Evaluating Software Design Patterns

— the “Gang of Four” patterns implemented in Java 6

Master’s Thesis, Computer Science

Gunni Rode

August 2007

Department of Computer Science

Faculty of Science

University of Copenhagen

Denmark

Dedicated to

Filur & Lurifax

- for keeping me company

Theodor Rode von Essen

- for keeping the smile on my face

But first, and foremost, dedicated to

Marina Rode von Essen

- for keeping me!

Preface

Life is what happens to you while
you're busy making other plans.
— ***John Lennon***

This thesis concludes a "work in progress" that has lasted over a decade – my Master's Degree in Computer Science; finally! I began studying in 1995, but was fortunate enough to get a very nice job within the first two years; a job that I hold and treasure to this day. Combined with life in general, this naturally slowed things down, but almost never to a complete halt, often because of friendly reminders from my family and friends.

Thesis — Formally, the workload of the project this thesis represents is 30 ECTS, and it was completed under the supervision of Professor Eric Jul at the University of Copenhagen, Denmark. The work carried out was done from late 2006 to August 2007, and the defence was held on November 2nd, 2007. This public version contains only small changes from the actual thesis handed in for grading, and the presentation delivered at the oral defence is available as [Rode07b]. During this period, many things happened in my life, good and bad. First and foremost, I became a father to a wonderful son named Theodor Rode von Essen; an eye—opener, to say the least, concerning such concepts as time, family, and especially oneself. Unfortunately, my family and I also experienced several deaths in the near family, especially that of my beloved father, Henning Rode, who died only a week before my defence. Secondarily, the company where I work was sold causing quite a few changes in my everyday life. I studied for the Java 5 certification, but never got around to take the exam (wonder why?). My wife and I had our kitchen completely renovated over a strenuous period of almost six months. And then some...! All these things one way or another influenced this thesis, but the reality is also that I at times was not focused enough. I did not manage to state precise and tangible goals for the work to be performed, causing me to pursue and writing about many different areas of interest related to OO and design patterns.

Originally, this project was intended to evaluate different aspects related to the two—way connection between the "Gang of Four" patterns and the programming languages used for implementation. Much work was put into formulating several evaluation criteria in a consistent format, albeit in broad terms. Simplified examples of criteria include how the use of natural language affects the applicability of pattern X in language Y, or vice versa; if the naming of pattern participants is consistent and independent of specific OO programming paradigms; or how easy is it to implement pattern X in language Y? Java 6, and possibly other languages, should act as the catalysts for the evaluations, but the criteria spawned more questions than answers because they were more theoretical than practical in nature. However, the intent had always been to make this a practical project with emphasis on the practical application of design patterns, but the evaluation approach seemed to collide with this. Hence, when I discovered several articles on pattern application utilising specific language features that caused "simpler" implementations, or even pattern componentizations, in various languages, the idea arose to compare such findings with Java 6 implementations of the "Gang of Four" patterns. I also realised that the "Gang of Four" patterns should be evaluated as a whole rather than an arbitrary sub—set because the patterns were published as a complete pattern system with many internal relationships and similarities. The work performed is still an evaluation, but focus thus moved from several forms of evaluation of a handful of the

"Gang of Four" patterns to more concise and practical investigation of how Java 6 paradigms can influence the application of all the "Gang of Four" patterns. The final work description for the work performed is approved by Eric Jul and is available as [Rode07a].

Thesis Website — This thesis has a dedicated website at http://www.rode.dk/thesis, which offers this thesis in a slightly modified online HTML version; the presentation given at the oral defence; the developed source code; generated JavaDoc; etc.

Acknowledgements — I can honestly say, without a doubt, that I would never have been able to complete my degree, and especially this thesis, without the love and continuous support from my beloved wife, Marina. Though alien to computer science, she also offered much appreciated assistance with proof—reading and layout. Jonna von Essen also helped with last minute, but very effective, proof—reading.

My workplace and colleagues also made this thesis possible as they graciously allowed me to take time off to complete it, thereby burdening themselves with even more work. Several people provided invaluable critique, some of which I unwisely ignored ☺. Morten Wolf assisted with harsh, but earnest proof—reading. Jesper Steen Møller provided much appreciated input and Erik W. Rasmussen did as well. Furthermore, Brian Grunnet lent me practically all the books cited in this thesis; some of them have since become mine due to coffee stains and undeniable traces of claws from a cat.

Finally, I wish to thank Eric Jul for allowing me to undertake a somewhat unorthodox but tangible and hands—on thesis that is actually highly relevant to me. I suspect this is also the case for my co—workers and the likes. Eric's pragmatic approach to this project, even after I changed focus half way through, as well as our many discussions, helped me overcome seemingly overwhelming obstacles to eventually complete this thesis.

Prerequisites — The reader is assumed to have an understanding of computer science corresponding to at least graduate level. Familiarity with Object—Orientation and Java is expected, but in—depth knowledge of pattern theory is not required as this thesis presents an introduction to pattern theory and how it relates to OO. However, practical experience with software design patterns and especially the "Gang of Four" design patterns is a definite plus. A sense of humour is not a bad thing either.

Keywords — Design Patterns; Gang of Four; Java 6; Object—Orientation; Language Features

Colophon

This thesis is written in UK English, set with Trebuchet MS, 9pt, using a line spacing of 1.5. Program listings are illustrated with Courier New, 6pt, and syntax highlighted. Program code inlined in normal text is written using Courier New, 10pt, in grey. Quotations are written using Times New Roman, 9pt, in italics; bold text within quotations identifies the author being quoted, or emphasises issues deemed important by the undersigned. Important terms or names, such as design pattern names, classifications, concepts, and type names are capitalised, as the Factory Method pattern or the Equilibrium property pertaining to pattern quality.

References are alphabetised by the surname of the primary author, followed by the year of publication if possible. Citations are written in square brackets, separated by semicolon in alphabetical order, including the name and possible year with two digits only, for example [Alexander77; Lea00]. Page, table, figure, or item references are prefixed with p, t, f, and i, respectively, for example [Lea00, i.12] for the twelfth FAQ item presented there. Page references are supplied if possible and only if the reference in question has explicit page numbers, for example [Gamma95, p.6]. An item will be referenced the first time encountered in the context at hand, and only again if the context warrants it.

Figures, tables, program listings, and examples are enumerated for easy reference. The enumeration format is the chapter number followed by a dot and a sequence number local to the chapter, e.g. 2.1, 2.2, 3.1, 4.1, 4.2, 4.3, etc. References to chapters, sections, figures, tables, program listings, and examples are set in bold—face, as figure **2.1**. Cross—references spanning several pages will generally be followed by the page number to the reference in question, as figure **2.1** on page 13.

Abstract

In this project, we perform an evaluation of the "Gang of Four" design patterns from a practical and experimental point of view using Java 6 as the implementation language. We investigate how Java 6 language features affect the application of the "Gang of Four" design patterns, individually and collectively. The investigation focuses on how the practical *use* of language features can affect the design pattern implementations, not how the features are constructed. To perform a reasonably structured and verifiable evaluation, we define a general evaluation approach on how to evaluate the "Gang of Four" patterns using a language as a catalyst. The premise is to implement *all* pattern functionality described in Implementation and Sample Code elements in the "Gang of Four" pattern descriptions, as these are the elements that primarily target the practical implementation, and evaluate the outcome.

Using the defined approach, we implement the "Gang of Four" patterns in Java 6 and investigate use of *core language features* (types, generics, closures, etc.), *reflection* (class literals, dynamic proxies, annotations, etc.), and *special language mechanisms* (synchronisation, serialization, cloning, etc.). The individual pattern evaluations show that with a few exceptions, all pattern functionality described in the Implementation and Sample Code elements, including Meta—information, can be implemented or simulated in Java 6 using the investigated features. The comparative evaluation shows that Java's mixture of static and dynamic features are very well suited to express the "Gang of Four" pattern functionality. Creational and especially Behavioural patterns benefit from dynamic usage, while the static features make the implementations more robust, possibly reusable, and clarify pattern intent. The implementations furthermore provide novel, or at least alternative, approaches on how to implement Abstract Factory, Factory Method, Memento, Observer, Proxy, Singleton, and State in Java 6.

Resumé

Dette projekt omhandler en evaluering af ”Gang of Four” designmønstrene (engelsk: *design patterns*) ud fra en praktisk og eksperimentel tilgang, hvor Java 6 er programmeringssproget, der bruges til implementeringen. Vi undersøger, hvordan sprogegenskaber i Java 6 påvirker anvendelsen og implementeringen af ”Gang of Four” designmønstrene, individuelt og sammenholdt for alle mønstrene. Evalueringen fokuserer på, hvordan den praktiske brug af konstruktioner i Java kan påvirke implementeringen af mønstrene, ikke hvordan konstruktionerne selv er konstrueret. For at udføre en rimelig struktureret og validerbar evaluering, definerer vi en generel fremgangsmåde til at evaluere ”Gang of Four” mønstrene ved brug af et givent programmeringssprog som katalysator. Udgangspunktet er, at *al* mønsterfunktionalitet beskrevet i ”Implementation” og ”Sample Code” elementerne i mønsterbeskrivelserne skal forsøges implementeret og resultatet derefter analyseres, idet disse er de primære elementer med fokus på den praktiske anvendelse.

Vi implementerer ”Gang of Four” mønstrene inden for rammerne af den definerede fremgangsmåde og undersøger brugen af *grundlæggende sprogegenskaber* (typer, parameteriserede typer, etc.), *refleksion* (klasser, dynamiske proxier, annoteringer, etc.), samt *specielle sprogmekanismer* (synkronisering, serialisering, kloning, etc.). De individuelle evalueringer af mønstrene viser, at al mønsterfunktionalitet fra ”Implementation” og ”Sample Code” elementerne kan implementeres eller simuleres i Java 6 ved brug af de undersøgte konstruktioner med få undtagelser. Den sammenlignende evaluering viser, at Javas blanding af statiske og dynamiske egenskaber er endog meget god til at udtrykke funktionaliteten beskrevet i ”Gang of Four” mønstrene. ”Creational”, men især ”Behavioural” mønstre drager fordel af de dynamiske egenskaber, mens statiske egenskaber medvirker til, at implementeringerne bliver mere robuste, muligvis genbrugelige, og tydeliggør mønsterfunktionalitet. Endeligt frembringer evalueringen nye, eller i det mindste alternative, tilgange til at implementere ”Abstract Factory”, ”Factory Method”, ”Memento”, ”Observer”, ”Proxy”, ”Singleton”, og ”State” i Java 6.

Table of Contents

List of Figures

List of Tables

Program Listings

1. Introduction

Life has been so much easier
since Science invented Magic.
— Marge Simpson

In this thesis, we evaluate software design patterns from a programming language and practical point of view in an Object—Oriented (OO) environment. We investigate how language paradigms in Java 6 [Gosling05] affect the application of all the "Gang of Four" (GoF) [Gamma95] design patterns. The investigation focuses on how the practical *use*, not the construction, of language features can affect the design pattern implementations. This chapter presents the motivation for undertaking this project, as well as the goals we want to achieve. We outline the work performed during this thesis, both theoretical and practical, and we conclude this introductory chapter with an extensive summary of the content presented in this thesis.

1.1. Motivation

Designing and developing complex software systems is not, and has never been, an easy task. On the contrary, the process is often very time consuming and requires interaction between many different people, skills, and roles, internally and externally. Many, often contradictory, factors must be addressed in the design process and at different levels, such as the need for maintainability versus quick delivery, flexibility versus speed, etc. The domain may offer tools, notations, principles, and methods to guide the development process, but they cannot shield against bad design decisions made by humans, and they may not even be standardised. For example, there is a lack of consensus on how to approach OO development, and several OO methods exist, each offering their take on how to design OO systems. The Unified Modelling Language (UML) [UML05] is commonly used to model the design, words like "class" and "object" denote commonly accepted concepts, and Gamma et al. suggest favouring object composition over inheritance [Gamma95, p.20]. However, this modus operandi is by no means a guarantee for good and durable designs. Experience helps, but careful decisions and meticulous work are always required. Therefore, and worst of all, the entire process tends to be error prone, not forgetting costly. The larger and more complex the system is, the worse these factors seem to become at an escalating rate.

Even the most complex systems are built by using smaller "parts", influencing the overall design directly or indirectly. A part can be anything from an entire sub—system to a specific component, native to the language or otherwise, that requires the need for a specific design. Such parts may in turn be built using even smaller parts and so forth and need to communicate to function as a whole. The key to any viable design is to identify the relevant parts, their functionality, and their interaction, but this is not a trivial matter. The OO approach attempts to manage the system complexity by abstracting out knowledge and encapsulating it within interacting *objects* [WirfsBrock90, p.5]. Hence, a part can be viewed as a single object (or rather its type) or a collection of interacting objects delivering a specific functionality. If we view a part as a design problem to be solved, regardless of the approach chosen, it is likely that others have already solved a similar problem in a satisfactory manner. If we can utilise this knowledge, the quality of the system may be improved. One approach to identify reoccurring design problems and their well—proven solutions is to use software design patterns. A design pattern is an abstraction of practical experience and empirical knowledge, but it is also a description of the problem it addresses and a solution to it [Alexander77; Lea93]. While the design pattern provides a canonical solution to the described problem, human interaction and interpretation is required to apply the solution in different contexts. Software design patterns are commonly associated with, but not limited to, OO environments. Patterns are uniquely named and written in a consistent format that allows designers, developers, and others to communicate using a common vocabulary. Related patterns are grouped in collections, or ideally languages. Design patterns can facilitate the entire design and development process because they express ideas and solutions founded in experience traditional methodologies cannot. They communicate architectural ideas in a consistent high—level language.

Nevertheless, design patterns must be applied with caution. Design patterns are neither completely static, nor completely dynamic in nature. To apply a design pattern, a problem similar to the one addressed by the pattern must have been identified in some context. Patterns are not reusable components, but guidelines on how to solve a given problem. This is an important fact, but based on our experience, one that is often forgotten in real life situations. As with any design discipline, the human factor is important because choices and interpretations must be made to adapt the pattern to a given situation. On the other hand, the environment or context may dictate behaviour that must be adhered to, and thus cannot be changed. The solution must be implemented for each context, perhaps differently and perhaps using different programming languages, but a given environment may also present standard implementations of a given pattern for easy reuse, depending on the pattern complexity. Patterns can be misunderstood, misused, not used at all, or convey incorrect information at the time of writing and/or at the time of application. While design patterns can lead to sound designs, they cannot offer any guarantees [Vlissides97, i.5]. The true benefit is only realised if a given collection of design patterns is used on a regular basis in a specific domain and context. The continued use will motivate a better understanding of how the patterns work and possibly evolve in the given context. In a practical sense, design patterns that repeatedly have been applied successfully are in our view equivalent to "teaching" or "Best Practices" for the domain in question, according to the philosophical approach offered by "Best Practices" based around continuous learning and continual improvement (see also [Vlissides97, i.6]). Qua this reasoning, we have used a number of design patterns extensively in our OO designs, but have experienced that regular practical usage in a given context is closely tied to the programming language used to implement a given pattern.

The motivation for this project is to gain a better understanding on how to use design patterns from a practical point of view in OO environments, specifically how the use of language features can influence the pattern application. To make this concrete, we investigate the "Gang of Four" design patterns using Java 6 as the programming language. In our view, "design patterns" as a concept is indeed a helpful tool. However, the choice is not *whether or not* to use "design patterns" in the design process, but *which* concrete patterns to use, if any. Design patterns as a practical tool are meaningless unless specific design patterns are known, because otherwise the knowledge cannot be utilised. A pattern can describe anything, but only specific patterns can solve a given problem. By virtue of our job, we wish to evaluate the "Gang of Four" patterns because we have used several of them extensively, but critically, in the design and development of large and quite complex Internet applications. The "Gang of Four" patterns are a collection of twenty—three design patterns described in the "Design Patterns" book [Gamma95]; by April 2005, the book was in its 32nd printing! The "Gang of Four" patterns describe communicating objects and classes that are customised to solve a general design problem in a particular context [Gamma95, p.3] in OO environments. The experience gained while working with these patterns over the years has shown us that they can be a valuable aid in shaping the design of successful applications, but we have also noted several issues that warrant a closer look. The "Gang of Four" patterns are well over a decade old, and seem to be targeted primarily for C++ [Stroustrup91] environments with rather dense and stringent descriptions on how to implement them. We have used them in different environments, using languages supporting other, or missing, features compared to those addressed in the pattern descriptions. We have also experienced problematic issues, such as concurrency related issues; for example, how thread—safe initialisation in the Singleton [Gamma95, p.127] pattern is ensured. Many of these issues are practical in nature, and seem related to how design patterns and a given programming language interact. Continuing with the Singleton example, Java has built—in support for synchronisation, which could solve the initialisation problem, but there are also other ways to solve the problem in Java. Unfortunately, because of time and money, real—life projects seldom allow in—depth investigations on such issues. The aim of this project is to remedy this by offering a subjective, but comprehensive, evaluation of the "Gang of Four" patterns implemented in Java 6. The result of the evaluation will give us a better understanding on how the practical use of certain language features may affect the evaluated design patterns. This is relevant as the "Gang of Four" patterns are frequently used in real—life systems, and so is Java, but Java 6 furthermore offers a range of versatile features that will be interesting to apply in the pattern implementations. As the evaluation centres on features found in Java 6, most observations will be relevant for Java only. It may be possible that some can be generalised to similar languages.

1.2. Goals

The primary objective of this Master's Thesis is for the undersigned to obtain a Master's Degree in Computer Science from the University of Copenhagen, Denmark.

This thesis represents a project with a formal workload of 30 ECTS. The purpose of the project is to evaluate practical application of the "Gang of Four" design patterns using Java 6 and present the findings in this thesis. **The premise is to investigate how the use of Java 6 features may affect pattern application**. By doing so, we hope to gain valuable experience that will enable us to understand and use design patterns better in "real—life situations", not just when applied in Java, but also in situations where the choice of programming language has already been made. The work includes theoretical and practical aspects.

The primary objective is achieved, if the project is approved based on this thesis. To fulfil the purpose of the project, we define the following sub—goals to be addressed in the project and in this thesis:

I. **Theory and Background** — Present an introduction to and a discussion about the theory deemed necessary to understand topics covered by the evaluation. This will include OO; patterns in general with focus on software design patterns, especially the "Gang of Four" design patterns; clarification on how concepts and themes described by Gamma et al. relate to Java 6; and a discussion on related work and topics.

II. **Evaluation Approach** — Define a simple, but reasonably structured approach on how to perform an evaluation of the "Gang of Four" patterns, where the choice of language will act as a catalyst for the evaluation. The approach must describe the overall evaluation set—up; how to focus the evaluation; and how to describe the specific criteria used to perform the various investigations. As the evaluation is subjective, this will enable others to judge the premise, execution, and result of the evaluation as well as perform a similar evaluation using a different language catalyst.

III. **Implementation** — Within the realm of the defined approach, implement and evaluate the "Gang of Four" design patterns using Java 6.

IV. **Evaluation** — Present the evaluation outcome and comment on the findings separately for each evaluated pattern and by juxtaposing the individual evaluations.

A secondary objective is the intention that this thesis can aid others, especially colleagues at work and like—minded, to reflect about software design patterns in general, but in particular in relation to the "Gang of Four" patterns implemented in Java 6. Whether or not this objective is achieved will not be evaluated, but this thesis and the implementations will be made publicly available for those interested.

1.2.1. Demarcations

This thesis will **not** cover:

- An evaluation on the validity of the abstractions the "Gang of Four" patterns describe. It is assumed that the patterns represent usable solutions to the problems they address, and the evaluation investigates only issues related to practical pattern application in Java 6. The pattern abstractions will only be commented during the investigations if deemed necessary.
- An in—depth description of Java 6 and its features, as the reader is assumed familiar with Java.
- The theory behind the construction of programming languages, specifically Java. While it is necessary to be familiar with language features in order to utilise them in pattern application, it is not necessary to know how these features are implemented in the language itself. For example, we do not care how Java's garbage collector or type system is implemented internally, only about the features they offer to the user and implementer of the design patterns. Relevant features are only described and discussed from a practical point of view.
- An in—depth analysis of Christopher Alexander's work on patterns and pattern languages within the field of architecture. In computer science, the "Gang of Four" patterns build on these concepts, as do others, but we only present a quick introduction, primarily based on [Appleton00; Lea93].

1.3. Thesis Summary

In this thesis, we present a subjective evaluation of the "Gang of Four" design patterns implemented in Java 6. The evaluation centres on how the use of specific language features may affect the pattern application. To make a reasonably structured evaluation across the different patterns using a given language as the catalyst, we address issues related to the implementation described by Gamma et al. in the Implementation and Sample Code elements in the "Gang of Four" pattern descriptions. If possible, we provide an example on how equivalent functionality can be implemented in Java 6, or explain why it cannot. We summarise our findings, and identify traits common to several patterns. Additionally, we present a thorough introduction to the background theory required to understand the "Gang of Four" patterns and the concepts they build on, such as OO development and pattern theory. We also discuss several articles related to application of the "Gang of Four" patterns in various different languages, both dynamic and static languages, and where deemed relevant compare the results to the outcome of this evaluation.

This thesis is divided into two parts, excluding the introduction and overall conclusion. The first part presents theory and background (chapters 2 – 4). The second part concerns the implementation and practical evaluation (chapters 5 – 9). In principle, each part can be read independently, but part one provides a solid foundation on related topics before the evaluation is undertaken in part two. Most of the theory presented in this thesis can be found in numerous other places in the literature as well, but we apply a practical viewpoint that focuses on the "Gang of Four" patterns and Java 6. By including, discussing, and focusing on various aspects of it here, we maintain an important perspective on points relevant to the evaluation.

1.3.1. Part One — Theory and Background

This part begins with an introduction to OO development. Focus is on how design patterns, particularly the "Gang of Four" patterns and the concepts they express, can aid the process and how they relate to Java 6. Pattern theory and the relation to software patterns are described and selected studies on related work are examined.

Chapter 2 — The "Gang of Four" patterns are design patterns targeting design problems related to OO. To understand the inner workings of the "Gang of Four" patterns, the OO methodology forming their foundation must be understood. Its importance is emphasized by Gamma et al. as the entire first chapter in [Gamma95] is dedicated to the OO concepts and themes that form the basis of the design patterns presented. Hence, chapter 2 gives an introductory, but focused, presentation to these issues, but goes even further and connects concepts, themes, Java 6, and usage of design patterns in the overall development life—cycle of OO systems. As explained in section 2.1, there is no formal consensus on the concepts that describe fundamental OO behaviour, but the themes and concepts described by Gamma et al. seem to be commonly accepted. This is illustrated by juxtaposing the concepts against a recent survey by Armstrong [Armstrong06] that investigates 239 texts on OO theory to try to identify the fundamental concepts inherent to OO. Because of this wide acceptance and considering how ubiquitous applicable the abstractions described in "Gang of Four" patterns have proven to be, the choice of OO method to guide the development process becomes less important. In many respects, as reasoned in section 2.2, we see design patterns as orthogonal to OO methods, because the objects and knowledge they represent are independent of which method produced the (initial) context to which a pattern can be applied. Regardless of the OO method used, the software lifecycle normally includes phases such as OO analysis (OOA), OO design (OOD), and OO programming (OOP), or implementation, perhaps re—iterated as needed as the design evolves. Section 2.4 explains that the analysis phase determines *what* is to be built, often in form of a conceptual model, and the design phase *how* it should be built, as pointed out in section 2.5. Design patterns are primarily used during the design phase, often modelled using UML as described in section 2.3, but also in the implementation phase as the patterns must be adapted to and implemented in the chosen language. As rationalised in section 2.6, the design and implementation phases are where the choice of language really becomes important, because it determines how the design is executed; what is possible, what is not, and ultimately how well the implementation reflects the desired concepts and themes.

Chapter 3 — Chapter 3 presents general pattern theory based on the ideas set forth by Christopher Alexander within the field of architecture, but also relates the theory to software design patterns, and in particular to the "Gang of Four" patterns. The "Gang of Four" design patterns are "just" one collection of software design patterns, and in order to understand software design patterns as a concept, at least the basic principles of Christopher Alexander's work on patterns and pattern languages must be known. This is necessary because software design patterns in general build on the basic ideas set forth by Alexander, in particular the "Gang of Four" patterns evaluated here [Gamma95, p.2]. Simplified greatly, a pattern is an abstraction of practical experience and basic knowledge on how to solve a given problem, described in a consistent format so it can be adapted for reuse in similar contexts. Section 3.1 contains an introduction to Alexander's work, describing the history and theory behind patterns and pattern languages; the information is mainly based on [Appleton00] and [Lea93], subsidiary on [Alexander77]. Many of Alexander's ideas are admittedly abstract, but computer science was not only reasonably quick to adapt several of his ideas, but also to introduce original pattern related concepts as explained in section 3.2. According to Alexander, a pattern must ideally possess certain properties to ensure the quality of the pattern and thus the quality of the (reusable) solution it generates, for example Abstraction, Composibility, and Encapsulation. Many of these properties have similar meaning to desirable constructs in OO, which could explain why software patterns first became popular within this domain. A class, for example, is an abstraction with encapsulated responsibilities that can be used as a component by other classes. Pattern qualities are explained in section 3.3, but a pattern must furthermore balance opposing forces, or constraints, within its context to reach a balance that implicitly will be present in the pattern and its application [Appleton00]. This implies, as elaborated in section 3.4, that a pattern may represent trade—offs between various forces, for example flexibility versus speed of an OO application. For a pattern to be useful, it must concisely communicate both the problem it tries to solve and the solution to it, including expressing the desired qualities and account for the forces at play. It does this by partitioning the description in various descriptive elements, such as Name, Intent, Consequences, Implementation, Sample Code, etc., but this is no trivial matter as the notion of patterns can be applied in various contexts. Section 3.5 describes common pattern formats used to describe patterns; in particularly the format used to describe the "Gang of Four" design patterns, where C++ and Smalltalk are used to illustrate key pattern points. This format is used extensively in the evaluation, in particular the Implementation and Sample Code elements as they pertain to the implementation and evaluation. Still, the lack of a formalised concept of a design pattern has long been a vigorously debated issue within the pattern community. It goes to the very core of understanding, or agreeing on, what software design patterns are. This is discussed in section **3.6**. Formalism is closely related to tool support for pattern mining, understanding, and application, and can therefore aid the implementation, but also limit the degree of freedom inherent in pattern descriptions. Section **3.7** describes how patterns can be grouped in collections, or ideally languages, where individual patterns may be interrelated in various cooperative ways. All twenty—three "Gang of Four" patterns are finally presented, including an illustration of how they may intricately connect and cooperate in numerous ways. The Gamma at el. classification scheme is also presented, which classifies patterns according to scope (Class, Object) and purpose (Creational, Structural, Behavioural). Section **3.8** describes how patterns can evolve, from discovery to ordinary usage to possibly becoming part of the language itself. Pattern collections may also evolve over time. Finally, section **3.9** discusses the practical application of patterns.

Chapter 4 — The final chapter in the first part of this thesis is chapter **4**, which discuss selected studies on related work. All revolve around the "Gang of Four" pattern application in a given language. Compared to chapter **2** and **3**, the chapter is much more technical and practical. It is discussed how specific languages via their paradigms and features affect individual "Gang of Four" patterns. As explained in section **4.1**, different languages have different support for various pattern abstractions. According to Norvig [Norvig96, p.7], patterns can be classified based on their (language) implementation level as Invisible, Informal, or Formal, where only the latter corresponds to pattern application as described by Alexander, i.e. anew from "scratch" for each context. The former two rely on built—in language support and/or components, respectively. The level of support can greatly influence the pattern application in the given language, and Java is no different. Several dynamic and primarily functional languages have been shown to provide simpler pattern implementations compared to the canonical "Gang of Four" implementations. Articles describing three such language studies are examined in section **4.2**, concerning Common Lisp, Dylan, and Scheme. The primary conclusion drawn from these studies is that dynamic features such as reflection, first—class types, multiple—dispatch, and higher—order functions have a positive impact on nearly all of the "Gang of Four" patterns [Norvig96, p.10; Sullivan02a, p.43]. This is interesting because even though Java is neither functional, nor dynamic, it supports reflection, closures, generics, and dynamic proxies, which possibly could be used to achieve similar implementations. As section **4.3** reveals, the "Gang of Four" patterns have been applied in several earlier Java versions, at least, but we have not found similar studies of all "Gang of Four" patterns in Java 5 or 6. Studies of AspectJ and Eiffel implementations are also discussed in section **4.3**. AspectJ uses Aspect Oriented Programming (AOP) features to allow Java to exhibit very dynamic features, such as open classes and support for method combination (advice). Utilising such features, the study claims that seventeen of the twenty—three implementations exhibit modularity improvements in terms of better code locality, reusability, composibility, and (un)pluggability [Hannemann02, p.1]. This is interesting, because Java by itself can simulate many of the features found in AspectJ, though requiring some work. The Eiffel studies are similar in that many of the features can also be found or simulated in Java 6, but also because of the success rate in fully or partly componentizing two—thirds of the "Gang of Four" patterns [Meyer06, p.3]. Interesting. Section **4.4** provides a comparison of the features examined in the aforementioned studies, comparing them to features found in Java 6 that can be used in the practical evaluation. It also tries to identify common traits of individual patterns as well as per pattern classification, e.g. scope (Class, Object) and purpose (Creational, Structural, Behavioural).

1.3.2. Part Two — Evaluation

The second part concerns the practical evaluation, which consists of individual pattern implementations as well as a comparative evaluation of all implementations and features used, to identify common traits and issues.

Chapter 5 — Chapter 5 defines a simple evaluation approach that can be used to investigate how well a given language expresses all pattern functionality described in the "Gang of Four" Implementation and Sample Code elements, and then applies it to define the evaluation goals used in this thesis. For others to judge the evaluation, its premise must be known. As described in section 5.1, the focus is practical and experimental as these elements focus implementation and language issues. Next, the evaluation approach is defined in section 5.2. It requires a detailed and a comparative evaluation. The detailed evaluation concerns the actual pattern implementation and participant usage, and describes the pattern implementations using familiar "Gang of Four" pattern elements, albeit in more detail. The comparative evaluation juxtaposes the individual observations and feature usage to identify common traits and issues. Section 5.3 uses the defined approach to establish the evaluation goals. The overall goal is to provide a realistic, but subjective, evaluation that may help understand how the "Gang of Four" patterns and Java 6 can cooperate. The focus is practical and technical, from the perspective of a practising designer and/or developer. Three broad categories of Java 6 features are examined: *core language features* (types, generics, closures, etc.), *reflection* (class literals, dynamic proxies, annotations, etc.), and *special language mechanisms* (synchronisation, serialization, cloning, etc.). The comparative evaluation will also analyse "Gang of Four" pattern relationships described by Gamma et al. compared to those actually expressed in the implementations. It also classifies the level of support individual patterns have in Java 6 within the realm of the evaluation performed.

Chapter 6 — Chapter 6 is dedicated to the practical aspects related to the implementation in Java 6. Section 6.1 outlines the technical set—up, such as the exact Java version and IDE used. Eclipse 3.3 is the primary IDE, but Sun's NetBeans 5.5.1 is used for comparison. Eclipse uses its own compiler implementation, whereas NetBeans uses the official compiler. No plug—ins of any kind are required to run the evaluation code or tests. All individual pattern implementations operate, directly or indirectly, on a set of model classes to imitate a larger "application" compared to what could be achieved by isolated pattern implementations alone. Individual implementations or parts thereof can thus more easily be used in other pattern implementations, expressing many of the pattern relationships described by Gamma et al. Section 6.2 explains how the implementations are modelled using UML Class diagrams. The diagrams are a big part of the detailed evaluation because they meticulously illustrate pattern participants, including attributes and operations. In section 6.3, the basic design for the overall evaluation is described, also illustrated in UML. Next, section 6.4 presents an overview of the developed source code, divided into relevant packages for each pattern implementation. Several Meta packages containing functionality such as model classes, loggers, reflection and general utilities have also been developed. To ensure that others can confirm the pattern behaviour in the implementations provided in this thesis, an absolute minimal "test framework" has been developed as explained in section 6.5. Each pattern implementation supplies a test class to illustrate the functionality. This is not a replacement for JUnit testing, but merely to report developed class usage via system out or file loggers. Complete source code, JavaDoc, and UML Class diagrams are available on the thesis website at http://www.rode.dk/thesis.

Chapter 7 — The first part of the evaluation is the comparative evaluation in chapter 7. The comparative evaluation offers a thorough analysis of which Java 6 features are used to implement which patterns. Section 7.1 presents all identified *pattern* × *feature* mappings in table 7.1, high—lighting the most interesting entries, which are also summarised separately in section 9.2. The features investigated are those established in chapter 5, categorised as *core language features*, *reflection*, or *special language mechanisms*. Each feature has a dedicated sub—section that explains its usage across all patterns, identifying possible common traits and alternatives, as well as a small conclusion to its usefulness with regards to help expressing "Gang of Four" pattern functionality. Numerous program listings are used to illustrate pattern functionality. The evaluation shows that the pattern implementations benefit from Java's mixture of static and dynamic features. As the last thing, section 7.1 presents observations from the pattern implementations on how to translate C++ features into Java 6 features relevant to several patterns. Next, section 7.2 compares the pattern relationships expressed in the evaluation to the "Gang of Four" described relationships identified in section 3.7. The expressed relationships are subjective based on the evaluation design rather than language features, but help illustrate how versatile the "Gang of Four" patters are. To conclude the comparative evaluation, section 7.3 classifies the patterns according to their implementation level as described by Norvig, explained in section 4.1.

Chapter 8 — Chapter 8 presents the individual pattern implementations. Section 8.1, 8.2, and 8.3 present the evaluations of Creational, Structural, and Behavioural patterns, respectively. Each pattern investigation is presented as required by the evaluation approach defined in chapter 5. The detailed evaluation shows that practically all pattern functionality described in the Implementation and Sample Code elements of the "Gang of Four" design patterns can be implemented or simulated in Java 6, including Meta information not used directly in the canonical implementations. The implementations express the concepts described in chapter 2 and 3.

Chapter 9 — The results of the comparative and detailed evaluations are summarised and presented in chapter 9. While chapter 7 and 8 provide summaries and conclusions where appropriate, chapter 9 comments on the evaluation as a whole, presents high—lights, and puts the evaluation and its results in perspective. Section 9.1 determines the level of compliance between the implementations and the "Gang of Four" concepts, themes, and pattern descriptions. Section 9.2 concludes that Java's core language features promote robustness, pattern intent, and reusability, and form the base of all the pattern implementations. Combined with reflection and annotations, this offers alternative and flexible pattern implementations. Next, section 9.3 presents the high—lights identified during the evaluation that utilises the Java 6 features in the manner just described. High—lights include generic factories in Abstract Factory and Factory Method, *guarded types* in Memento, annotated observers in Observer, dynamic proxies in Proxy and State, and enumerations in Singleton (*Singleton—as—Single—Constant* idiom). We end the evaluation conclusions with an evaluation of the defined evaluation approach itself in section 9.4. We conclude that the evaluation approach offers a way to investigate and judge how well a given language can express the "Gang of Four" functionality expressed in the Implementation and Sample Code elements, but we do not draw any conclusions as to whether the language catalyst should be used in a given scenario, here Java 6. The evaluation serves as a tool from which experience can be drawn.

Following part two, chapter 10 contains an overall conclusion to this thesis and the work performed. Section 10.1 explains the perspective in which this thesis and its conclusions must be understood: from a practical and experimental point of view, explanatory in nature. The initial goals have all been achieved and the specific results and contributions made by this thesis are listed in section 10.2. Primary contributions include the detailed evaluation in chapter 8, which shows that practically all pattern functionality described in the Implementation and Sample Code elements of the "Gang of Four" patterns can be implemented or simulated in Java 6, as well as the pattern and Java 6 functionality high—lights from section 9.2. Before we conclude this thesis with a final remark in section 10.4, an outlook on possible future work is provided in section 10.3.

1.3.3. Work Performed

The work performed during the project and presented in this thesis has been both theoretical and practical, with emphasis on the latter. The amount of hours put into the project has been substantial; the time spent reading, writing, experimenting, shouting, and programming is hard to put into words, but has been spent nonetheless.

