By combining a set of different metrics, we get a fairly reliable indicator of quality.

There are two related notions of quality in the context of software engineering; functional quality (whether the system fulfills its purpose or not), and structural quality (non-functional qualities observable at runtime or related to the internal structure of components). We will focus on structural quality in this thesis, and specificially metrics that measure the maintainability, reusability and testability characteristics of software, as this aligns well with our goals (see section 1.2 on page 3). For the sake of completeness, however, we give a short description of functional quality in section 12.2 on the next page.

According to [Pre], the noun "metric" can refer to any standard of measurement and is not necessarily related to the metric system of measurement. We observe that "measure" and "metric" is often used interchangeably in the litterature. Hence, we will use them both in this thesis to refer to any standard of measurement.

12.2 Functional Quality

"The system shall do . . . "

Often derived from use cases, functional requirements define the functions that the system and its components are comprised of. Use cases illustrate how the system is supposed to behave in given scenarios. The sum of all the functional requirements define the purpose of the system; what it is supposed to do. How a functional requirement should be implemented is usually detailed in the system design. Thus, functional quality evaluates conformance and compliance with regards to functional requirements (in addition to regulatory requirements and international standards). Also referred to as "fitness for purpose", functional quality is typically enforced through software testing, which is not covered in this thesis.

12.3 Structural Quality

"The system shall be . . . "

While functional requirements define what the system is supposed to do, non-functional requirements, or "quality attributes", define how the system is supposed to be. The non-functional requirements can be divided into qualities observable at runtime ("execution" qualities: i.e. security and usability), and qualities caused by the structure of the system ("evolution" qualities: i.e. maintainability, reusability, and testability). "Structural quality" refers to how well the system meets the non-functional requirements, which gives an indication as to the quality of the implementation of the system.

An important part of the software engineering discipline is the ability to measure aspects of software, and express results with numbers. Measuring product quality, and assessing the efficiency of the people and tools involved in the process of manufacturing is fundamental to any engineering discipline. Statistical data allows us to measure improvement over time.

12.4 Complexity Metrics

This section briefly presents some of the software metrics that claim to measure the complexity of source code. The metrics have been selected because they are available in Visual Studio® (and there are far too many to cover all of them).

Metric	**Range**
Coupling Between Objects	Good $<= 14 <$ Bad
Cyclomatic Complexity	Good $<= 10 <$ Bad
Depth of Inheritance	Good $<= 6 <$ Bad
Lines of Code	Good $<= 24 <$ Bad
Maintainability Index	Good $>= 20 >$ Bad

Table 12.1
This table summarises the metrics we use in this thesis, and the upper and lower bounds we use to interpret their output.

12.4.1 Lines of Code

The Lines of Code (NLOC) metric is a basic measure of size based on the length of code in lines. Although it is often used by management as a measure of productivity, its value lies at the method level where it can be applied to identify methods that may be trying to accomplish too much. A well designed method should have one distinct purpose. Additionally, the entire body of a method should fit comfortably on the screen without having to scroll up and down.

Some tools also measure the Number of Statements (NOS) which has the advantages of allowing programmers to better format their code without getting punished with a bad score (because the statement count remains the same regardless of the number of lines they span). A high NLOC may possibly be considered better if the NOS is low, because it may indicate that the code is well formatted. The NOS measure is unfortunately, however, not supported by Visual Studio®, and will not be covered further as such.

Interpretation

Developers working on the Linux kernel adheres to a simple style guide written by Linus Torvalds. He suggests that a function should be short and have a single purpose. Furthermore, he argues that an entire function definitions should fit one or two “screenfuls of text”, where one screenful is 80 characters wide and 24 lines high [Tor]. We have determined it to be unfeasible to strictly adhere to the width restriction in C$^\sharp$, as the language is relatively verbose compared to plain C, and often produces very long lines. We believe that 24 lines of code should be sufficient on the method level (methods $\times$ 24 on the class level), and thus we have adopted it as out maximum limit on NLOC.

Behaviour Visual Studio®

The NLOC measure in Visual Studio® is an approximation based on the executable Common Intermediate Language (CIL) code generated by the compiler after processing the source code, and thus it may differ slightly from the number of line breaks in the file [Mica].[1] Consequently, files containing C$\sharp$ code that does not compile to executable code (such as interfaces) may appear to have zero lines of code. This carries the same advantages as NOS in that it allows programmers to better format their code without getting punished with a bad score.

12.4.2 Halstead Complexity Measures

The Halstead Complexity Measures (HCM) is a family of measurements introduced by Halstead in 1977. It is comprised of three primitive metrics and three derived metrics [Hal77]. Although not measured directly in this thesis, we include this section because of another, related metric (MI), which we present in section 12.5.1 on page 67.

Primitive Measurements

- Halstead Difficulty (HDIF) gives a suggestion on how difficult some code is to write and maintain by considering the number of unique operators (UOP), the total number of operands (OP) and the number of distinct operands (UOD).

$$\mathrm{HDIF} = \left(\frac{\mathrm{UOP}}{2}\right) \times \left(\frac{\mathrm{OD}}{\mathrm{UOD}}\right)$$

- Halstead Length (HLTH) is the total amount of operators (OP) and operands (OD).

$$\mathrm{HLTH} = \mathrm{OP} + \mathrm{OD}$$

- Halstead Vocabulary (HVOC) measures the complexity by looking at the number of variables used (the vocabulary). A small number of different variables used repeatedly indicates a lower complexity, while higher complexity indicates that more variables are in use. The vocabulary is simply put sum of the number of unique operators (UOP) and the number of unique operands (UOD).

$$\mathrm{HVOC} = \mathrm{UOP} + \mathrm{UOD}$$

[1]CIL is a standardised [ECM06] low level, human readable, stack-based and object-oriented assembly language used by the .NET Framework and Mono. CIL bytecode is typically executed by a virtual machine, but can also be translated to native code.

Derived Measurements

- Halstead Bugs (HBUG) attempts to estimate the number of bugs likely to be present in code. Some argue that the new formula is a more precise metric in the context of object oriented source code.

$$\text{HBUG} = \text{(new) } \frac{\text{HVOL}}{3000} \text{ or } \frac{\text{HEFF}^{(2/3)}}{3000} \text{ (original)}$$

- Halstead Effort (HEFF) is generally seen as a metric on the amount of mental effort required to rewrite some code from scratch.

$$\text{HEFF} = \text{HDIF} \times \text{HVOL}$$

- Halstead Volume (HVOL) measures the amount of code that has been developed by multiplying the HLTH with the binary logarithm of the HVOL.

$$\text{HVOL} = \text{HLTH} \times \log_2(\text{HVOC})$$

  The purpose of HVOL is to indicate how much information the reader of some code has to absorb to grasp its meaning.

12.4.3 McCabe's Cyclomatic Complexity

How many test cases are needed?

The Cyclomatic Complexity (CC) measure, introduced by McCabe in 1976, is a metric for the number of independent paths through a module. It is calculated by subtracting number of nodes from the number of edges plus 2. Nodes represent one or more actions and edges represent the flow of control [McC76].

$$\text{CC} = \text{Edges} - \text{Nodes} + 2$$

CC is often used as an indication as to the complexity of an algorithm and the number of test cases required to comprehensively test it. In practice the metric seems to be roughly equivalent to the number of loops and other conditional statements plus one [Hum14].

Interpretation

CC seems to be the most useful at the method level, as it may be easier to visualize and conceptualize the paths through a method, than paths through a large class or an entire application. A method with a single clearly defined purpose, composed of simple sequential statements, will generally be considered better because it requires fewer test cases to achieve comprehensive test coverage. As the number of conditional statements and looping constructs increases, the number of independent paths (and thus the effort required for comprehensive testing) increase as well. A CC value no larger than 10 is generally considered acceptable.

12.4.4 Coupling Between Objects

Is the class general enough?

The Coupling Between Objects (CBO) measure, introduced in 1994 by Chidamber and Kemerer, is a bidirectional count of the number of classes that are coupled to a class [CK94]. Classes can be coupled through parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, fields defined on external types, and attribute decoration [Mica]. The CBO of a class A is the union of all the classes that reference A, as well as all the classes A references. A biderectional reference is counted only once. CBO is known as "Class Coupling" in Visual Studio® [Nab11a].

Interpretation

A class with a CBO of zero has no relationship to any other class in the system, indicating that it should be considered for removal. Based on their research, Sahraoui, Godin and Miceli suggest that a maximum CBO value of 14 is desireable [SGM00]. A CBO higher than 14 may be an indication that the class is too tightly coupled to other classes, which may have a negative impact on future modifications and testing, thus limiting its potential for reuse because of its many interdependencies on other types.

12.4.5 Depth of Inheritance

How complex is the class?

The Depth of Inheritance (DIT) measure, introduced in 1994 by Chidamber and Kemerer, is defined as "the maximum length from the node to the root of the tree" and is predicated on three fundamental assumptions [CK94]:

- Predicting the behaviour of the class becomes harder the deeper it is in the class hierarchy.
- Design complexity increases with the depth of the inheritance tree because more classes and methods are involved.
- Potential for reuse of inherited methods increases with the depth of the inheritance tree.

Interpretation

Low DIT values may suggest low complexity and less potential errors, but it also implies that the potential for reuse through inheritance is low. Higher DIT values may indicate that the potential reuse is higher, but also that the complexity is higher, with which a higher probability of errors follows.

In [Nab11b], Microsoft® engineer Naboulsi observes that there is currently no accepted standard for DIT values. He suggests that a DIT value of around 5 or 6 should be an acceptable upper limit, but stresses that there was no empirical evidence to support it at the time of writing.

12.5 Measuring Maintainability

Maintainability refers to how easily and inexpensively upkeep of products can be carried out in response to deterioration of their components. There are three categories of maintenance activities in the context of software engineering: **corrective** (removal of defects), **adaptive** (adjustments in response to environmental changes), and **perfective** (improvement of software qualities) maintenance [NT11].

In this section, we present one measure for maintainability in object oriented software applications. We use the Maintainability Index because it is supported by many tools (including Visual Studio®), and because it is well defined and easy to understand.

12.5.1 Oman's Maintainability Index

How maintainable is the system overall?

The Maintainability Index (MI) was introduced in 1992 by Oman and Hagemeister at the International Conference on Software Maintenance [OH92], and was refined in 1994 with the paper 'Using Metrics to Evaluate Software System Maintainability' [Col+94].

$$\begin{aligned} \mathrm{MI} = 171 &- 5.2 \times \ln \langle \mathrm{HVOL} \rangle \\ &- 0.23 \times \ln \langle \mathrm{CC} \rangle \\ &- 16.2 \times \ln \langle \mathrm{NLOC} \rangle \\ &+ 50 \times \sin \left(\sqrt{2.46 \times \mathrm{PCL}} \right) \end{aligned}$$

MI is a blend of several metrics, including HVOL, CC and NLOC (which we have discussed), in addition to the Percentage of Comment

MI	Colour	Maintainability
20 to 100	Green	High
10 to 19	Yellow	Moderate
0 to 9	Red	Low

Table 12.2
Maintainability Index in Visual Studio®.