1.3.3.1. Theoretical

The theoretical aspect covers the research, books and articles read, not forgetting the summation and discussion of the relevant material presented in this thesis. Making clear demarcations proved no easy task because, in our view, everybody within the field of computer science seem to have an opinion on software design patterns, perhaps because of the apparent lack of a common understanding and formal methodology. Much new material had to be covered, understood, and some of it paraphrased for this thesis; and some material that was expected could not be found. All this initially came as a bit of surprise; while design patterns are easy to use, ordinary use normally does not warrant in—depth scrutiny, research, and evaluation based on scientific theories. As this thesis concludes, the use of design patterns is very much a practical discipline. Due to the shift in focus of this thesis, much early research and work unfortunately had to be discarded, but this process also caused much improved (practical) focus and structure in the thesis. [Rode07a] is the final work description.

The bibliography contains the list of references used in this thesis. Pivotal among them are [Gamma95], [Lea93; Lea00], [Appleton00], [PPR], [Buschmann96], and [WirfsBrock90] for the theory and background; [Norvig96], [Sullivan02a; Sullivan02b], [Hannemann02], and [Arnout06; Meyer06] for the related work; and [Stroustrup91], [Gosling05], and [Bloch01] for the implementation and evaluation. The choice to use the "Gang of Four" patterns was made because of experience and their widespread use. The (re—) reading of the [Gamma95] book gave us much new insight into the workings of several familiar "Gang of Four" design patterns. The book is very dense and covers a lot of information, some of which can easily be missed on casual reading. This is one of the reasons why the evaluation investigates *all* functionality described in the Implementation and Sample Code elements, and not just the canonical implementations.

1.3.3.2. Practical

The practical part covers everything related to the evaluation and implementation(s). It took substantial effort to figure out how to conduct a meaningful evaluation of design patterns. An evaluation only makes sense if the premise for the evaluation can be viewed and judged by others, so they themselves can conduct a similar evaluation, or at least judge the outcome in the proper context. Because of the shift in focus, we deemphasised a formalised evaluation approach compared to the practical work performed in the evaluation. Unfortunately, this was not done until after we had developed a semi—formal approach, which was then completely discarded. On the plus side is that this gave the entire project an aura of realism because unlike the "Gang of Four" patterns, software projects and systems rarely get things right on the first try.

The choice to use Java 6 was because of personal experience with Java, but also because we know of no other study analysing the use of Java 6, or 5 for that matter, as the language to implement the "Gang of Four" patterns. Experience is essential in a project like this; the "tricks of the trade" cannot be utilised otherwise and implementations may become trivial. The overall implementation has produced over 300 Java source files, yielding 400+ compiled class files (including enumerations and inner classes). The design, implementation, test, and documentation took longer than expected, as always. We feel it is important to establish that the pattern implementations are not trivial shells unless explicitly warranted by the design, but realistic and sometimes quite complex. Because of the scale of the implementations, the evaluations also took quite some time. In reality, the theoretical and practical work performed exceeds 30 ECTS, though in part because of the shift in focus. This indicates that the thesis scope is perhaps too wide and/or too ambitious, but done is done.

2. Object—Oriented Development

The object—oriented model makes it easy
to build up programs by accretion.
What this often means, in practice, is that
it provides a structured way to write spaghetti code.
— Paul Graham

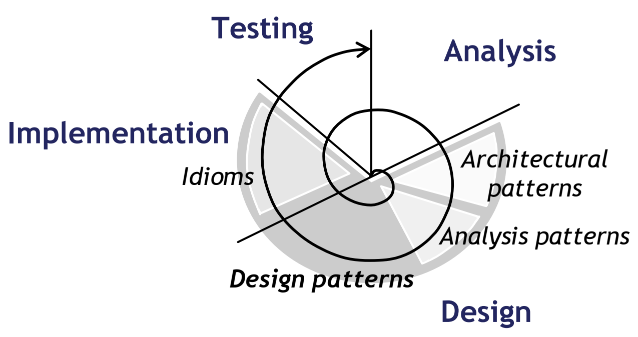
In a room full of top software designers,
if any two of them agree, that is a majority.
— Bill Curtis

Object—Oriented (OO) development is entirely possible without the use of OO design patterns, but if OO design patterns are used, they must be applied within the realm of OO development. This chapter gives a short presentation to OO development and to the process of designing OO systems with focus on how OO and design patterns interact, especially in a Java context. The "Gang of Four" design patterns we evaluate in this thesis can be used as a tool to aid the design of OO systems, regardless of the Object—Oriented Method (OOM) used. The patterns represent solutions to problems related to the design of OO systems, but at the same time express this knowledge using OO concepts and principles. Hence, the general OO concepts must be understood in order to understand the design patterns and to perform the evaluation in a consistent manner, including understanding the approach utilised by Gamma et al. in the "Gang of Four" design patterns themselves. We also link the themes and concepts described by Gamma et al. to Java. To understand how and when design patterns can be utilised when designing OO systems, we present abridgements on Object—Oriented Analysis (OOA), Object—Oriented Design, and Object—Oriented Programming (OOP) as well.

The Object—Oriented (OO) approach to software design attempts to manage the complexity inherent in real—world problems by abstracting out knowledge and encapsulating it within objects [WirfsBrock90, p.5]. Identifying the proper objects, relationships, and interactions are the key objectives to any successful OO design, but this is no trivial matter. The granularity of the design is thus a (complex) object, but an object may also represent an interaction with a complete sub—structure, for example a reusable component or a software design pattern such as a "Gang of Four" pattern. Numerous OO methods have been developed, each offering more or less proprietary procedures on how to approach the design and development in order to fulfil these objectives, but no common standard exists. Up until deployment, and regardless of the method used, the OO development life—cycle generally consists of analysis, design, implementation, and testing phases in some form. The phases may be overlapping or re—iterated, each time refining the design and implementation. This is dictated by the OO method and procedures used or more likely by (ever) changing demands and specifications. Compared to other forms of software development the design phase is considerable larger, because OO systems are designed for easy reuse, maintenance, and modification [WirfsBrock90, p.9].

As the design phase is so central to OO development, it is paramount that the design is sound and durable. While the OO method may guide the design process, it cannot offer the specific knowledge represented by a pattern. Patterns known by the designer can be used as a tool in the design process because they offer proven solutions to common problems, which ideally heighten the quality of the design. Part of the pattern knowledge is describing the objects and their relationships relevant for the given scenario, thereby making the job of the designer a little easier. As a benefit, the application of well—known patterns will probably make the design seem more familiar to other designers as well. Figure 2.1 illustrates the OO software development life—cycle commonly used excluding phases such as deployment and evaluation and the relation to patterns.

Figure 2.1 — OO development life—cycle and patterns (modified from [WirfsBrock90, f.1—2])



The OO software development life—cycle traditionally consists of an analysis, design, implementation, and testing phase, which may be overlapping or re—iterated as dictated by the OO method used, each time refining the design and implementation.

Different categories of patterns are used in different phases of the life—cycle. *Architectural patterns* have large design granularity and are used early in the design phase. *Analysis patterns* target the domain. *Design patterns* have medium granularity and can be used throughout the entire design phase, but are also closely related to the implementation. *Idioms* have the smallest granularity and are connected with a specific language.

Different categories of patterns are used at different times in the development process, but their usage can overlap as illustrated in figure 2.1 above. As explained in section 2.5.1, design patterns are patterns targeting design problems with medium granularity, used to refine the sub—systems or components of an OO system, or the relationships between them [Buschmann96, p.13]. The "Gang of Four" patterns are classified as design patterns, which is thus the category of patterns this thesis investigates. From a practical point of view, design patterns are also closely related to the implementation because their descriptions contain source code and must in any case be implemented. Any type of pattern used in OO development inherently reflects OO concepts such as objects, classes, inheritance, encapsulation, polymorphism, etc. To understand such patterns these concepts need to be understood as well. Hence, the next section presents an introduction to OO concepts as understood in this thesis before we describe the processes pertaining to OO development and the relation to patterns.

2.1. Object—Oriented Concepts

The general lack of consensus regarding fundamental OO concepts is clearly illustrated by a recent survey of existing literature related to OO development performed by Armstrong [Armstrong06]. Two hundred and thirty nine articles, books, and conference proceedings related to OO development were examined by Armstrong to try to identify the essential elements of OO development. Thirty—nine concepts were identified, but only eight of these were utilised by the majority of the sources reviewed [Armstrong06, p.124]. Armstrong argues that the lack of consensus may be because we do not yet thoroughly understand the fundamental concepts that define the OO approach. Many authors suggest concepts that define OO, taking for granted that the concepts are known, or that no universal concepts exist; others acknowledge the need for a consensus [Armstrong06, p.123]. Few works offer methods of precise specification for OO design, and none are commonly recognised as standards.

Armstrong defines a two—construct taxonomy containing the eight fundamental concepts identified, also known as *quarks* [Armstrong06, t.3]. The taxonomy is reproduced in table 2.1 below.

Table 2.1 — Armstrong's two—construct OO taxonomy (modified from [Armstrong06, t.3])

Construct	Description
Structural Construct	
Abstraction	Creating classes to simplify aspects of reality using distinctions inherent to the problem.
Class	A description of the organisation and actions shared by one or more similar objects.
Encapsulation	Designing classes and objects to restrict access to the data and behaviour by defining a limited set of messages that an object can receive.
Inheritance	The data and behaviour of one class is included in or used as the basis for another class.
Object	An individual, identifiable item, either real or abstract, which contains data about itself and the descriptions of its manipulations of the data.
Behavioural Construct	
Message	A way to access, set, or manipulate information about an object.
Message Passing	An object sends data to another object or asks another object to invoke a method.
Polymorphism	Different classes may respond to the same message and each implement it appropriately.

By using an OO perspective to classify the individual concepts, they are placed in one of two constructs, namely the Structural or Behavioural construct. Armstrong describes Structural concepts as focused on the relationship between classes and objects, as well as the mechanisms that support the class/object structure. A class is an abstraction of an object. The class/object encapsulates data and behaviour and inheritance allows the encapsulated data and behaviour of one class to be based on an existing class [Armstrong06, p.127]. On the other hand, Behavioural concepts are focused on object actions. Armstrong describes message passing as the process in which an object sends information to another object, or asks the other object to invoke a method. Last, polymorphism enacts behaviour by allowing different objects to respond to the same message differently [Armstrong06, p.127]. Behaviour and structure are interconnected in the sense that behaviour is a way of manipulating structure, but behaviour must also support the actions of the system. The OO perspective used in the taxonomy to identify concepts as either Structural or Behavioural matches very well with the "Gang of Four" classification concerning pattern purpose, namely Structural, Behavioural, or Creational as described in section 3.7.1 on page 44. It also matches quite well with the types of UML diagrams targeting Structural and Behavioural conduct as described in section 2.3.

In order to perform a meaningful evaluation of the "Gang of Four" design patterns, the general concepts and themes inherently expressed by the design patterns must be understood. The pattern authors understanding of OO concepts will naturally be reflected in the pattern descriptions, but pattern users may have a different understanding as Armstrong's survey explains. We must therefore establish the basic concepts and themes reflected in the "Gang of Four" patterns. Luckily, this is not as difficult as it sounds. Several concepts related to OO development in class—based languages are summarised in chapter one of the "Design Patterns" book [Gamma95, p.11-28]. Obviously, this thesis adapts the concepts and themes described by Gamma et al., especially because Java is a class—based language like C++ and Smalltalk. Not doing so would be a topic for a different thesis altogether, for example an evaluation targeting prototyped—based languages, where OO concepts such as classes and inheritance has no or at least a different meaning.

2.1.1. Concepts

The OO concepts described in chapter one of the "Design Patterns" book [Gamma95, p.11-28] are explained in relation to the languages used, i.e. C++ and Smalltalk, as well as the problems the "Gang of Four" patterns are designed to solve. For example, the concept of mixin classes seems only relevant in a language like C++ that allows multiple functional inheritance (as opposed to mixin types, in Java in form of interface implementation that requires composition). The delegation and acquaintance concepts directly refers to one of the general "Gang of Four" design principles as described in the next section. The number of concepts is around forty in total of varying granularity, though many of them have fine granularity. Thirty—eight is the number of **bold—faced** words, i.e. concepts, with associated explanations on pages 11-28 in [Gamma95]. Some have identical meanings, though, for instance request and message. In addition, a few concepts are introduced as part of a figure or section heading, for example application. Table 2.2 lists the identified concepts in alphabetical order. Because the concepts are described in relation to C++, the table also supplies comments related to Java.

Table 2.2 — "Gang of Four" concepts

Concept	Description	Java 6 Remarks
Abstract class	A class whose main purpose is to define a common interface for its sub—classes [Gamma95, p.15].	Supported.
Abstract operation	The methods an abstract class declares but does not implement [Gamma95, p.15].	Supported. Abstract methods can only be declared in abstract classes. Interfaces also declare methods with no corresponding implementation.
Acquaintance	An object uses another object in a loosely coupled fashion [Gamma95, p.22].	Composition, supported.
Aggregatee	The object owned by the aggregator [Gamma95, p.23].	Composition, supported. Also called Aggregate Member.
Aggregation	An object owns or is responsible for another object [Gamma95, p.22].	Composition, supported.
Aggregator	The object owning the aggregatee [Gamma95, p.23].	Composition, supported.
Application	Type of program where internal reuse is important [Gamma95, p.25].	
Black—box reuse	Reuse by object composition [Gamma95, p.19].	
Class	An object's implementation is defined by its class [Gamma95, p.14].	Supported since Java is a class—based language. Java also provides access to an object's class at runtime.
Class inheritance	Defining new classes in terms of existing classes for code and representation sharing [Gamma95, p.15,17].	Java supports single inheritance only, but a class can implement several interfaces.
Client	The object that issues a request [Gamma95, p.11].	
Concrete class	A class that is not abstract [Gamma95, p.15].	Supported.

Table 2.2 — "Gang of Four" concepts

Concept	Description	Java 6 Remarks
Delegate	The object being forwarded a message in delegation is called a delegate [Gamma95, p.20].	Another form of composition, supported.
Delegation	Using object composition, an object receiving a message forwards the message to its delegate passing itself along as an argument [Gamma95, p.20].	Supported. Delegation implies composition, but composition does not imply delegation as aggregation and acquaintance could also be used.
Dynamic binding	Runtime association of a message to an object and one of its methods [Gamma95, p.14].	Supported via polymorphism. The signature of the method is determined at compile—time, but the actual type of the (polymorphic) object is determined at runtime [Sierra06, p.111].
Encapsulation	The internal state of an object cannot be accessed directly, and its representation is invisible from outside the object [Gamma95, p.11].	Supported, but must be enforced by access modifiers.
Framework	A set of cooperating classes that makes up a reusable design for a specific class of software [Gamma95, p.26].	
Generics	Parameterised types as used in certain languages [Gamma95, p.21].	Supported, including support for bounds and wild—card types (not found in C++). Type information is not always present at runtime (erasure), and generics do not allow (static) template specialisation as in C++. Corresponds to parameterised types.
Instance	A created object is a unique instance of its class [Gamma95, p.15].	Supported. Instances can be compared based on identity or based on equivalence (equals).
Instance variable	The internal data of an object are represented as instance variables [Gamma95, p.15].	Supported. Can also be accessed via reflection.
Instantiation	Objects are created by instantiating a class [Gamma95, p.15].	Supported. Objects can also be created reflectively.
Interface	The set of all signatures for a given object [Gamma95, p.13].	Interface as a type is supported, but a class may also represent the set of all signatures of an object.
Message	An object invokes a method when it receives a message. Messages are the only way to get an object to invoke a method [Gamma95, p.11].	Supported.
Method	A typical name used to describe the procedures that operate on object data. If encapsulation is enforced, methods are the only way to change the internal state of an object [Gamma95, p.11].	Supported. Can also be accessed and/or invoked reflectively.
Mixin class	A class providing an optional interface or functionality to other classes, but it is not intended to be instantiated and requires multiple (functional) inheritance [Gamma95, p.16].	Mixin classes are not supported, but mixin types in form of interfaces that require composition are[1]. Java supports dynamic proxies that allow implementation of interfaces at runtime (reflection).

[1] The `java.io.Serializable` and `java.lang.Cloneable` interfaces are each a hybrid between a mixin class and a mixin interface. Java has built—in support for both of these special interfaces that cannot be described by standard interface semantics, for example default serializable behaviour in form of private inherited methods.

Table 2.2 — "Gang of Four" concepts

Concept	Description	Java 6 Remarks
Object	An object packages both data and procedures that operate on the data [Gamma95, p.11].	Supported. All classes inherit `java.lang.Object`.
Object composition	An alternative to class inheritance that composes (assembles) objects to obtain complex functionality [Gamma95, p.18].	Supported.
Operation	Synonym for method.	Supported.
Override	A sub—class may override a method defined in its parent class [Gamma95, p.16].	Supported unless the method is declared final. Java supports covariant return types.
Parameterised type	A type that is declared without specifying all the types it uses until the point of usage [Gamma95, p.21].	In Java a synonym for generics.
Parent—class	A parent—class defines data and methods sub—classes can inherit [Gamma95, p.15].	Supported, also called super—class. Java provides access to the super—class at runtime as well as the actual instance.
Polymorphism	Substitution of objects with similar interfaces at runtime using dynamic binding [Gamma95, p.14].	Supported. All non—primitive classes are polymorphic in Java as they inherit `java.lang.Object` and define their own type. See dynamic binding.
Request	Synonym for message.	
Signature	The name, parameter, and return type of a method [Gamma95, p.13].	Supported. Can be accessed reflectively.
Sub—class	A sub—class inherits (all) data and methods from its super—class [Gamma95, p.15].	Supported, but access modifiers determine data and methods inherited.
Sub—type	A type is a sub—type of another type if its interface contains the interface of its super—type [Gamma95, p.13].	Supported.
Super—type	A type is a super—type of another type if its interface is included in the interface of a sub—type [Gamma95, p.13].	Supported.
Template	Parameterised types as used in C++ [Gamma95, p.21].	Not supported by Java.
Toolkit	A class library [Gamma95, p.26].	
Type	A name used to denote a particular interface [Gamma95, p.14].	Supported, but type is usually used to describe the functionality listed under Interface. A type is thus a class or interface.
White—box reuse	Reuse by sub—classing [Gamma95, p.19].	

Of the eight fundamental concepts identified by Armstrong listed in table **2.1**, all but message passing are described as a distinct concept in some form by Gamma et al., though some using slightly different names and meanings, for example polymorphism and dynamic binding. However, message passing is implicitly part of the message (request) and method (operation) "Gang of Four" concepts. This is similar to method invocation not being described either. We therefore conclude that the concepts are encompassed by the taxonomy suggested by Armstrong. As the "Design Patterns" book predates Armstrong's taxonomy, it is possible that the tight resemblance is an indication of how influential and/or how widely used the "Gang of Four" patterns have been – and still are. On the other hand, many of the concepts described are well—known OO principles that any developer **has** to know to design and implement durable OO designs. Concepts such as classes, inheritance, polymorphism, etc., cannot be ascribed to Gamma et al.

There is only one concept we disagree with the definition of, namely encapsulation. From our perspective, the merging of the different meanings of encapsulation and information hiding by Armstrong is flawed, even though Gamma et al. do the same [Gamma95, p.11]. Consequently, the Gamma et al. definition of sub—class is faulty as well, because information hiding will determine the data and methods to inherit (see Java remark). We consider encapsulation and information hiding as two distinct concepts as explained by Rogers [Rogers01]:

> "***Encapsulation is a language construct*** *that facilitates the bundling of data with the methods operating on that data.* ***Information hiding is a design principle*** *that strives to shield client classes from the internal workings of a class.* ***Encapsulation facilitates, but does not guarantee, information hiding****.* ***Smearing the two into one concept prevents a clear understanding of either****.*

The remarks regarding Java 6 functionality in table 2.2 clearly indicates that the concepts are well—suited for a Java environment. Hence, the concepts are adapted to represent our understanding of OO concepts as well, keeping the distinction between encapsulation and information in mind.

2.1.2. Themes

The first chapter of the "Design Patterns" book also describes a set of reoccurring themes that permeate the "Gang of Four" approach to OO development and their design patterns [Gamma95, p.11-31]. The concepts listed in the previous section facilitate the themes, but these themes must also be understood in order to understand the "Gang of Four" design patterns. Two important principles summarise their ideas:

1. Program to an interface, not an implementation [Gamma95, p.18]; and
2. Favour object composition over class inheritance [Gamma95, p.20].

Perhaps more than the design patterns themselves, we consider these principles evidence of how significant the "Design Patterns" book has been in OO development. They cover the concepts listed in table 2.2, and express the need for abstraction, loose coupling, and flexibility in OO (re—) designs. By using interfaces, clients remain unaware of the specific types (and classes) of objects they use [Gamma95, p.18]. Interfaces are directly supported as a concept in Java. Gamma et al. promote indirection as a mean to achieve decoupling, flexibility, and reuse, and encapsulation, information hiding, and parameterised types may aid in achieving this as well [Gamma95, p.19,22]. They prefer dynamic (e.g. runtime) relationships as opposed to static ones and thus favour object composition over implementation inheritance [Gamma95, p.20]. Delegation is the extreme example of composition, which can always be used to replace inheritance [Gamma95, p.21]. However, dynamic, highly parameterised software is harder to understand than more static software [Gamma95, p.21], which thus may influence the pattern descriptions. The need for re—design may still arise, but by following the two principles and utilising relevant design patterns expressing them, the process of re—design becomes easier, because then aspects of a system structure may vary independently of other aspects [Gamma95, p.24]. As Java 6 supports the concepts from the previous section, these themes can be expressed in Java providing a prudent designer.

Many of the principles and themes described by Gamma et al. are represented by the General Responsibility Assignment Software Patterns (GRASP) [Larman04]. Grand provides a Java version of these patterns in [Grand99, p.51-87]. These patterns are *not* design patterns as such. They do not target a specific problem, but provide insight into how responsibilities should be assigned to classes to achieve a well—structured design, which is easily understood and maintained [Grand99, p.52]. For example, Low Coupling and High Cohesion [Grand99, p.53] is closely related to both of the above principles, and Polymorphism [Grand99, p.69] is naturally related to concepts such as polymorphism, super—class, sub—class, inheritance, etc. Several of these themes have by some been promoted to design patterns. Grand provides Delegation [Grand98, p.53] and Interface [Grand98, p.61] patterns, but whether such fundamental concepts are best expressed as design patterns is doubtful in our view.

2.2. Object—Oriented Methods

An Object—Oriented Method (OOM) provides a set of techniques for analysing, decomposing, and modularising software system architectures [Schmidt, p.4]. The techniques may be applied in different phases of the software lifecycle, e.g. in the analysis, design, and implementation phases (see figure 2.1) [Schmidt, p.6]. An OOM can for example describe how the requirements found in the analysis can be transformed into a software model consisting of objects [SEI]. Despite the widespread use of OO as explained in section 2.1 on page 13, there is not only a lack of consensus regarding the formalisation of the relevant concepts and principles inherent in OO, but also on how to approach the overall design process. Hence, numerous OO methods have been developed, each trying to remedy this, for example Rational Unified Process (RUP) [RUP] or Model—Driven Architecture (MDA) [MDA], but none are an accepted industry standard. Different software development processes are used in various OO methods, such as the sequential Waterfall model, or the Iterative, Spiral, or Agile development. All but the first are based on the idea of repair and evolution and are in some form iterative in nature, while the Waterfall model is more static and employs replacement. RUP, for example, uses iterative development.

2.2.1. Patterns

The traits of a given OOM and the procedures used will guide the OO development. It is difficult to speculate on the impact a given OOM has on the application of design patterns, if any, without in—depth knowledge and experience with each method. Vlissides, one of the "Gang of Four" members, argues that patterns do not need tools or methodologies to be effective [Vlissides97, i.4]. Based on experience we agree. However, certain methodologies directly address the use of patterns or other techniques, such as UML. Responsibility Driven Design has no mention of patterns what so ever [WirfsBrock90], while Extreme Programming (XP), for example, de—emphasises or even ignores the need for patterns [Fowler04].

Nevertheless, we do not even see XP as incompatible with design patterns. XP is a software engineering methodology developed mainly by Kent Beck and Ward Cunningham, the duo that also introduced software design patterns [Beck87] as described in section 3.2. It is typically used in Agile development, and is iterative in nature. It advocates the use of Evolutionary Design contra to Planned Design under certain preconditions [Fowler04]. Central is the use of several enabling practices, such as testing, refactoring, and continuous integration that embodies and encourages certain values, such as simplicity and communication. This allows changes to be performed much faster and cheaper, thus reinforcing the enabling practices [Fowler04; PPR]. Due to the evolutionary nature of this methodology, it is often believed that Object—Oriented Analysis (OOA), Design (OOD), and design patterns are incompatible with XP. Others, such as Fowler, think that patterns are underrated within XP, and are in no way contradictory to the paradigms of XP and that program code developed using the methodologies can evolve into patterns during refactoring. We agree, and conclude that the enabling practices of XP to some extent can be viewed as a form of pattern discovery, or mining (see section 3.8.1).

Designers often feel strongly about their preferred development method, OO or otherwise, sometimes to the point of a religious belief. In many respects, we see design patterns as orthogonal to OO methods, because the objects and knowledge they represent are independent of which method produced the (initial) context to which a pattern can be applied. While design patterns can be grouped in collections, such as pattern systems and languages as explained in section 3.7, the effect of this in our experience rarely influences their practical application when used in a specific process. Their application is thus largely independent of the OOM used.

2.3. Unified Modelling Language

Regardless of OO method and processes used, the Unified Modelling Language (UML) is generally used for object modelling and illustration [UML05]. UML is an extensible general—purpose object modelling and specification language used to create abstract (design) models illustrated graphically. It is not limited to modelling software, but is widely used in various OO methods. The model of the system can be described using a Functional Model (user's point of view); using an Object Model (structural); and/or using a Dynamic Model (internal behaviour). Different models use different types of diagrams, for example a Use Case Diagram for the Functional Model; a Class or Object Diagram for the Object Model; and a Sequence Diagram for the Dynamic Model [UML05].

UML can be used in various development phases. Use Case Diagrams can specify demands the analysis must adhere to (see also [Cockburn01]). Class and Object Diagrams can be used in the design phase to describe the identified classes and objects, and Sequence Diagrams can illustrate the behaviour of classes, objects, and methods. As the design evolves, so must the diagrams. UML does not have built—in notations for all features found in Java 6, such as annotations, but can be adapted by user—defined extensions.

2.3.1. Patterns

Patterns related to OO development commonly use UML models, because the pattern participants (i.e. classes and objects) are easily illustrated using the UML models. Graphical illustrations of pattern functionality are a requirement to ensure proper quality of the pattern as well as a meaningful description of its functionality as explained in section 3.3 and 3.5, respectively. The "Gang of Four" patterns predate UML, but use other forms of closely related types of illustrations. In this thesis, only UML Class diagrams are used. Section 6.2 details the usage, but the evaluation produces a Class diagram for each pattern implementation.

2.4. Object—Oriented Analysis

As illustrated in figure 2.1, Object—Oriented Analysis (OOA) is the first phase in OO development, excluding mundane tasks such as sale, legal affairs, project planning, and management in real corporate environments. A typical scenario is that a given client has produced a (far from complete) list of demands identifying the overall behaviour of the system that must be built. The demands can be specified in a number of ways, for example as Use Cases [Cockburn01, p.1-3]. The analysis is concerned with developing software engineering requirements and specifications from these demands, often expressed in form of a conceptual object model, as opposed to the traditional data or functional views of systems [Larman04; SEI]. The analysis is a discovery process that determines *what* is to be built, and the design determines *how* it is done [Schmidt, p.6; SEI; WirfsBrock90, p.5]. This is done by identifying the (real—life) abstractions, concepts, responsibilities, and relationships present in the system in order to form a conceptual model of the system while adhering to the demands. The practical procedures on how to do this as well as how the model is described are typically dictated by the Object—Oriented Method (OOM) used.

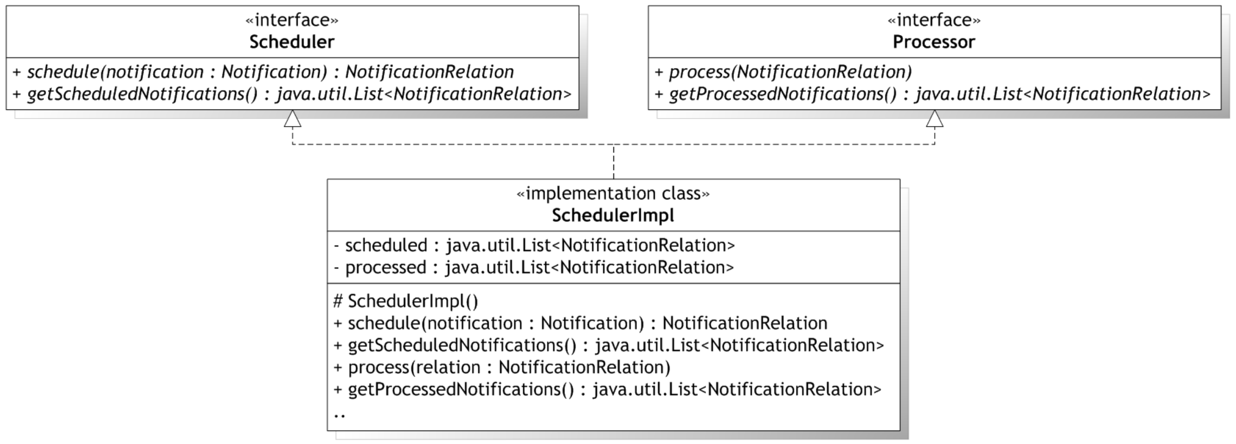
Example 2.1 — Consider the task of designing a sophisticated notification mechanism able to notify subscribers when certain events occur with support for different means of deliveries. Example usage could be in Internet applications that must notify users when certain events occur, data driven or otherwise, or as a mean to monitor application usage and abnormalities. The demands set forth by the client will (or should) specify the overall context and desired functionality. From these, the analysis must identify the relevant concepts and their interactions forming the conceptual model of the notification mechanism. The notification mechanism is used as a continuous example in the first part of this thesis. This chapter offers a number of examples illustrating how different development phases and patterns may influence the development of such a mechanism.

Simplified, the abstractions and concepts could include User, Subscription, Notifiable, Event, Notification, Scheduler, Processor, Delivery, Formatter, and Message; a User, for instance, could be an abstraction of a logical entity known to the system, such as an identified human or program, while Notifiable is a more abstract concept related to functionality rather than a physical entity. To express the relationships, the model could specify that a User can have different Subscriptions pertaining to different Notifiable contexts, e.g. subscriptions to receive different kinds of notifications. When a certain Event occurs related to a Notifiable context, a Notification will be created and scheduled by a Scheduler. Based on the Notification, relevant Subscriptions will be identified by the Processor handling the Notification, and Messages will be created and formatted by a Formatter as required by the Delivery mechanism preferred by the User. Furthermore, the conceptual model must describe the responsibilities related to the core functionality of each concept; for example, the type of Delivery must ensure that a proper type of Message is formatted and delivered, or perhaps even create it. Once the conceptual model is defined and described, the design phase will determine how the model must be utilised from a software perspective, i.e. how it should be transformed into program code ■

2.5. Object—Oriented Design

Object—Oriented Design (OOD) is the process of defining the software objects and collaborations forming an OO model of a software system in order to implement the identified requirements found during the analysis [Larman04; SEI; WirfsBrock90, p.10]. The design phase is thus the second phase in OO development and where the analysis determines *what* is to be built, the design is a process of invention and adaptation that determines *how* it is done [Schmidt, p.6; SEI; WirfsBrock90, p.5]. While the conceptual model identified during the analysis describes conceptual objects unrelated to software terminologies, the OO model describes the computational software objects needed to implement the functionality of the model instead. The mapping between objects is rarely or never one—to—one. The system is decomposed into (complex) software objects of relevant granularity, some perhaps mapping to existing re—usable components. Detailed descriptions consisting of message protocols ("patterns of communication"), attributes, and methods ("public behaviour") at the level of individual objects should be specified [WirfsBrock90, p.10,28].

Example 2.2 — To implement a design for the notification mechanism described in example 2.1 the conceptual model must be transformed into a model of collaborating software objects. Model objects such as User, Subscription, Notification, Message, Formatter, and Delivery may map directly to similar software objects, or *types*, e.g. to `User`, `Subscription`, `Notification`, `Message`, `Formatter`, and `Delivery` software objects, respectively. A software object may be designated as *abstract*, which will require specific implementations for usage as well. For example, the Delivery object could map to a `Delivery` interface with specific implementations such as `EmailDelivery`, `SMSDelivery`, and `SNMPDelivery`, which in turn could require abstract `Message`, `Formatter`, and `Subscription` types as well. Coarse or complex model objects may require numerous software objects or even libraries to represent the functionality. For example, an object doubling as both a Scheduler and Processor must implement a `Scheduler` and a `Processor` interface. The UML Class diagram below shows such a scenario.


«interface»
Scheduler
+ schedule(notification : Notification) : NotificationRelation
+ getScheduledNotifications() : java.util.List<NotificationRelation>
«interface»
Processor
+ process(NotificationRelation)
+ getProcessedNotifications() : java.util.List<NotificationRelation>
«implementation class»
SchedulerImpl
- scheduled : java.util.List<NotificationRelation>
- processed : java.util.List<NotificationRelation>
# SchedulerImpl()
+ schedule(notification : Notification) : NotificationRelation
+ getScheduledNotifications() : java.util.List<NotificationRelation>
+ process(relation : NotificationRelation)
+ getProcessedNotifications() : java.util.List<NotificationRelation>
..

Conversely, certain model objects may not even require a structural counterpart such as a class/object; this could be the case with an Event object, which could be defined as the executing context creating and scheduling a `Notification` object simply using method invocations. On the other hand, some software objects may have no direct conceptual counterpart, as for instance a `NotificationRelation` object expressing a specific relationship between a `Notification` scheduled for later processing using a certain `Delivery` type.

Once the software objects have been identified, their responsibilities and relationships must be established and described ("fine design granularity"). As seen in the UML diagram above, the `Scheduler` object could have a `schedule(Notification)` method as well as a `getScheduledNotifications()` method to return the `Notification` objects scheduled by *that* `Scheduler` contained in `NotificationRelation` objects. Similar, the `Processor` could have a `process(NotificationRelation)` method as well as a `getProcessedNotifications()` method to return relations processed by *that* `Processor`. Here, the Scheduler and Processor are the same type (and instance), but that is not a requirement. The design will not only identify the attributes and methods, but also the overall internal logic of the methods. Finally, depending on the demands at hand, the mechanism could be designed as a standalone library used in a large OO systems, or as part of the system itself ("large design granularity"). If designed as a library, it could be used in other design scenarios, but this raises the need for a good, durable, and flexible design even more ■

The practical procedures on how to execute the design phase, i.e. how objects and responsibilities are identified as well as how the design is presented, are typically described by OOM used combined with personal experience, for example using the Responsibility Driven Design process suggested by Wirfs—Brock et al. [WirfsBrock90]. However, Fowler states that it can be hard to distinguish between the analysis and design phase in practice [Fowler97]. Wirfs—Brock et al. do not even label the initial phase as the analysis phase, but as part of the design phase. Nevertheless, the design phase requires the specification of concepts nonexistent in analysis, such as the logic of object methods or the types of the attributes of an object or class [SEI], as for example a `name` attribute of the `User` class identified in example **2.2** having the type `java.lang.String`. Furthermore, the design may seem closely related to the implementation, and in particular OO Programming Languages (OOPL), because it will typically be represented by diagrams such as UML Class and Object Diagrams sharing similar notions [UML05]. The design does not require an OOPL for implementation, but an OOPL will facilitate the implementation considerably, though the variant of the OO paradigm supported by it will also play a role.

The choice of programming language is important already in the design phase. The language may implicitly affect the design if it affects the design patterns used. In [Norvig96], Norvig differentiates between three types of programming relevant for the design: a) Programming In a language; b) Programming Into a language; and c) Programming On a language [Norvig96, p.58]. In case a), the design is constrained by what the language offers. In case b), the design is done independently of any language, then implemented using features available in the chosen language. In case c), the design and language meet half way. Norvig explains this as programming into the language you wish you had; a language built on the actual language chosen. Ideally, patterns represent case b). Unfortunately, the choice of programming language may already have been made, for example based on client demands, and this may force a given type of design and programming relation.

2.5.1. Patterns

In our experience, real—world problems seldom map to software objects representing real—life entities, but rather to programmatic abstractions, i.e. objects of varying granularity, of required functionality. As stated, identifying the objects, their relationships, and interactions is no trivial matter. This is where software patterns come in handy, because they provide solutions to many of the problems faced while designing such objects (or "components"): a given pattern has already identified a set of objects, relationships, and responsibilities required in a given scenario. The knowledge represented by the patterns can be adapted and utilised by the designer yielding familiar variants of already well—known scenarios. Chapter 3 provides a thorough introduction to pattern theory, not necessarily related to OO.