Lines (PCL). For all of the aforementioned metrics, the average per module is combined into the formula above

Behaviour Visual Studio®

Microsoft® engineers observed that the original formula from 1992 produced a value ranging from 171 to an unbounded negative number. They decided that the difference between code at 0 and some negative value was not useful, because code tending toward 0 is clearly hard to maintain. As such, they decided to treat all 0 or less indexes as 0 and then rebase the 171 or less range to be from 0 to 100 [Mor07]. This is the version of MI used by Visual Studio®.

$$\begin{aligned} \mathrm{MI} = \max(0, (171 &- 5.2 \times \ln \langle \mathrm{HVOL} \rangle \\ &- 0.23 \times \ln \langle \mathrm{CC} \rangle \\ &- 16.2 \times \ln \langle \mathrm{NLOC} \rangle) \times \tfrac{100}{171}) \end{aligned}$$

Additionally, Microsoft® provides an interpretation of the MI values produced by Visual Studio® which we have reproduced in table 12.2.

12.6 Criticism

A team of researchers hired six developers and conducted an experiment in which a set of structural software quality measurements were collected from four functionally equivalent enterprise systems. After comparing the measurements with each other, and with the subjective opinion of the developers, the researches concluded that most software maintenance metrics are mutually inconsistent, and that the best predictor of maintainability was the size of the system (NLOC) [SAM12].

12.7 Summary

Structural quality is concerned with the design of components, their roles and how they interact in source code. Structural metrics aim to express various software quality characteristics as a single number, and are typically measured with tools for static analysis, which is what we do in this thesis, using Visual Studio®.

We have covered five structural quality metrics: Coupling Between Objects (CBO), Cyclomatic Complexity (CC), Depth of Inheritance (DIT), Lines of Code (NLOC) and Maintainability Index (MI).

- CBO indicates whether a class is loosely or tightly coupled by counting the number of classes that references or are referenced from it. Maintainability, reusability and testability suffers with an increase in CBO due to the increase in complexity and lack of encapsulation. We consider a value equal to or less than 14 to be ideal.
- CC indicates the complexity of a class, method or module, by counting the number of independent paths through it. It is often interpreted as the number of test cases required, and as such, it affects maintainability and testability. A CC equal to or less than 10 is generally considered acceptable on the method level.
- DIT indicates the number of classes in the chain of inheritance. Complexity increases with depth, but so does reusability. A maximum DIT of 6 seems to be an acceptable limit.
- NLOC is a pure measurement of size. It indicates the number of non-blank, non-comment lines of code. It can be computed by counting newline characters, or it can be approximated by looking at the code generated by the compiler, thus allowing the programmer to format the code better, without negatively impacting the measure. A function should, ideally, consist of no more than 24 lines of code, though 48 is also acceptable.
- MI, a composite of other metrics, indicates how maintainable code is. In Visual Studio®, the value ranges from 0 (least maintainable) to 100 (highly maintainable), where 20 is the lower bound for "good" maintainability.

13

Part III: Evaluation

Measurements

13.1 Introduction

In this chapter, we present the measurements that we have collected from the source code of the example program.

We have organised the results into two main categories: “class average” and “method average”. Method average measurements have been collected by measuring each method in a class and by manually computing the average for that class. Thus, the “method average” at the bottom of the table is the average of all the averages. Class average, on the other hand, has been collected on the class level directly by Visual Studio®, and at the bottom of the table is the average of these class level measurements. Note that the DIT measure has been omitted from the method average, as it is meaningless on the method level (a method cannot inherit from another method).

The measurements include all getter and setter methods, which are generated automatically by the compiler to implement C$^\sharp$ class properties. Hence, one property may result in two “methods”. The ✓symbol at the bottom of the tables indicate that, on average, the results in that column are within the acceptable bounds specified in chapter 12 on page 61.

Category	Name	CBO	CC	DIT	NLOC	MI
Interface	`IConsumer`	2	1	0	0	100
	`IPerformer`	2	5	0	0	100
Base Class	`BaseRequest`	7	17	1	23	90
	`BaseResponse`	5	5	1	9	89
Concrete Class	`SimplePerformer`	21	20	1	58	65
Class Average		7.4	9.6	0.6	18	88.8
		✓	✓	✓	✓	✓

Table 13.1
Class level measurements of the basic classes and interfaces.

Category	Name	CBO	CC	NLOC	MI
Interface	`IConsumer`	2	1	0	100
	`IPerformer`	0.6	1	0	100
Base Class	`BaseRequest`	1	1.1	1.5	92.3
	`BaseResponse`	1.8	1	1.8	90.2
Concrete Class	`SimplePerformer`	4.8	4	7.3	75
Method Average		2	1.6	2	91.5
		✓	✓	✓	✓

Table 13.2
Method level measurements of the basic classes and interfaces.

API Module	**BaseRequest**	**CBO**	**CC**	**DIT**	**NLOC**	**MI**
Favourites	`GetFavourites`	7	2	2	5	82
Hearbeat	`Index`	5	2	2	4	85
Line	`GetLines`	5	2	2	4	85
	`GetLinesByStopID`	5	2	2	4	84
	`GetDataByLineID`	5	2	2	4	84
	`GetLinesRuter`	6	2	2	5	83
Meta	`GetValidities`	5	2	2	4	85
Place	`GetStop`	5	2	2	4	84
	`GetPlaces`	8	5	2	8	73
	`GetStopsByLineID`	5	2	2	4	84
	`GetClosestStops`	7	2	2	7	77
	`GetStopsByArea`	7	2	2	9	72
	`GetSalePointsByArea`	6	2	2	8	73
	`GetStopsRuter`	5	2	2	4	85
StopVisit	`GetDepartures`	5	2	2	4	84
Street	`GetStreet`	5	2	2	4	84
Travel	`GetTravels`	11	6	2	20	59
Trip	`GetTrip`	8	3	2	6	77
Class Average		6.1 ✓	2.4 ✓	2 ✓	6 ✓	80 ✓

Table 13.3
Class level measurements of the basic request classes.