Patterns can describe solutions to various areas and the pattern concept originated within the field or architecture. According to Vlissides, a common misconception is that software patterns are just for OO design and implementation, but patterns can be applied in numerous areas [Vlissides97, i.7]. Furthermore, it is commonplace that any pattern related to design, software or otherwise, is dubbed design pattern. However, according to Lea, within computer science, the term design pattern reflects a categorisation to identify a specific range of patterns related to the design of software systems [Lea00, i.3]. In accordance with this, Buschmann et al. suggest a pattern taxonomy that categorises patterns pertaining to design as architectural patterns, design patterns, or idioms depending on their range of scale or abstraction [Buschmann96, p.12-15]. Others, for example Hohmann [Hohmann98], extend the taxonomy to include analysis patterns as described by Fowler [Fowler97]. Table 2.3 illustrates the extended and slightly modified taxonomy used in this thesis.

Table 2.3 — Pattern taxonomy

Category	Description	Target
Architectural Patterns	An architectural pattern expresses a fundamental structural organisation schema for software systems. It provides a set of predefined sub—systems, specifies their responsibilities, and include rules and guidelines for organising relationships between them [Buschmann96, p.12].	Entire (sub—) systems, applications, and frameworks
Analysis Patterns	An analysis pattern reflects the conceptual structures of business processes rather than actual software implementations [Fowler97, p.XV].	Domain and Business Object Model
Design Patterns	A design pattern provides a scheme for refining the sub—systems or components of a system, or the relationships between them [Buschmann96, p.13]. It does so by describing communicating objects and classes that are customised to solve a general design problem in a particular context [Gamma95, p.3].	Micro—architectures within sub—systems or components
Idioms	An idiom is as a low—level pattern, specific to a particular programming language that describes how to implement particular aspects of components or the relationships between them using the features of the given language [Buschmann96, p.14]. An implementation of a design pattern that is unique to the language chosen is also considered an idiom in this thesis.	Classes, Objects, and Methods

Architectural patterns have large design granularity and are thus used early in the design phase. Analysis patterns are related to the domain and business object model of the system, if any, while design patterns can be used throughout the entire design phase. Though design patterns target sub—systems, they are only used to define specific and encapsulated functionality *within* the systems. Hence, the granularity of analysis patterns if often larger than design patterns. Idioms have fine granularity and are closely related to the implementation phase. This is illustrated in figure 2.1 on page 13. Design patterns (indicated with the grey row in table 2.3) are the category of patterns evaluated in this thesis, and unless explicitly stated otherwise, the term design pattern as used in this thesis indicates this category of software patterns.

2.5.1.1. Architectural Patterns

Patterns categorised as architectural express fundamental structural organisation schemas for software systems. They have large granularity and their introduction into the early stage of the design phase will greatly influence the system, including the detailed design of sub—systems and how different parts collaborate and communicate [Buschmann96, p.25-26]. Architectural patterns are still only applicable for a given scenario and do not represent a complete software architecture. Hence, several patterns may need to be applied to form the entire system. As is the case with design patterns, architectural patterns may be classified according to their overall purpose, for example as Adaptable Systems, Interactive Systems, or Distributed Systems [Buschmann96, p.26]. Buschmann et al. also suggest a number of patterns, known as the "POSA" patterns, for instance the architectural Model—View—Controller Pattern [Buschmann96, p.125], which we have used extensively in the design of numerous applications[2].

Example 2.3 — The notification mechanism described in example **2.1** on page 21 can be designed as a standalone library, or even framework allowing for customisation in form of an API. It is reasonable to assume that Users, Subscriptions, scheduled Notifications, and Messages must be serialised to a perhaps permanent store to handle identification of pre—existing users and subscriptions, application shutdown, and re—deliveries. The Layers [Buschmann96, p.31] architectural pattern suggest to divide the architecture into layers dedicated to different tasks, for example a database layer handling the persistence of the objects and a layer handling the application logic. An API can by it self be considered a variant of the Layers pattern [Buschmann96, p.46]. Fowler identifies specific variants of the Layers pattern, for example Two—Tier Architecture [Fowler97, p.240] corresponding to the scenario in this example ■

While architectural patterns can have tremendous impact on the design of the software system, we for the most part see design patterns as autonomous from their application. Of course, a design pattern such as the View Handler Pattern [Buschmann96, p.291] is a refinement relevant to the infrastructure offered by the Model—View—Controller Pattern, and Creational "Gang of Four" design patterns behaviour could be affected if the architectural Reflection Pattern is applied [Buschmann96, p.293]. However, the granularity of design patterns and their general versatility makes them useful and relevant in many different architectural contexts, for example the "ever—applicable" Iterator [Gamma95, p.257] and Decorator [Gamma95, p.175] patterns.

2.5.1.2. Analysis Patterns

At the core of many Information Systems (IS) is the business object model that do represent real—world entities, for example a mapping of a company to a `Company` object. The business object model is said to represent the *domain* of the system. Hence, the business object model is often closely related to the conceptual model constructed in the analysis phase, but it is often just a relative small part of the entire system. However, the

[2] The "POSA" pattern system contains architectural patterns, design patterns, as well as idioms.

business object model is a truly pivotal part because it effectively defines the system behaviour: instances of model objects are to be manipulated by the system while adhering to the business rules, thus defining the overall behaviour. Because a well—designed business model is so important, patterns have even been developed targeting the domain specifically. Fowler presents a comprehensive set of analysis patterns targeting reusable parts of business object models [Fowler97], some even at a granularity level corresponding to architectural patterns as illustrated with the Two—Tier Architecture [Fowler97, p.240] pattern in example **2.3**. However, a business object model cannot stand by it self, or may not even be utilised in a design. The total functionality required to manipulate the business object model, directly or indirectly, will in terms of objects vastly out number objects in the model. For example, in Internet applications auxiliary objects are required to handle of incoming browser requests, security, logging, persistence of data, errors, rendering, etc. Such objects rarely have real—world counterparts. Examples could be a `Logger` object to log diagnostic messages; a `Request` object to represent input to the application; or in the case of example **2.2**, a `NotificationRelation` object. Even if a system does not use a business object model as such, it will always have a core functionality that requires many auxiliary objects with additional functionality. Hence, the design of business system is not just about designing the business object model, but naturally about designing the entire system. The use of analysis patterns does not exclude the need for design patterns.

2.5.1.3. Design Patterns

The design pattern categorisation is almost directly based on the "Gang of Four" definition of design patterns. The precise definition used in this thesis is shown in table **2.3**. The "Gang of Four" patterns [Gamma95] are a collection of patterns targeting the domain of design problems closely related to pragmatic problems found in general OO designs. Gamma et al. define design patterns as [Gamma95, p.3]:

> *“ **Descriptions** of **communicating objects** and **classes** that are **customized** to solve a **general design problem** in a **particular context**.*

Gamma et al. further explain that the domain of design patterns is describing concepts and structures beyond individual objects and classes up to the granularity level of refinement of OO sub—systems. Algorithms are not considered a pattern by this, or other, definitions; they solve computational problems, not design problems. This definition of design patterns is roughly equivalent to the domain of the design pattern categorisation described by Buschmann et al. [Buschmann96, p.13], except that Buschmann et al. do not explicitly mention OO. Our definition implies an overall OO domain. Borchers [Borchers99, p.2] offers a broader definition that does not require class—based languages, or even a specific domain:

> *“ A software design pattern is generally considered to be a **proven solution** of a **recurring software engineering problem** that balances the **competing design constraints optimally** for a **certain type of situation**.*

This broader definition implies the choice of pattern has consequences. This is an important aspect of patterns as explained in chapter **3**. As the "Gang of Four" patterns are evaluated in this thesis, we see no reason not to use the "Gang of Four" definition. Hence, design patterns both describe the "Gang of Four" patterns, but also the category of patterns targeting the same domain. Many other commonly used design patterns have been published as well, for example the "POSA" patterns[3] [Buschmann96]. The problems design patterns address arise more frequently than issues purely related to the business object model as targeted by analysis patterns.

Example 2.4 — In example 2.2 on page 22, we identified the need for an abstract `Delivery` type with concrete implementations representing different means of delivery mechanisms, such as `EmailDelivery`, `SMSDelivery`, and `SNMPDelivery` to deliver messages via email, SMS, or the Simple Network Management Protocol (SNMP), respectively. The design must ensure that only the proper types of `Message` objects will be delivered using a given delivery mechanism; that the messages will be formatted to a representation suited for such a delivery; and that additional means of delivery could fairly easy be added – but how?

A designer familiar with the "Gang of Four" patterns will immediately recognise that the Abstract Factory [Gamma95, p.87] and Factory Method [Gamma95, p.107], and perhaps the Singleton [Gamma95, p.127] and Template Method [Gamma95, p.325], design patterns could be utilised here. The Abstract Factory pattern can be used to ensure that the `Delivery` and `Formatter` types used together are to correct ones, making use of the Factory Method to defer the actual creation elsewhere, which also allows for easy introduction of new `Delivery` and `Formatter` types. The Singleton pattern can be used to ensure that the notification mechanism creates `Delivery` and `Formatter` objects in a uniform way not breaking the loose coupling offered by the factory patterns by ensuring that only a single factory is available. Finally, if the notification mechanism is designed as a library, the Template Method pattern can be used to define hooks in various objects that the client can override to add additional functionality or means of delivery ■

2.5.1.4. Idioms

Buschmann et al. describe an idiom as a low—level pattern, specific to a particular programming language that describes how to implement particular aspects of components or the relationships between them using the features of the given language [Buschmann96, p.14]. The classification is based on the work by Coplien in [Coplien91]. We furthermore claim that any design pattern is *implemented* as an idiom if the specific implementation is unique to the language. An example is a Java implementation of the Singleton pattern using the `synchronized` statement to ensure that only a single instance is created. While the implementation can be considered a Java idiom, the abstraction is still a design pattern. This indicates a closer relation between design patterns and idioms and thus the implementation, which is illustrated in figure 2.1 on page 13. Buschmann et al. also note that certain design patterns provide a source for idioms [Buschmann96, p.350].

2.6. Object—Oriented Programming

The objects described in the design phase must be transferred into program code. Languages supporting an OO paradigm will facilitate this process, for example by directly offering language constructs such as objects,

[3] The Command Processor [Buschmann96, p.277] design pattern implemented as part of the Command pattern in section 8.3.2.3 is in fact a "POSA" pattern.

classes, and inheritance. Other types of languages can be used as well, but will require more work during implementation. The implementation phase is often called Object—Oriented Programming (OOP), but OOP is also commonly used to denote OO in general. This thesis refers to the implementation phase as OOP. More so, Sethi describes OOP as a programming paradigm, where execution is normally imperative and data is conceptualised in cooperating and communicating objects representing logical entities [Sethi96, p.15-16], i.e. closely related to the programming language chosen. Languages supporting an OO paradigm are called OOP languages (OOPL).

Example 2.5 — To implement the design of the notification mechanism described in example 2.2 on page 22 a programming language must be chosen. If the design is described using UML Class Diagrams, the conceptual model entities are represented by classes. All User objects are thus represented by the `User` class, Delivery objects by a `Delivery` interface, etc. A class—based language like Java would be the obvious choice to transform the design to program code, because the language directly support classes and inheritance as part of the syntax. The User type could directly be defined as `class User`, and a specific Delivery implementation as `class EmailDelivery implements Delivery`, for example. On the other hand, if a prototype—based language is chosen as the programming language, the relationship between `Delivery` and `EmailDelivery` would have to be expressed differently ■

Implementation is not the last phase in the software life—cycle, but the last phase relating to the design. Testing, deployment, and evaluation are key phases that might spawn new demands, which in turn may cause the development cycle to re—iterate.

2.6.1. Object—Oriented Programming Language

Ideally, the relation between the design and implementation should be in form of Programming Into a language as described by Norvig [Norvig96, p.58], i.e. the design should be designed independently of any programming language (see section 2.5). A programming language that has built—in support for an OOP paradigm is an obvious choice to use when the design must be implemented, because such languages directly support the object notions of objects, encapsulation, information hiding, polymorphism, and in case of class—based languages classes and inheritance [SEI; WirfsBrock90, p.10]. In short, most of the concepts from the concepts presented in section 2.1, which Java 6 does. Below, table 2.4 offers a quick comparison of some of the more interesting features found in C++, Smalltalk, and Java 6 based on [Gamma95; Gosling05; Stroustrup91].

Table 2.4 — Comparing C++, Smalltalk, and Java 6

Language	Paradigms	Type System	Example Features	Implementation
C++	Hybrid, Class—based, Imperative, Procedural	Strong, Static	Templates, Multiple Inheritance, Overloading, Overloaded Operators	Static (RTTI), Compiled
Smalltalk	Pure, Class—based, Imperative, Reflection	Strong, Dynamic	Duck Typing, Inheritance, Overloading, Overloaded Operators	Runtime, Bytecode, JIT
Java 6	Hybrid, Class—based, Imperative, Reflection, Concurrent	Strong, Static	Generics, Inheritance, Interfaces, Overloading, Dynamic Proxies, Annotations	Runtime, Bytecode, JIT

Other types of languages can be used as well to implement a design, though not as easily. For example, a class—based design implemented in a language not supporting classes must utilise, or even invent, means to represent classes and inheritance. This corresponds to Programming On a language [Norvig96, p.58]. Programmatic features must be introduced to support the requirements of the design. OOP languages will be easier to use in conjunction with design patterns as well because design patterns build on the fundamental OO concepts.

2.6.2. Patterns

If design patterns were utilised in the design phase, the patterns will supply canonical implementations or at least examples on how to implement the required functionality. As Gamma et al. already note, the choice of language will affect the pattern application because the language will ultimately decide what can and what cannot be done (easily) in light of supported programming paradigms [Gamma95, p.4]. In case of a Programming In a language or Programming On a language relation between the design and programming language [Norvig96, p.58], design patterns can help establish wanted features. That is, to avoid limitations of the implementation language [Norvig96, p.4]. However, the pattern examples must be modified to the language chosen and to the scenario at hand, which may raise issues in case the language does not support features utilised in the examples or the problems inherent to the scenario. We have already established that Java 6 supports practically all concepts from section **2.1.1**. Still, the concepts do not describe all specific programmatic features used in the examples, such as multiple inheritance in C++ or codeblocks in Smalltalk. The Java 6 implementations must find alternative ways to implement the desired functionality.

The patterns used may also reflect part of the author's approach to OO development, for example the two important principles for OO development defined by Gamma et al. that are listed in section **2.1.2** on page 18: 1) program to an interface, not an implementation [Gamma95, p.18]; and 2) favour object composition over class inheritance [Gamma95, p.20]. By applying the "Gang of Four" patterns, these principles will be reflected in the developed source code. By repeatedly using the "Gang of Four" patterns, these principles may be promoted by the developer to core principles that will be applied elsewhere in the design process as well.

The knowledge represented by some design patterns can be implemented as reusable components. The process of implementing patterns as reusable components is called "componentization" by Meyer and Arnout [Arnout06]. This is discussed in chapter **4**. Pattern components make the implementation phase much easier, but also fixate the behaviour to the functionality available. Certain design patterns are so universally applicable that programming languages offer implementations of them as part of the language or its core libraries. For example, it is widely known that Java has built—in support for the Iterator, Observer, and Proxy patterns. The `java.util.Iterator<E>` interface describes the Iterator pattern functionality *as understood in Java* with numerous standard implementations in the Java Collections Framework[4]. Iterators in Java, for example, are defined to fail immediately in case of concurrent modification, thus addressing, but fixating behaviour only discussed in [Gamma95, p.261]. Additionally, any class implementing the `java.lang.Iterable<T>` interface

[4] See http://java.sun.com/javase/6/docs/technotes/guides/collections/overview.html.

must return an iterator, which can be used directly in the *for—each* loop introduced in Java 5. The `java.util.Observer` interface combined with the `java.util.Observable` class describes the functionality needed to implement the Observer pattern, but is in our experience rarely used. Perhaps because it utilises deprecated collection types; that the default implementation is too simple; or because developers prefer unique method names to identity different types of events. We do not think it is unreasonable to consider that poor implementations may cause developers to become biased towards not using a pattern at all or at least in the given language. As a side note concerning the Observer pattern, the Java thread notification model described by the `wait()` and `notify()` methods in `java.lang.Object` can be viewed as a variant of it. Finally, the `java.lang.reflect.Proxy` class combined with the `java.lang.reflect.InvocationHandler` interface are an advanced implementation of the Proxy pattern that exploits Java's reflection mechanism in a manner totally different from the canonical implementation supplied in [Gamma95, p.210-215].

A programming language should be chosen from the device "the right tool for the right job" once the initial design has been established. In reality, the choice is often made beforehand. The regular usage of a programming language supporting certain design patterns will affect the way the developer thinks of the specific design patterns. It may ease the development process, but it may also fixate how the developer perceives pattern behaviour. The choice of programming language is therefore important to establish as early as possible.

2.7. Summary

By **abstracting out knowledge and encapsulating it within objects**, the OO approach to software design attempts to manage the complexity inherent in real—world problems. Identifying the proper **objects**, their **relationships**, and **collaborations** is the key to a successful design of any OO system.

Object—Oriented **analysis** (OOA), **design** (OOD), and **implementation** (OOP) is part of OO development and the software lifecycle for OO systems. The **analysis develops the software engineering requirements and specifications**, often expressed in form of **conceptual object model**. The **design must define the software objects and collaborations** forming an OO model of a software system to implement the identified requirements. Compared to other forms of software development, the design phase is larger, emphasising the need for good and durable designs even more. The **analysis determines what** must be built; the **design determines how** it should be done. The **implementation must implement the design using a programming language**. A programming language that has built—in support for an OOP paradigm (OOPL) will be easier to use, for example **C++ and Java supporting class—based programming**, but other languages can be used as well. **Different OO methods** (OOM) **can be used** to guide the design and development process, offering procedures and principles to follow within the realm of OO development. A given method may dictate that the development phases may be re—iterated and/or overlapping.

Software patterns can be used as a tool in the design and implementation process regardless of the OOM chosen, because we view patterns as orthogonal to the OOM used in many respects. **Different pattern categories may be utilised in different phases of the design**. **Architectural patterns** have large design granularity and are thus used early in the design phase, while **design patterns can be used throughout the entire design phase**. **Analysis patterns** are related to the business object model, or domain, of the system. **Idioms** are closely related to the implementation phase. The **"Gang of Four" patterns are classified as design patterns**. Regardless of the OOM chosen, **UML is often used to model the design**, including pattern implementations, visually. The strength of patterns is that **they represent well—proven solutions to commonly known and re—occurring problems based on empirical knowledge**, thus **aiding and facilitating the design process**. Several languages have **built—in support for commonly applied patterns,** such as the Iterator pattern in Java, which **makes the implementation and usage easy**, **but may also fixate pattern behaviour** and affect the way the developer perceives the patterns in question. **This is an indication of patterns and programming languages influence each other**.

This thesis investigates the "Gang of Four" design patterns described by Gamma et al. As the design patterns build upon OO, the fundamental OO concepts must be understood. **We adapt OO concepts identified by Gamma et al.** because they are inherent to the pattern application and **very well suited for Java environments**.

3. Patterns

*A pattern foreshadows the product:
it is the rule for making the thing, but it is also,
in many respects, the thing itself.*
*— **Jim Coplien***

Christopher Alexander originally described *patterns* and *pattern languages* as a mean to improve twentieth century architectural design methods and practices. Patterns have since been shown to be applicable in many other areas as well, perhaps most notably within the field of computer science and especially manifested as software design patterns related to OO development. This chapter presents the ideas set forth by Alexander and the connection to software design patterns. We describe the core pattern concepts, such as pattern languages, entries, qualities, forces, descriptions, and formats, which must all be understood in order to understand what a pattern represents, and hence to perform a meaningful evaluation. We explain how the general pattern concepts relate to software patterns and in particular to OO and the "Gang of Four" design patterns. We also present the "Gang of Four" pattern system containing the twenty—three "Gang of Four" design patterns, and explain how the patterns are classified and related. Throughout this chapter, we try to make the theory concrete by supplying several practical examples, and we present our views on many of the discussed topics. This will help understand the practical application of the "Gang of Four" patterns in the evaluation.

3.1. Christopher Alexander

Software design patterns are based on the ideas set forth by Christopher Alexander, a licensed contractor and an architect, who introduced and explained patterns and pattern languages in [Alexander77; Alexander79]. These texts were preceded and followed by a rather large number of others on closely related topics. We only give a short (and far from complete) introduction to Alexander's numerous writings, primarily based on [Appleton97; Appleton00; Lea93] unless specifically noted otherwise.

According to Lea, Alexander postulates that there is something fundamentally wrong with twentieth century architectural design methods and practices; a certain Quality Without A Name (QWAN) is missing from constructed environments. QWAN cannot be summarised briefly and no single term exist to convey or capture its meaning, but Alexander explains QWAN using partial synonyms closely related to the human impact on the design process like freedom, life, wholeness, and harmony [Lea93]. Consequently, constructions do not satisfy the real demands of users and society, because the generated environment does not have a coherent form, thereby failing the basic requirement that design and engineering improve the human condition. His ultimate goal is to build viable living structures for the people who live and work there. To remedy these shortcomings, Alexander suggests letting the inhabitants of the towns and buildings themselves take part in the design and practices using easily understandable patterns and pattern languages. This will ensure that far more inhabitable constructions will be made – structurally and spiritually – that will have that certain nameless quality we should strive for, thus reaching a coherent form. Alexander's patterns are design patterns concerning architecture.

Alexander never gives a formal definition of a pattern or a pattern language [Lea93], but offers the following explanation [Alexander79, p.247]:

> *As **an element in the real world**, **each pattern is a relationship** between a certain **context**, a certain **system of forces**, which occurs repeatedly in that context, and a certain **spatial configuration**, which allows these forces to resolve themselves. As **an element of language**, **a pattern is an instruction**, which **shows** how this spatial configuration can be used, over and over again, to **resolve the given system of forces**, wherever the context makes it relevant.*

The notion of a pattern is thus two–fold. Firstly, a pattern is an abstraction of practical experience and basic knowledge; it is not invented as such, but discovered (or *mined*), and Alexander even states that some patterns are universally known [Alexander79; Lea93]. The idea is to identify the conflicting forces within a given context, and then find a solution that brings them into harmony. A pattern not only identifies a solution, it also explains why the solution is needed [Appleton00]. Applying a pattern is the process that generates such a solution, but variant solutions may be generated. Alexander therefore emphasises letting the inhabitants (e.g. end–users) take part in the design and stresses that human interaction is an absolute necessity in applying patterns. In Alexander's domain of designing and constructing buildings and towns ("neighbourhoods" or "urban planning"), a context could be an entire town or just a house. Conflicting forces could be the known problems related to building, say, a house. This implies that patterns may be applicable at different levels in the design and therefore have different granularity, ordered in a hierarchical structure.

Secondly, the solution must be recorded, or described, so it can be reused in similar contexts. Alexander suggests a format to describe patterns in a literary non–mathematical form having the elements Name, Example, Context, Problem, and Solution [Lea93]. However, not everything that can be described using a pattern format can be considered a pattern. A pattern (entry) must ideally have a set of properties to ensure its quality, namely Abstraction, Composibility, Encapsulation, Equilibrium, Generativity, and Openness [Appleton00]. The pattern must also describe the forces that it balances. If well written, each description describes a whole that is greater than the sum of its parts [Lea93]. The presence of these properties combined with all the required pattern elements is what makes pattern entries more than just principles, heuristics, rules, or algorithms [Coplien, i.8-9; Lea93]. On the other hand, a pattern description will often contain the former, i.e. heuristics, etc., and use them as part of the pattern [Appleton00]. However, pattern descriptions leave room for interpretation. As Alexander desires living and constantly evolving architectures, patterns may be applied differently in equivalent contexts to reflect subtle changes. In [Alexander77], he writes:

> *Each pattern **describes a problem** which occurs over and over again in our environment, and then **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**.*

The idea of using "descriptive manuals" of "Best Practices" as aids to solve a given problem is nothing new and cannot be attributed to Alexander, but Alexander views patterns as "a timeless way of building" (see [Alexander79]) rather than merely offering factual descriptions on how to solve various design problems. In

Danish, there is even a specific word to describe "Best Practices" within various engineering areas, *ståbi*, meaning (an) *assist*, *aid*, or (a) *stand by* in form of some sort of manual describing optimal solutions to various technical problems[5].

A pattern description can be an entry in a pattern language. As such, entries are considered elements of a language. It is therefore essential their representation is easily understandable and recognisable so the described pattern can be applied in other applicable contexts. A pattern language is comprised of a fixed number of such pattern entries, each describing a well—proven solution to a reoccurring problem within a specific context inside a larger domain. Furthermore, a pattern language should describe its context in full, but different languages can use the same (sub—) set of patterns, combined and collaborating in different ways and perhaps in some order, depending on the context. Combined, the patterns can solve a more fundamental problem that is not explicitly addressed by any individual pattern [Appleton00; Lea93]. A pattern language describing the entire domain is said to be complete. Mathematically, to our knowledge, no pattern language has ever been proven complete, which would also seem contradictory to the idea of patterns being discovered since based on practical experience.

Alexander constructs a pattern language containing 253 pattern entries of varying granularity, from regional patterns down to patterns pertaining to some small part of a house[6]. Alexander claims the entries form a complete architectural pattern language for his domain [Alexander77; Lea93]. From Alexander's language, smaller languages can be constructed using several of the contained patterns to describe sub—contexts. For instance, to construct a pattern language to describe a given house, some of these patterns must be used, for example patterns describing light, transitions, colours, surfaces, etc., while these and others would be required to describe an entire region. The end—user, i.e. the inhabitant, help decide which patterns to include in the language to construct the house [Lea93]. However, during several experiments using his pattern language, Alexander and others realised that it did not really work as well as intended in practice; the pattern language alone did not succeed in constructing coherent form because of too many unknowns, for example the order in which to apply the patterns. Alexander therefore introduced morphogenetic sequences, or just sequences (see [Alexander05b]). A morphogenetic sequence is a pattern language that adheres to a certain order of unfolding, i.e. the order in which patterns are applied one after another. A sequence causes a repeatable coherent order to unfold, which also contains the patterns and therefore is well behaved as an environment [Alexander05b]. Alexander's, as of yet, final modification to pattern languages is generative codes [Alexander05a]. Generative codes are also morphogenetic sequences, but include *all* information needed for practical implementation, especially concerning human interaction, as well as practical, legal, and procedural details. Alexander states that without the use of generative codes, the practical work cannot be done successfully.

[5] It is not that uncommon to hear a Danish engineer turned developer pondering: "Why is there not a *ståbi* for this problem?". But there often is – in form of a software design pattern.
[6] Alexander's patterns can be viewed online at http://www.patternlanguage.com/leveltwo/patterns.htm.

3.2. Software Patterns

While Alexander describes patterns to supply solutions to problems related to his domain of designing and constructing buildings and towns, patterns and pattern languages can be, and have been, applied elsewhere, especially within the field of Computer Science.

Around 1987, Beck and Cunningham were among the first to apply the Alexander's ideas to computer science. They constructed a pattern language containing five pattern entries describing how to design simple Graphical User Interfaces (GUIs) in Smalltalk, targeted at novice Smalltalk programmers [Beck87]. The patterns were not only related to design, but also to Human Computer Interaction (HCI) in the sense that they focused on the usability of the resulting design. The patterns had varying granularity and were hierarchical related, yielding an order of application.

In 1991, Jim Coplien published a book containing a large collection of C++ idioms [Coplien91]. The book does not explicitly use the term pattern, but it was published years before the patterns became popular within computer science. However, the idiom classification of design patterns from the taxonomy listed in table **2.3** seems quite influenced by his work. In the early nineties, various people now considered pattern notables began collecting and discussing software patterns, but software patterns first became truly popular after the "Design Patterns" book by Gamma et al. was released in 1995. The four authors became known as the "Gang of Four", and the patterns presented in the book as the "Gang of Four" design patterns. The "Design Patterns" book not only describe twenty—three software design patterns describing communicating objects customised to solve a general design problem in a particular context [Gamma95, p.3], but also discuss the overall OO concepts and themes the patterns express (as explained in section **2.1.1** and **2.1.2**). The "Gang of Four" patterns have long since become famous and used extensively within the OO community. Many other books on patterns have since been published, far too many to give a meaningful and comprehensive list, and there are conferences dedicated to patterns held regularly, such as the Pattern Languages of Programs (PLoP) conferences. The "POSA" books by Buschmann et al. [Buschmann96; Schmidt00] are also widely used; Buschmann et al. formulated the design pattern categories commonly used to this day, i.e. the architectural patterns, design patterns, and idioms categories from table **2.3**. Besides books and conferences, online pattern repositories such as [PPR] and [Hillside] also provide much information regarding software patterns.

It is important to state that software patterns are not restricted to software design patterns, but it is hard to estimate how influential Alexander's work has been on different kinds of patterns and individual collections, whether intentionally or not. For example, many of Coplien's C++ idioms are not really patterns, while Beck and Cunningham directly references Alexander and claims that their five patterns form a complete pattern language [Beck87]. Though not to the same extent as Beck and Cunningham, Gamma et al. clearly state that they build on the work by Alexander [Gamma95, p.2-4], and Buschmann et al. relate their work to Alexander as well [Buschmann96, p.360,414; Schmidt00, p.505-526]. But one thing is saying so, another thing is doing so. Alexander's work related to software (design) patterns is debated heartily within the community, also with respects to the "Gang of Four" patterns (see for example [PPR]). On the other hand, pattern related concepts originating in computer science have also emerged, such as The Rule of Three and Proto Patterns as explained in section 3.8.3 on page 49. Furthermore, as discussed in section 3.6, efforts are also being made to formalise pattern descriptions and include patterns as language features or implement them as reusable components in accordance with the principles in OO. This approach to design patterns deviates from the original concept defined by Alexander in the sense that the human factor is less important in applying the patterns, but is much more tangible and structured.

More recently, different kinds of software patterns have emerged outside the scope of "mere" OO design, such as Analysis, HCI, Organisational, or Process Patterns, used to describe other or more specialised aspects of software engineering [Appleton]. Analysis patterns as described by Fowler are still closely related to design [Fowler97], and are therefore included in our pattern taxonomy, while HCI patterns seem quite well suited to expand on Alexander's ideas in different ways than OO design has. This is because several verbose, but exactingly formulated, HCI methodologies already exist, based on empirically proven design guidelines, such as User—Centred Design (UCD) in which the end—user must participate wholeheartedly in the design of the solution [Borchers99]. This is reminiscent of Alexander letting the end—user participate in the design process. HCI is very much based on practical experiences, and because of characteristic similarities with patterns, HCI methodologies could be expressed using pattern descriptions, especially since the methodologies already have a written form.

Still, software design patterns related to OO are very likely the most used kind of software patterns. People within the software community neither accept the usefulness of individual patterns or collections, nor the need for such a thing as software design patterns at all. OO has always acknowledged the need for meticulous analysis (OOA) and design (OOD), but prior to design patterns, descriptions of design problems where mostly of a rather abstract nature describing from scratch how to identify the individual parts of the system, their relationships, and collaborations. In our view, a textbook example of this is the otherwise good book "Designing Object—Oriented Software" by Wirfs—Brock et al. from 1990 [WirfsBrock90]. With patterns, problems of varying granularities have already been solved and described, giving the designer a new set of "broader" tools to use in the analysis and design phase. The abstraction need no longer be just focused at the individual class and object level, but also at a higher level describing functionality, relations, and coherency traditional OO constructs cannot. The principles are thus separated from the implementation. We think this is a key reason behind the popularity of software design patterns.

3.3. Pattern Qualities

A pattern entry must ideally possess the set of properties listed in table 3.1 to ensure the quality of the pattern, namely Abstraction, Composibility, Encapsulation, Equilibrium, Generativity, and Openness [Appleton00]. Many of these properties have similar meaning to desirable constructs in OO, which could explain why software patterns first became popular within this domain. As an example, consider a class. A class is an abstraction with encapsulated responsibilities representing some equilibrium. It can be used as a component by other classes and is normally generative in its usage. It can be implemented in different languages or may even be parameterised with other types (openness) [Lea93].

Lea even writes that patterns may be viewed as extending the definitional features of classes, and that classes and patterns have two analogous aspects [Lea93]:

I. **The external, problem—space view**: descriptions of properties, responsibilities, capabilities, and supported services as seen by the external context; and

II. **The internal, solution—space view**: static and dynamic descriptions, constraints, and contracts among components known only with respect to a possibly incomplete external view (interface).

The need for these qualities implies that there is no guarantee that a given problem can be solved using a pattern. Not every solution can be captured in a pattern, and not everything described by a pattern entry can be considered a pattern [Hohmann98]. Accordingly, a class will only express these properties if well designed.

Table 3.1 — Pattern qualities

Name	Description	Computer Science
Abstraction	A pattern represents a general abstraction of knowledge and experience within a given domain [Lea93]. The use of natural language, diagrams, illustrations, etc., is required.	Objects are programmatic abstractions of functionality, real—world or otherwise. A pattern abstraction is a higher—level abstraction compared to what can be described by programming language constructs alone. The use of programming language in examples augments the pattern description, but the examples cannot stand—alone.
Composibility	Patterns of different granularity are hierarchically related (in a pattern system or language), indicating a rough application order to be adhered to when the patterns are unfolded. Patterns at a given level of abstraction and granularity may lead to, or be composed with, other patterns [Alexander77; Appleton00; Lea93].	Objects share similar traits, and can be composed to achieve complex functionality. For example, a recurring theme in [Gamma95] is to prefer delegation to inheritance, which allows for dynamic composibility.
Encapsulation	A pattern must encapsulate an independent, well—defined real—world problem and solution within a given domain [Alexander77; Lea93].	An object uses encapsulation to ensure that both data and the methods that operate on the data are correlated. Combined with information hiding, this ensures that the responsibilities of the objects are well—defined. However, an object need not represent a real—world problem.
Equilibrium	Indicates a balance between forces and constraints that minimises the conflicts in solution space identified by the pattern, and may be based on invariants and/or heuristics. Equilibrium provides a rationale for each individual step in the pattern when applied [Alexander77; Appleton00; Lea93].	The responsibilities of an object represent the trade—offs made when designing it, and the functionality implemented by the object represents the equilibrium.
Generativity	When a pattern is applied, as described by its description, it provides the solution to a given context thereby generating a new resulting context, which in turn can be used to apply other patterns, and so forth, leading to the overall generation of the solution to the domain in question. More	Classes can be viewed as being generative as well; they support parameterised instance construction and perhaps parameterised types (e.g. generics and templates). Objects in prototype—based languages may also support parameterised instance construction.

Table 3.1 — Pattern qualities

Name	Description	Computer Science
	than one pattern may be applicable to a given context [Alexander77; Lea93].	
Openness	Each pattern should be open for extension and parameterisation by other patterns, to work together to solve a larger problem. Realisation of the pattern should be possible using any number of implementations, alone or in conjunction with other patterns [Alexander77; Lea93]. Applying a pattern is the process that generates a solution, but it may generate variant solutions [Appleton00]. In theory, a pattern entry should be implemented for each usage.	Some languages contain built—in support for several patterns, and libraries are commonly used to supply well—proven pattern implementations (see section 2.6.2 on page 29). Combined with parameterised types (e.g. generics or templates) even built—in classes may be considered open, for example `java.util.Iterator<E>` in Java.

If a pattern exhibits these qualities, the source code implementation is likely to reflect them as well.

Example 3.1 — In example 2.4 on page 27, we claimed that the Abstract Factory, Factory Method, and Singleton patterns could aid in the design of the notification mechanism from example 2.1. To add value to the design the patterns must express the desired qualities. The Abstract Factory pattern is an abstraction of knowledge about creating objects without explicitly knowing their type; its description contains text, illustrations, as well as program code (Abstraction). The pattern functionality is required in many different types of flexible real—world systems, and the pattern encapsulates this task by providing a description of the problem as well as a proven solution to it (Encapsulation). The description explains the trade—offs in using it, for example that the pattern promotes consistency, but also that it can be hard to add new types of object to a given factory (Equilibrium). The Abstract Factory can defer the actual creation of new objects elsewhere, typically to Factory Method or Prototype [Gamma95, p.117] pattern implementations (Composibility and Openness); as a variant, it could also choose to implement the functionality by it self, for example using reflection in Java (Openness). Finally, Abstract Factory implementations are often suitable as candidates for the Singleton pattern (Generativity) ■

3.4. Pattern Forces

A pattern must balance opposing forces within its context to reach a balance that implicitly will be present in the pattern and in its application [Appleton00]. The described solution must bring the identified forces into harmony, or the pattern is not warranted. This implies that a pattern may represent trade—offs between various forces.

The type of forces depends entirely on the domain and context, but forces can generally be thought of as goals and constraints. In computer science, the notion of force generalises the kinds of criteria used to justify designs and implementations [Lea00, i.12]. According to Buschmann et al., the most important non—functional forces regarding OO development are Changeability, Interoperability, Efficiency, Reliability, Testability, and Reusability [Buschmann96, p.404-410], and Lea lists a set of similar forces, such as Portability, Extensibility, Fairness, Maintainability [Lea00, i.12], etc. More explicit functional forces are closely tied to the domain [Appleton00; Lea00, i.12-13]. A functional force can be visible to the users of the system by means of a particular function, or it may represent aspects of the implementation, such as the algorithm used to compute the function [Buschmann96, p.389].

Design patterns, e.g. the "Gang of Four" patterns, primarily express non—functional forces, as example **3.2** below also illustrates. As patterns are used to implement system functionality, the forces balanced in the pattern may influence the system unless fully encapsulated, intentionally or otherwise. Similar, the traits of the system will dictate the type of applicable patterns.

Example 3.2 — A set of non—functional forces relevant for the notification mechanism from example **2.1** on page 21 could be Reusability, Changeability, and Extensibility related to the various design issues contemplated in example **2.2**. If designed as an open—source library or API, Reusability becomes an important factor, as well as Changeability and Extensibility to manage or add new means of deliveries or new functionality. On the other hand, Efficiency and Fairness is not that important as long as a delivery is made eventually. As means of deliveries, we considered email and SMS deliveries in form of the `EmailDelivery` and `SMSDelivery` implementations. They are *store—and—forward* services, and once a message has been delivered successfully to the gateway, nothing more can be done from the application's point of view. However, other types of deliveries could require scheduling and processing guarantees, for example the order of delivery. A delivery writing to an event or audit table in a database is one example. Patterns used in the design of the notification mechanism should match these forces and preferably enforce them, for example using the Abstract Factory [Gamma95, p.87] and Factory Method [Gamma95, p.107] patterns as described in example **2.4** to ensure Changeability and Extensibility of associated `Delivery` and `Formatter` types.

Functional forces will be closely related to the core functionality of the notification mechanism, which is a library for delivery of messages to subscriptions using various means of deliveries. This indicates that there will be an overlap between functional and non—functional forces in this case, e.g. Fairness and Extensibility. A more explicit functional force could be an algorithm used to correlate and concatenate related `Notification` objects to be delivered in a single delivery ■

An example of an unresolved force relevant to the "Gang of Four" patterns is Multithreaded Safety as suggested by Lea [Lea00, i.12]. In general, concurrency is not an issue discussed much in the "Design Patterns" book [Gamma95]. This does by no means imply that the "Gang of Four" patterns are faulty, but that care must be taken when applying them in modern concurrent systems. For example, what is the result in case of concurrent modification to the underlying representation used by the Iterator pattern (*robustness*), or how do we ensure that only a single instance of a Singleton type is created in a concurrent environment?

3.5. Pattern Elements

Alexander's description of patterns contains certain vital elements to ensure that it conveys the relationship between the context and forces, and implicitly the qualities as well [Appleton00]. A pattern format, or just form, is a template dictating the elements and structure of pattern descriptions. To be able to reuse a pattern in a design, the pattern description must contain the decisions, alternatives, and trade—offs (forces) that led to it [Gamma95, p.6]. A well—written pattern must also express the desired qualities [Lea93], and is more than a simple recipe as Fowler explains [Fowler06]:

> ***Recipes tend to be more particular**, usually **tied to a particular programming language** and platform. Even when **patterns** are tied to a platform, they try to **describe more general concepts**.*

Many different formats exist, some just slight variations of others, but no official standard is acknowledged [Lea00]. However, several de—facto standards exist. Alexander's format, Alexandrian Form, is used to describe architectural patterns and it contains the elements Name, Example, Context, Problem, and Solution [Lea93]. In computer science, the "Gang of Four" (GoF) and Canonical Forms are widely used [Lea00; Appleton], but many other exist (for a list, see [Lea00; PPR]). For instance, the "POSA" patterns are described using a variant of the Canonical Form [Buschmann96, p.20-21], while the "Gang of Four" patterns are described using the "Gang of Four" form, surprisingly enough. All forms in some way seem to present the elements required by Alexandrian Form, but not necessarily in, or as, their own sections. Some formats make these elements explicit, while others do not. For example, the form used by Fowler to describe his analysis patterns in [Fowler97; Fowler03] has just three named elements, where only the Name element corresponds to a pattern element as defined by Alexander. Hence, different pattern formats describe different elements, and elements differently, but Appleton states that the elements from the Canonical Form should be clearly recognisable upon reading a pattern description. The elements are Name, Problem, Context, Forces, Solution, Examples, Resulting Context, Rationale, Related Patterns, and Known Uses [Appleton00]. The naming of patterns is especially interesting. By giving a pattern a meaningful and concise name, designers, developers, and others share a common vocabulary (easy naming of solutions to common problems) that can be utilised in the development process, and which extends beyond other more traditional methods [Gamma95, p.6; Fowler06].

The pattern description will be affected by the domain targeted by the pattern. The "Gang of Four" design patterns operate in OO environments, and OO concepts and themes utilised by Gamma et al. will be reflected in the patterns and their application, i.e. implementation. The concepts and themes thus become important in order to understand the patterns as a whole. However, the format used to describe the patterns can also affect the pattern, because not all formats are appropriate for a given domain [Vlissides97, i.7].

3.5.1. "Gang of Four" Format

The format used by Gamma et al. in the "Design Patterns" book has since been named the "Gang of Four" format, or GoF form. The Canonical Form builds on the format, and shares many elements; it can be viewed as a generalised version of the "Gang of Four" format. The format is commonly used, and often used as a base for variant forms [Fowler06]. The format is highly structured compared to the Alexandrian form of writing, which is narrative and almost lyrical [Vlissides97, i.7]. Table 3.2 explains the general purpose of the different elements. It also relates them to the most relevant qualities from table 3.1 as we see it.

Table 3.2 – "Gang of Four" pattern format (modified from [Gamma95, p.6-7])

Element	Description	Qualities
Name	A concise pattern name that conveys the pattern essence.	Abstraction
Classification	The classification of the pattern according to the two dimensions Scope (Class and/or Object) and Purpose (Creational, Structural, or Behavioural).	Abstraction, Composibility, Generativity
Intent	A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?	Abstraction, Equilibrium
Also Known As	Alternative names, if any.	Abstraction
Motivation	An example that illustrates a design problem and how the class and object structures in the pattern solve the problem.	Abstraction
Applicability	In which situations can the pattern be applied? What are examples of poor designs that the pattern can address? How can you recognise these situations?	Abstraction, Composibility, Generativity
Structure	A graphical representation of the classes and objects in the pattern.	Abstraction, Encapsulation
Participants	The classes and/or objects participating in the design pattern and their responsibilities.	Abstraction, Encapsulation
Collaborations	How do the participants collaborate to carry out their responsibilities?	Abstraction, Encapsulation
Consequences	How does the pattern support its objectives? What are the trade–offs and results of using the pattern? What aspect of the system structure does it let you vary independently?	Equilibrium, Openness
Implementation	What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language specific issues?	Composibility, Equilibrium, Generativity, Openness
Sample Code	Code fragments that illustrate how you might implement the pattern.	Generativity, Openness
Known Uses	Examples from real systems.	Composibility, Generativity, Openness
Related Patterns	Related patterns, if any.	Composibility, Generativity, Openness

The elements listed in light grey are the elements most closely related to pattern implementation, i.e. Implementation and Sample Code. As described in chapter 5, the evaluation pays special attention to these elements. The format as described in the "Design Patterns" book explicitly mentions that the Sample Code element will supply source code in C++ or Smalltalk [Gamma95, p.7], because the "Gang of Four" patterns are illustrated in these languages. Similar, the Implementation element is closely related to these languages as well, or at least the features of the languages. The use of these languages in the pattern description may influence the pattern application using other languages, e.g. Java 6, because they tie the patterns to specific languages. This is also noted by Gamma et al. [Gamma95, p.4]. As patterns are discovered in existing source code (see section 3.8.1 on page 47), the Implementation and/or Sample Code elements may very well represent extracts from real systems written in the same programming language. Both the problem and solution may thus have originated in, or because of, the language in question.

3.6. Pattern Formalism

The lack of a formalised concept of a design pattern has long been a vigorously debated issue within the pattern community (see for example [Eden98, p.3; Eden02, p.380] and much of [Hillside; PPR]). It goes to the very core of understanding, or agreeing on, what software design patterns are. Formalism is closely related to tool support for pattern mining, understanding, and application. The efforts to formalise design patterns intentionally oppose Alexander's ideas of patterns languages and patterns expressing the QWAN in favour of more analytic and structural approaches [Eden98, p.3]. Practically all software design pattern formats are already much more structured compared to the Alexandrian format. Stated bluntly, Eden says most followers of Alexander's ideas treat software design patterns as sacred cows, no less, which cannot, and should not, be formalised, while followers of the efforts to bring structure, formalism, and tools to the pattern community are rational [Eden98, p.3]. In this feud, we take the middle ground. Formalism can be a valuable tool to aid the practical implementation of design patterns, i.e. componentization, tooling, and ease of understanding, while remembering that software design patterns ideally express more than program code, i.e. part of a vocabulary, highly adaptable, used for teaching and understanding of concepts, etc.

While formal specifications may clarify pattern functionality, we fail to see how it can describe the human aspect in patterns and in their application, expressed in, and as a combination of, various pattern elements. Strict formalisation of patterns will deemphasise the human aspect greatly, which goes against Alexander's original ideas. Vlissides agrees and writes [Vlissides97, i.4]:

> *In short,* ***patterns are primarily food for the brain****, not fodder for a tool. There may yet be* ***latent benefit*** *in methodological or automated support, but I'm convinced it'll be* ***icing on the cake****,* ***not the cake itself*** *or even a layer thereof.*

Even more so, as described in section **3.5**, different pattern formats describe different elements and elements differently. If formalism is to succeed, we believe it will be at the expense of variety of pattern formats. This could pose a problem, as a single pattern format does not fit all [Vlissides97, i.7].

Baroni et al. discuss numerous OO and pattern formalisation methods in [Baroni03], and conclude that all the reviewed mechanisms have drawbacks, and cannot capture all the concepts related to patterns [Baroni03, p.11,53]. In light of the reviews, Baroni et al. also conclude that certain pattern elements in the "Gang of Four" format are easier used in the pattern formalisation process, namely Participants, Collaborations, Structure, and in part Implementation [Baroni03, p.8,53]. As explained in table **3.2** on page 41, the first three elements relate to the actual design and relationships of the classes and objects used in the pattern. Common relations like inheritance, creation, and forwarding are labelled as *simple* [Baroni03, p.11] and are clearly encompassed by the concepts and themes described by Gamma et al. The Implementation element primarily express programming language constructs, which are highly structured. The four elements all favour structured over unstructured information. In our view, this is a clear indication that formalisation is closer related to fundamental OO concepts as opposed to pattern concepts, such as qualities and forces that cannot easily be described. The human factor is missing. Pattern concepts are what make patterns powerful abstractions and tools, not just the OO mechanisms used to implement them. Formalism may be able to resolve some ambiguities, but literal descriptions are still warranted to describe the human aspect.

3.7. Pattern Collections

A pattern cannot describe a complete OO system by itself, but targets a specific problem within the system. Instead, patterns are often interrelated and rely on the behaviour of other patterns to achieve their own goals, especially so for patterns targeting the same domain or even context. The relations between patterns are important because applying a specific pattern may generate the need to apply other patterns; a relation can also indicate different possible solutions in form of other patterns. In the "Gang of Four" format, as described in section **3.5.1**, the Related Patterns element expresses such pattern relationships. Based on how coherent a collection of patterns are, including how they individually are described, patterns can be correlated in different types of collections as explained in table **3.3** below.

Table 3.3 — Pattern collections

Name	Description
Catalogue	A pattern catalogue is a collection of loosely and/or informally related patterns. The contained patterns are often divided into broad categories and are not necessarily written using uniform pattern entries or even format [Buschmann96, p.23].
System	A pattern system is a cohesive set of related patterns described in a consistent format, working together to support construction and evolution of whole architectures [Buschmann96, p.361].
Language	A pattern language can be viewed as a pattern system covering a complete domain with rules and guidelines, which explain how and when to apply its patterns to solve a problem that is larger than any individual pattern can solve [Appleton00].

Pattern catalogues can evolve into pattern systems, and due to the obvious benefits of systems over catalogues, catalogues are rarely used because the knowledge they represent may be too unstructured to be truly useful in the design process. Gamma et al. identify the "Gang of Four" pattern collection as a "catalogue" [Gamma95, p.8], but according to the Buschmann et al. definition, they constitute a pattern system. This is because the "Design Patterns" book predates the "POSA" books. The "Gang of Four" design patterns all target the same domain; they are interrelated in intricate ways; many depend on other patterns to supply functionality; and they are all written using the same format.

Though pattern systems share many desirable traits with pattern languages[7], they can at most be considered incomplete pattern languages [Buschmann96, p.360]. Pattern systems lack the robustness and wholeness of pattern languages. Because of narrower focus, most are described using only a sub—set of the pattern elements in the Canonical Form, but may eventually evolve into a pattern language. Pattern languages are not created all at once, but evolve from pattern systems. In practice, however, the difference can be very hard to detect

[7] The first "POSA" book uses the term *pattern systems* as almost a synonym for *pattern languages* as described by Alexander [Buschmann96, p.360-362], while the second book explicitly differentiates between systems and languages [Schmidt00, p.524-526].

[Appleton00]. Dominus, for example, states that systems and even catalogues are what many people mistake for pattern languages [Dominus02].

A pattern language can be thought of as a "super pattern" that can be applied to solve a problem in its entirety. The contained patterns solve sub—problems in a *divide—and—conquer* fashion [Appleton00; Buschmann96, p.403]. Very few authors claim to have, or have indeed, constructed a complete pattern language. Alexander claims his 253—entry pattern language is complete for his domain, while Beck and Cunningham were among the first to create a pattern language within the field of computer science, containing only five pattern entries [Beck87]. According to Buschmann et al., other small languages from computer science include Crossing Chasms for connecting OO systems to relational database, and CHECKS by Cunningham for information integrity [Buschmann96, p.360]; in [Schmidt00, p.510-524], Buschmann et al. themselves claim to present a pattern language for middleware and applications in relation to concurrency and networking. By Alexander's definition, the general case is that pattern languages are very rare in any field. In computer science, catalogues and systems are much more common because of their lenient definitions. As we understand it, this is also a key issue pointed out by several critics of software patterns (see for example [Dominus02]): pattern languages are not used in computer science, merely the patterns themselves in a loosely organised fashion. Regardless, whether or not such languages indeed are pattern languages is open for debate, because there is no mathematical way to determine it.

3.7.1. "Gang of Four" Pattern System

In 1995, Gamma et al. published the "Design Patterns" book [Gamma95], describing twenty—three individual design patterns contained in a pattern system pertaining to OO, which popularised the use of patterns in computer science [Appleton00]. As explained in section **2.5.1.3**, the "Gang of Four" design patterns describe concepts and structures beyond individual objects and classes up to the granularity level of refinement of OO sub—systems, customised to solve a general design problem in a particular context [Gamma95, p.3]. Below, table **3.4** lists the twenty—three "Gang of Four" design patterns from [Gamma95], including their classifications and relationships.

Gamma et al. classify the "Gang of Four" patterns in two dimensions according to Scope (Class and/or Object) and Purpose (Creational, Structural, or Behavioural) [Gamma95, p.10]. The Scope criterion identifies whether the pattern applies primarily to classes or objects. Class patterns deal with relationships between classes and their sub—classes. Object patterns are more dynamic, and deal with objects and their relationships, but almost all the patterns uses inheritance, and thus classes to some extent. Purpose is a problem—based criterion that classifies the "Gang of Four" patterns according to what they do. Creational patterns focus on the instantiation process of objects [Gamma95, p.81]; Structural patterns focus on how classes and objects are composed to form larger structures [Gamma95, p.137]; and finally Behavioural patterns focus on algorithms and assignment of responsibilities between objects [Gamma95, p.221]. Other types of problem—based classifications exist, for instance Concurrency patterns (see for example [Schmidt00]).

Table 3.4 — "Gang of Four" pattern system

Name	Description	Scope	Related Patterns
Creational Patterns			
Abstract Factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes [Gamma95, p.87].	Object	⇒ creates Bridge ⇒ alternative to Builder ⇒ collaborates with or alternative to Facade ⇒ uses Factory Method ⇒ uses or alternative to Prototype ⇒ is a Singleton
Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations [Gamma95, p.97].	Object	⇒ alternative to Abstract Factory ⇒ creates Bridge ⇒ creates Composite ⇒ is a Singleton
Factory Method	Define an interface for creating an object, but let sub—classes decide which class to instantiate. Factory Method lets a class defer instantiation to sub—classes [Gamma95, p.107].	Class	⇒ used by Abstract Factory ⇒ used by Iterator ⇒ alternative to Prototype ⇒ used by Template Method
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype [Gamma95, p.117].	Object	⇒ used by or alternative to Abstract Factory ⇒ implemented by Command ⇒ collaborates with Decorator ⇒ alternative to Factory Method ⇒ is a Singleton ⇒ collaborates with Template Method
Singleton	Ensure a class only has one instance, and provide a global point of access to it [Gamma95, p.127].	Object	⇒ implemented by Abstract Factory ⇒ implemented by Builder ⇒ implemented by Facade ⇒ implemented by Mediator ⇒ implemented by Prototype ⇒ implemented by Observer ⇒ implemented by State
Structural Patterns			
Adapter	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces [Gamma95, p.139].	Class, Object	⇒ alternative to Bridge ⇒ alternative to Decorator ⇒ alternative to Proxy
Bridge	Decouple an abstraction from its implementation so that the two can vary independently [Gamma95, p.151].	Object	⇒ created by Abstract Factory ⇒ alternative to Adapter ⇒ created by Builder
Composite	Compose objects into tree structures to represent part—whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly [Gamma95, p.163].	Object	⇒ created by Builder ⇒ collaborates with Chain of Responsibility ⇒ collaborates with Decorator ⇒ collaborates with Flyweight ⇒ used by Interpreter ⇒ uses or collaborates with Iterator ⇒ collaborates with Visitor
Decorator	Attach additional responsibilities to an object dynamically. Decorators provide a flexible	Object	⇒ alternative to Adapter ⇒ collaborates with Prototype

Table 3.4 — "Gang of Four" pattern system

Name	Description	Scope	Related Patterns
	alternative to sub—classing for extending functionality [Gamma95, p.175].		⇒ collaborates with Composite ⇒ alternative to Strategy
Facade	Provide a unified interface to a set of interfaces in a sub—system. Facade defines a higher—level interface that makes the sub—system easier to use [Gamma95, p.185].	Object	⇒ collaborates or alternative to Abstract Factory ⇒ alternative to Mediator ⇒ is a Singleton
Flyweight	Use sharing to support large numbers of fine—grained objects efficiently [Gamma95, p.195].	Object	⇒ collaborates with Composite ⇒ used by Interpreter ⇒ implemented or used by State ⇒ implemented by Strategy
Proxy	Provide a surrogate placeholder for another object to control access to it [Gamma95, p.207].	Object	⇒ alternative to Adapter ⇒ alternative to Decorator
Behavioural Patterns			
Chain of Responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it [Gamma95, p.223].	Object	⇒ collaborates with Composite
Command	Encapsulate a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations [Gamma95, p.233].	Object	⇒ is a Composite ⇒ uses Memento ⇒ is a Prototype
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language [Gamma95, p.243].	Class	⇒ uses Composite ⇒ uses Flyweight ⇒ uses Iterator ⇒ uses Visitor
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation [Gamma95, p.257].	Object	⇒ used by or collaborates with Composite ⇒ uses Factory Method ⇒ used by Interpreter ⇒ uses or alternative to Memento
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently [Gamma95, p.273].	Object	⇒ alternative to Facade ⇒ collaborates with Observer ⇒ is a Singleton
Memento	Without violating encapsulation, capture and externalise an objects internal state so that the object can be restored to this state later [Gamma95, p.283].	Object	⇒ used by Command ⇒ used by or alternative to Iterator
Observer	Define a one—to—many dependency between objects so that when one object changes state, all dependants are notified and updated automatically [Gamma95, p.293].	Object	⇒ collaborates with Mediator ⇒ is a Singleton
State	Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class [Gamma95, p.305].	Object	⇒ is a or uses Flyweight ⇒ is a Singleton
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy	Object	⇒ alternative to Decorator

Table 3.4 — "Gang of Four" pattern system

Name	Description	Scope	Related Patterns
	lets the algorithm vary independently from clients that use it [Gamma95, p.315].		⇒ is a Flyweight ⇒ alternative to Template Method
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to sub—classes. Template Method lets sub—classes redefine certain steps of an algorithm without changing the algorithm's structure [Gamma95, p.325].	Class	⇒ uses Factory Method ⇒ collaborates with Prototype ⇒ alternative to Strategy
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates [Gamma95, p.331].	Object	⇒ collaborates with Composite ⇒ used by Interpreter

The legends used to describe the pattern relationships indicate the type of relationship. We have deciphered the relationships by examining all the pattern descriptions, especially the Related Patterns element, as well as [Gamma95, f.1—1,p.9-13]. The relationships do not indicate that the patterns must be used together as illustrated, merely that they can be. They are by no means a formal specification of the "Gang of Four" relationships, but help provide an overview in the practical application. The *Uses* and *Used by* legends typically indicates a strong relationship, often a "has—a" relationship corresponding to composition and/or delegation in compliance with the general "Gang of Four" themes as described in section 2.1.2 on page 18. The *Is a* and *Implemented by* legends indicates an "is—a" relationship, corresponding to class—based inheritance or interface implementation. The *Collaborates with* legend indicates some form of collaboration between the patterns, for example that both can be used by in conjunction by other patterns; the term is broadly defined and could refer to a stronger relationship such as *Used by* depending on the actual application. The *Creates* and *Created by* legends indicates a special form of creational collaboration. Finally, the *Alternative to* legend indicates that alternative, but not identical, solutions to a problem exist; however, applying one pattern over an alternative one may generate considerable changes to the design.

3.8. Pattern Evolution

Individual patterns evolve over time, but so too can pattern catalogues, systems, and languages. This is of pivotal importance because the patterns must reflect their environment, which according to Alexander is constantly evolving.

3.8.1. Mining

Mining is the non—trivial art of discovering new patterns within systems in a given domain *and* describing them. This is a term originating in computer science, but Alexander present similar ideas. The general idea is that true patterns are discovered, not invented, due to the duality in the definition of a pattern as explained in section 3.1. Coplien states that patterns observed in an existing system may not be desirable. Some patterns are non—generative, descriptive, and passive, i.e. recipe—like, which is not good and do not lead to desirable results [Coplien, i.3]. Only good patterns should be mined for actual (re—) use, which can then help generate new systems that will contain the pattern traits. Such patterns are generative, prescriptive, and active; they are more than simple recipes.

Since patterns represents common best—practices to reoccurring solutions, the pattern naturally has to be found in more than one design. The Rule of Three simply states that any pattern must have been found in at least three real—world systems for it to be considered a valid pattern [PPR, p.RuleOfThree]. Gamma et al. state that none of the "Gang of Four" design patterns represents new or unproved designs, but that they are elements of some successful OO system, or part of the folklore of the OO community [Gamma95, p.2].

3.8.2. Anti—Patterns

Anti—Patterns focus on existing software failures in an attempt to understand, prevent, and recover from them [McCormick01]. The term and its meaning were originally coined by Brown et al. as a counterpart to design patterns [Brown98]. Since they represent specific pitfalls to avoid during software development, they can naturally be found within any imaginable area, for instance management, organisation, design, programming, etc. Anti—Patterns are described using structured formats and each description is based on existing (bad) solutions [McCormick01]. Anti—patterns are sometimes referred to as *code smells*.

Design patterns are often the "cure" for anti—patterns. They describe a solution that will remedy the problems inherent in the anti—pattern. Pattern mining is therefore closely related to anti—patterns: new patterns may produce commonly accepted design patterns that can be used to avoid common pitfalls, but on the other hand, overly or wrong use of design patterns may be an anti—pattern by itself. Rarely, design patterns can thus be the very "symptom" described by an anti—pattern. The optimal solution is to evolve from designs containing anti—patterns – well, preferably containing none – to designs utilising well—described design patterns without constructing new anti—patterns in the process.

Example 3.3 — The Layers pattern [Buschmann96, p.31] discussed in example **2.3** on page 25 is a design pattern offering a solution to pitfalls described by the Big Ball of Mud [PPR, p.BigBallOfMud] anti—pattern [Buschmann96, p.29]. It offers structure instead of chaos. On the other hand, the Singletonitis [Vieiro06] anti—pattern describes overly or wrong use of the Singleton [Gamma95, p.127] pattern; it exists because the Singleton pattern exist, and designers using the Singleton pattern should be aware of this. Example **2.4** on page 27 considered applying the Singleton pattern in the design of the described notification mechanism, specifically to ensure unique `Delivery` and `Formatter` factories. Forcing singleton objects into libraries may cause unforeseen runtime consequences, such as class loading issues in Java, but it may also cause undesirable behaviour, such as severely restricting how clients can use and combine factories. The latter may be fine, but the consequences must naturally be thought through. In any case, the evolution of the notification mechanism could even require refactoring causing less frequent usage of the Singleton pattern ■

There is no precise checklist specifying what constitutes an anti—pattern, but [PPR, p.AntiPatternsCatalog] lists many commonly accepted anti—patterns. Nevertheless, functionality some people regard as anti—patterns, others do not; even more so, functionality some regard as patterns, others regard as anti—patterns! A simple example is invoking an overridden method in a super class from inside the overriding method in the sub—class, i.e. an extra tight coupling between a super and sub—class that must be enforced by the *developer*. In Java, *finalizer chaining* is an example of this: when overriding `java.lang.Object.finalize()`, the developer must ensure that the `finalize()` method in the super—class is invoked [Bloch01, p.22-23]. Fowler identifies this as the Call Super [Fowler05] anti—pattern because there is no way to guarantee that the sub—class will invoke the overridden method in the super class (as opposed to method combination, for example in CLOS [DeMichiel87]). However, according to Grand [Grand99, p.179], Beck identifies this as a design pattern called Extend Super, though with a slightly different context. As a practical example, Livshits has identified misuse of the Extend Super pattern in the Eclipse project, which corresponds to the situation described by the Call Super anti—pattern [Livshits05, p.1-2].

3.8.3. Proto Patterns

Ideally, a newly discovered and initially described pattern is called a proto pattern until its qualities and elements have been validated and acknowledged by others, if at all, for example at a PLoP conference. This is also a term originating in computer science. A proto pattern will be investigated to see if it is meaningful within its domain; if it describes the forces at play; if it has the required elements and qualities; if the Rule of Three is adhered to; etc. Even if a proto pattern is accepted as a valid pattern, there is no guarantee that it will ever be commonly used. Many patterns are left unused. This does not necessarily mean that they are not useful, though as a concept design patterns are often misused to denote anything that has the slightest touch of recognisability, but perhaps rather that their context and problem is too specific to be truly valuable. On the other hand, many so—called patterns have been published violating the needed elements and qualities, not to mention the Rule of Three, or representing a solution in which no forces are at play. They could also be passive as described by Coplien, not generating quality solutions.

Once a proto pattern has been established to represent a valid pattern, it is no longer considered a proto pattern. The problem is naturally "who" decides this. Furthermore, since Alexander describes patterns as being non—static, we claim they will always function as "prototypes" in form of their knowledge and descriptions. For example, the "Gang of Four" Command [Gamma95, p.233] pattern has at least spawned the "POSA" Command Processor [Buschmann96, p.277] pattern, and in the evaluation we even present a variant of the Command Processor pattern that might eliminate the need to use Composite [Gamma95, p.163] (or macro) commands (see section **8.3.2.3** on page 159).

3.8.4. Piecemeal Growth

Catalogues can mature and evolve into pattern systems over time as well as systems can mature and evolve into pattern languages via a process Alexander calls *piecemeal growth*: patterns are applied in an ordered sequence of piecemeal growth, progressively evolving an initial architecture, which will then flourish into a "live" design possessing the QWAN (see also table **3.1**). As patterns are applied by the means of piecemeal growth, applying one pattern provides a context for the application of the next pattern [Appleton00]. This implies that both the collection and the design will evolve; if an individual pattern evolves, it may thus affect the entire collection.

According to Appleton [Appleton00], Alexander explains that piecemeal growth is based on the idea of repair as opposed to traditional architectural development processes that are based on the idea of replacement. Traditional processes assume that each act of design or construction can be viewed in isolation, "perfect" at the time of construction. Alexander calls such processes *large lump development*. Piecemeal growth acknowledges that environments are continuously changing and growing in order to keep its use in balance. Appleton notes that there are similarities between piecemeal growth and spiral development processes involving prototyping and iterative/evolutionary development, such as XP, as well as large lump development and the Waterfall model [Appleton00]. As explained in section **2.2**, iterative OOMs acknowledge that the design is not static, but dynamic in nature. By using design patterns sharing similar traits the design will be able to evolve more easily.

A decade ago, Buschmann et al. speculated that the development of complete software design pattern languages was an optimistic, but worthwhile goal [Buschmann96, p.422]. To this day, the goal has not been achieved, not even using the "Gang of Four" patterns as the system to evolve into a proper pattern language. Buschmann et al. estimate that the "Gang of Four" pattern system may cover as much as half of the general—purpose design patterns of its domain [Buschmann96, p.422], i.e. at the granularity level of a few number of cooperating classes. Even though the "Gang of Four" patterns are also over a decade old, no additions have been added to the system by the authors. Many other design patterns have been published since then, however, claiming to target the same domain as the "Gang of Four" patterns, for example the well—known "POSA" patterns. To our knowledge, no unified attempt has yet been made to combine the vast number of design patterns into a unified language, or even system. This does not mean that the "Gang of Four" pattern system is static, or has not evolved. As stated, many individual "Gang of Four" design patterns have spawned variants or other related patterns. Furthermore, variants of the system itself could also evolve, for example a system balancing the Multithreaded Safety force described by Lea [Lea93, i.12]. To handle this force explicitly, each "Gang of Four" pattern would have to be re—engineered, causing at least changes to the description and sample code, but perhaps also to the pattern it self.

3.9. Pattern Application

As any tool or method, design patterns must be used correctly, i.e. when the design warrants it. It is as simple as that. It is as difficult as that. Patterns cannot really offer any guarantees that the application design will be a successful one [Vlissides97, i.5], and a critical, or at least careful, approach to any pattern is warranted in our opinion. Usage is closely related to how design patterns are perceived; i.e., as a practical tool; formally; more abstract along Alexander's original ideas; or somewhere in between.

3.9.1. Usage

Several anti—patterns can help describe misuse of design patterns. The Cargo Cult [PPR, p.CargoCult] anti—pattern can explain the dangers of using design patterns without understanding why, and on a software engineering level, it can describe the dangers of following the procedures dictated by an OO Method (OOM) without understanding why. This is relevant for the evaluation in case the investigated "Gang of Four" patterns advocate the use of specific language features. The Golden Hammer [PPR, p.GoldenHammer] anti—pattern can describe the overuse of design patterns, i.e. applying patterns were they are not needed, perhaps adding unnecessary layers to the code (over—engineering). As a practical example, Livshits describes misuse of the Extend Super (or the anti—pattern Call Super) and Observer patterns in the Eclipse project [Livshits05, p.1-3].

At [PPR, p.DesignPatternsConsideredHarmful], these very issues are discussed, especially over—engineering. No conclusions are drawn, naturally, as the use and understanding of design patterns is the individual designer's prerogative, in part because of the lack of formalism. However, we do think that critics often neglect the fact that design patterns are more than recipes, especially concerning the "Gang of Four" patterns. A common misconception is what Vlissides calls "the belittling dismissal" [Vlissides97, i.2]: patterns are only seen as recipes containing jargon, rules, programming tricks, data structures, etc., without acknowledging important pattern aspects such as problem, context, teaching, and naming. The value of the "Gang of Four" vocabulary should not be underestimated because it is a powerful tool for communicating design issues [Vlissides97, i.4]. While the "Gang of Four" patterns may not be applicable for all contexts, the concepts and themes identified in section **2.1.1** and **2.1.2** can still be utilised, because designers familiar with the patterns should also be familiar with these themes and concepts.

Pattern usage is closely tied to the implementation. Another issue raised is whether the use of design patterns result in duplicate code within a system or not [PPR, p.PatternBacklash]. Some of the main design goals in any OO system are reuse, maintenance, and modification [WirfsBrock90, p.9]. The goals are reflected in the most important non—functional forces regarding OO development as defined by Buschmann et al., namely Changeability, Interoperability, Efficiency, Reliability, Testability, and Reusability [Buschmann96, p.404-410] (see also section **3.4** on page 38). Ideally, functionality should be referenced, not copied [PPR, p.OnceAndOnlyOnce]. Duplication of code does not mix well with these principles and forces. Therefore, the principle in applying patterns can seem contradictory to the very forces the patterns should heed. General refactoring, pattern componentization, and language support are possible solutions to duplicate code problems. Componentization and language support are discussed in some detail in chapter **4**.

3.9.2. Understanding

Dominus has a bleak, but practical and perhaps more realistic, view on software design patterns and their usage, and insists that software design patterns as described by the "Gang of Four" and many others are fundamentally different from Alexander's ideas of patterns and especially pattern languages [Dominus02]. According to Dominus, the "Gang of Four" idea is to discover existing design patterns (mining), and then program people to implement them habitually. Contrary to this, Alexander's pattern language help decide what should be designed, but does not dictate how to design anything; the user can decide what patterns will lead to a good design. Hence, Dominus concludes that the two approaches are completely different, representing two different meanings assigned to the term *design pattern*. The "Gang of Four" approach is much less profound and human, and he strongly advocates that the software pattern community needs to re—implement Alexander's ideas.

As we understand Alexander, we agree with the statement that software pattern collections really do not express the ideas set forth by Alexander concerning pattern languages, e.g. QWAN, order of unfolding, generativity, etc. We perceive design patterns as a valuable and practical tool to aid the design process, but one that do not generate fixed solutions. Ergo, we once again agree with Vlissides, who writes [Vlissides97, i.6]:

> ***The key to generativity is in the parts of a pattern dedicated to teaching*** *— the forces and their resolution, for example, or the discussion of consequences. These insights are particularly useful as you define and refine an architecture. But* ***believing that patterns themselves can generate architectures or anything else is definitely over the top****.* ***Patterns don't generate anything; people do****, and they do so only if both they and the patterns they use are up to snuff.*

On the other hand, we think that software design patterns in practice generally try to express the ideas of forces and pattern elements, and to some extent pattern qualities. A testament to this is the large number of references to Alexander's writings on these exact topics, for example [Appleton00; Gamma95; Lea93; Lea00; PPR], but more importantly that the pattern formats commonly used all express these notions. Dominus seems close to performing "a belittling dismissal" [Vlissides97, i.2] of the "Gang of Four" patterns. We also think Dominus neglects the fact that Alexander's patterns really cannot be chosen completely at random because of the order of unfolding, including granularity. Because software pattern collections differ from Alexander's ideas, they rarely have such restrictions. The designer is still free to choose relevant design patterns, and should be able to decide how to implement them. In any case, even if the meaning of software design patterns differs from Alexander's notion, they can still be used (as Dominus also notes).

3.10. Summary

A software design pattern is a pattern related to the design of software systems, but patterns can be applied in different areas and fields. The **term "design pattern" refers to a classification** of software design patterns that can be **used throughout the OOD phase for OO systems**, targeting **communicating objects and classes that are customised to solve a general design problem in a particular context**. Design patterns thus **rely heavily on OO concepts**, and **separate the principles from the implementation**. Different languages can thus be used to implement the solution described by a given pattern.