API Module	BaseRequest	CBO	CC	NLOC	MI
Favourites	GetFavourites	3.5	1	2.5	81.5
Hearbeat	Index	2.5	1	2	87
Line	GetLines	2.5	1	2	87
	GetLinesByStopID	2.5	1	2	85
	GetDataByLineID	2.5	1	2	85
	GetLinesRuter	3	1	2.5	82
Meta	GetValidities	2.5	1	2	87
Place	GetStop	2.5	1	2	85
	GetPlaces	4	2.5	4	76
	GetStopsByLineID	2.5	1	2	85
	GetClosestStops	3.5	1	3.5	78
	GetStopsByArea	3.5	1	4.5	75
	GetSalePointsByArea	3	1	4	76.5
	GetStopsRuter	2.5	1	2	87
StopVisit	GetDepartures	2.5	1	2	85
Street	GetStreet	2.5	1	2	85
Travel	GetTravels	5.5	3	10	68
Trip	GetTrip	4	1.5	3	79
Method Average		3.1 ✓	1.2 ✓	3 ✓	81.9 ✓

Table 13.4
Method level measurements of the basic request classes.

BaseResponse	**BaseRequest**	**CBO**	**CC**	**DIT**	**NLOC**	**MI**
`DepartureListResponse`	`GetDepartures`	9	4	2	8	82
`FavouriteListResponse`	`GetFavourites`	14	8	2	13	73
`IndexResponse`	`Index`	3	3	2	4	91

`LineListResponse`	`GetLines`
	`GetLinesByStopID`
	`GetLinesRuter`
		9	4	2	8	82

`LineResponse`	`GetDataByLineID`	9	4	2	8	82

`PlaceListResponse`	`GetClosestStops`
	`GetPlaces`
	`GetStopsByArea`
	`GetStopsByLineID`
	`GetStopsRuter`
		9	4	2	8	82

`PlaceResponse`	`GetStop` `GetStreet`	9	4	2	8	82
`SalePointListResponse`	`GetSalePointsByArea`	9	4	2	8	82
`TravelListResponse`	`GetTravels`	11	4	2	8	81
`TripResponse`	`GetTrip`	9	4	2	8	82
`ValiditiesResponse`	`GetValidities`	9	4	2	8	82
Class Average		9.1 ✓	4.3 ✓	2 ✓	8.1 ✓	82 ✓

Table 13.5
Class level measurements of the basic response classes.

BaseResponse	**BaseRequest**	**CBO**	**CC**	**NLOC**	**MI**
`DepartureListResponse`	`GetDepartures`	3.7	1.3	2.7	87
`FavouriteListResponse`	`GetFavourites`	5	2.7	4.3	84.3
`IndexResponse`	`Index`	1	1	1.3	92

`LineListResponse`	`GetLines`
	`GetLinesByStopID`
	`GetLinesRuter`
		3.7	1.3	2.7	87

`LineResponse`	`GetDataByLineID`	3.7	1.3	2.7	87

`PlaceListResponse`	`GetClosestStops`
	`GetPlaces`
	`GetStopsByArea`
	`GetStopsByLineID`
	`GetStopsRuter`
		3.7	1.3	2.7	87

`PlaceResponse`	`GetStop` `GetStreet`	3.7	1.3	2.7	87
`SalePointListResponse`	`GetSalePointsByArea`	3.7	1.3	2.7	87
`TravelListResponse`	`GetTravels`	3.7	1.3	2.7	87
`TripResponse`	`GetTrip`	3.7	1.3	2.7	87
`ValiditiesResponse`	`GetValidities`	3.7	1.3	2.7	87
Method Average		3.6 ✓	1.4 ✓	2.7 ✓	87.2 ✓

Table 13.6
Method level measurements of the basic response classes.

13.2 Base Classes and Interfaces

13.2.1 Interfaces

As we noted in section 12.4.1 on page 63, an interface will always be reported as having zero source lines of code, because Visual Studio® approximates the value by looking at the executable CIL code generated by the compiler. An interface does not compile to executable code, and as such, the NLOC is zero for all of them.

The CC of the interfaces is in this case equivalent with the number of methods they define, which is to be expected. Both interfaces reference each other, in addition to either the `BaseRequest` or the `BaseResponse` class, and as such their CBO is low. It is possible for interfaces in C$^\sharp$ to inherit from other interfaces, though as the DIT of zero demonstrates, we have not exploited that fact (because `IConsumer` and `IPerformer` share no functionality).

IConsumer

The `IConsumer` interface defines a single method that allows a class to be notified when the system has received a response from the server related to a previously issued request. The method accepts two parameters; a reference to the invoking performer and a reference to the request object.

IPerformer

`IPerformer` defines the interface of an "engine" that accepts request objects, combines them with its knowledge of an API and performs them on a separate thread. The interface supports five operations: attaching and detaching (subscribing and unsubscribing to notifications), starting and stopping, and request submissions.

13.2.2 Base Classes

BaseRequest

`BaseRequest` is a class that encapsulates all of the information that is required to invoke an HTTP method on a resource. Specifically, `BaseRequest` stores the information that is not shared among different requests to the same API: the HTTP method, path, query parameters, headers and content body. The base URI belonging to the API is a common property of all requests, and as such it is stored only in classes implementing the `IPerformer` interface.

BaseResponse

`BaseResponse` contains three kinds of data; the HTTP status code returned from the server, the headers that were included in the response, and the content body of the response. Although some systems may benefit from storing more information in `BaseResponse` subclasses, this is the minimum amount of information that might be enough for the majority of APIs at this maturity level.

13.2.3 Concrete Classes

SimplePerformer

`SimplePerformer` is the reference implementation of the `IPerformer` interface which we use in our ReisAPI example. The implementation offers a simple First In, First Out (FIFO) model where requests are fetched and executed by n threads in chronological order.