The notion of **a pattern is two—fold**: **a pattern is an abstraction of practical experience and basic knowledge**, but it is also **a literary description of this knowledge**, written in a **consistent format.** The pattern describes the **problem it solves** as well as a **solution to it**; hence, the pattern can be applied for similar problems in other contexts. As such, design patterns are not invented, but **discovered in existing solutions**. Different formats exist, containing required **pattern elements to describe different important aspects** of the pattern functionality, such as a **concise name**, **forces**, **related patterns**, etc. A **format traditionally uses natural language**, **illustrations**, and **examples** as opposed to formal specifications. **The naming of patterns allows designers and others to communicate architectural ideas in a high—level consistent language**. Hence, **human interaction is paramount in pattern application** because patterns are not *out—of—the—box* reusable components; **pattern application as described by Alexander requires interpretation and adaptation** to apply in them in the design at hand. Within computer science, however, efforts are being made to **include patterns as language features** or implement them as **reusable components**. Not everything that can be written using a pattern format constitutes a valid pattern, because **patterns must possess a certain set of qualities** to ensure the overall quality of the design. The qualities are **Abstraction**, **Composibility**, **Encapsulation**, **Equilibrium**, **Generativity**, and **Openness**, and many of **these qualities have similar constructs in OO**.

A pattern can be an entry in a collection. A **pattern catalogue** is a collection of loosely and/or informally related patterns, while a **pattern system** is a cohesive set of related patterns described in a consistent format that works closely together. A **pattern language can be viewed as a pattern system covering a complete domain with rules and guidelines**, which explain how and when to apply its patterns **to solve a problem that is larger than any individual pattern can solve**. Patterns and pattern collections will **evolve over time**, reflecting **knowledge gained through continued use and adaptation**. Patterns can present solutions to known software failures that are recorded as **anti—patterns**, but **unwise application of patterns may be an anti—pattern** by itself. As patterns are implemented using a given programming language, the **features of the language may influence the application** and perhaps bring new insights to the pattern description. On the other hand, the **pattern description may also dictate behaviour that has direct impact on the implementation**.

Within the pattern community, there is some **debate about what a design pattern is**. Some people are followers of **Alexander's ideas, which emphasise the human interaction**, while others prefer **more structural approaches in order to analyse and apply patterns**. Pattern formalism tries to bring rigid structure to design patterns at the expense of human interaction. In this thesis, we try to **apply the best from both worlds**. We perceive **design patterns as a valuable and practical tool to aid the design process, but one that do not generate fixed solutions**. As any tool or method, **design patterns must be used correctly: only when the design warrants it**.

The **"Gang of Four" design pattern system contains twenty—three design patterns classified in two dimensions**: **Scope** and **Purpose**. The **Scope** criterion identifies whether the **pattern applies to Classes** and/or **Objects**. **Purpose** is a **problem—based criterion** that classifies the "Gang of Four" patterns according to **what they do**. **Creational patterns** focus on the **instantiation process of objects**, **Structural patterns** on how **classes and objects are composed to form larger structures**, and **Behavioural patterns on algorithms and assignment of responsibilities** between objects. The "Gang of Four" patterns are **described using the "Gang of Four" format**, using C++ and Smalltalk as example code, and we deciphered the pattern descriptions **to clarify and label the relationships** between the individual "Gang of Four" patterns. The "Gang of Four" patterns **express the OO themes and concepts described** in chapter 2. The **concepts and C++ constructs** used in the "Gang of Four" canonical pattern implementations **will be used extensively in the evaluation as reference points for the features used in the Java 6 implementations**.

4. Related Work

When the only tool you have is a hammer,
everything looks like a nail.
*— **Abraham Maslow***

The connection between design patterns and programming languages has been debated since the "Gang of Four" patterns were published. In this chapter, we discuss selected studies regarding application of the "Gang of Four" patterns using specific languages, as well as how different features may help provide simpler implementations or even components. The level of language support for a given pattern will influence the implementation because it determines how much work is required to apply the pattern. We focus on both dynamic and static languages as we consider Java 6 a hybrid: though static, its special language mechanisms and runtime capabilities allow Java to exhibit very dynamic behaviour at runtime. We compare the different studies and relate the observations to issues deemed relevant for Java 6. As always, a summary concludes this chapter.

The choice of language will affect the pattern application because the language will ultimately decide what can and what cannot be done (easily) in light of supported programming paradigms [Gamma95, p.4]. Several studies have been undertaken to investigate "Gang of Four" pattern application in various programming languages. This chapter discusses studies of implementations in dynamic languages like Common Lisp, Dylan, and Scheme, and in static languages like C++, Java, Java + AspectJ, and Eiffel. Common features used in the various studies as well as discovered common pattern behaviour are compared in section **4.4**, but this is not an easy task as the studies have different focus. We start by establishing the level of support a given language has for a given pattern.

4.1. Language Support

The traditional close connection between design patterns and statically typed languages is criticised by some, mainly because static languages often lack advanced runtime constructs. The "human compiler" is put to work, repeatedly writing Meta—programs, e.g. patterns, to cope with the missing (runtime) features [Graham02]. Even more specific, some believe the "Gang of Four" design patterns are simply a library of C++ code templates [Dominus02; PPR, p.DesignPatternsInDynamicProgramming]. However, such claims seem to neglect that several of the "Gang of Four" patterns were exemplified using Smalltalk that has advanced runtime features, including dynamic typing and reflection. The patterns are still relevant even if implemented in Smalltalk. Still, others regard traces of the "Gang of Four" patterns in the source code as *code smells*; an indication of the language used is not powerful enough and/or developers blindly using design patterns [Halloway07]. This view assumes that the entire pattern abstraction can be represented as language features. The point is moot as already discussed in section **3.9** because like any other tool, design patterns should be used only when the design merits it. Furthermore, we have yet to a see a language that has built—in support for all the "Gang of Four" patterns.

At [PPR, p.AreDesignPatternsMissingLanguageFeatures], it is discussed whether a pattern stops being a pattern in the context of a language that has some kind of built—in support for it. The discussion concerns the verbal use of the term pattern and as well as its meaning. There is no definitive conclusion presented, but it is suggested that design patterns are one way programming languages can evolve. The consensus seems to be that a pattern does not stop being a pattern because a given language has support for it, but that designers stop referring to it explicitly as a pattern, effectively altering the common vocabulary of patterns for the given domain. From this follows that developers will stop referring to the pattern description as well; therefore, we conclude, the ultimate consequence must then be that when all programming languages implement a given pattern, or have support for it as a component, there will no longer be use for its description. These issues indicate a very strong connection between patterns and languages, but one that may eventually be discarded. Meyer calls this the Pattern Elimination Conjecture: any useful pattern should in the long term be discarded as a pattern, and replaced by reusable components with a clear, simple, directly usable interface [Meyer03, p.41]. This corresponds well with the efforts being made regarding pattern formalism as described in section **3.6** at the expense of human interaction in the application process.

4.1.1. Implementation Level

In [Norvig96], Norvig classifies the level of implementation a pattern can have in a given programming language as paraphrased in table **4.1** below. Built—in support for a pattern thus corresponds to either Invisible or Formal if part of the standard libraries, while Informal corresponds to Alexander's view on pattern application.

Table 4.1 — Pattern implementation level (modified from [Norvig96, p.7])

Level	Description	Java 6 Example
Invisible	A pattern is so much a part of the language that its usage is not noticed by the user.	The *for—each* loop help hide explicit usage of the Iterator pattern.
Formal	A pattern is implemented in a language, but must be instantiated or called for each use (component).	The Iterator pattern can still be explicitly implemented and/or used.
Informal	A pattern is part of a common shared vocabulary and referred to by name, but must be implemented from scratch for each use based on its description.	The Singleton pattern must be implemented for each relevant class.

Note, that even though a pattern is invisible on average use does not mean that it cannot be used formally. Invisible and Formal does not exclude a pattern from a common vocabulary or from being implemented alternatively either, e.g. Informally. The classification is rather subjective because different users may notice different things, depending on their point of view: an API developer may need to create different Iterator implementations, but the API user may not need to. In our view, the distinction between Invisible and Formal is vaguely defined, whereas it is easier to distinguish between Formal and Informal.

As summarised in the section **4.2.1**, Norvig implements the "Gang of Four" patterns in Common Lisp and Dylan and arrives at "simpler" implementations. At [PPR], Norvig's simpler implementations of the "Gang of Four" patterns are seen as augmenting Graham's critique about the "human compiler" at work. However, while Norvig is in agreement with Graham in using certain dynamic features to implement functionality, Graham is directly criticising the concept of design patterns. By using a proper language, says Graham, the need for design patterns is non—existent. We disagree, and to our understanding, so does Norvig, Meyer [Meyer03, p.41], and [PPR].

4.1.2. Discussion

In our view, the validity of a design pattern is not lessened because a given language has full, partial, or just easier support for it. The knowledge represented by a design pattern should ideally be independent of any specific programming language, what Norvig calls Programming Into a language [Norvig96, p.58] as already described in section 2.5. Lea states that a pattern is not an implementation, but instead describes when, why, and how to go about creating an implementation or other engineering product [Lea00, i.6]. We believe the danger of equalising design patterns with the implementation is forgetting that the human factor is paramount in understanding and applying design patterns in different contexts, which is exactly what makes design patterns a very flexible tool indeed. On the other hand, the ease and practicality of using design patterns is also important. For example, nobody wants to implement the Iterator pattern for each system. In Java, the Java Collections framework is used. In Java, however, everybody has to implement the Singleton pattern for each applicable class because the language does not support the abstraction described by the pattern (which Java actually in part does, as discovered during the evaluation and explained in section 7.1.1.4 on page 92). Hence, the sheer practicality in the frequent application of certain simple design patterns in our view warrants Formal and/or Invisible implementations, respectively componentization and/or language support. Even though Coplien is in favour of pattern componentization [Coplien, i.3], he equalises the human factor (still required) with creativity and claims it will always be needed [Coplien, i.11]. Along these lines, Fowler argues that patterns are needed because real—world solutions have failed despite using the latest technology for lack of ordinary solutions. Patterns provide a way to organise and name those ordinary solutions to make it easier for ordinary people to use them [Fowler06]. This is contrary to Graham's claim of design pattern usage being "institutionalised" [Graham02], especially considering no standard formalisation of patterns has been agreed upon as discussed in section 3.6 on page 42.

Practical pattern implementation, however, is dependent on the pattern granularity. A pattern can be applied *across* systems, but also *within* systems. Typically, architectural design patterns, having large granularity, are applied once per system, for example the Two—Tier Architecture pattern from example 2.3 on page 25. We find it reasonable to assume their level of granularity and abstraction will make them difficult to componentize compared to the "Gang of Four" patterns with finer granularity. Therefore, they *must* be adapted to the system at hand. Hence, it is also unlikely they will evolve into language features. Ergo, they will not cause duplicate code. Conversely, the "Gang of Four" design patterns describe problems and solutions that are so common they occur in many different contexts with relatively fine granularity within the same system. It is unreasonable to assume that such patterns, for example Iterator, would be applicable only once in a system, even more so for idioms as they are very tightly connected to a given programming language.

On the other hand, even some "Gang of Four" patterns like Facade and Template Method pose problems because of abstraction and granularity level. Hence, componentization and language support in form of Formal and Invisible patterns, respectively, can augment reuse, but patterns that for one reason or another remain Informal still generate specific implementations for each usage even within the same system. We consider the Singleton pattern the archetypal example of this in Java. This raises the issue if Informal design patterns collide with the principles of OO as discussed at [PPR]. In our view, this is not the case. Each application of the pattern will cause an implementation targeted for a *specific* problem in a *specific* context within the system. Two *different* contexts will thus cause two *different* implementations, albeit similar. Common functionality can still be factored out. What appears as duplicate code is in reality not; the true semantics come from the pattern description combined with the specific adaptation. In Alexander's domain, a house could be an analogy to an OO system: is it unreasonable to assume that within a given house, a "window pattern" can be applied more than once? Obviously not, and this results in duplicate functionality. In case the design of a window is discovered to be flawed, for example if the glass does not provide sufficient insulation, all windows have to be repaired or replaced eventually. Still, the alternative is surely not to apply the pattern only once.

We believe the success of componentizing a given pattern into a language or library depends perhaps more so on its abstraction and granularity level than the language in which it is implemented. It is possible, though, that the language will dictate behaviour that makes componentization difficult, if not impossible. While the human factor is important in understanding and applying patterns, we still think simple design patterns should be componentized, if possible; or even better evolve into language features.

4.2. Dynamic Languages

The studies by Norvig [Norvig96] and Sullivan [Sullivan02a; Sullivan02b] emphasise that dynamic features of Common Lisp, Dylan, and Scheme, respectively, have a large impact in providing simpler implementations. However, we cannot find a standard precise definition of what a dynamic language is or what it must support. A generalisation is that a dynamic language possesses one or more of the following overall features: dynamic typing, runtime code modification, and interpretation [Hacknot07]. Dynamic typing (or dynamic type binding) enforces type rules at runtime as opposed to compile—time. The type of a variable is not determined until the variable is actually used at runtime [Sethi96, p.137]. Runtime code modification allows changes to the structure of executing code, for example adding new methods to an object. Interpretation is the process of reading and evaluating program code at runtime without prior compilation; an interpreter runs the program directly [Sethi96, p.20]. It is also worth noting that Common Lisp, Dylan, and Scheme all are functional languages.

4.2.1. Common Lisp and Dylan

Not long after the "Design Patterns" book was published, Norvig showed that sixteen of the twenty—three "Gang of Four" patterns have qualitatively simpler implementation in Common Lisp or Dylan compared to C++ for at least some uses of each pattern [Norvig96, p.9]. Common Lisp and Dylan are dynamic languages, and many of the language features found in dynamic languages are exactly what makes the pattern application simpler, such as first—class types [Norvig96, p.10]. Table **4.2** illustrates the specific features Norvig found that influenced specific "Gang of Four" patterns.

Unfortunately, Norvig does not directly apply his implementation level classification to the "Gang of Four" patterns, nor does he discuss why seven patterns cannot be made simpler in dynamic languages. Though note that four of them are Structural patterns, i.e. Adapter, Bridge, Composite, and Decorator, two are Creational, i.e. Prototype and Singleton, and only one is Behavioural, namely Memento. At first, this seems to make sense:

dynamic languages are all about runtime behaviour whereas Structural patterns represent (static) structure. However, the ability to perform runtime code modification would seem to affect at least Adapter, and Decorator is an ideal candidate for method combination. Furthermore, the two Creational patterns would be almost directly supported in prototype—based languages, which Common Lisp and Dylan are not; Common Lisp utilises CLOS for OO capabilities (e.g. multiple—dispatch), and Dylan utilises built—in classes.

Table 4.2 — "Gang of Four" patterns in Common Lisp and Dylan (modified from [Norvig96, p.10])

Feature	Description	"Gang of Four" patterns
First—class types	Types can be used without restrictions and are treated as any other (first—class) object, i.e. can be constructed at runtime, stored in variables, have identities, etc. [Norvig96, p.11].	Abstract Factory, Chain of Responsibility, Factory Method, Flyweight, Proxy, State
First—class functions	A first—class function is a first—class object, and can for example be created at runtime [Norvig96, p.14].	Command, Strategy, Template Method, Visitor
Macros	Macros provide syntactic abstraction [Norvig96, p.17].	Interpreter, Iterator
Method combination	Combination of methods having the same signature to execute in a given order [Sullivan02a, p.9]. Enforced by the language as in CLOS or explicitly, e.g. using the Extend Super pattern (see section 3.8.2 on page 48).	Mediator, Observer
Multi—methods (multiple—dispatch, generic function)	In multiple—dispatch, methods are grouped based on their name into multi—methods, and the correct method to invoke is determined based on all the arguments [Sullivan02a, p.8-9].	Builder
Modules	A module explicitly encapsulates data and operations [Sethi96, p.209]. May also represent namespaces [Norvig96, p.28].	Facade
Not discussed	Adapter, Bridge, Composite, Decorator, Memento, Prototype, Singleton	

Some understand Norvig's work as a criticism of design patterns, but in our view, Norvig is not criticising the concept of design patterns, merely stressing the impact of the programming language, advocating the use of dynamic languages. Norvig even suggests several other pattern variants for dynamic languages as well [Norvig96, p.31]. He states that design patterns are higher—order abstractions for program organisation that help discuss, weigh, and record design trade—offs [Norvig96, p.4].

4.2.2. Scheme

In [Sullivan02a] and [Sullivan02b], Sullivan studies if language features can move design patterns away from the Informal implementation level into the Invisible or Formal levels; that is, how the basic capabilities of reflection and dynamism affect the need for, use of, and implementation of the "Gang of Four" design patterns. To try to establish a connection between modelling and programming languages, Sullivan investigates how languages can enable more abstraction in a declarative style, i.e. abstraction expressed using language constructs, for example in form of multi—methods. Sullivan emphasises the need for modelling as models enable abstraction, are declarative in style, and can allow for pre—runtime verification, but warns that dynamic features make it more difficult to analyse program statically [Sullivan02b, p.3,35]. As the language, Scheme is used with the GLOS library that adds certain OOP facilities. Sullivan reasons that reflection is closely related to first—class values as reflection refers to the ability of a program to reason about its own structure and behaviour [Sullivan02a, p.3]. Table 4.3 summarises the outcome of Sullivan's investigations based on the summaries for each investigated pattern in [Sullivan02a]. It illustrates which features where useful in a given pattern implementation, and there is a large overlap with the features discussed by Norvig from table 4.2.

Table 4.3 — "Gang of Four" patterns in Scheme + GLOS

Feature	Description	"Gang of Four" patterns
First—class types	See table 4.2 on page 58.	Abstract Factory, Builder, Prototype
First—class functions	See table 4.2.	Adapter, Builder, Chain of Responsibility, Command, Iterator, Mediator, Strategy
Macros	See table 4.2.	Proxy
Method combination	See table 4.2.	Decorator, Proxy, Memento
Multi—methods (multiple—dispatch, generic functions)	See table 4.2.	Abstract Factory, Adapter, Builder, Chain of Responsibility, Factory Method, Mediator, Observer, Strategy, Visitor
Modules	See table 4.2.	Adapter
Reflection	Reflection refers to the ability of a program to reason about its own structure and behaviour [Sullivan02a, p.3].	Abstract Factory, Prototype, Chain of Responsibility, Memento
Instantiation protocols	Controls how objects are created, either explicitly or implicitly (hidden or built—in).	Factory Method, Singleton, Flyweight, Proxy
Singleton types	A type that matches exactly one value [Sullivan02a, p.6], e.g. an instance of `java.lang.Class` in Java.	Abstract Factory, Factory Method, Proxy
Predicate types	Predicate types are based on predicate functions and thus resolved at runtime [Sullivan02a, p.9].	State
Closures	A closure consists of an expression (function) and its saved environment [Sethi96, p.534].	Command, Flyweight, Iterator, Strategy
Prototype—based	Has no notion of classes. Behaviour reuse is achieved by cloning existing objects that act as prototypes.	Prototype, State
None (similar)	Bridge (universal), Composite, Facade (universal), Interpreter, Template Method (universal)	

In accordance with Norvig, Sullivan concludes that dynamic features such as reflection, multiple—dispatch, higher—order functions, and predicate types have a positive impact on nearly all of the "Gang of Four" patterns [Sullivan02a, p.43]. Underlined patterns in the table above represent similar usage by Norvig. Instantiation and method protocols are also effective [Sullivan02b, p.34]. Sullivan states that the need for explicit patterns may disappear or the implementation may become much simpler, but mention that the Scheme implementations do not always capture the entire pattern functionality. Emphasis is clearly on the implementation aspect at the expense of pattern abstraction. Factory Method and Singleton, for example, are described as easily implemented in any language supporting an extensible instantiation protocol, but C++ already supports modification of the instantiation protocol by overloading `new` [Stroustrup91, p.215]. This indicates that the patterns describe more than code. Command and Strategy are implemented using closures, but closures do not capture the abstraction of an object with additional functionality, i.e. extra functionality, polymorphism, identity, etc. Just because a language supports a given feature does not mean the feature is the pattern by itself. What it does mean is that the feature in certain cases can represent the pattern implementation in the language in question.

Comparing the results from Sullivan and Norvig, it puzzles us that Norvig only lists the patterns as utilising a single feature. It is probably for educational or practical purposes, i.e. listed according to primary exploited feature. We think this is why there is little overlap in features for the individual patterns (underlined patterns in table **4.3**), while the overall conclusions are the same. Unfortunately, it makes it hard to conclude anything based on Norvig's study. Sullivan is more detailed, and several of the patterns not covered by Norvig are addressed, for example Adapter, Decorator, Prototype, and Singleton. Sullivan's conclusion, however, comes as no surprise as Scheme is closely related to Lisp. Like Norvig, Sullivan accedes that the "Gang of Four" patterns are closely related to design and modelling as the patterns discuss design trade—offs. Even more so, certain patterns represent universal programming concepts that cannot be solved with language features alone [Sullivan02a, p.43; Sullivan02b, p.36].

4.3. Static Languages

C++ and Java are statically typed languages. Type errors are detected at compile—time. The advanced features discussed by Gamma et al. for the "Gang of Four" implementations all but a few exclusively targets Smalltalk, which uses dynamic typing.

4.3.1. C++

The "Gang of Four" patterns all supply implementation or sample code in C++. The features used are those presented in the Implementation and Sample Code pattern elements in [Gamma95]. Gamma et al. primarily use C++ constructs found commonly elsewhere as well, e.g. classes, inheritance, access modifiers, etc., but more exotic features like templates, multiple inheritance, friends, overloaded operators are also utilised. These features are not found in Java 6, and hence alternative ways to implement the pattern in question must be applied.

4.3.2. Java

All the "Gang of Four" design patterns have been implemented in at least Java 1.2, 1.3, and 1.4, some even at an Invisible implementation level as exemplified in section **2.6.2** on page 29. Many different Java implementations of individual "Gang of Four" patterns exist. Grand, for example, presents almost exact Java 1.2 versions of all the "Gang of Four" design patterns in [Grand98; Grand99], and Hannemann et al. have implemented pure, albeit very simple, Java 1.4 versions used for comparison with the AspectJ implementations discussed next [Hannemann02]. Another example is [Eckel03], where some of Java's more advanced features such as reflection and dynamic proxies are used. However, the implementation level of the individual patterns is primarily Informal, and we know of no efforts to componentize the "Gang of Four" patterns in Java 5 or 6.

Hence, all "Gang of Four" patterns are known to be applicable in Java. This is expected as Java adheres to the fundamental OO concepts and can directly express many of the themes discussed by Gamma et al. Java is not considered a dynamic language, but it still possesses advanced runtime features like reflection and dynamic class loading. Bearing Norvig and Sullivan's work in mind, we therefore assume that the practical application will be easier and/or present alternatives to the canonical C++ implementations. On the other hand, runtime features in class—based languages often yield verbose source code, which could imply more work, and possibly clutter the core functionality and intent of the pattern when reviewing the source code. Reusable libraries and components, however, can shield the pattern implementations from much of this.

4.3.3. Java and AspectJ

Sullivan notes that *crosscutting concerns* of Aspect—Oriented Programming (AOP) matches well design patterns because patterns are primarily concerned with the coordination of multiple "parts" of a system, typically via classes and abstract methods [Sullivan02a, p.3]. Patterns are the glue that connects the joints [Sullivan02b, p.6]. Hannemann et al. have shown this in practice by implementing the "Gang of Four" patterns in Java 1.4 and AspectJ [AspectJ], claiming that seventeen of the twenty—three implementations exhibit modularity improvements in terms of better code locality, reusability, composibility, and (un)pluggablity. The improvements vary, but with the greatest improvement coming when the pattern solution structure involves crosscutting concerns, e.g. one object playing multiple roles, many objects playing one role, or an object playing roles in multiple pattern instances [Hannemann02, p.1]. Besides locality and reusability, and following code—level benefits, Hannemann et al. state that modular pattern implementations ensure that the entire pattern description of a pattern instance is localised and does not "get lost" or "degenerate" in a system as could otherwise pose a problem [Hannemann02, p.7]. Twelve of the implementations constitute reusable components with respect to abstract *aspects* [Hannemann02, t.1].

AspectJ uses aspects to encapsulate crosscutting concerns in one place. They can apply additional behaviour, or advice, to various *joint points*, for example constructors or methods. Joint points are specified using *pointcuts*, either directly or in form of a "query" to detect if a given joint point matches the aspect based on signatures. Furthermore, to encapsulate all code related to a given concern in a single aspect, the *open class* mechanism might be used to declare members or parents of another class. For a full introduction to AspectJ, see [AspectJ]. Table 4.4 on page 62 lists the different AspectJ features used to improve the various "Gang of Four" implementations.

As expected, the dynamic features of AspectJ are what facilitate easier implementations. Advice is equivalent to the method combination features found in CLOS and GLOS, clearly illustrated in the ability to execute the contained code before, after, and around join points, though advice cannot be added or removed at runtime [Sullivan02a, p.20]. Hannemann et al. utilise this feature extensively, for example to intercept calls to `new` for Singleton classes, thereby creating a specific instantiation protocol. Pointcuts can be seen as macros. In pure

Java, the reflection mechanism does not allow for structural changes to classes or objects, only behavioural ("read—only"). By exposing the structure of the executing program as objects, for example the `java.lang.Class` and `java.lang.reflect.Method` classes representing a class and a method, respectively, such objects can be accessed like any other first—class object. In AspectJ, however, the open class mechanism is a work—around to modify Java's otherwise static classes, approaching true first—class behaviour. AspectJ still adheres to the standard Java semantics, which gives certain static advantages such as compile—time type errors. Some errors are still only seen at runtime, for example if a given pointcut does not capture a jointpoint as expected.

Table 4.4 — "Gang of Four" patterns in Java + AspectJ

Feature	Description	"Gang of Four" patterns
Roles only used within pattern aspect	Abstract aspect per pattern defines the roles and default implementations where possible, local to the pattern realisations. Abstract pointcuts specify hooks for additional specialisation [Hannemann02, p.5].	Chain of Responsibility, Composite, Command, Mediator, Observer
Aspects as object factories	Patterns are abstracted into aspects containing code for the factory functionality; the factory methods used are contained either in the abstract aspect or in the participants [Hannemann02, p.5].	Flyweight, Iterator, Memento, Prototype, Singleton
Language constructs	Pattern implementations are directly affected by language constructs such as the open class mechanism or by attaching advice [Hannemann02, p.6].	Adapter, Decorator, Proxy, Strategy, Visitor
Multiple inheritance	Pattern implementations can implement any number of interfaces and use the open class mechanism to attach default functionality [Hannemann02, p.6].	Abstract Factory, Factory Method, Bridge, Builder, Template Method
Scattered code modularised	Attaching advice to be break tight coupling between participants [Hannemann02, p.7].	Interpreter, State
None (similar)	Facade	

Some of the implementations in AspectJ result in a completely new design structure. We find it difficult to identify the actual role Java occupies in this study as opposed to specific AspectJ features. Most of the Java features used in the implementations are trivial, such as classes and interfaces. Very few advanced features such as inner classes and weak references are used. The pattern functionality is achieved with the AspectJ features, which may mimic C++ features such as multiple inheritance and private (functional) inheritance.

By using Java's built—in reflection mechanism and annotations as of version five, we believe much of the same dynamic functionality could be achieved without the use of AspectJ, though at some expense. Classes could implement advice functionality that can be attached to any accessible object (jointpoint), i.e. field, constructor, or method. Pointcuts could be specified by annotations. Unfortunately, all access to enriched objects must go through proxy objects to intercept invocations to apply the advice, but it would allow the advice to change at runtime. This indicates a need for a framework to handle the execution. Furthermore, as reflection would be utilised extensively, runtime errors are in effect unavoidable, but probably manageable. It reminds us of existing products using similar ideas, such as Hibernate, JBoss Seam, or Google Guice. In any case, in accordance with Sullivan's conclusions, Java's reflection mechanism could have a significant impact on the "Gang of Four" pattern implementations as most are exemplified using C++ that has few runtime features.

4.3.4. Eiffel

In section 2.6.2 on page 29, we discussed how patterns could aid in the implementation phase during OO development. Options are adaptation and application, componentization, and language support, corresponding with Norvig's implementation levels Informal, Formal, and Invisible, respectively. Using Eiffel as the programming language, Meyer and Arnout claim to provide full componentization for eleven of the "Gang of Four" patterns and partial componentization for an additional four [Meyer06, p.3], totalling two thirds of the patterns. Full componentization is defined by Meyer and Arnout to include all the original pattern functionality [Meyer06, p.2] and is equivalent to what Norvig calls first—class patterns [Norvig96, p.31]. Their implementation level is Formal because as components they can be treated like any other object [Norvig96, p.32], or Invisible as part of the language. Componentization is an effective way to avoid duplicate code as discussed in section 3.9.1 on page 50. It is more difficult than ad—hoc pattern application as examined by Norvig and Sullivan, however, because it focuses extensively on reusability. The focus of Meyer and Arnout is closer to pure Java 6 implementations compared to the AspectJ implementations by Hannemann et al., because their components rely on reusable classes, not reusable aspects [Arnout06]. Furthermore, the specific Eiffel features utilised in the components are described in [Arnout06, t.1].

Examples of full componentization achieved by Meyer include Composite, Command, Abstract Factory, and Visitor [Meyer06, p.10-11]. Six patterns required some form of automated support to help integrate them into libraries through reusable skeletons, or though components that address part of the problem. Only two patterns could not even be partially componentized or handled through some automated support, namely Facade and Interpreter; Facade is obvious, because it is completely dependent on the context and abstraction used, it seems universal and language independent. Componentization makes pattern application in the implementation phase much easier, but also fixates the behaviour to the functionality available. A componentized pattern is only *applied* once, and then *reused*, possibly in a specialised fashion; it becomes a mere recipe instead of a full—fledged description. Partial componentization does not express the full knowledge expressed in the pattern descriptions, thereby limiting the pattern applicability unless the component itself expresses pattern—like qualities such as Openness and Generativity. The same is true for any Invisible or Formal implementation.

Meyer and Arnout recognise that the language used clearly affect the componentization process [Arnout06; Meyer06, p.3,11], and that componentization affects pattern applicability [Meyer06, p.11]. Eiffel is not a dynamic language as it employs static and strong typing, but many of its special features are used, such as multiple inheritance, generics with or without bounds, contracts, agents, and cloning facilities [Arnout06, t.1]. Meyer and Arnout compare the Eiffel features used with features in Java (1.4), and question if the ideas used in the Eiffel implementations can be used in Java, although they suggest that reflection might provide some solutions [Arnout06]. Language impact and componentization are therefore closely related, which is also demonstrated by the fact that AspectJ features augment the entire componentization process in the study by Hannemann et al. As of Java 5, generics have been added to Java, but multiple inheritance and agents are not supported. However, we fail to see how the lack of these features should prevent componentization of the "Gang of Four" patterns in Java, but agree that different solutions must be implemented.

4.4. Comparison

This section offers a quick comparison between the features utilises in the examined studies and the features found in Java 6. It also summarises common pattern behaviour identified in the individual studies as well as can be expected bearing the different scope of the studies in mind. The features listed in this section is not the final list of Java 6 features and mechanisms used in the evaluation, but provide clues to which features may be useful.

The studies on Common Lisp, Dylan, and Scheme all focus on making the "Gang of Four" implementations simpler using explicit language features found in the respective languages. The AspectJ and Eiffel implementations focus on traditional OO values, primarily reusability, where the language features used are simply the means to an end. This evaluation is somewhere in between: we do not strive to make the implementations simpler – since that depends on the eye of the beholder – but to illustrate how features in Java 6 can be utilised in the implementation process, which may spawn reusable components.

4.4.1. Features

Sullivan concludes that dynamic features such as reflection, multiple—dispatch, higher—order functions, and predicate types have a positive impact on nearly all of the "Gang of Four" patterns [Sullivan02a, p.43]. Norvig agrees, and claims dynamic features found in dynamic languages is exactly what makes the pattern application simpler [Norvig96, p.10]. Java 6 in part supports two of these three dynamic traits described in section **4.2**, i.e. dynamic typing, runtime code modification, and interpretation. Java employs static typing in favour of dynamic typing. Baring instrumentation (see the `java.lang.instrumentation` package), runtime code modification is not directly supported by Java. The reflection mechanism does not allow for structural changes to classes or objects, only behavioural ("read—only"). Java objects can access Meta data reflectively, such as classes and methods, and dynamic proxies can be used to create new types at runtime. Java is compiled into byte—code that is interpreted at runtime.

The studies by Norvig and Sullivan suggest that Java's reflective capabilities will be useful in the pattern implementation. Of the features listed in table **4.2** and table **4.3**, Java 6 supports several of them but to a varying degree. First—class types and functions are only partly supported. There is no way to create a regular class or method on—the—fly, but dynamic proxies can create duck types at runtime (see section **7.1.2.4**). Besides creational restrictions, types and methods can be manipulated like any other object ("second—class objects"). Modules correspond to packages. Multi—methods are not supported, but generic methods can be used in a type safe manner for any applicable type. Closures are partly supported in form of inner classes. Instantiation protocols and method combination must be explicit enforced by the developer, which is unfortunate, as these features are found useful in Common Lisp, Scheme, and AspectJ.

Meyer and Arnout list the features used for the "Gang of Four" patterns they succeeded to componentize [Arnout06, t.1]. The features used include "design—by—contract" (invariants), inheritance, multiple inheritance, generics, bounded generics, agents, and cloning facilities. Java 6 has direct support for inheritance, generics, bounded generics, and cloning. Agents, or delegates, are not directly supported by Java 6, but can be emulated by closures or using reflection. Design—by—contract and multiple inheritance are not supported. Java assertions are useless as they must be turned on to work. It may be possible to use dynamic proxies to emulate multiple inheritance.

The Hannemann et al. study does not offer much concerning which Java features to use. The comparison between features in C++ and Java 6 is already indirectly made in table 2.2 on page 15.

4.4.2. Patterns

It is interesting that Hannemann et al., Meyer, and Arnout cannot componentize at least the same set of patterns, namely Adapter, Bridge, Decorator, Facade, Interpreter, and Template Method [Arnout06; Hannemann02, t.1]. As illustrated in table 4.3, Sullivan has trouble providing simpler implementations for all of these patterns except Adapter and Decorator. Norvig does not discuss Adapter, Bridge, and Decorator, perhaps an indication of simpler implementations could not be made. This indicates the pattern abstractions are very context and problem specific. It is also interesting that out of the twenty—three "Gang of Four" patterns, only four have Class scope – and three of these are included in the above list, namely Adapter, Interpreter, and Template Method. Hannemann et al. cannot componentize the last pattern with Class scope either, Factory Method. The implementation level of these patterns thus corresponds to Informal. However, it does not say anything conclusive about language dependencies. It could seem reasonable to assume that the same language features are required regardless of language used, for example abstract classes in Template Method, package—like functionality in Facade, and composition in Interpreter. Nonetheless, Common Lisp and Scheme have no notion of classes, so this is clearly not the case for Template Method, for example. Other language features may also be applied. As an example, decoration and adaptation can be performed using dynamic proxies in Java 6. In our view, no definitive conclusions can be drawn in this respect. This corresponds with our initial belief from section 4.1.2 that the success of componentizing a given pattern into a language or library feature depends more on its abstraction and granularity level than the language in which it is implemented.

For Java and AspectJ, it is clear that Behavioural patterns are most easily componentized, with eight out of the twelve: Chain of Responsibility, Command, Composite, Iterator, Mediator, Memento, Observer, and Strategy. The last four are Composite and Flyweight (Structural) and Prototype and Singleton (Creational) [Hannemann02, t.1]. Many of the Behavioural patterns have a *container—like* structure, or operate on a container—like structure, for example Observer and Visitor, respectively. In our opinion, this makes them ideal for componentization, as the abstraction is not that complicated. Of the fifteen patterns componentized by Meyer and Arnout in Eiffel, there is an overlap with ten patterns from the AspectJ components. The only difference is Iterator and Singleton, while Meyer and Arnout also provide components for Abstract Factory, Builder, Factory Method (Creational), Proxy (Structural), and State (Behavioural) [Arnout06, t.1]. This is indeed a close match, and a strong indication that the abstractions described by Behavioural patterns are easily implemented in different languages. Common features used by Meyer and Arnout in these patterns include design—by—contract invariants, inheritance, generics, and to some extent agents. Features used by Norvig and Sullivan for Behavioural patterns primarily include first—class objects, multi—methods, method combination, closures, and reflection.

While Hannemann et al. have trouble with Creational patterns, e.g. Abstract Factory, Builder, and Factory Method, Meyer and Arnout provide componentization of these patterns as well. Sullivan and Norvig have no problems with Creational patterns either, and utilise many of the same features, for example first—class types and instantiation and method protocols. Singleton in AspectJ is also implemented using instantiation protocols.

Structural patterns seem to be the classification of patterns that generally causes most problems, regardless of the scope of the study in question. Gamma et al. state that Structural patterns rely on a small set of language features, namely single and multiple inheritance for patterns with Class scope, and object composition for Object scoped patterns [Gamma95, p.219]. This indicates that alternative solutions may be hard to implement. Meyer and Arnout only provide componentization of a single Structural pattern, namely Proxy, and Hannemann et al. of only two as mentioned above, e.g. Composite and Flyweight. Still, several Structural patterns provide very decent implementations according to Hannemann et al. in form of locality and (un—)pluggability, for example Adapter, Decorator, and Proxy [Hannemann02, t.1]. In unison, Sullivan and Norvig agree on simpler implementations for Adapter, Decorator, Facade, Flyweight, and Proxy. Again, there is an overlap of patterns, e.g. Adapter, Decorator, Flyweight, and Proxy.

The examined studies show that languages have great impact on the pattern implementations. The studies by Hannemann et al. and Meyer and Arnout also show that implementations can also express many of the desired pattern forces, such as Reusability, Interoperability, and Changeability, which are closely related to traditional OO concepts.

4.5. Summary

Below, we list and then summarise the most important points from this chapter:

- As already noted by Gamma et al., the **choice of language will affect the pattern application** because of inherent language features and the level of support for the patterns.
- The studies related to dynamic languages examined conclude that **dynamic features** and **reflection have a positive impact on nearly all of the "Gang of Four" patterns**.
- The studies related to static languages examined conclude that **many of the "Gang of Four" design patterns can be componentized**.
- Based on the examined studies and personal experience, we conclude that **Java 6 will be useful** for the evaluation because of its **mixture of static and runtime features**, but that it is the **pattern abstraction more so than the language that determines the ease of implementation** and componentization.

The **level of implementation a given pattern can have in a programming language is classified as Invisible**, **Formal**, or **Informal**. Invisible indicates a **pattern is so much a part of the language that its usage is not noticed** by the user. Formal indicates a **pattern has an implementation** in the language, but **must be instantiated or called for each use**. Informal indicates a **pattern is part of a common shared vocabulary** and referred to by name, but **must be implemented from scratch for each use** based on its description.

The studies related to dynamic languages examined conclude that **dynamic features have a positive impact on nearly all of the "Gang of Four" patterns**, for example **reflection**, **first—class objects**, method combination, multiple—dispatch, and higher—order functions. **Several of the dynamic features discussed are present in Java 6**, such as **reflection**, or can be simulated to some extent, for example via **dynamic proxies**.

The studies related to static languages examined conclude that it is **possible to componentize several of the "Gang of Four" design patterns**, but the **language and pattern abstraction will determine if a given pattern can be implemented as a component**. We believe the **success of componentizing** a given pattern into a language or library **depends perhaps more on its abstraction and granularity level than the language** in which it is implemented. **Behavioural patterns seem more manageable compared to Structural patterns**, with Creational patterns somewhere in between. This indicates support for advanced **runtime features will be beneficial**. **Patterns having Class scope are more difficult to work with** compared to patterns with Object scope. Several of the **language features used in the pattern components are present in Java 6**, for example **generics**.

5. Evaluation Approach

Whenever anyone says, "theoretically",
they really mean, "not really".
*— **Dave Parnas***

Because of the versatility of design patterns and the extensive human interaction required for utilisation, there is no straightforward way to benchmark the correlation between patterns and their implementations using test frameworks, simulations, or other automated tools. Ideally, applying design patterns require human interaction in all phases of the software development life—cycle, including in the final evaluation of the developed system. The evaluation approach defined here, however, focuses on the practical application of the "Gang of Four" patterns using a given language catalyst. This chapter defines and explains a simple evaluation approach that is independent of any given language. It can thus be used in similar evaluations with different language catalysts and perhaps different pattern systems providing they are described in the "Gang of Four" format. The goal of the evaluation is simply to implement *all representative* pattern functionality described in the Implementation and Sample Code elements for each pattern, if possible, using a single language. The evaluation outcome is then reported using a sub—set of the familiar "Gang of Four" format. Using Java 6 as the catalyst, this will allow us to perform a reasonably structured evaluation of the entire "Gang of Four" pattern system, because the individual implementations must be juxtaposed to identify common traits as well. We start by establishing the focus of the evaluation approach before we outline the approach itself. The approach requires both individual and collective evaluations of the "Gang of Four" patterns. Once the approach is defined, we use it to state the goals for this evaluation using Java 6 as the language catalyst, and we determine the language features that will be used.

5.1. Focus

Design patterns are not an exact science. There is no mathematical way to deduce if a pattern is correct or not since it is based on empirical knowledge and experience, though several formalisation techniques have emerged within the last few years (see for example [Baroni03; Eden04; Taibi07]). The concept of patterns cannot exist without human interaction, as patterns are described and interpreted by humans. The idea of a pattern must be captured and described by the author ("what does it do?"); based on it, pattern behaviour and applicability may be inferred by the user ("how is it done?"), but the interpretation will be based on the user's point of view. Neither part can be excluded. It is hard to speculate upon, which part is easier to evaluate. Evaluating well—written pattern descriptions and/or implementations could be easier than evaluating pattern abstractions because well—written descriptions could be more tangible than the concept they describe. The reverse could also be true. The evaluation performed here does not evaluate the validity of the abstractions, merely practical issues encountered during application from our point of view. How a user views the pattern will affect the application of it, and only through implementation and testing in the given scenario can the desired behaviour be confirmed. Because of the human factor and the versatility of patterns, there is no straightforward way to benchmark patterns using test frameworks, simulations, or other benchmarking tools. To evaluate patterns is to implement them from a specific point of view, which is what the evaluation approach conveys. This implies that any evaluation of patterns will be subjective and that its conclusions must adhere to the initial point of view and interpretation. Hence, the goals of the evaluation can only make sense if the viewpoints used are established and explained.

Example 5.1 — To illustrate how different points of views can affect the evaluation, consider evaluating a car, say an Aston Martin, for some magazine from the point of view of a mechanic and from the point of view of the owner. The mechanic may approach the evaluation in a technical fashion, focusing on the design of the engine, e.g. engine performance. The evaluation could investigate different parts of the engine in turn, e.g. specific criteria, and comment on issues deemed relevant by the mechanic, as well as state a conclusion to the overall performance. The conclusion might be that the engine is inferior to a car in its price range. Furthermore, certain issues could be independent of the specific engine (and car), and related to the general design of a combustion engine. The owner of the car could instead evaluate the car based on its physical design compared to other cars, perhaps focusing on the front, rear, interior, etc. The subjective conclusion could be that the car is the most beautiful one. The result is two evaluations of the same car with completely different results, one negative, and one positive. For others to use the evaluations to anything meaningful, the premise, e.g. point of view, and the specific criteria used must be known. The point of view alone is not enough, because different criteria could be used for the same point of view. For example, a vaguely formulated criteria such as "How durable is *it*?", where *it* thus means engine or car design depending on the viewpoint, yielding a positive evaluation for the design of a combustion engine, whereas car designs traditionally have a much shorter lifespan, i.e. less durable ■

The general idea is that the evaluation and pattern implementations as a whole must try to express the Gamma et al. themes and concepts described in section 2.1 on page 13. This makes sense because the individual patterns by definition must express the themes and concepts regardless of the language used. Determining if this is indeed the case is not easy. However, if we assume that the individual patterns as described by Gamma et al. express the desired properties, then their implementation should as well. By trying to implement all functionality described in the Implementation and Sample Code pattern elements, the pattern implementations attempt to express the largest possible set of desirable pattern qualities. These pattern elements are chosen because they explicitly focus on the practical application in context of specific languages and features. The contained information can rather easily be compared to other languages. The focus is on the practical *use* of the programming language to implement the design patterns, not on how the features are *constructed* internally.

The focus of the evaluation is practical and applied from the perspective of a practising designer and/or developer. The "Gang of Four" patterns should be used in a realistic, varied, and a practical manner. This requires an "application" of some size and complexity. In our view, this will produce much more realistic pattern applications than merely isolating individual pattern implementations in trivial shell—like implementations; such implementations are plentiful to be found on the Internet. Our evaluation contains no enervating "Dogs and Cats" examples; this is a Master's Thesis, not a petting zoo ☺. As such, the evaluation merits rather advanced and complex implementations.

5.2. Description

The approach demands that all implementation issues related to pattern functionality described in the Implementation and Sample Code elements in the "Gang of Four" patterns must be addressed, and if possible, provide a solution in the language catalyst. It is sufficient to refer to similar solutions in other patterns, but the features used must in any case be established. As both the implementation and the selection of features used may be determined by the evaluator, the evaluation and its conclusions will be subjective. The detailed evaluation of the solutions in the given language must be expressed using the Name, Intent, Structure, Participant, and Implementation elements from the "Gang of Four" pattern format. This includes at least an UML Class diagram in the Structure element and identification of the pattern participants expressed in the solutions. While familiar pattern elements are used to describe the evaluation outcome, the contents are much more detailed and specific compared to the "Gang of Four" pattern descriptions. The comparative evaluation must identify common traits in the pattern implementations and establish where various features are used and what their purposes are. Common traits include both pattern and language behaviour. The format of the comparative evaluation is not defined since it is completely dependent of the language and features investigated. It must be defined by the evaluation in question.

5.3. Evaluation Goals

The purpose of the evaluation is to investigate how the use of languages features indigenous to Java 6 can affect application of the "Gang of Four" patterns, individually and collectively. As the whole concept of pattern correctness and behaviour is so elusive, the evaluation and its conclusions will be subjective. Hence, the objective is not to provide a definitive conclusion as this goes against the very idea of design patterns. Instead, the objective is to provide a realistic, but subjective, evaluation, which may be useful in disclosing how the "Gang of Four" patterns and Java 6 can cooperate. The goal is not to establish that a given pattern should be implemented using a set of specific features, but to illustrate that a given set of features may be useful in the application of the pattern.

In order to perform a reasonably structured evaluation of the entire "Gang of Four" pattern system using Java 6, we use the defined approach to implement all representative pattern functionality described in the Implementation and Sample Code pattern elements (in compliance with sub—goal II from the introduction). For each pattern, the outcome of the detailed evaluation will thus be (sub—goal III and IV):

– An introduction to the pattern, describing it using the participants and wording found in [Gamma95] (described in Name, Intent, and Participants elements);

– A simple description on how the pattern is implemented in this thesis, relating in particular the pattern participants to implementation entities (Participants and Structure);

– A detailed UML Class diagram of the implementation, where pattern participants and behaviour are clearly identifiable (Structure); and

– An explanation of how all information in the Implementation and Sample Code elements has been addressed and possibly solved (Implementation).

The outcome of the comparative evaluation will be (sub—goal IV):

- A schematic presentation describing the use of Java 6 features in the pattern implementations;
- A thorough, comparative analysis on the use of the investigated features, including examples, program listings, and in—part conclusions and identification of representative high—lights;
- A comparative analysis between the pattern relationships described by Gamma et al. and relationships expressed in the evaluation; and
- A casual classification on the level of pattern support Java 6 has based on the evaluation outcome.

The comparative evaluation is presented in chapter 7, while the detailed evaluations are presented in chapter 8. Furthermore, based on all evaluation results, overall evaluation conclusions will be made in chapter 9.

5.3.1. Features

As the last thing before we can conduct the evaluation, we need to select the set of features to investigate. A fixed set is a necessity to keep the evaluation focused, but it must be realistic. Excluding interfaces, for example, is not an option. The following core features will at least be investigated: type usage (classes, enumerations, interfaces, abstract classes, and exceptions), implementation and inheritance, generics and generic methods, inner and anonymous classes (closures), covariant return types, and varargs. Many of these features have similar constructs in C++, such as classes, generics, and covariant return types (for virtual functions [Stroustrup91, p.647]), while others do not, such as generic methods and anonymous classes. Many of these features are given, as writing any form of code in Java would otherwise not be possible. These features also encompass many of the Eiffel features used in the study by Meyer and Arnout from section 4.3.4.

As the related work examined in section 4.2 all concluded that runtime dynamic features aid in the application of the "Gang of Four" patterns, it is obvious to examine the use of Java's reflective capabilities in this evaluation. Reflective usage of class literals, constructors, and methods is examined, as well as *dynamic proxies* that allow a type at runtime to implement a given interface using reflective methods for dispatching. The use of annotations is also examined, especially when used reflectively at runtime. These features cannot be matched by C++, but Smalltalk possesses several similar features. Numerous "Gang of Four" descriptions illustrate or discuss pattern functionality relying on runtime features that cannot be directly implemented in C++, for example using classes to create objects in Abstract Factory [Gamma95, p.90-91] and Factory Method [Gamma95, p.112], or changing the class of an object runtime for State behaviour [Gamma95, p.309].

Java's built—in mechanisms for synchronisation, serialization, and cloning are also examined. C++ cannot match these mechanisms either.

The comparative evaluation will provide short descriptions of the relevant features where deemed necessary.

5.4. Summary

Below, we summarise the most important points related to the defined evaluation approach and its practical use in this thesis:

- The evaluation approach has a **practical and experimental approach** and **investigates if a given language catalyst can express all representative pattern functionality** described in the **Implementation** and **Sample Code** elements. Whether or not specific functionality can be implemented using the language catalyst must be documented.
- The evaluation approach requires **detailed and comparative evaluations**. **Detailed evaluations are more structured** and express evaluation outcome using the familiar Intent, Structure, Participants, and Implementation elements from the "Gang of Four" format. The **comparative evaluation identifies common traits pertaining to pattern and language behaviour**.
- The evaluation goals include a **schematic presentation of the *pattern* × *feature* usage** and an in—depth comparative language feature analysis of **core language features**, **reflection**, and **special language mechanisms**.
- The evaluation goals also include a **comparative analysis between the pattern relationships described by Gamma et al. and relationships expressed** in the evaluation and a **casual classification on the level of pattern support** in Java 6 based on the evaluation outcome.

The evaluation tries to **express the themes and concepts described by Gamma et al**. as realistic as possible. The pattern implementations will be **non—trivial**, and all **relate to a few common model classes** to convey the sense of a stand—alone "application". This requires more effort on behalf of the reader. On the other hand, we will strive to **produce better and fully documented program code**. The **implementation in Java 6 will try to express "Best Practices"** as described by Bloch [Bloch01].

The **objective of the evaluation is to provide a subjective investigation**, not a definitive conclusion as this goes against the very idea of design patterns. The evaluation may help **identify how the "Gang of Four" design patterns and Java 6 can cooperate** by illustrating how **a given set of features may be useful in the application** of a pattern. Three categories of features will be examined: **core language features**, **reflective capabilities**, and **special language mechanisms**. Core language features include **types**, **generics**, **closures**, **covariant return types**, and **varargs**. Reflective capabilities include **class literals**, **methods**, **dynamic proxies**, and **annotations**. Special language mechanisms include **synchronisation**, **serialization**, and **cloning**.

6. Implementation

```
       LDX #$00
loop:  LDA data,X
       STA $0400,X
       INX
       CPX #$0A
       BNE loop
       RTS
data:.BYTE $03,$36,$34,$20,$34,$05,$16,$05,$12,$21
```
— ***Gunni Rode***

This chapter presents the practical details regarding the pattern implementations. We present the environment set—up used for the evaluation, including precise Java version and IDE. Since UML cannot describe all Java features, we explain the UML extensions defined and used during the evaluation to aid the construction of UML Class diagrams. To simulate a larger "application" than standalone pattern implementations can achieve, we present the core model classes used directly or indirectly in all pattern implementations. The developed source code and documentation is available on the thesis website; here, we only present a package overview. Finally, we describe how the pattern implementations can be executed and tested.

6.1. Software

JDK 1.6.0_01 is used with compiler compliance to version 6. The JDK is available for download at http://java.sun.com/javase/downloads/index.jsp. JavaDoc is used to document the source code and is bundled with the JDK. The standard doclet is used with a compliance level to version 6.

Eclipse 3.3 (Europa) is used as the main IDE. It is available at http://www.eclipse.org/downloads. NetBeans 5.5.1 from Sun is used as the secondary IDE since Eclipse utilises its own compiler. To ensure compatibility with the standard compiler, NetBeans is used to verify compilation - the compilers do not behave exactly alike. Known issues are documented with `//ISSUE:`. NetBeans is available for download at http://www.netbeans.org. There is a single compiler error in NetBeans in the `bridge.SequenceAbstraction<E>` class, line 305, but it does not concern core pattern functionality. In our opinion, it is a compiler bug (well, at least in one of the compilers…). There are no problems in Eclipse.

A deliberate choice is that no plug—ins for Eclipse or NetBeans are required, not even JUnit. The OS used during development is Microsoft XP Professional, SP2.

6.2. Modelling

Each pattern implementation is only illustrated with an UML Class diagram, similar to the Class diagram shown in figure **6.1** on page 76. Standard UML notations are not described here, but UML cannot describe all Java features, such as final methods, annotations, or genetic bounds. Fortunately, it is extensible. Additional data types, stereotypes, and attributes are thus defined and used as explained below in table **6.1**.

Packages are rarely depicted. If so, it is only to illustrate a clear separation between patterns and/or classes.

Types include attribute and operation (constructor and method) information as deemed necessary. Two dots (..) indicate additional attributes/operations not depicted. Open—ended sub—classes are generally not depicted, but implied. All Class diagrams use Java types for clarity, pictured with their fully qualified generic name such as `java.util.List<E>`. All types defined in this thesis are presented by their simple generic name, such as `Sequence<E>` for `dk.rode.thesis.meta.model.Sequence<E>`. Inner classes are qualified by their enclosing class, for example `Sequence.State`. Parameterised type realisations are depicted with rounded corners («bind» relations), which is a deviation from the UML standard. Realisations from type parameter `E` to `E` are not illustrated, only when bound to a concrete type such as `java.lang.Integer`, or to a type parameter with a different name, e.g. `E` to `T`. Bounds on type parameters use Java syntax, like `E extends T`, `E super S`, or even wild—cards like `? super E`. All this is illustrated in figure **6.1** on page 76. Comments are light grey.

Table 6.1 — UML stereotypes and properties

Name	**Description**
«static class»	Indicates a static inner class.
«enumeration»	Indicates an enumeration, depicted like a class, but with enumeration constants before attributes.
«final class»	Indicates a final class.
{final}	A property indicating a final attribute or method.
«exception»	Indicates an exception type.
«throws»	Indicates a relationship via a thrown exception.
{exception}	Indicates a method that might throw an exception as {exception = *type*}.
«annotation»	Indicates a Java annotation type. Depicted like a class, using this stereotype.
«annotated»	Indicates a realisation of an annotated type. The non—default fields of the annotation are bound like type parameters, for example «annotated» *name*::*value*, ..
{synchronised}	A property indicating that a given method is synchronised, alternatively {synchronised = *lock*}.
{unmodifiable}	A property indicating that an object is unmodifiable, e.g. read—only.

The UML Class diagrams identify the pattern participants in a manner similar to a format suggested by Vlissides, one of the "Gang of Four" authors. Here, a participant is identified by a dark blue rectangle containing the participant name in the upper left corner of the type.

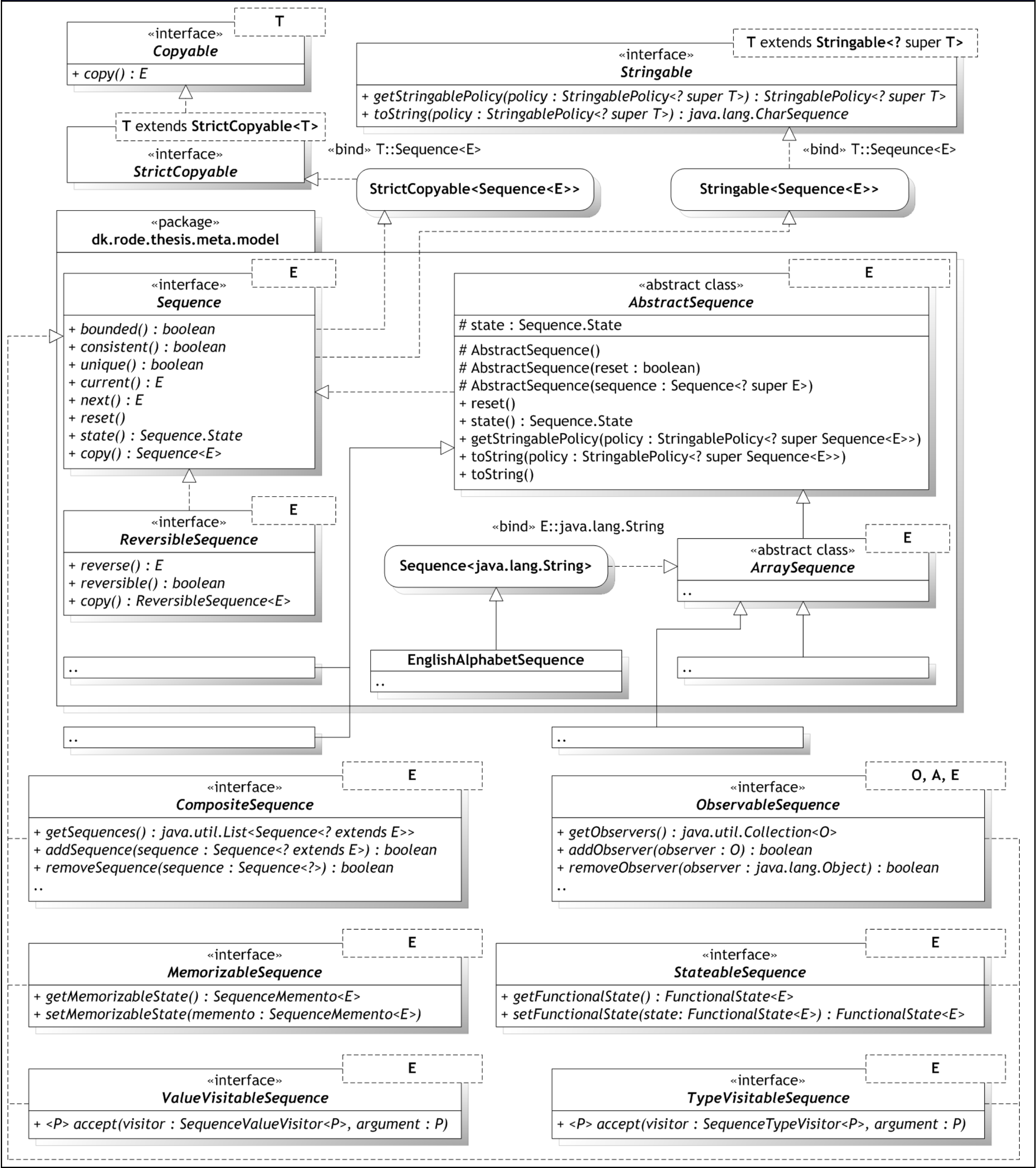
6.3. Design

All pattern implementations in this evaluation relate to a few common model classes defined in the `dk.rode.thesis.meta.model` package. This is part of the deliberate design choice to simulate larger and more complex applications than could be achieved by disjoint stand—alone pattern implementations, but also to keep the project within reason, time and development wise. Individual implementations can thus be used in other pattern implementations as well, expressing many of the pattern relationships described by Gamma et al. The primary type is the `Sequence<E>` interface, which represents a *sequence* that will deliver the next, or current, value in given sequence on demand, such as for example a Fibonacci sequence or a sequence delivering the names of the Simpsons family members. Sequences are more than mere iterators; they are value centric and have a number of interesting properties that makes them useful in an evaluation such as this. Sequences always have a lower bound, i.e. the initial sequence value, and may have an upper bound, limiting the number of possible values it can deliver. If a sequence is bounded and deliver consistent values, it will restart when the upper bound is reached on an invocation of `next()`, i.e. the same values will be delivered again, in order. The sequence values of bounded, consistent sequences are thus deterministic: two instances of the same sequence type initialised identically will return the same sequence values if utilised in the same manner. Sequences may also deliver unique values until reset or restarted. Sequences can be reset explicitly, which will cause the sequence to restart if it is consistent. The complete `Sequence<E>` interface is illustrated in figure **6.1**, including closely related types, but please refer to the JavaDoc for an in—depth description.

A given pattern implementation will either *use* or *implement* sequence functionality, but the `Sequence<E>` type is often merely a catalyst to make the implementation and evaluation concrete. The Abstract Factory implementation, for example, provides a reusable factory that can create any type of product, but the products created are sequence related. Other implementations are more entangled with sequence functionality. The implementation of the State pattern is a sequence implementation delivering prime numbers, where each concrete state represents internal sequence functionality, such as calculating prime numbers, delivering the next prime number, restarting the sequence, etc. Usage includes the Adapter implementation, which adapts the `Sequence<E>` interface to the `java.util.Iterator<E>` interface via composition, and the Interpreter implementation, which evaluates expressions that directly or indirectly manipulate sequences.

Arguably, the choice to centre all pattern implementations on a few core model types may seem contrived. There is no guarantee that "one—size—fits—all", especially considering the scope of this thesis. The evaluation of design patterns from a practical point of view requires a real context to be truly educational. Through real, practical application of a pattern using a given language will the connection between the two become apparent. Design patterns should be applied only where relevant. A design forcing the use of certain design patterns is not only contrived, it goes against the very idea of design patterns. An evaluation like this one can only try to imitate a real context. It has no choice but to implement each pattern within that context as it is the very purpose of the evaluation. On the other hand, sequence functionality as described above is generic enough to allow for many different applications of it, which we think the evaluation demonstrates. It helps convey the idea of an overall "application". At first glance, a general impression of the design and implementation as whole could be that it suffers from *featuritis*, but this is in fact not the case. On the contrary, reusing common components such as sequences allow individual pattern application to become focused, added only what is needed while still participating in non—trivial overall implementations. Accordingly, several pattern implementations define sub—interfaces of `Sequence<E>` to express the required functionality, and such types represent the focus of the given pattern implementation. Examples include Composite, Observer, and Visitor that defines the `composite.CompositeSequence<E>`, `observer.ObservableSequence<O,A,E>`, and `visitor.TypeVisitableSequence<E>` interfaces, respectively. The actual implementations need only be concerned with specific pattern functionality as general sequence functionality can be reused or inherited.

Figure 6.1 — Primary model classes


T
«interface»
Copyable
+ copy() : E
T extends StrictCopyable<T>
«interface»
StrictCopyable
T extends Stringable<? super T>
«interface»
Stringable
+ getStringablePolicy(policy : StringablePolicy<? super T>) : StringablePolicy<? super T>
+ toString(policy : StringablePolicy<? super T>) : java.lang.CharSequence
«bind» T::Sequence<E>
«bind» T::Seqeunce<E>
StrictCopyable<Sequence<E>>
Stringable<Sequence<E>>
«package»
dk.rode.thesis.meta.model
E
«interface»
Sequence
+ bounded() : boolean
+ consistent() : boolean
+ unique() : boolean
+ current() : E
+ next() : E
+ reset()
+ state() : Sequence.State
+ copy() : Sequence<E>
E
«abstract class»
AbstractSequence
# state : Sequence.State
# AbstractSequence()
# AbstractSequence(reset : boolean)
# AbstractSequence(sequence : Sequence<? super E>)
+ reset()
+ state() : Sequence.State
+ getStringablePolicy(policy : StringablePolicy<? super Sequence<E>>)
+ toString(policy : StringablePolicy<? super Sequence<E>>)
+ toString()
E
«interface»
ReversibleSequence
+ reverse() : E
+ reversible() : boolean
+ copy() : ReversibleSequence<E>
«bind» E::java.lang.String
Sequence<java.lang.String>
E
«abstract class»
ArraySequence
..
EnglishAlphabetSequence
..
..
..
..
..
E
«interface»
CompositeSequence
+ getSequences() : java.util.List<Sequence<? extends E>>
+ addSequence(sequence : Sequence<? extends E>) : boolean
+ removeSequence(sequence : Sequence<?>) : boolean
..
O, A, E
«interface»
ObservableSequence
+ getObservers() : java.util.Collection<O>
+ addObserver(observer : O) : boolean
+ removeObserver(observer : java.lang.Object) : boolean
..
E
«interface»
MemorizableSequence
+ getMemorizableState() : SequenceMemento<E>
+ setMemorizableState(memento : SequenceMemento<E>)
E
«interface»
StateableSequence
+ getFunctionalState() : FunctionalState<E>
+ setFunctionalState(state: FunctionalState<E>) : FunctionalState<E>
E
«interface»
ValueVisitableSequence
+ <P> accept(visitor : SequenceValueVisitor<P>, argument : P)
E
«interface»
TypeVisitableSequence
+ <P> accept(visitor : SequenceTypeVisitor<P>, argument : P)

6.4. Source Code

Table 6.2 lists the packages containing the developed source code. Approximately 300+ Java files have been developed, yielding approximately 400+ class files (including inner classes and enumeration constants). All types are fully documented using JavaDoc, including packages. The source code can be downloaded from the thesis website at http://www.rode.dk/thesis.

The reader of this thesis is expected to browse the generated JavaDoc to get a better understanding of the different implementations. The primary implementation of each pattern is implemented in each own aptly named package, e.g. `dk.rode.thesis.abstractfactory` for Abstract Factory. A given pattern implementation may naturally be utilised by other patterns, and additional applications of a given pattern may be present in the source code as well, for example anonymous classes used as on–the–fly adapters. Several Meta packages and classes have been developed to aid the individual pattern implementations. An example is the `dk.rode.thesis.meta.model` package as described in the previous section, or the `dk.rode.thesis.meta.reflect` package that supplies the core reflection functionality used in different pattern implementations. They are listed in grey cells in table **6.2** below. Refactoring common code into Meta classes is also a good design choice in compliance with traditional OO concepts instead of implementing everything from scratch in each pattern implementation. It is the contents of each "pattern package" that are evaluated in chapter **8**, but Meta class functionality will be included if it is essential for the pattern functionality.

Table 6.2 – Source code packages

Package	Description
`dk.rode.thesis.abstractfactory`	Implementation of Abstract Factory.
`dk.rode.thesis.adapter`	Implementation of Adapter.
`dk.rode.thesis.bridge`	Implementation of Bridge.
`dk.rode.thesis.builder`	Implementation of Builder.
`dk.rode.thesis.chainofresponsibility`	Implementation of Chain of Responsibility.
`dk.rode.thesis.command`	Implementation of Command.
`dk.rode.thesis.composite`	Implementation of Composite.
`dk.rode.thesis.decorator`	Implementation of Decorator.
`dk.rode.thesis.facade`	Implementation of Facade.
`dk.rode.thesis.factorymethod`	Implementation of Factory Method.
`dk.rode.thesis.flyweight`	Implementation of Flyweight.
`dk.rode.thesis.interpreter`	Implementation of Intepreter.
`dk.rode.thesis.iterator`	Implementation of Iterator.
`dk.rode.thesis.mediator`	Not implemented (but evaluated).
`dk.rode.thesis.memento`	Implementation of Memento.
`dk.rode.thesis.meta`	Annotations to identify and classify pattern participants.
`dk.rode.thesis.meta.log`	The log framework used.
`dk.rode.thesis.meta.model`	The core model used as the base for all patterns.
`dk.rode.thesis.meta.reflect`	Reflection utilities.
`dk.rode.thesis.meta.reflect.proxy`	Dynamic proxy utilities.
`dk.rode.thesis.meta.test`	Defines the test setup.
`dk.rode.thesis.meta.util`	Various general utilities.
`dk.rode.thesis.observer`	Implementation of Observer.

Table 6.2 — Source code packages

Package	Description
`dk.rode.thesis.prototype`	Implementation of Prototype.
`dk.rode.thesis.proxy`	Implementation of Proxy.
`dk.rode.thesis.singleton`	Implementation of Singleton.
`dk.rode.thesis.state`	Implementation of State.
`dk.rode.thesis.strategy`	Implementation of Strategy.
`dk.rode.thesis.templatemethod`	Implementation of Template Method.
`dk.rode.thesis.visitor`	Implementation of Visitor.

Each pattern package contains a `doc-files` folder containing an UML Class diagram for the implementation. This ensures that the diagram is included in the generated JavaDoc. The diagrams are included in this thesis as well, in the relevant evaluation sections.

In the evaluation, full package names will rarely be used for the developed source code. Package prefix `dk.rode.thesis` is always implied if not already included in a given type name. Once a given type or package has been referenced in a context, e.g. section or paragraph, the remaining package information will be ignored as well. Example: if the full type name is `dk.rode.thesis.composite.CompositeSequence<E>`, `composite.CompositeSequence<E>` will suffice, and additional references in the same context will thereafter simply reference `CompositeSequence<E>`. Java types are fully qualified, or go by their simple name when referenced again within the same context, as for example `java.util.NavigableMap<K,V>` and `NavigableMap<K,V>`.

6.5. Testing

Each pattern package includes a `Main` class that will execute the tests devised to illustrate the developed pattern functionality, e.g. `dk.rode.thesis.abstractfactory.Main`. **The tests are not meant as a replacement for JUnit testing, but to illustrate pattern functionality**. They can each be run directly, but the `dk.rode.thesis.meta.test` package furthermore includes two separate test classes, namely `AllTests` and `IntegrityTests`. The first runs all individual pattern tests in alphabetical order, while the latter perform integrity tests on all accessible `dk.rode.thesis.meta.model.Sequence<E>` implementations defined in the individual pattern implementations. Note that certain test files are required to run the Template Method tests and that Bridge and Memento, as well as the logger, will write to disk.

To record the outcome of the tests, two types of logs exist: a global log and logs associated with a specific class. The output is generally verbose as all objects in the evaluation implement meaningful `toString()` representations as recommended by Bloch [Bloch01, p.42-44]. It is possible to control the log level explicitly in the individual tests by altering the source code, of course, but it is easier to supply a proper boolean value for the `-log` argument to each test class, indicating whether or not logs associated with individual classes should be activated or not, e.g. `java Main -log true`. For additional verbose logging, the `-log.verbose` parameter can be used in a similar fashion. Notice there is no `=` operator between argument keys and associated values. The tests are designed so the global log should suffice to convey the intent.

The logs will output to either `System.out` or to a file. This is determined by the system property `dk.rode.thesis.log`. A string value of "file", excluding quotation marks, will log to a file in the relative directory `log`; all other values will cause logging to `System.out`. File logs will append to existing logs. System properties are supplied with the `–D` option during execution, as in `java –Ddk.rode.thesis.log=file Main –log true`. Notice the use of `=` unlike normal supplied arguments.

6.6. Summary

We have implemented the "Gang of Four" patterns in Java 6, **fully documented with JavaDoc**. The source code **expresses the "Best Practices" described by Bloch** [Bloch01] whenever possible. Each specific **pattern implementation has a dedicated package**, but may be used in other pattern implementations as well. All **implementations basically operate on the same core model classes** to simulate "application" usage. **Additional use of the "Gang of Four" patterns is applied where warranted**, for example in **Meta classes** or as part of another pattern implementation. **Test classes** have been developed **to illustrate pattern usage**.

7. Comparative Evaluation

*Program testing can be used to show the presence of bugs,
but never to show their absence!*
— Edsger W. Dijkstra

The comparative evaluation provided in this chapter presents an analysis of the pattern implementations that correlate patterns based on the Java 6 features and mechanisms used in their application. Three categories of Java 6 features are examined: *core language features* (types, generics, inheritance, etc.), *reflection* (class literals, dynamic proxies, annotations, etc.), and *special language mechanisms* (synchronisation, serialization, cloning, etc.). The core language features category primarily encompass static features, the reflection category primarily runtime features, while special language mechanisms category targets both. Based on the analysis, we provide observations on how C++ features used to implement core pattern functionality can be implemented in Java 6. We also present a schematic illustration of the pattern relationships expressed in the implementations, comparing them to the relationships described by Gamma et al. We furthermore outline traits of each pattern implementation in relation to pattern implementation levels as described by Norvig [Norvig96, p.7].