From looking at tables 13.1 and 13.2, it appears that the `SimplePerformer` implementation violates the upper limit on the CBO measure of generality, on the class average. The class is, in other words, strongly coupled because it is related to a total 21 unique classes, 7 more than the suggested upper bound of 14. However, many of these classes are related to the classes that support the HTTP functionality in the .NET Framework or the synchronised data structures we use to implement support for threading, and as such it is hard to reduce. An alternative approach would be to outsource this functionality to a separate class. However, the aforementioned classes will always be available, and other languages may provide the same functionality using less classes. Hence, we conclude that the CBO of the `SimplePerformer` class is acceptable.

Even though the MI of `SimplePerformer` is relatively low (65/75) compared to all the other classes we have developed (with the notable exception of `GetTravels` with an MI of 59/68), it is still quite good when we consider that the lowest value Microsoft® considers good, or "green", is 20 [Mica].

A more advanced implementation of `IPerformer` may offer priorities, which is useful when one request must be performed for subsequent requests to succeed; for example in the event that an authentication token expires and causes all other requests to fail until it has been renewed by a high priority request.

13.3 Subclasses of BaseRequest

Roughly half of the concrete `BaseRequest` subclasses we listed in tables 13.3 to 13.4 on pages 73–74 has the exact same CBO value, which is because they are virtually identical, with the only differences being the request path they specify, and the type of the object they receive from ReisAPI. The other half of the requests accept parameters and use them to perform some processing in their constructors, which impact their CBO.

The worst "offender" with regards to high CBO is the `GetTravels` class which accepts 13 parameters, of which we use 12.[1] To improve the readability of the code we moved some of what we believe to be the least used options to a `TravelSettings` class. We have provided the code for the aforementioned constructor in listing 13.1. We make no attempt to improve the structure of the API.

```
public GetTravels(uint from, uint to, DateTime? time = null,
                  bool isAfter = true, TravelSettings settings = null)
{
    base.Path = "/Travel/GetTravels";
    settings = (null != settings) ? settings : new TravelSettings();
    time = time.HasValue ? time : DateTime.UtcNow;

    if (settings.LineNames.Count() > 0)
    {
        Parameters.Add(
            "linenames",
            string.Join(",", settings.LineNames));
    }

    if (settings.Transportation.Count() > 0)
    {
        Parameters.Add(
            "transporttypes",
            string.Join<TransportationType>(",", settings.Transportation));
    }

    Parameters.Add("changemargin", settings.ChangeMargin.ToString());
    Parameters.Add("changepunish", settings.ChangePunish.ToString());
    Parameters.Add("fromPlace", from.ToString());
    Parameters.Add("isafter", isAfter.ToString());
    Parameters.Add("maxwalkingminutes", settings.MaxWalkingMinutes.ToString());
    Parameters.Add("proposals", settings.MaxProposals.ToString());
    Parameters.Add("time", time.Value.ToString("ddMMyyyyHHmmss"));
    Parameters.Add("toPlace", to.ToString());
    Parameters.Add("walkingfactor", settings.WalkingFactor.ToString());
    Parameters.Add("walkreluctance", settings.WalkReluctance.ToString());
}
```

Listing 13.1
The `GetTravels` resource accepts a wide range of query parameters which cause the client implementation to be assigned a high CBO and a relatively low MI as a result.

[1] At the time of writing, the 13th option, "waitAtBeginningFactor", is undocumented in [Rutb]. It is not clear to us what its purpose is or what values it accepts.

To reduce the amount of repeated boilerplate code among the request classes, it may be possible to implement `BaseRequest` as a generic class; `BaseRequest<T>`. Doing so would make it possible to, for instance, create a new request using `new Request<PlaceResponse->("/Place/GetStop/3010930")`, or one could define `GetStop` as a subclass of `BaseRequest<PlaceResponse>`. This is something that may be worth considering in the context of C♯ and the .NET Framework or similar environments, but we do not investigate it further in this thesis.

13.4 Subclasses of BaseResponse

Overall, the `BaseResponse` subclasses are all fairly similar, with the exception of a few that employ private helper classes to satisfy WCF. Only a single class meets the suggested upper bound on CBO, namely the `IndexResponse` class.

In section 13.3 on page 79, we suggested that the request classes could be simplified by implementing `BaseRequest` as a generic class. Much in the same way, we may be able to reduce the amount of repeated boilerplate code among the response classes by implementing `Base-Response` as a generic class; `BaseResponse<T>`. Doing so would make it possible to define the simplest responses with a single line of code. Like before, this is something that may be worth considering in the context of C♯ and the .NET Framework or similar environments, but we do not investigate it further in this thesis.

13.5 Summary

An evaluation of the base classes and interfaces defined by Performer suggests that they are highly maintainable and reusable. As the classes perform no function other than storing data (with the exception of `SimplePerformer`), there is not much to test. Thus, the testability is excellent. `SimplePerformer`, on the other hand, is the most complex class in the entire system, not only because of what it does, but how it does it. The class uses threading (which is complex and prone to errors) to parallelise execution of requests. CC suggests that 20 test cases are required to achieve complete test coverage of `SimplePerformer`.

With the notable exception of `GetTravels` (with an MI of 59/68 due to offering many parameters), all `BaseRequest` and `BaseResponse` subclasses gets perfect scores in all measures. The DIT measure is always 2, because we only have one level of inheritance.

14

Part III: Evaluation

Summary

In the previous chapter, we performed an evaluation of the structural (non-functional) qualities of a ReisAPI client developed in C$^\sharp$ using Performer. We collected five software quality metrics and used the measurements to determine the maintainability, reusability, testability and size of the code. The metrics we applied were: Coupling Between Objects (CBO), Cyclomatic Complexity (CC), Depth of Inheritance (DIT), Lines of Code (NLOC) and Maintainability Index (MI).

14.1 Maintainability

The "worst" class in the entire code base, with regards to MI, was `GetTravels` with 59, which was relatively low compared to the other classes, though very good, considering that 20 is the lowest "good" value, according to Microsoft®. The reason for this is that `GetTravels` accepted 12 parameters, which is a lot more than most of the other classes did. The MI of the `SimplePerformer` class was much better, even though it was considerably larger in size, with an NLOC of 58 (a difference of 38). However, `SimplePerformer` exhibited excessive coupling, with a CBO of 21. This was largely caused by the number of classes needed to use HTTP in C$^\sharp$, and as such, would be difficult to reduce.