7.1. Language Features

Table 7.1 on page 82 summarises the most important language features and mechanisms applied in the implementations of the "Gang of Four" patterns. For comparison, pattern application and features utilised in developed Meta classes are also illustrated. A set of legends is used to describe the feature use in the specific pattern implementations. Regardless of the legend used, an entry in the table indicates that the feature was somehow used in the pattern implementation. The most interesting *representative* pattern functionality and feature combinations are highlighted with a dark—blue background. They are addressed individually in section 9.2, after the general feature usage has been investigated. The legends are:

- **X**: the feature is used directly in the pattern implementation. For example, the Singleton pattern uses inheritance to allow specialisation of singleton types in the `singleton` package, and the legend used for the *Singleton* × *Inheritance* table entry is thus **X**.
- **P**: the feature is used to implement the pattern functionality in another **P**attern implementation because of the close relationships between patterns. For example, anonymous inner classes are used to define concrete adapter strategies in the Apapter implementation in the `adapter` package, but the actual Strategy implementation in the `strategy` package uses enumerations to define concrete strategies. Hence, **P** is used as the legend for the *Strategy* × *Inner classes* table entry (**P** thus refers to Adapter).
- **M**: the feature is used in **M**eta classes essential for the pattern implementation. This is a design issue related to refactoring performed during the evaluation: had the feature not been used in the Meta classes, it would have been used directly in the pattern implementation. For example, *class—like adapters* are not implemented directly in the Adapter pattern package, but class—like adapters

facilitated by dynamic proxies are used extensively in the `meta.reflect.proxy.ProxyFactory` Meta class. **M** is therefore used as the legend for the *Adapter* × *Dynamic proxies* table entry.

- **E**: the feature is only used in classes **E**xternal to the core pattern participants described by Gamma et al., but such classes are still implemented in the specific pattern package. For example, the Command pattern implementation includes the `command.CommandProcessor` class, which is not described by Gamma et al., but by the "POSA" Command Processor [Buschmann96, p.277] pattern. As only the processor implementation use generic methods, and not the actual `command.Command<E>` types themselves, the legend is **E** for the *Command* × *Generic Methods* table entry.
- **D**: the feature is **D**erived because it depends on the design of other patterns and/or classes. Had the used classes not utilised the feature, the pattern implementation would (probably) not have used it either. This is the only legend that counts as a "maybe". For example, the Decorator pattern uses generics because the decorated type is the generic `meta.model.Sequence<E>` type. Hence, the table entry *Decorator* × *Generics* is labelled **D**.

Functionality "inherited" from other patterns is **not** included in table 7.1. For example, most pattern implementations in some form or another operate on the Meta model classes, in particular the `Sequence<E>` interface. As this interface extends the `prototype.StrictCopyable<T>` interface to become a prototype, most `Sequence<E>` implementations will use covariant return types to specify the precise type of sequence from the inherited `copy()` method. This is registered for the Prototype pattern, but not for other patterns that use covariant return types for this purpose only.

Sections 7.1.1 – 7.1.3 discuss the observed use of features in more detail. The program listings all represent actual program code, albeit truncated as needed. Several listings represent multiple features, but will be presented in the section deemed most relevant; some cross—referencing is thus required. Table 7.1 and the summaries presented really cannot stand alone. When reading this chapter, the evaluation chapters for each pattern will in all likelihood frequently have to be consulted because of the large amount of information that has to be described: patterns, participants, features, etc. Consulting the JavaDoc is not a bad idea either.

7.1.1. Core Language Features

This section describes the core language features used in the various pattern implementations. Java has many features in common with C++, lacking some, but also provides others not found in C++.

7.1.1.1. Inheritance, Abstract Classes, and Interfaces

The evaluation differentiates between (abstract) class—based inheritance and interface implementation. Standard use of polymorphism and inheritance is not explicitly addressed, as it is fundamental in any OO design. Inheritance is included only if it is part of the core pattern functionality as for example the Template Method pattern; this is also true for interfaces and abstract classes. In our experience, abstract classes have a slightly different purpose in Java compared to similar C++ designs: in Java, abstract classes often implement the basic traits of an interface for convenience while C++ use (abstract) classes for implementation inheritance.

Table 7.1 — Use of Java 6 features in the "Gang of Four" pattern implementations

Pattern / Feature	Abstract Factory	Adapter	Bridge	Builder	Chain of Responsibility	Command	Composite	Decorator	Facade	Factory Method	Flyweight	Interpreter	Iterator	Mediator *(not implemented)*	Memento	Observer	Prototype	Proxy	Singleton	State	Strategy	Template Method	Visitor	*Meta classes*
Core Language Features																								
Inheritance	X		X	X	X	X	X	X		X	X	X	X		X	X			X	X		X	X	X
Abstract classes				X	X	X	X	X		X		X				X				X		X	X	X
Interfaces	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X	E	X	X		X	X
Generics	X	D	D	D	X	D	D	D		X	D	D	X		D	X	X	X	E	D	X	D	X	X
Generics (bounded)	X	D	D	D			D					D	X				X		E		X			X
Packages									X		X													X
Nested classes		M						M								X	M	X	X	X	P	X		X
Anonymous classes		X			X		X			X							X	X			P		X	X
Enumerations							X					X				X			X	X	X			X
Exception handling						X		M				X			X	X		M	X			X		X
Generic methods	X			D		E	X				X	D	X			X	X	X	E				X	X
Covariant return	X			X		X				X		X					X		X			X		X
Varargs				X		X	X					X				X		X				X		X
Reflection																								
Class literals			X	D		X	X				X	X			X	X	X	X	X				X	X
Type literals	X									X							D	D	E				D	X
Constructors	X									X							X	X						X
Methods																X	X	X	X				X	X
Dynamic proxies		M															X	X		X				X
Annotations	X			X						X		X			X	X			X			X	X	X
Special Language Mechanisms																								
Synchronisation			X		E				X	M	X				X			X	X			X		X
Serialization			X												X				D					D
Cloning																	X		X					
Class loader																	X	X	E					X
Weak references					E											X								X
Meta classes		X						X		X			X					X	X		X	X		

X: used directly in a pattern participant; **P**: used as part of another pattern implementation; **M**: used in Meta classes; **D**: derived usage; **E**: used in related, but non—participant, classes; dark—blue squares: high—lights.

Bloch denotes such abstract classes as *skeletal implementations* [Bloch01, p.85]. This kind of usage is because Java does not support multiple (functional) inheritance, and interfaces are the way to define mixin types, but only publicly [Bloch01, p.84]. Forcing clients to inherit a class limits the sub—class from inheriting other classes, which can be problematic. Abstract classes are not used very much in general APIs, such as the Java libraries, unless they represent implementation inheritance reminiscent of C++ or if used in frameworks for call—backs, handlers, etc. The distinction is that frameworks offer a more controlled environment, and using abstract (skeletal) classes allows the default implementation to change (almost) without violating the contract, for example adding a new method [Bloch01, p.88]. Interfaces do not have this advantage. More so, it is not possible to use more than a single (abstract) class as a generic bound, while any number of interfaces can be used. Even more exotically, classes cannot be used to construct dynamic proxies, only interfaces as described in section **7.1.2.4**. On the plus side counts that it is possible to invoke abstract methods from the constructor in Java, which is not possible in C++ because of multiple inheritance [Stroustrup91, p.582]. This is utilised in the Template Method implementation as illustrated in listing **7.13** on page 100. However, this feature has to be used with caution because sub—classes will not have been fully initialised yet [Bloch01, p.80], but in a controlled design like the Template Method, it generally works fine. Furthermore, as explained in section **7.1.2.1** on page 102, advanced reflection techniques can be used to determine if a given method is being invoked as part of the super—class initialisation process or because of ordinary use, which is utilised in the Bridge implementation.

Java and C++ share the access modifiers public, protected, and private, but use them a bit differently. Access modifiers influence inheritance and implementation as they are used to enforce information hiding. Java also supports packages and package private access, which is utilised in the Facade implementation. Compared to C++, packages also influence the use of the protected access modifier. Public and protected members are inherited in both C++ and Java, but C++ allows private implementation inheritance. Very little core pattern functionality described by Gamma et al. depends on multiple inheritance, but Adapter (private) and Observer (public) do. A couple of canonical pattern implementations use friends, such as Memento and State. In Java, composition must be used, perhaps combined with public mixin interfaces, and package private access or inner classes may be able to solve certain issues. The Memento implementation defined in this evaluation rely on public interfaces and delegation, but it also enforces dynamic access checks at runtime to identify the caller as illustrated in listing **7.26** on page 120.

In the evaluation, the abstract `flyweight.AbstractCharacter` class in the Flyweight implementation is just a convenience class that does not define any abstract methods, but simply implements the basic traits of the `flyweight.Character` interface; it is a skeletal implementation. Conversely, in the Command implementation, the `command.SequenceCommand<E>` class supplies needed undo functionality common to all commands based on mementos and defines abstract methods. In the Factory Method implementation, the `factorymethod.CommandCreator<E,T>` is an abstract class that defers the instantiation of the products to the sub—class as described by Gamma et al. [Gamma95, p.107]. This is illustrated in listing **7.1** below.

Listing 7.1 — Abstract classes in Factory Method

```java
public abstract class CommandCreator<E,T> {

  protected CommandCreator() {..}

  public final Command<E> getCommand(Sequence<E> sequence, T token) {
    ..
    try {
      if ((command = this.create(sequence, token)) != null) { // call sub-class implementation
        out.println("Created command: ", token, " -> ", command);
      }
    } catch (Exception e) {
      out.warn("Command creation failed: ", token, " -> ", sequence).warn(e);
      // fall-through
    }
    if (command == null) {
      if ((command = this.createDefault(sequence)) != null) { // call possible sub-class implementation
        ..
      }
      ..
    }
    return command;
  }

  protected abstract Command<E> create(Sequence<E> sequence, T token) throws Exception; // abstract factory method

  protected Command<E> createDefault(Sequence<E> sequence) throws Exception { // hook operation
    return new NullCommand<E>();
  }
  ..
}
```

Whether or not a given abstract class is truly needed is often a judgement call, as is the case with the Builder and Bridge implementations. The use of abstract classes is therefore generally more restricted and interface implementation, object composition, and delegation is often the viable alternative in Java. Bloch claims that interfaces should be preferred to abstract classes [Bloch01, p.84-88], which corresponds well with the Gamma et al. themes from section 2.1.2. The pattern implementations authored in the evaluation try to express this. An example of interface and inheritance usage is supplied in listing 7.2, taken from the Flyweight implementation.

Listing 7.2 — Interface usage in Flyweight

```java
public interface Textual<T> extends CharSequence, Comparable<E> {
  public String lowerCaseFirst(Locale locale);
  public String upperCaseFirst(Locale locale);
  ..
}

public interface Character extends Textual<Character>, Stringable<Character> {..}

public abstract class AbstractCharacter implements Character { // implements basic traits of Character

  protected AbstractCharacter() {..}

  public String lowerCaseFirst(Locale locale) {
    return Strings.lowerCaseFirst(this, locale).toString(); // as Character extends Textual that extends CharSequence
  }
  ..
}

public class Letter     extends AbstractCharacter {..}
public class Symbol     extends AbstractCharacter {..}
public class Whitespace extends AbstractCharacter {..}

public class Word extends ArraySequence<Character> implements Sequence<Character>, Iterable<Character>, Textual<Word> {

  public String lowerCaseFirst(Locale locale) {
    return this.elements[0].lowerCaseFirst(locale); // elements contains letters in this word
  }
  ..
}
```

All pattern implementations but the Singleton and Template Method make explicit use of interface functionality. Template Method has Class scope and relies on inheritance and abstract classes [Gamma95, p.325]. Common interfaces could be implemented by the AbstractClass and ConcreteClass participants, but this would not influence the pattern functionality assuming the abstract operations and hooks were still defined as (abstract) protected methods in the abstract class, and not declared in the interface. Otherwise, the protected methods would become public, which would violate information hiding. An alternative is to break the tight coupling between the AbstractClass and ConcreteClass and make them stand—alone classes that interact through composition or delegation, but interface functionality would thus be required. This is true for all patterns relying on inheritance. Bloch calls this simulated multiple inheritance, because numerous interfaces can be implemented while the work is performed by delegating requests to privately stored objects of proper type [Bloch01, p.87]. Sub—classing an internal Iterator as described by Gamma et al. [Gamma95, p.267-270] corresponds to a degenerate application of the Template Method, but the internal Iterator implementation supplied in the evaluation breaks the tight coupling and uses composition and interfaces instead, to demonstrate the alternative as illustrated in listing 7.3.

Listing 7.3 — Composition as an alternative to abstract classes in Iterator

```
01 public interface ValueProcessor<E> {
02   public void initialise();
03   public boolean process(E value);
04 }
05
06 public class ProcessableSequence<E> extends SequenceDecorator<E> implements Sequence<E> {
07   ..
08
09   public boolean process(ValueProcessor<? super E> processor) { // internal iterator functionality
10     // invariant enforced in constructor
11     IterableSequence<E> sequence = (IterableSequence<E>)this.getSequence(false);
12     sequence.reset();
13     processor.initialise();
14     for (E value : sequence) { // language support for external iterators
15       if (!processor.process(value)) { // forwarding
16         return false;
17       }
18     }
19     return true;
20   }
21   ..
22 }
```

As a side note, listing 7.3 above (line 14) as well as listing 7.11 on page 98 (line 7, 12, and 21) illustrate how elegantly Java 6 supports iteration over user—defined types, corresponding to external Iterator functionality.

The Singleton pattern relies on sub—classing as well since specialisation of singleton types is rare and far between. Defining a common interface was not needed in this evaluation.

Conclusion — As expected, the evaluation shows that interface implementation combined with composition is used in favour of class—based inheritance. Behaviour expressed with multiple and/or private inheritance in the canonical C++ examples by Gamma et al. can be achieved with mixin interfaces, composition, and delegation, though only publicly. The lack of private inheritance in Java actually promotes composition in the pattern implementations where applied.

7.1.1.2. Generic Types and Methods

There are two main uses of generic types in this evaluation, which we have dubbed *static* and *dynamic usage*. Static usage implies compile—time behaviour, inheritance, and/or implementation, while dynamic usage implies a close relationship with runtime behaviour, but neither is exclusive. Static usage is to *implement* or *use* a generic type with a compile–time known type parameter, for example `class SimpsonsFamilySequence implements Sequence<java.lang.String>`, while dynamic usage could be to *instantiate* a generic type with a specific type at runtime, for example `new StatelessSingletonRegistry<Sequence<String>>`, or *use* the generic information in other ways, e.g. class literals. Compile–time bounded types are also considered dynamic usage because the exact type must still be supplied at runtime. The Factory Method implementation employs both types of usage. Consider the abstract `factorymethod.CommandCreator<E,T>` class illustrated in listing **7.1** on page 84. It expects concrete sub–classes via inheritance specify the type of token, `T`, used to identify the concrete `command.Command<E>` types to create, for example a `command.NextCommand<E>` based on a specific instance of `T`. The actual type of `T` is thus available in the factory method implementation in the `CommandCreator<E,T>` sub–class at compile–time, unless the sub–class is generic with respect to `T` as well. However, that makes little sense, as it would make it difficult to use `T` to anything meaningful during the creation process. Below, listing **7.4** illustrates two specific `CommandCreator<E,T>` sub–classes that supply the expected compile–time type of `T`. As the type of `T`, the `factorymethod.SequenceCommandCreator<E>` uses the `meta.model.Sequence.State` type, and line 9 illustrates that the `State` type is available at compile–time as part of the factory method signature to identify the `Command<E>` types to create. At runtime, the type parameter `E` of `SequenceCommandCreator<E>` is bound to the type parameter `E` from `CommandCreator<E,T>`, which is bound to the type parameter `E` from `Command<E>`. This is a dynamic binding that allows the same `SequenceCommandCreator<E>` class to create different types of parameterised commands as illustrated in line 15 and 17. Had it employed a static binding of `E` as well, only commands matching the static binding of `E` could have been created, which is clearly less flexible.

Listing 7.4 — Static and dynamic usage of generics in Factory Method

```
01 public abstract class CommandCreator<E,T> { // E = dynamic, T = dynamic
02   public final Command<E> getCommand(Sequence<E> sequence, T token) {..}
03   protected abstract Command<E> create(Sequence<E> sequence, T token) throws Exception; // abstract factory method
04   ..
05 }
06 
07 public class SequenceCommandCreator<E> extends CommandCreator<E,State> { // E = dynamic, T = static
08 
09   protected Command<E> create(Sequence<E> sequence, State state) throws Exception { // T = known at compile-time
10     if (state == null) {
11       return null; // super
12     }
13     switch (state) {
14       case NORMAL:
15         return new NextCommand<E>(sequence); // E = unknown at compile-time
16       case RESET:
17         return new ResetCommand<E>(sequence); // E = unknown at compile-time
18       default:
19         return null; // super
20     }
21   }
22   ..
23 }
24 
25 public class ReflectiveCommandCreator<E>
26      extends CommandCreator<E,TypeLiteral<? extends Command<E>>> { // E = dynamic, T = static, but dependent of E!
27 
28   protected Command<E> create(Sequence<E> sequence, TypeLiteral<? extends Command<E>> type) throws Exception {..}
29   ..
30 }
```

In the listing above, line 26 shows that the `factorymethod.ReflectiveCommandCreator<E>` class has a dependency between `E` and `T` because the static `T` type is generic with respect to the runtime type of `E`. `T` is represented by a *type literal* representing any sub–class of the generic type `Command<E>`, e.g. `meta.reflect.TypeLiteral<? extends Command<E>>`. This is a powerful way to correlate reflectively created *generic* commands with the actual type of `E` in a type safe manner; something class literals alone cannot achieve because of type erasure. The use of type literals is described in detail in section **7.1.2.1** on page 102, but they themselves depend on static usage to ensure the type information is present at runtime. The Factory Method example illustrate that the choice of static and/or dynamic usage of generics is a design choice, but often a choice that is highly influenced by inter—class relationships. Here, the static usage of `T` allows the same abstract `CommandCreator<E,T>` class to work in a type—safe manner for any type of sub—class, while still allowing runtime parameterisation of `E` for the created products. Static usage of `T` to identify products can be used in any factory implementation, but dynamic usage of `E` is a consequence of the product type being generic, e.g. `Command<E>`.

All but the Facade implementations employ generics, but several patterns do so only because they operate on genetic model types. This corresponds to a **D** legend in table **7.1** on page 82[8]. The patterns are Adapter, Bridge, Builder, Command, Composite, Decorator, Flyweight, Interpreter, Memento, State, and Template Method. In a different evaluation, they could as easily provide implementations that do not use generics. Conversely, Abstract Factory, Chain of Responsibility, Factory Method, Iterator, Observer, Prototype, Proxy, Singleton (registry), Strategy, and Visitor use generic types in relation to the core pattern functionality. In most cases, this coincides with static usage. Comparing generic usage with C++ template usage in the canonical examples, Command, Factory Method, Iterator, and Visitor declare (and use) template types, while Facade, Mediator, and Observer use templates (corresponding to a **D** legend). This illustrates a clear bias towards Behavioural patterns, primarily corresponding to dynamic usage in Java.

The evaluation revealed that static usage works fine in simple cases, but quickly makes things tricky. Consider a scenario where a specific type must be passed as an argument to a method, like the Observer or Memento pattern. Assume such an element is a `meta.model.Sequence<E>` type. An initial attempt could be to supply the specific type as a type parameter, say `memento.MemorizableSequence<S>`, which could then be used directly by the `memento.SequenceMemento<S>` class. But as `Sequence<E>` is generic, there has to be a relationship between `E` and `S`, like `MemorizableSequence<E,S extends MemorizableSequence<E>>` and `SequenceMemento<E,S extends MemorizableSequence<E>>`. Each memorizable sequence type must then implement the `MemorizableSequence<E,S extends MemorizableSequence<E>>` interface supplying its own class as the type parameter `S`. This yields verbose class definitions, especially if even more type parameters are at play. Worse, it makes it hard to use sequence mementos anonymously or via reflection because the explicit sequence types must be known. Using `SequenceMemento<E,?>` will not work because wild—card capture cannot be guaranteed. The Memento implementation, therefore, does not use generics in this fashion, but has to resolve to casting to get the actual sequence types. Its implementation is illustrated in listing **7.26** on page 120. In general, the static usage of generics is therefore kept rather simple in the implementations compared to dynamic usage.

[8] The implementation consistently tries to name type parameters `E` if they are related to the model classes, while names like `T` or `P` are used for (typically static) type parameters unrelated to (dynamic) model classes.

The evaluation shows that dynamic usage is well suited for *container types*, but it is generally not easy to mix generics with reflection because of type erasure. Polymorphism can also be tricky. Still, table 7.1 illustrates that generics were used in most pattern implementations. Behavioural patterns such as Chain of Responsibility, Composite (Command), Interpreter, Iterator, and Observer can all be considered to exhibit container—like behaviour, and Visitor operates on container types, so the use of generics is expected. Generics can help enforce type safety and help pave the way for possible reusable components that can operate on different types. Listing 7.5 illustrates the dynamic usage in Chain of Responsibility. Creational patterns should employ dynamic usage if the products they construct are generic; static usage is already discussed above. As the `meta.model.Sequence<E>` type is the core model object used, it is understandable that Abstract Factory, Builder, Factory Method, and Prototype are generic as they use this or a derived type, such as the `command.Command<E>` type. Listing 7.1 in the previous section illustrates how the `factorymethod.CommandCreator<E,T>` class creates generic `Command<E>` types. Singleton is also creational, but much more static and class—centric than the other Creational patterns. In our experience, generics are rarely used in singleton classes; this is also true in this evaluation. The only use of generics in the Singleton implementation is the Singleton Registry. Structural patterns such as Adapter, Decorator, Facade, and Proxy should employ dynamic usage if the original types are generic, which – again – is the case here. Template Method is Behavioural, but not a container type unless specifically designed to be so, as for example an internal Iterator. The implementation here uses Template Method to construct `Sequence<E>` types delivering complex value types and is reminiscent of Structural patterns such as Decorator.

Listing 7.5 — Dynamic usage of generics in Chain of Responsibility

```java
public interface Handler<R> { // R = runtime type of request
  public Handler<R> handle(R request, HandlerLink<R> link);
}

public interface HandlerLink<R> {
  public Handler<R> forward(R request, Handler<R> current);
}

public interface HandlerChain<R> extends Handler<R>, HandlerLink<R> {
  public void register(Handler<R> handler);
  public Handler<R> handle(R request);
  ..
}

public abstract class AbstractHandlerChain<R> implements HandlerChain<R> { // container-like functionality
  ..
  public Handler<R> handle(R request) {
    return this.forward(request, (Handler<R>)null); // from the start...
  }

  public Handler<R> forward(R request, Handler<R> current) {
    Log out = LogFactory.getLog(this);
    for (Handler<R> handler : this.getHandlers(current)) { // fetch handlers after "current"
      Handler<R> h = handler.handle(request, null); // null link = no recurse into this chain
      if (h != null) {
        out.println("Handled request ", ObjectPolicy.ID.toString(this), ": ", request, " -> ", handler);
        return h;
      }
    }
    out.println("Could not handle request ", ObjectPolicy.ID.toString(this), ": ", request);
    return null;
  }

  public Handler<R> handle(R request, HandlerLink<R> link) {
    Handler<R> handler = this.handle(request);
    if (handler != null) {
      return handler;
    }
    return link == null ? null : link.forward(request, this);
  }
  ..
}
```

Generic usage, especially combined with bounds on the type parameters, can also cause problems. Class literals cannot describe a generic type, only *raw types*. Erasure may also cause problems with unchecked casts. Standard use of polymorphism for raw types do not apply to generics [Sierra06, p.582-597]: while it is true that `java.lang.Integer` is a sub—class of `java.lang.Number` and `java.util.ArrayList` is an implementation of `java.util.List`, for example, `ArrayList<Integer>` is **not** a sub—type of `List<Number>`. Bounded wild—cards have to be used for polymorphism of type parameters, as in `Expression<? extends Number>`. This is illustrated in listing **7.4** on page 86, where the `? extends Command<E>` bound in line 26 ensures that any sub—type of `command.Command<E>` can be created by the factory. As the Interpreter implementation uses generic expression types, this presents certain problems. A method to replace a given expression in an expression sub—tree as described by Gamma et al. [Gamma95, p.251-255] is not possible in the developed implementation because wild—cards have to be used. Wild—cards cannot be used in expression creation because a concrete type must be known, but there is no way to determine a common type for all type parameters. The Chain of Responsibility implementation originally used a `? super R` bound instead of just `R` on the set of contained handlers in a given handler chain. This proved too cumbersome because wild—card capture was not possible on ordinary use. On the other hand, generic bounds are vital for correct pattern functionality in several implementations, such as Abstract Factory, Prototype, and Interpreter.

Implementations can use generic types without using generic methods, which is illustrated by the Chain of Responsibility implementation. On the other hand, it is possible that generic types augment the use of generic methods. Generic methods can still be used unrelated to specific generic types. The Flyweight implementation is an example where generic methods are used to look up non—generic flyweight types based on class literals, while the Abstract Factory as illustrated in listing **7.17** on page 108 uses bounded generic methods to enforce type safety for contained prototypes (line 32 + 39). The Builder implementation is an example of a close relationship between generic types and generic methods. The Singleton and Visitor implementations define interfaces declaring generic methods that are clearly related to static usage of generic types. This is illustrated by the generic `visitor.SequenceValueVisitor<P>` type, where `P` identifies the type of argument to supply to the visitors. The `visitor.ValueVisitableSequence.accept(SequenceValueVisitor<P>, P)` method must be generic in order to handle any type of `P`. This is illustrated in listing **7.20** on page 110.

Conclusion — Static usage of generics ensures compile—time type safety useful for especially Creational patterns because the creation process can utilise the knowledge as illustrated by Factory Method. Structural patterns rely heavily on object composition and can use generics to ensure proper cooperating types, including usage of generic bounds for greater flexibility as illustrated by Bridge. This is also true for Behavioural patterns, especially container—like patterns like Observer and Chain of Responsibility. Both static and dynamic usage is useful. The choice to use which, or both, is a design choice that depends on the type of objects manipulated by the patterns, which is also illustrated in the "Gang of Four" template usage. Generics pave the way for possible pattern componentizations because the same components can operate on different types in a type safe manner, especially if aided by generic methods and bounds.

7.1.1.3. Nested and Anonymous Classes

There are four kinds of nested classes in Java: *static member* classes, *non—static member* classes, *local* classes, and *anonymous* classes [Bloch01, p.91]. Nested classes in table 7.1 refer to any nested class that is *named*. This corresponds to one of the first three types. An anonymous class is a class without a name. They are by design nested classes as they are always defined within the scope of another class, but they are not registered as such in table 7.1. The distinction is made because named classes are reusable, while anonymous classes act as unique *closures*. They are declared and instantiated at the same time [Bloch01, p.93]. Non—static, local, and anonymous classes are also called *inner classes* as they share state with the enclosing class. In C++ terminology, they can be considered a (localised) friend class [Stroustrup91, p.566-568], but in C++ nested and local classes follow the standard C++ access rules and have no special access to state of the enclosing class or function, respectively [Stroustrup91, p.551-553]. Inner classes are in Java often used to apply the Adapter pattern [Bloch01, p.92]. Java does not have function pointers, or *callbacks*, like C++ because the functionality can be achieved using object references: a *function object* in Java is a stateless instance of a class that export a single method performing operations on other objects [Bloch01, p.115]. Nested classes are often used to create function objects and Java has in this sense direct support for the Strategy pattern. Anonymous classes acting as closures are sometimes referred to as *process objects* [Bloch01, p.93]. The canonical C++ implementations do not utilise nested or local classes.

Adapter, Decorator, Observer, and Template Method use non—static member classes. Adaptation using non—static member classes is used in the `meta.reflect.proxy.ProxyFactory` class that will adapt any proxy to conform to the private `ProxyFactory.Proxy` interface, which will allow retrieval of the proxy factory *instance* that created the proxy. The `ProxyFactory` class uses the `ProxyFactory.Handler` class to decorate all invocation handlers used by dynamic proxies to achieve a form of method combination (see listing 7.10 on page 96). Observer uses non—static member classes for adaptation of the `observer.BirthdayRegistry` class to the `observer.SequenceObserver<A>` interface without exposing this information externally for misuse. As noted by Bloch, this is common use in Java [Bloch01, p.92]. This is illustrated in listing 7.6 below. Template Method use non—static member classes as *protection proxies* to guard access to delivered sequence values (see also section 7.1.2.4).

Listing 7.6 — Inner classes used for adaptation in Observer

```java
public class BirthdayRegistry { // does not implement SequenceObserver

  private final class RegistryObserver<A> implements SequenceObserver<A> { // inner class acts as reusable adapter class

    public void sequenceEvent(Sequence<?> sequence, A aspect) {
      Object current = sequence.current();
      if (current instanceof Date) {
        if (BirthdayRegistry.this.size() > 0) {
          BirthdayRegistry.this.checkDate(getFullBirthday((Date)current));
        }
      }
    }
    ..
  }
  ..

  // registration method
  public <A> boolean register(AspectObservableSequence<SequenceObserver<A>,A,Date> sequence, A... aspects) {
    RegistryObserver<A> ro = new RegistryObserver<A>(); // implicit reference to enclosing BirthdayRegistry instance
    ..
    boolean added = false;
    for (A aspect : aspects) {
      added = sequence.addObserver(ro, aspect) || added; // may contain null as well = all
```

Listing 7.6 — Inner classes used for adaptation in Observer

```
    }
    return added;
  }

  private void checkDate(Date date) {..} // actual notification method
  ..
}
```

The Adapter implementation in the `adapter` package uses the Strategy pattern to choose an adaptation strategy in form of the `adapter.AdapterDelegate<S,T>` type. A set of reusable adapter strategies is defined in the `adapter.AdapterStrategy` class, but others can naturally be defined as needed. As illustrated in listing **7.7** below, each concrete strategy is an instance of a stateless anonymous inner class with a distinct type, thus acting as a function object. If all concrete strategy types are identical, enumeration constants may be an alternative to using anonymous inner classes. The actual Strategy implementation in the `strategy` package uses enumerations for concrete strategies in form of the `strategy.SequencePolicy` and `strategy.ObjectPolicy` enumerations (Policy is a synonym for Strategy [Gamma95, p.315]). Listing **7.21** on page 112 illustrates the use of anonymous local classes in the Prototype implementation (line 17-25).

Listing 7.7 — Anonymous classes used as concrete strategies in Adapter

```
public class AdapterStrategy {

  public final static AdapterDelegate<Number,String> NUMBER_TO_STRING = // concrete strategy
    new AdapterDelegate<Number,String>() { // function object (+ toString)
      public String adapt(Number value) {
        return value == null ? null : value.toString();
      }
      public String toString() {
        return "Number->String";
      }
    };

  public final static AdapterDelegate<CharSequence,Long> CHAR_TO_LONG = // concrete strategy
    new AdapterDelegate<CharSequence,Long>() { // function object (+ toString)
      public Long adapt(CharSequence value) {
        return value == null ? null : Long.valueOf(value.toString());
      }
      public String toString() {
        return "CharSequence->Long";
      }
    };

  ..
}
```

The evaluation shows that nested classes can promote advanced functionality not directly related to the "Gang of Four" patterns, but which can make several patterns more flexible and robust. The State implementation uses a public static nested class as factory to create `state.StepSequence` instances, which are actually dynamic proxies. The Singleton implementation uses private static member classes to allow for thread—safe lazy initialisation of the Singleton instances without enforcing explicit synchronisation and resulting overhead. The technique is well—known and called the *Initialise—on—demand holder class* idiom [Bloch01, p.194]. It is illustrated in listing **7.15** in section **7.1.1.5**. A nested static class must be used because inner classes cannot declare static attributes. Nested static classes in Java make lazy initialisation of singletons easy and is in fact the best way to do so without requiring synchronisation on each call to the static singleton method, as the Double—Checked Locking Optimisation [Schmidt00, p.353] pattern for lazy initialisation is broken in Java prior to version 5 [Bacon; Bloch01, p.193]. The pattern can be applied in Java 5 and 6, though [Bacon; Bloch08, p.283].

The Prototype and Proxy implementations use anonymous (method—) local classes to construct on—the—fly adaptation of invocation handlers to use for dynamic proxies. Like normal inner classes, local classes have access to private members of the enclosing class object, but they also act as *closures*, keeping references to required variables supplied as final arguments to the enclosing method. Even more so, because private state is required for proxy functionality, the `proxy.SequenceProxyFactory` class declares a named method—local class.

Anonymous classes also form the base of type literals as described in section **7.1.2.2** on page 107. They allow dynamic usage of generics to become static via inheritance. An anonymous class will inherit an abstract generic super—class (e.g. type literal) and instantiation requires an actual type, which will be expressed through inheritance. The type information can therefore be used reflectively as it is not erased.

Conclusion — Much of the Prototype and Proxy functionality implemented would practically have been less feasible without inner classes acting as closures. Patterns, e.g. Observer and Template Method, can be implemented without the use of inner classes, but yielding a less elegant and more verbose design. Nested, including anonymous, classes possess several qualities that mix well with the "Gang of Four" patterns and can aid in at least encapsulation, information hiding, adaptation, decoration, and delegation.

7.1.1.4. Enumerations

Enumerations were added to Java as of version 5. Each enumeration is a class, but each constant in the enumeration is also a class and can implement or override functionality required by that constant alone [Gosling05, p.256]. This is used to implement and name different traversal strategies for the Composite implementation, for example `composite.CompositeStrategy.DEPTH_FIRST`. Enumerations are also used to implement compile—time known strategies in the Strategy and Observer implementations.

However, enumerations have several semantic rules that set them apart from normal classes [Gosling05, p.249], the most obvious being that it is a compile—time error to explicitly instantiate an enumeration type. Each instance, e.g. constant, is known and fixed at compile—time, but it also means that comparison of for example traversal strategies is easy as it can be done on identity. Enumerations inherit the `java.lang.Enum<E>` class, and cannot inherit other classes (single inheritance). They are furthermore guaranteed final; not cloneable; serializable; and reflective instantiation of enumeration types is disallowed. The static nature of enumerations is applied to ensure that all internal states are non—instantiable in the State pattern implementation; they are stateless and all operate on a given delegate sequence instance supplied as an argument. This is illustrated in listing **7.8** below. Since defined as an inner enumeration, the states have access to all sequence attributes as in line 36-38. As each constant is named, calling other states is easy as in line 15. Usage of enumerations also allows `switch` statements, which is very useful when modelling states (line 10-11, for example). Each unique state also defines its own functionality similar to traversal strategies described above. Private nested classes cannot offer the same guarantee as enumerations as even private classes with private constructors can be instantiated from the enclosing class, or externally via reflection if the security manager allows it. On the other hand, states defined as enumerations cannot be sub—classed, but that is no different from private inner classes.

Listing 7.8 — Enumeration constants as states in State

```java
public interface FunctionalState<E> {
  public E action(State internalState, StateableSequence<E> sequence);
}

public class ReversiblePrimeSequence extends AbstractStateableSequence<Integer> implements ReversibleSequence<Integer> {

  private static enum PrimeState implements FunctionalState<Integer> {
    Start { // concrete state
      protected void action(State state, ReversiblePrimeSequence sequence) { // State class = enum
        switch (state) {
          case START: {
            if (sequence.maximum < 2) throw new IllegalStateException("Maximum number must be > 2");
            sequence.calculate = (sequence.maximum > 3);
            ..
            ResetLow.action(state, sequence); break; // initialize, but keep state
          }
          case NORMAL: {
            sequence.setFunctionalState(sequence.calculate ? Calculate : Next, state); break;
          }
          ..
          default: throw new IllegalStateException(state.name()); // should never happen
        }
      }
    },
    ResetLow { // concrete state
      protected void action(State state, ReversiblePrimeSequence sequence) {
        switch (state) {
          case NORMAL: {
            sequence.setFunctionalState(sequence.calculate ? Calculate : Next, state); break;
          }
          case START: {..}
          case RESTART: {
            .. // fall-through!
          }
          case RESET: {
            sequence.state = state;
            sequence.index = 3;
            sequence.prime = 2;
          }
        }
      }
    },
    ResetHigh {..}, // concrete state
    Calculate {..}, // concrete state
    Next      {..}, // concrete state
    Previous  {..}; // concrete state

    public final Integer action(State internalState, StateableSequence<Integer> sequence) { // delegation
      ReversiblePrimeSequence rps = (ReversiblePrimeSequence)sequence;
      if (internalState != null) this.action(internalState, rps);
      return rps.prime;
    }
    protected abstract void action(State internalState, ReversiblePrimeSequence sequence);
  }

  public ReversiblePrimeSequence(int maximum) {..}

  public Integer reverse() {
    FunctionalState<Integer> fs = this.getFunctionalState();
    if (fs == PrimeState.Next) // identity
      this.setFunctionalState(PrimeState.Previous, State.NORMAL);
    } else if (fs == PrimeState.Previous) {
      this.setFunctionalState(PrimeState.Next, State.NORMAL);
    } else if (..) {
      ..
    }
    ..
  }
  ..
}
```

The State implementation led us to the novel, or at least one we have not seen before, and yet so simple approach of creating a singleton type as an enumeration with only a single constant; the singleton instance. We call this the *Singleton—as—Single—Constant* idiom. However, as enumerations cannot extend other classes, more work on part of the developer may have to be required. This means that such singleton types cannot be sub—classed. Functionality from abstract classes that would otherwise have been inherited must be used via composition. It is not possible to supply arguments to the singleton constructor, unless dubious design decisions allow usage of system properties, JNDI, and the likes, to determine the singleton behaviour. On the other hand, much Singleton behaviour is enforced by Java directly, for example non—instantiable, global point of access, object identity, etc. The `singleton.DanishAlphabetSequence` is a Singleton implementation using enumerations as illustrated in listing **7.9** below. The singleton instance is created and/or acquired using the single enumeration constant `DanishAlphabetSequence.Instance`. The instance is not loaded until the constant is first referenced. The actual sequence functionality is represented by a final private `meta.model.Sequence<java.lang.String>` aggregate member that stores the letters in the Danish alphabet. However, Singleton implementations using enumerations cannot be sub—classed, nor be generic.