14.2 Reusability

All classes involved in the implementation had a DIT of 2, with the obvious exception of the basic classes: `BaseRequest`, `BaseResponse` and `SimplePerformer`. The latter is the only class that exceeds the CBO limit, which in general is an indication that the class may be hard to reuse. The high CBO was caused by a large number of classes involved in the use of HTTP, but because they are a part of the .NET Framework (always available), they have no impact on the reusability of `SimplePerformer`.

14.3 Testability

The number of possible test paths is low, because of the low coupling (CBO), and the low complexity (CC). The low DIT means that classes exhibit predictable behaviour. The low NLOC and high MI are also indicators of good testability.

14.4 Size

A function should, ideally, be no longer than 24 lines, although up to 48 lines is acceptable. This is to make sure that the entire function can be visible on the screen at the same time. The biggest class in terms of NLOC was `SimplePerformer`, with its 58 lines (7.3 on the method average), which is well within the limit. The entire project was comprised of 2 interfaces and 32 classes.

15

Part III: Evaluation

Conclusion

In this chapter, we provide a brief summary of the work we have presented in this thesis, our main contributions to the field of software engineering, and possibilities for further study regarding Performer.

15.1 Summary

In this thesis we have described and implemented the composite software design pattern, Performer. We have provided sample code in the statically typed C$^\sharp$, and the dynamically typed Python programming languages. We used the C$^\sharp$ implementation to build a REST client for the Norwegian travel planner API, ReisAPI, and used that code base to extract various indicators of software quality.

We begin by introducing the architectural style known as Representational State Transfer (REST) in chapter 3 on page 11. REST APIs can be classified according to three levels of service maturity defined in the Richardson Maturity Model (RMM). The level of least maturity, level 1, is the maturity level we have been working with in this thesis.

In chapter 4 on page 17, we present the fundamental purpose of naming and documenting design patterns in writing, and how the history of design patterns began with the release of a philosophical book about architecture, not long ago. The patterns we concern ourselves in this thesis are classified by purpose (creational, structural or behavioural) and scope (classes or objects). We looked at the Command, Factory

Method and Observer patterns which provide the basis for the composite pattern that is described in this thesis.

Chapter 5 on page 25 provides a short, high-level, presentation of the Norwegian travel planner API, ReisAPI, its nine modules and their purpose. The travel planner itself represents a fairly small portion of the API with only one documented resource. The rest of ReisAPI provides information about geographical locations, departures in fixed- and real-time, and metadata.

We conclude part I on page 9 with chapter 6 on page 29, a short presentation of Windows Communication Foundation (WCF), a programming model for building service-oriented applications on top of the .NET Framework using C$^\sharp$ and other Common Language Infrastructure (CLI) languages.

In part II on page 37 we describe the potential composite software design pattern, Performer. Performer is a class that accepts objects representing API requests and executes them using multiple threads. Any object can submit requests, and all objects that implement the right interface can receive notifications when responses arrive from the server.

Finally, in part III on page 59, we collected five measurements of software quality from the ReisAPI client source code; Coupling Between Objects (CBO), Cyclomatic Complexity (CC), Depth of Inheritance (DIT), Lines of Code (NLOC) and the Maintainability Index (MI). We used these measurements to evaluate the complexity and maintainability of the solution, which was found to be well within acceptable limits for the bulk of the solution.

15.2 Main Contributions

We have documented what we believe to be a good way of designing systems that consume resources on the internet, including, but not limited to REST APIs on service maturity level 1. We were familiar with the three basis patterns from previous study, and wanted to see whether a combination of them in this context would prove to be beneficial.

The composite pattern we describe in this thesis works as a high-level abstraction layer on top of a multithreaded execution “engine”, thus allowing programmers to get the performance benefits of multicore processing in their applications without having to manage threads and synchronisation directly. It can be implemented in any programming language that supports object orientation and multithreading.

15.3 Further Work

During the evaluation phase of this thesis, we identified some areas of Performer that could be improved. In this chapter, we explain some non-optimal aspects of the Performer design, and give suggestions on how they may be addressed by future research.

15.3.1 Generic Base Classes

Many of the `BaseRequest` and `BaseResponse` subclasses are almost identical. This results in a lot of duplicated boilerplate code, because most of their implementation is code that is required to properly create a subclass and override methods in C♯. One possible way to alleviate this issue may be to implement `BaseRequest` and `BaseResponse` as the generic classes `BaseRequest<T>` and `BaseResponse<T>`. This may reduce the amount of repeated boilerplate code significantly and could contribute to both improved complexity and maintainability. For the simplest of `GET` requests, only the request path would have to be changed in the subclass, as is exemplified in 15.1.

```
public class GetStop : BaseRequest<PlaceResponse>
{
    public GetStop(uint id)
    {
        base.Path = string.Format("/Place/GetStop/{0}", id);
    }
}
```

Listing 15.1
The `GetStop` class implemented using a generic `BaseRequest` base class. Only the constructor must be implemented explicitly.

15.3.2 Identify Performer in Other Systems

Search for independent and unrelated systems that use Performer, in order to determine whether it is a real design pattern or not.

15.3.3 Pluggable Representation Formats

An API may provide different representation formats (e.g.JSON and XML), and it would be an advantage if there was a mechanism that supported this possibility. Support for more than one representation format could be implemented by applying the Strategy pattern [Gam+94]. This would allow different parsers, with a generic interface, to be "plugged in" as necessary. The desired response format could be indicated by the performer or by each request using the `Accept` HTTP header.

15.3.4 Reduced Dependence on Type Checking

The biggest issue with Performer is, perhaps, the need to always know what kind of response one has received (which usually involves downcasting the object to a subtype). Because each subclass of `BaseRequest` is supposed to provide a corresponding subclass of `BaseResponse`, accessing the decoded data in the response depends on a type check; "if the type of the response is StopResponse, access its Stop property, else if the response is LineResponse, access its Line property, . . . " To identify the type one must make use of whatever facilities the programming language offers for this purpose. The only entity that ever knew the actual type was, after all, the factory method that created it. Finding a better way to achieve the same thing would be great for improved decoupling.

15.3.5 Transparent Authentication Mechanism

Whereas a fully RESTful API performs complete authentication in every request (stateless), typically using HTTP Basic Authentication (BA) or Hash-based Message Authentication Code (HMAC), one of the hallmarks of a level 1 API is that it uses authentication tokens that must be renewed periodically (stateful). It may be of interest to research how this authentication should be handled to be as transparent and non-intrusive as possible. Authentication could be performed by the performer itself, but it could also be carried out by a reqular request, or perhaps using the Strategy pattern [Gam+94]. One would need a mechanism for detecting that a request failed due to an expired token, pause execution of all requests, and then resume execution upon successful renewal of the token.

15.3.6 Use in Server Applications

Investigate the feasibility of developing REST server software using the architecture we have described in Performer. One thread listens for incoming requests, takes care of creating request objects and submits them to a performer, which carries it out. The response could then be sent by another thread, after it has carried out the request, (assuming a reference to the connection is included with the request object). Another approach would be to have a separate thread dequeue response objects, sending them sequentially to clients.

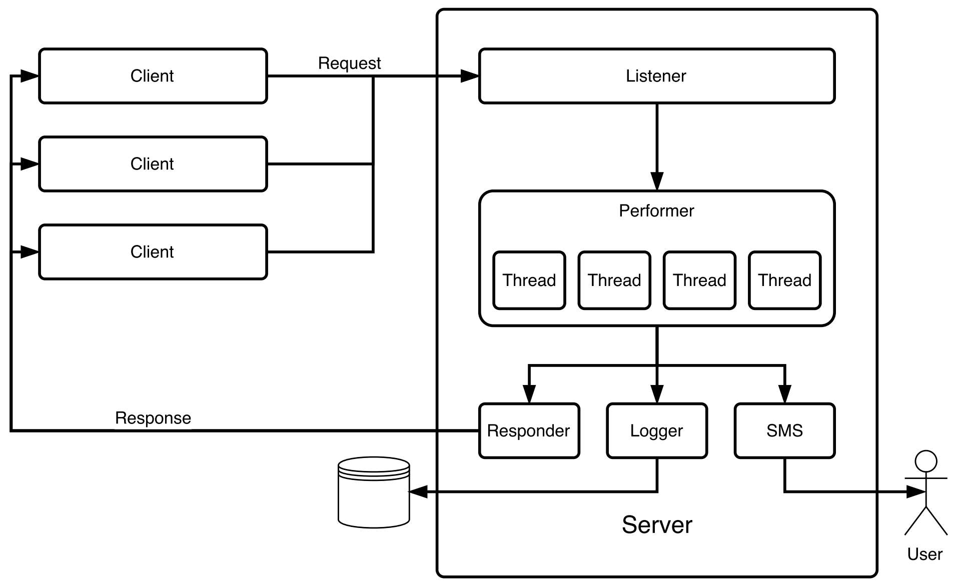

Figure 15.1
Performer may be useful on the server as well as the client. One thread accepts incoming requests, forwards them to a multithreaded performer which submits the response objects to an outgoing response queue which is serviced by a separate thread. Other services, like logging and notifications, may also be attached.

Acronyms

API	Application Programming Interface
CalDAV	Calendaring Extensions to WebDAV
CardDAV	vCard Extensions to WebDAV
CBO	Coupling Between Objects
CC	Cyclomatic Complexity
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CPU	Central Processing Unit
CRUD	Create Read Update Delete
DIT	Depth of Inheritance
FIFO	First In, First Out
FILO	First In, Last Out
GoF	Gang of Four
GUID	Globally Unique Identifier
GUI	Graphical User Interface
HATEOAS	Hypermedia as the Engine of Application State
HBUG	Halstead Bugs
HCM	Halstead Complexity Measures
HDIF	Halstead Difficulty
HEFF	Halstead Effort
HLTH	Halstead Length
HMAC	Hash-based Message Authentication Code
HTTP	Hypertext Transfer Protocol
HVOC	Halstead Vocabulary
HVOL	Halstead Volume

IRI	International Resource Identifier
JSON	JavaScript Object Notation
LCOM	Lack of Cohesion in Methods
LIFO	Last In, First Out
LILO	Last In, Last Out
MM	Maintainability Model
NLOC	Lines of Code
NLOD	Norwegian Licence for Open Government Data
NOS	Number of Statements
OJD	Ole Johan Dahl's house
MI	Maintainability Index
PCL	Percentage of Comment Lines
PSE	Programming & Software Engineering
PSF	Python Software Foundation
REST	Representational State Transfer
RFC	Response For Class
RMM	Richardson Maturity Model
SIRI	Service Interface for Real Time Information
SOAP	Simple Object Access Protocol
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
WCF	Windows Communication Foundation
WebDAV	Web Distributed Authoring and Versioning
WSDL	Web Services Description Language
WWW	World Wide Web
XML	Extensible Markup Language
BA	HTTP Basic Authentication

Bibliography

[Ale77] C. Alexander. *A Pattern Language*. 1st ed. Oxford University Press, 1977. ISBN: 0-19-501919-9.

[Ale79] C. Alexander. *The Timeless Way of Building*. 1st ed. Oxford University Press, 1979. ISBN: 0-19-502402-8.

[CK94] S. R. Chidamber and C. F. Kemerer. 'A Metrics Suite for Object Oriented Design'. In: *IEEE Transactions on Software Engineering* 20.6 (June 1994), pp. 476–493. ISSN: 0098-5589. DOI: 10.1109/32.295895. URL: http://dx.doi.org/10.1109/32.295895.

[Col+94] Don Coleman et al. 'Using Metrics to Evaluate Software System Maintainability'. In: *Computer* 27.8 (Aug. 1994), pp. 44–49. ISSN: 0018-9162. DOI: 10.1109/2.303623. URL: http://dx.doi.org/10.1109/2.303623.

[ECM06] ECMA. *ECMA-335: Common Language Infrastructure (CLI)*. Fourth. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), June 2006, vii + 104 (Part I), viii + 191 (Part II), iv + 138 (Part III), ii + 20 (Part IV), i + 4 (Part V), ii + 57 (Part VI). URL: http://www.ecma-international.org/publications/standards/Ecma-335.htm;%20http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf;%20http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.zip.

[Fie00] Roy Thomas Fielding. 'Architectural Styles and the Design of Network-based Software Architectures'. AAI9980887. PhD thesis. 2000. ISBN: 0-599-87118-0.

[Fie08] Roy Thomas Fielding. *REST APIs must be hypertext-driven*. English. Oct. 2008. URL: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (visited on 14/05/2016).

[Gam+94] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley, 1994. ISBN: 0-201-63361-2.

[Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977. ISBN: 0444002057.

[Hum14] Benjamin Hummel. *McCabe's Cyclomatic Complexity and Why We Don't Use It*. English. May 2014. URL: https://www.cqse.eu/en/blog/mccabe-cyclomatic-complexity/ (visited on 12/01/2016).

[Kel89] W.T.B. Kelvin. *Popular Lectures and Addresses*. Nature series v. 1. Macmillan & Company, 1889. URL: https://books.google.no/books?id=7685AAAAMAAJ.

[McC76] T.J. McCabe. 'A Complexity Measure'. In: *IEEE Transactions on Software Engineering* SE-2.4 (Dec. 1976), pp. 308–320. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837.

[Mica] Microsoft. *Code Metrics Values*. URL: https://msdn.microsoft.com/en-us/library/bb385914.aspx (visited on 03/06/2016).

[Micb] Microsoft. *How to: Generate Code Metrics Data*. URL: https://msdn.microsoft.com/en-us/library/bb385908.aspx (visited on 26/07/2016).

[Micc] Microsoft. *Windows Communication Foundation*. URL: https://msdn.microsoft.com/en-us/library/dd456779.aspx (visited on 05/06/2016).

[Mor07] Conor Morrison. *Maintainability Index Range and Meaning*. Nov. 2007. URL: https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning/ (visited on 28/05/2016).

[Nab11a] Zain Naboulsi. *Code Metrics – Class Coupling*. May 2011. URL: https://blogs.msdn.microsoft.com/zainnab/2011/05/25/code-metrics-class-coupling/ (visited on 28/05/2016).

[Nab11b] Zain Naboulsi. *Code Metrics – Depth of Inheritance*. May 2011. URL: https://blogs.msdn.microsoft.com/zainnab/2011/05/19/code-metrics-depth-of-inheritance-dit/ (visited on 28/05/2016).

[NT11] K. Naik and P. Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. Wiley, 2011. ISBN: 9781118211632. URL: https://books.google.no/books?id=neWaoJKSkvgC.

[OH92] P. Oman and J. Hagemeister. 'Metrics for assessing a software system's maintainability'. In: *Software Maintenance, 1992. Proceerdings., Conference on*. Nov. 1992, pp. 337–344. DOI: 10.1109/ICSM.1992.242525.

[Pre] Oxford University Press. *Definition of "metric" in English*. English. Oxford Dictionaries. URL: http://www.oxforddictionaries.com/definition/english/metric (visited on 19/05/2016).

[Pyt16] Python Software Foundation. *Abstract Base Classes*. English. Jan. 2016. URL: https://docs.python.org/3/library/abc.html (visited on 17/03/2016).

[Ric08] Leonard Richardson. ‘Justice Will Take Us Millions Of Intricate Moves’. QCon San Francisco. 2008. URL: https://www.crummy.com/writing/speaking/2008-QCon/.

[Ruta] Ruter AS. *About us*. English. URL: https://ruter.no/en/about-ruter/about-us/ (visited on 19/06/2016).

[Rutb] Ruter AS. *API Ruter Reise*. English. URL: http://reisapi.ruter.no/help (visited on 16/05/2016).

[Rutc] Ruter AS. *Ruter Labs*. Norwegian. URL: https://ruter.no/labs/ (visited on 16/05/2016).

[Sal] N. Salingaros. *Some Notes on Christopher Alexander*. English. URL: http://zeta.math.utsa.edu/~yxk833/Chris.text.html (visited on 12/05/2016).

[SAM12] Dag I.K. Sjøberg, Bente Anda and Audris Mockus. ‘Questioning Software Maintenance Metrics: A Comparative Case Study’. In: *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM ’12. Lund, Sweden: ACM, 2012, pp. 107–110. ISBN: 978-1-4503-1056-7. DOI: 10.1145/2372251.2372269. URL: http://doi.acm.org/10.1145/2372251.2372269.

[SGM00] H. A. Sahraoui, R. Godin and T. Miceli. ‘Can metrics help to bridge the gap between the improvement of OO design quality and its automation?’ In: *Software Maintenance, 2000. Proceedings. International Conference on*. 2000, pp. 154–162. DOI: 10.1109/ICSM.2000.883034.

[Tor] Linus Torvalds. *Linux kernel coding style*. URL: https://www.kernel.org/doc/Documentation/CodingStyle (visited on 03/07/2016).

[WPR10] Jim Webber, Savas Parastatidis and Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture*. Beijing: O’Reilly, 2010. ISBN: 978-0-596-80582-1.