Listing 7.9 — *Singleton—as—Single—Constant* idiom used to implement Singleton

```java
public enum DanishAlphabetSequence implements Sequence<String> { // extends java.lang.Enum<DanishAlphabetSequence>

  Instance; // single constant

  private final Sequence<String> sequence = new ArraySequence<String>( // aggregatee (adapter)
    "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m",
    "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z",
    "æ", "ø", "å") {

      public Sequence<String> copy() {..}
  };

  private DanishAlphabetSequence() {..} // private enumeration constructor required by Java

  public String next() {
    return this.sequence.next(); // forward call to aggregatee
  }
  ..
}
```

The use of enumerations in the evaluation is for all pattern implementations associated with interface implementation except for the Interpreter pattern as well as usage in Meta classes. The enumeration constants are mainly used for implementation expediency, not to define a separate type, save the Singleton and State patterns. Their usage is not paramount for any pattern implementation. This indicates that normal classes could have been used instead, but perhaps requiring more work. The reverse is not true: enumerations cannot be used if inheritance must be used, and each constant has to have the same type. The Adapter implementation therefore has to use anonymous inner classes in the `adapter.AdapterStrategy` class, because each strategy has a different (generic) type. Examples of regular usage of enumerations are the `meta.util.Primitive` Meta enumeration and the annotations used to describe the "Gang of Four" participants, for example the `meta.Scope` enumeration. These enumerations are used to define both a type and a fixed set of constants.

Conclusion — Enumerations proved surprisingly useful in the specialised cases where a few instances of a given type are required because of their unique behaviour guaranteed by the compiler. Patterns having a static structure will benefit most from their usage. They were used continually to represent Strategy implementations, but are also limited in conjunction with generic types. Most significantly, enumerations illustrated that Java has built—in support for Singleton pattern.

7.1.1.5. Exception Handling

The evaluation shows that exceptions and *try—catch—finally* blocks in Java can be used in the "Gang of Four" implementations in a number of ways. The features have a very versatile use and are embedded in several different pattern implementations for different purposes as summarised below.

C++ supports the *try—catch* idiom, while Java supports *try—catch*, *try—finally*, and *try—catch—finally*

[Gosling05, p.396-401]. Unlike C++, Java supports checked as well as unchecked exceptions [Gosling05, p.299]. The "Gang of Four" canonical implementations do not use error or exception handling in any way, but that is a deliberate choice to simplify the examples. The use of checked exceptions augments the intent of the declaring method. From a design point of view, they signal intent, much like annotations. Checked exceptions are used in the evaluation when the caller can reasonably be expected to recover [Bloch01, p.172]. Gamma et al. do not use exceptions, nor address error handling in general, except certain more or less obscure scenarios using Smalltalk's `doesNotUnderstand` mechanism, for example in the Chain of Responsibility pattern [Gamma95, p.229]. The lack of error handling in the "Gang of Four" patterns is – of course – a deliberate omission because of clarity, space, time, money, etc.

Though exception handling is frequent in most Java programs, including the pattern implementations in this evaluation, few specific exception types have been declared. Command, Interpreter, Memento, Singleton, and Template Method declare specific exception types. All except the `singleton.SingletonException` type are used to signal a type of ***execution error***, for example a `command.CommandException` to signal command execution failure, and `memento.MemorizableException` to signal an inapplicable memento state, or an `interpreter.ExpressionException` to signal the evaluation failure. Such errors are all data centric and the context must be prepared for errors caused by invalid data. The use of the `ExpressionException` type is illustrated in listing **7.14** on page 103; some errors are recoverable (even intentional), while others are not. However, the pattern implementations **must** ensure *failure atomicity* to prevent illegal internal states of pattern participants in case of errors [Bloch01, p.185]. This is especially true for Memento as it goes to core functionality of it: updating only part of the internal state of Originator participant can have potentially disastrous effects. Enforcing failure atomicity is illustrated in listing **7.26** on page 120.

The `SingletonException` is runtime because ordinary use of singleton registries is assumed correct by the context using them. The context, for example, should never test for a null return value from a registry, because either the singleton instance exist or it does not; a null value does not make any sense, and would constitute a poor design decision. Hence, in the general case, the context should not have to worry about checked exceptions, as they would never be thrown. In Template Method, a `SequenceValueException` represents a sequence value that is null or cannot be delivered by a primitive operation; again, something that should never occur and therefore declared as runtime. That said, the evaluation shows that the pattern implementations frequently employ *exception chaining* by transforming a checked exception to a runtime exception to comply with Java's *handle or declare* mantra [Bloch01, p.178-180]. This is trivial and easily done with a *try—catch* (and re—throw) block.

The Interpreter implementation can use a specific type of exception to transfer control to a new expression to be evaluated, thereby discarding the current evaluation stack while reusing existing variables and constants. This is indeed a crude way to transfer control, but quite effective for small languages. Java's built—in *try—catch—finally* idiom is in general very useful for manipulating exceptions, but especially so in a case like this. As the generic expressions are manipulated, class literals have to be used to supply the expression type at runtime.
The *try—finally* or *try—catch—finally* idioms are very useful for method combination and sub—class hooks. The

meta.reflect.proxy.ProxyFactory.Handler class, which is a Decorator, uses a finally block to perform method combination: it times (and logs) each invocation of the inner method, even if the inner method fails; this is illustrated in listing 7.10 below. The Template Method use finally blocks to ensure a proper state after primitive operations and hooks have been executed in the sub–class. In the Command implementation, the command.CommandProcessor class uses finally blocks to ensure logging of spawned commands is always performed. In Factory Method, the factorymethod.CommandCreator<E,T> class uses a *try–catch* block to ensure a default command.Command<E> is created in case the sub–class creation fails; this is illustrated in listing 7.1 on page 84. The Observer implementation uses an error handler strategy in a catch block to delegate the error handling elsewhere.

Listing 7.10 — *try–catch–finally* idiom used for method combination in Proxy

```java
public class ProxyFactory {

  private class Handler implements InvocationHandler { // decorator
    ..
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
      ..
      Log out = LogFactory.getLog(InvocationHandler.class).println(id, " -> execute start: ", method);
      long start = System.nanoTime();
      try {
        return this.handler.invoke(proxy, method, args); // call method in decorated handler

      } catch (Throwable t) {
        out.warn(id, " -> execute exception: ").warn(t);
        throw t; // will also trigger finally block

      } finally { // always executed, even in case of errors
        long time = System.nanoTime() - start;
        out.println(id, " -> execute end: ", method, ": ", time, " ns");
      }
    }
    ..
  }
  ..
}
```

Exceptions are objects and can be manipulated (almost) like any other object. Exception types cannot be generic, and cannot inherit any other class than a sub–class of java.lang.Throwable. By *creating* exceptions without *throwing* them a stack trace representing the current execution context can be acquired. Combining this with class literals and reflection, the java.lang.Class<T> instance representing the caller can be acquired. This is exploited by the logger, which uses the information to log the member name and line number from the source code file. More importantly, the Memento and Singleton implementations rely on exception stack traces to protect access to methods that are only allowed to be invoked by "friends", such as a sub–class constructor in Singleton. We refer to the Singleton and Memento types as *guarded types*. Unlike C++, the access check has to be made at runtime. The usage is closely related to class literals and thus explained in section 7.1.2.1 on page 102.

Conclusion — Java's support for exceptions and exception handling is used extensively throughout the pattern implementations to support pattern invariants in one way or another. Instantiation protocols and method combination are implemented using these features. Exception stack traces are also useful for applying additional simple access checks to the pattern implementations, such as identifying the caller, but the problem is that the security it provides is runtime, not compile–time. If it fails at runtime, the solution may be compromised.

7.1.1.6. Covariant Return Types

While it has always been possible to return a polymorphic type prior to Java 5, covariant return types now allow an overriding method to formally narrow the return type at compile—time by changing the method signature [Gosling05, p.220-221]. Abstract Factory, Builder, Command, Factory Method, Interpreter, Prototype, Singleton, and Template Method use covariant return types. The evaluation shows that *factory—like* methods can benefit from covariant return types, for example Prototype as the client statically has access to the precise type, but also that careful considerations must be made before changing the participant types. Several implementations narrow the return type to a specific *class* in favour of an abstract *type* (interface), for example Abstract Factory. In some cases, it is not possible to return an interface because no interface is available. However, care must be taken to ensure that the themes described by Gamma et al. are adhered to, here especially programming to an interface as opposed to the implementation (section **2.1.2**). If covariant return types are used *internally* in a participant, as for example in the Template Method implementation, using actual classes is not a problem. It is when the public signature is changed problems may arise. Finally, covariant return types may augment overloading as the compile—time type is known. In theory, for example, this could aid Visitor to allow all visitation methods to be named `visit(..)` if the exact type of visited objects are known at compile—time. Note, however, that while C++ supports covariant return types for virtual functions [Stroustrup91, p.647], this is not utilised in a single canonical pattern implementation by Gamma et al. as far as we can tell.

The Abstract Factory implementation use covariant return types to specify the precise type of created objects where meaningful and if possible. This is *external* use, but important because knowing that a factory creates `bridge.MemorizableSequenceAbstraction<E>` objects, for example, ensures that the Memento functionality is visible. In Factory Method, the `factorymethod.CommandCreator<E,T>` abstract class allows default creation of commands in case regular creation fails. Sub—classes can override the default creation hook, or the creation method itself for that matter, to specify the exact created type. This is internal usage. The `meta.reflect.CallerClass` and its sub—class `meta.reflect.Caller` utilise covariant return types when returning the caller of a given execution context: `CallerClass.getCaller()` returns a `CallerClass` type, while `Caller.getCaller()` returns a `Caller` type. The `getCaller()` method can be seen as a Factory Method for public usage, but there is no proper interface to return, so a class is returned. Template Methods often utilise several primitive and/or hook operations that it may be possible to specialise, as is the case in the `templatemethod.ZipSequence` class. This allows both template, primitive, and hook operations to internally use the precise types without casting, which allows for compile—time safety in the implementation.

Strict prototypical types will in their `copy()` declaration or implementation specify the actual implementing type as the return type, and sub—classes of the given type will refine the type even further to a specific class. The `prototype.Copyable<T>` interface offers a more lenient contract compared to returning the precise type as required by the `prototype.StrictCopyable<T>` interface. For example, `command.Command<E>` implements `StrictCopyable<Command<E>>` and the method signature becomes `Command<E> copy()`. The `command.CompositeCommand<E>` class implements `Command<E>`, but declares that it returns a `CompositeCommand<E>` from its `copy()` method, not just any `Command<E>` type. After copy, for example, the client can access the `getCommands()` method without having to cast, but `CompositeCommand<E>` is a concrete class. This is similar to Abstract Factory, and is illustrated in listing 7.11 below. The `meta.model.ReversibleSequence<E>` interface extends `meta.model.Sequence<E>` and refines its `copy()` method to return a `ReversibleSequence<E>` type, but actual reversible sequences will narrow it even further to the concrete class.

Listing 7.11 — Covariant return types for prototypical and composite behaviour in Command

```java
public interface Command<E> extends StrictCopyable<Command<E>> { // prototype

  public Command<E> copy();
  ..
}

public class CompositeCommand<E> implements Command<E>, Iterable<Command<E>> { // composite
  ..

  public CompositeCommand(CompositeCommand<E> command) { // copy constructor
    this();
    for (Command<E> c : command) { // language support for iteration
      this.addCommand(c.copy());   // copy contained command
    }
  }

  public CompositeCommand<E> copy() { // covariant return type
    return new CompositeCommand<E>(this);
  }

  public Iterator<Command<E>> iterator() {..}

  public List<Command<E>> getCommands() {..} // this.copy().getCommands() compiles

  public boolean addCommand(Command<E> command) {..}
  ..
}
```

The Singleton implementation is interesting because it utilises a covariant return type on a redefined static method. Static methods cannot be overridden in Java, but they can be redefined, or *hidden*, unless declared final [Sierra06, p.147]. The `singleton.SimpsonsFamilySequence` class is a singleton type that allow sub–classing. The singleton `SimpsonsFamilySequence` instance is acquired using the static `getFamily()` method. The `singleton.SimpsonsAndBouvierFamilySequence` class is a sub–class of `SimpsonsFamilySequence` and defines a static `getFamily()` method as well, but it returns the `SimpsonsAndBouvierFamilySequence` type, e.g. instance. This is more flexible that what the canonical "Gang of Four" implementations can offer in C++.

The Builder implementation is an example of some of the problems that may be involved when using covariant return types in the design. The `builder.ExpressionBuilder<E>` builds `interpreter.Expression<E>` types, while the `builder.TypedExpressionBuilder<E>` builds `interpreter.TypedExpression<E>` types, a sub–type of `Expression<E>`. `TypedExpressionBuilder<E>` uses an aggregate builder to perform the actual construction, where after the constructed expressions are enriched (decorated) with the required type information. `ExpressionBuilder<E>` returns the `Expression<E>` interface from most build methods, which shields the client from knowing the precise expression types. However, certain expression types must be known to utilise their functionality, such as the `interpreter.FlowExpression<E>` used to build composite expressions. To treat composites and non–composites alike, Gamma et al. suggest definining a method in the Component participant to return the composite (`this` or null) [Gamma95, p.168]. Here, `Expression<E>` is the component and `FlowExpression<E>` the composite. However, this is not a viable solution, as other specific types of expressions must sometimes be known; defining specific methods to acquire each type is not an option.

So, how can `TypedExpressionBuilder<E>` narrow the return type for `FlowExpression<E>` without loosing any type information? Inheritance, dynamic proxies, and/or decoration/adaptation must be used. Dynamic proxies can only be used if an interface is used. Decoration and adaptation can only be used if a non—final type is known. Inheritance only if a concrete class is known, regardless if an interface is known. As `FlowExpression<E>` is concrete, but not final, we have to make a decorator wrapping the created `FlowExpression<E>`, overriding all methods, and adapting it to the `TypedExpression<E>` interface. This is tedious at best, but cannot be avoided. This type of solution is illustrated in listing 7.12 below. In the general case, it cannot be guaranteed to work because the type to be narrowed can be either an interface with no accessible implementation or a (final) class with no interface. On the plus side, examples such as this illustrate how other patterns can aid in a possible solution, particularly Decorator and Adapter, but also Composite.

Listing 7.12 — Decoration and adaptation required for covariant return types in Builder

```
public interface Expression<E> extends StrictCopyable<Expression<E>> {..}

public class FlowExpression<E> extends AbstractExpression<E> // concrete composite type
  implements NonTerminalExpression<E>, InitialisableExpression<E> {
  public FlowExpression<E> add(Expression<? extends E> expression) throws ExpressionException {..}
  ..
}

public interface TypedExpression<E> extends Expression<E> {..}

public class TypedFlowExpression<E> extends FlowExpression<E> implements TypedExpression<E> {..} // decorator/adapter

public interface ExpressionBuilder<E> extends StrictCopyable<ExpressionBuilder<E>> {
  public FlowExpression<E> buildFlowExpression(); // concrete return type required
  public Expression<Boolean> buildEqualExpression(Expression<?> first, Expression<?> second);
  ..
}

public class TypedExpressionBuilder<E> extends AbstractExpressionBuilder<E> implements ExpressionBuilder<E> {
  protected final Class<E> type;
  protected final ExpressionBuilder<E> builder; // aggregate builder
  ..

  protected final static <T> TypedExpression<T> getTypedExpression(Expression<T> expression, Class<T> type) {..}

  public TypedFlowExpression<E> buildFlowExpression() { // specific concrete type required as not to loose information
    FlowExpression<E> fe = this.builder.buildFlowExpression();
    if (fe instanceof TypedFlowExpression) {
      return (TypedFlowExpression<E>)fe;
    }
    return new TypedFlowExpression<E>(this.type, fe);
  }

  public TypedExpression<Boolean> buildEqualExpression(Expression<?> first, Expression<?> second) { // simple covariant
    return getTypedExpression(this.builder.buildEqualExpression(first, second), Boolean.class);
  }
  ..
}
```

Conclusion — Covariant return types can eliminate the need for casts in several pattern implementations, which boosts the overall type safety, but also the pattern intent. Publicly, the feature must be used wisely because it changes the static structure of the pattern participant(s) and may cause access to implementations rather than interfaces. The evaluation shows that covariant return types are especially useful combined with *static usage* of generics (see section 7.1.1.2 on page 85) or with class—based inheritance. Creational patterns seem to benefit most from this feature in this evaluation because of their *factory—like* behaviour.

7.1.1.7. Varargs

Like enumerations, varargs[9] were added as of Java 5 [Sierra06, p.46]. C++ supports varargs using the functionality in the `<cstdarg>` (or the C `<stdarg.h>`) header, but this is not utilised in the canonical examples. A reason could be that no header files are used at all in the relatively simple examples. In Java, varargs are used for Builder, Command, Composite, Interpreter, Observer, Proxy, and Template Method. The usage is for all except the last two related to the *container—like* behaviour of pivotal pattern participants, such as the `command.CompositeCommand<E>` or `observer.ObserverManager` classes. By using varargs, the lazy developer does not have to construct a temporary array or collection to add to a given container to supply the actual elements. This type of usage can always be replaced by use of arrays or collections in local program code. The use of varargs may also cause compiler warnings if combined with generics, because it is not possible to create a generic array in Java. This is a clear indication of the close connection between varargs and arrays.

Varargs is used extensively in the Java reflection API. Because of this, several Meta classes utilise varargs as well, for example `meta.reflect.Reflection` and `meta.reflect.proxy.ProxyFactory`. Consequently, the Proxy implementation also uses varargs. As illustrated in several program listings in this chapter, the `meta.log.Log` implementation uses varargs to avoid string concatenation. However, the Template Method implementation is perhaps the finest example of varargs usage: the `templatemethod.SequenceTemplate<K,E>` abstract class permits sub—classes to supply any number of arguments that it will forward to the initialisation hook in the sub—class itself. This is illustrated in listing **7.13**. C++ does not allow abstract (pure virtual) methods to be invoked from a constructor, so such a scenario is not possible in C++ [Stroustrup91, p.582]. However, this feature has to be used with caution because sub—class constructors will not have been initialised yet [Bloch01, p.80], but it has nothing to do with varargs in general. In a controlled design like the Template Method implementation, it works fine and makes the Template Method implementation very flexible.

Listing 7.13 — Varargs usage in Template Method

```java
public abstract class SequenceTemplate<K,E> extends AbstractSequence<E> implements Closeable {
  ..
  protected SequenceTemplate(Object... arguments) throws SequenceValueException {
    ..
    this.closed = !this.sequenceOpen(arguments); // perform sub-class specific sequence initialisation
    ..
  }

  protected abstract boolean sequenceOpen(Object... arguments) throws Exception; // abstract operation

  protected void sequenceClose() throws Exception {..} // hook operation with default behaviour
  ..
}
```

Conclusion — Varargs can be useful for patterns that need to forward arguments to sub—classes, delegates, and/or aggregates. They can also be useful for container—like patterns when adding elements, but may generate compiler warnings in situations involving generics. They do not offer anything new as arrays can achieve the same result.

[9] Formally, a function that accepts a variable number of arguments, i.e. has variable *arity*, is a *variadic function*. Varargs in Java is basically syntactic sugar, albeit sweet tasting in our opinion, for arrays.

7.1.1.8. Summary

C++ and Java 6 share many static features, but the canonical pattern examples do not express several of the features examined in this evaluation. Still, we have illustrated that features such as exceptions and error handling, covariant return types, and varargs may be useful to augment pattern functionality as shown in table 7.1 on page 82. Narrowing the return type is closely related to inheritance and implementation. It is used in around a third of the pattern implementations. Varargs is useful for Creational patterns to support different instantiation protocols and convenient for Behavioural container—like patterns to manipulate contained objects. Exception handling idioms like *try—catch—finally* allow for explicit flow control and are useful for method combination, while usage of exceptions strengthens the pattern robustness by statically documenting pattern intent and enforcing it at runtime.

Use of packages, interfaces, and classes form the foundation for any pattern implementation, regardless of purpose and scope. Interfaces are used explicitly in all pattern implementations except Template Method and Singleton. This is a clear indication that the pattern implementations express the "program to an interface, not an implementation" theme expressed by Gamma et al. as explained in section 2.1.2. Inheritance is exploited in seventeen pattern implementations, while abstract classes in around half. Use of abstract classes is always associated with interface usage. The evaluation shows that inheritance does not exclude composition and the two are often used hand—in—hand, combined with interface usage. Hence, the "favour object composition over class inheritance" theme by Gamma et al. is also expressed. Inner classes are very useful because of their versatility. They can express adaptation and decoration, thus facilitating object composition; they augment static usage of generics; and they can aid in encapsulation and information hiding. Enumerations are useful for patterns that require explicit control over compile—time known instances of a given type.

Generics are used in all implementations except Facade, but only half are unrelated to the evaluation context. This indicates that the context to which the "Gang of Four" design patterns are applied will influence the implementation. Both static and dynamic usage of generics is useful to express pattern functionality. Static usage of generics ensures compile—time type safety useful for especially Creational patterns. Structural patterns rely heavily on object composition and can use generics to ensure proper cooperating types, including usage of generic bounds for greater flexibility. This is also true for Behavioural patterns.

7.1.2. Reflection

This section discusses the use of Java's reflective capabilities to implement pattern functionality. Gamma et al. provide several examples that use or discuss specialised runtime behaviour, albeit in languages like Smalltalk and Self as C++ excels in the lack of runtime features. Examples include using classes to create objects in Abstract Factory [Gamma95, p.90-91] and Factory Method [Gamma95, p.112]; changing the class of an object runtime for State behaviour [Gamma95, p.309]; interception and/or forwarding of messages in Proxy [Gamma95, p215-216] and Chain of Responsibility [Gamma95, p.229]; as well as closures (code blocks in Smalltalk) in Adapter [Gamma95, p.145]. Java supports all these features: class literals can be used for type safety at runtime, and to create new instances of a given class; dynamic proxies can intercept messages and effectively change the class of an object at runtime; and inner classes provide direct support for a limited form of closures.

Gamma et al. also provide several examples that use C++ templates for static creational and/or behavioural purposes in manners not supported directly by Java, but which can be mimicked at runtime. There is no requirement for a C++ template parameter to be defined as an abstract type, which is reminiscent of (static) *duck typing* (see also section **7.1.2.4**) and used in the Strategy implementation [Gamma95, p.319]. Statically, this cannot be enforced in Java without using a common type in form of an upper bound, but can be simulated using annotations at runtime to identify the method of interest. This is also true for the Command [Gamma95, p.240-241] example, which uses function pointers. Directly using the `new` operator to create instances of an actual C++ template parameter as in Factory Method [Gamma95, p.113] is not possible in Java because of type erasure. It can be simulated through class or *type literals*, the latter even supporting creation of exact generic types. A type literal is a Meta class defined in this evaluation that in certain situations allow generic type information to be retained at runtime. Furthermore, privileged access (friends) to certain operations as in the Memento example [Gamma95, p.287] can be achieved statically through package private access, or using reflection at runtime. Overloading of the member access operator can be achieved by dynamic proxies in Java to mimic the behaviour described in the Proxy example [Gamma95, p.221].

7.1.2.1. Class Literals

Builder, Command, Composite, Flyweight, Interpreter, Singleton, and Strategy use *class literals* for runtime type safety, in particular inquiries about type information possibly followed by casts. A class literal is a `java.lang.Class<T>` instance representing the type `T`. For example, `java.lang.String.class` is represented by `Class<String>`. A class represents structural Meta data and need not be used reflectively. Here, we consider reflective usage. Class literals are frequently used to access methods and constructors as described separately in section **7.1.2.3**, or to load classes dynamically. The latter is only used in the `meta.reflect.CallerClass` type to load classes based on stack trace elements as explained below. Otherwise, class loading is handled explicitly by class loaders, the use of which is discussed in section **7.1.3.4**. Class literals can be directly used to instantiate new instances of the type it describes if the class declares a public no—argument constructor. In the evaluation, all reflective creation is handled explicitly by constructors. Because of the extensive use of generic types as described in section **7.1.1.2** on page 85, *type literals* are used in favour of class literals for creational purposes regarding generic types as described in the next section.

Type safe casting is closely related to generic types and methods: because of erasure, it is unsafe to cast to a type parameter. Instead, the `java.lang.Class.cast(java.lang.Object)` method has to be used to cast to a given type, but it requires a class literal, `Class<T>`, at runtime to allow type safe casting to `T`. This is important for any pattern implementation that operates on generic types to keep it type—safe, even if it does not directly relate to the core pattern functionality. The Abstract Factory implementation illustrates the use of class literals for type safe casting in listing **7.17** on page 108 (line 47). Command, Composite, and Flyweight use class literals in generic methods to allow a specific type of contained objects to be retrieved, but this is not vital for the core pattern functionality. It is a principle, though, that can be applied to most container—like Behavioural patterns. Builder only uses class literals because it builds generic `interpreter.Expression<E>` instances from the Interpreter implementation. A cast is required to deliver the correct singleton type because the Singleton implementation uses annotations to identify singleton types and methods (see section **7.1.2.5**).

One of the more direct uses of class literals in this evaluation is the Interpreter implementation. As described, generic expression types are used in form of the `Expression<E>` type. This causes the need for class literals to ensure type safety when casting to `E`. The `interpreter.Context` class stores variables and constants of any type with their assigned values. The only guaranteed super type of the values is `java.lang.Object`, which is how they are stored internally in `Context`. When a value is requested, it must be cast into the proper type to ensure type safety. Simply casting to `E` will cause an unchecked warning because of type erasure. For this to work, the `interpreter.VariableExpression<E>` and `interpreter.ConstantExpression<E>` classes used to represent variables and constants store their generic type internally as a class literal, i.e. `Class<E>`. The assigned value can then be cast to `E` using the `Class.cast(Object)` method from the class literal. Since other types of expressions require access to runtime type information as well, a specific interface describes the functionality. All such types implement the `interpreter.TypedExpression<E>` interface.

Listing 7.14 — Class literals in Interpreter

```java
01 public interface Expression<E> extends StrictCopyable<Expression<E>> { // expression
02   public E evaluate(Context context) throws ExpressionException;
03   public String asSymbol(Context context) throws ExpressionException;
04   public List<Expression<?>> operands();
05   ..
06 }
07
08 public interface TypedExpression<E> extends Expression<E> { // typed expression using class literals
09   public Class<E> type();
10   ..
11 }
12
13 public class Interpreter<T> {
14   private final Class<T> type; // class literal
15   ..
16
17   public <E extends T> T interpret(Context context, Expression<E> expression) throws ExpressionException {
18     return this.interpret(context, expression, this.type);
19   }
20
21   @SuppressWarnings("unchecked")
22   public <E extends T> E interpret(Context context, Expression<E> expression, Class<? super E> type)
23     throws ExpressionException {
24     ..
25     try {
26       ..
27       out.println("Interpreting: ", expression.name(), " -> ", expression.asSymbol(context.reset()));
28       return expression.evaluate(context);
29
30     } catch (BreakExpression.BreakException be) {
31       if (be.expression != null) {
32         if (type.isAssignableFrom(be.expression.type())) { // check type
33           out.warn("Breaking and transfering control to: ", be.expression.asSymbol(context.reset()));
34           return this.interpret(context, (Expression<E>)be.expression, type); // cast
35         } else {
36           out.warn("Cannot handle expression: ", be.expression.type(), " -> ", be.expression.asSymbol(context.reset()));
37           // fall-through
38         }
39       }
40       out.warn("Interpretation exited");
41       return null;
42
43     } catch (ExpressionException ee) {
44       out.println().error("Interpretation failed").error(ee);
45       throw ee;
46     }
47   }
48   ..
49 }
```

As illustrated in listing **7.14** above, the Interpreter implementation also allows the interpretation to exit the current evaluation and perhaps transfer control to a new expression. This is achieved by using a special exception type, `interpreter.BreakExpression.BreakException` as seen in line 30, which stores the expression to transfer control to. A `BreakException` will effectively discard the current evaluation stack while reusing existing variables and constants if a new expression is to be evaluated; crude, but effective for small languages. As exception types cannot be generic, an expression must be stored with a wild—card type parameter. To retain the actual type information, a class literal is used to store the type parameter within the expression; the exception can therefore only store `TypedExpression<?>` instances (line 31-32). This information is enough to determine if the interpreter can handle the actual expression type, and if so make an appropriate cast (line 32-34) before interpreting the expression (line 34). If the type cannot be handled, the interpretation must be aborted (line 41). While the cast to `Expression<E>` is known to succeed because of the test in line 32, the actual type of `E` is unknown, and hence warnings must be suppressed. It is not possible to use `Class.cast(Object)` because `Expression<E>` is generic, but type literals could have been used to remedy this. The `interpreter.Main` test class provide tests that exemplify expressions transferring control to other expressions.

The Bridge, Memento, Observer, and Singleton implementations uses class literals in more advanced ways. The `observer.ObserverManager` class use annotations to identify the notification methods to notify observers with, and class literals determine if a given annotation is applicable for the `ObserverManager` or some other context using the general `@Executor` annotation (section **7.1.2.5** covers annotation usage). For example, the `@Executor(ObserverManager.class)` annotation indicates that only the `ObserverManager` will use the annotated method, regardless of which class declares the annotated method.

Bridge, Memento, and Singleton use class literals to identify the *caller* of a given context based on exception stack traces. The `meta.reflect.CallerClass` and `meta.reflect.Caller` Meta classes *create* exceptions, but do not *throw* them. The stack trace is used to identify the current caller. Combining this with class literals and reflection, the `java.lang.Class<T>` instance representing the caller can be acquired. Even more so, `Caller` can identify a Java *member object* representing the caller as a static initialisation block, initialisation block/constructor, or method, e.g. a `java.lang.reflect.Method` instance[10]. This is similar to joint points in AspectJ as discussed in section **4.3.3**. This is used in the Singleton implementations to allow sub—classing, an issue discussed to some extent by Gamma et al. [Gamma95, p.130-133], but also to ensure that the private singleton constructor cannot be invoked via reflection. This is a novel approach to allow sub—classing of singleton types that we have not seen elsewhere, and is more flexible than using the *Singleton—as—Single—Constant* idiom as described in section **7.1.1.4**. The *Singleton—as—Single—Constant* idiom cannot apply this technique, because enumerations cannot be sub—classed. The principle is illustrated in listing **7.15** below.

Listing 7.15 — Stack trace and identification of class members used for sub—classing in Singleton

```java
public class SimpsonsFamilySequence extends ArraySequence<String> {
  ..

  private final static class Instance { // initialize-on-demand holder class idiom
    private final static SimpsonsFamilySequence instance = new SimpsonsFamilySequence();
  }

  public final static SimpsonsFamilySequence getFamily() {
    return Instance.instance;
  }

  private SimpsonsFamilySequence() { // singleton constructor
```

[10] A fitting idiom name to describe the use of callers as exemplified by the `meta.reflect.Caller` class could be *69. ☺

Listing 7.15 — Stack trace and identification of class members used for sub—classing in Singleton

```java
    ..
    Caller caller = new Caller().getNonClassCaller(); // caller created based on exception stack trace
    if ((caller.callerClass != Instance.class) || (!caller.isStatic())) {
      LogFactory.getLog(this).error("Illegal access to singleton constructor: ", caller);
      throw new SingletonError(caller);
    }
  }

  protected SimpsonsFamilySequence(String... members) { // singleton sub-class constructor
    ..
    Caller caller = new Caller(); // caller created based on exception stack trace
    Caller subClass = caller.getSubClassCaller();
    if ((subClass == null) || (!subClass.isConstructor())) {
      LogFactory.getLog(this).error("Illegal access to sub-class constructor: ", caller);
      throw new SingletonError(caller);
    }
  }
  ..
}
```

The `singleton.SimpsonsFamilySequence` allows a sub—class constructor to invoke its protected sub—class constructor or its protected sub—class copy constructor. It does so by identifying the first caller that is a constructor in a sub—class of `SimpsonsFamilySequence`, if any. If no such caller is found, a `singleton.SingletonError` is thrown. `SingletonError` extends `java.lang.Error` because invoking a sub—class constructor from outside a sub—class is most definitely a big "no—no" and programmer error. However, any singleton type that allows sub—classing in this manner has to trust the sub—class to some extent, because it is not possible to determine if the sub—class constructor invokes the super constructor as it should, e.g. `super(..)`, or creates a new instance of the super—class in the sub—class constructor directly using `new`. Making the super—class abstract is not an option, because then not even a single instance of it can be created.

Using caller information in this way is basically an attempt to imitate friends from C++. The canonical Memento implementation by Gamma et al. uses friends to ensure the Memento participant has "two interfaces", a *narrow* (public) and a *wide* (public and private) with respect to the functionality offered. The private parts of the wide interface are accessed by friends [Gamma95, p.287] as Gamma et al. stress the importance of implementing Memento without violating encapsulation. The same principle is applied in the canonical State implementation [Gamma95, p.305]. This is not possible in Java because Java does not support friends and private interface signatures and/or private implementation in this manner. If the functionality described by the Memento participant, e.g. `memento.SequenceMemento<E>`, must be used outside the declaring package, it must be public unless reflectively accessed. To ensure that only a (memorizable) sequence has access to the state methods in the memento, `CallerClass` is used once again. The `memento.GuardedSequenceMemento<E>` (sub—)class simply checks if the caller class corresponds to the sequence type stored in the memento; internally, a snap—shot (copy) of the sequence represents the state to set. We call `GuardedSequenceMemento<E>` and `SimpsonsFamilySequence` ***guarded types***. Below, listing **7.16** illustrates the `GuardedSequenceMemento<E>` implementation. Its usage is illustrated in listing **7.26** on page 120, where line 19 calls the `GuardedSequenceMemento.getSequence()` state method declared in line 3 below. It will throw an exception if it is invoked from a context class different from the one that stored the sequence in the memento in the first place. In listing **7.26**, any other class than `memento.RangeSequence` calling the method on the memento instance will cause an exception as `RangeSequence` created the memento (line 14).

Listing 7.16 — Stack trace and class literals used to identify caller in Memento

```java
01 public class GuardedSequenceMemento<E> extends SequenceMemento<E> {
02   ..
03   public final Sequence<E> getSequence() { // override
04     Sequence<E> sequence = super.getSequence();
05     CallerClass caller = new CallerClass().getNonClassCaller(); // may be called from this class
06     if (!sequence.getClass().equals(caller.callerClass)) {       // only type of "sequence" is allowed to call
07       LogFactory.getLog(this).error("Caller does not have access: ", caller);
08       throw new UnsupportedOperationException("Method not supported for caller: " + caller);
09     }
10     return sequence;
11   }
12 
13   public final void setSequence(Sequence<E> sequence) { // override
14     CallerClass caller = new CallerClass();
15     if (caller.isSuperClass()) { // may be called from super class constructor
16       caller = caller.getCaller();
17     }
18     caller = caller.getNonAssignableCaller();
19     if (!sequence.getClass().equals(caller.callerClass)) { // caller type must equal type of "sequence"
20       LogFactory.getLog(this).error("Caller does not have access: ", caller);
21       throw new UnsupportedOperationException("Method not supported for caller: " + caller);
22     }
23     super.setSequence(sequence);
24   }
25   ..
26 }
```

This principle could also be used in *protection proxies* [Gamma95, p.208]. Unfortunately, the friend—check is not exactly cheap. It has to be enforced runtime and cannot be employed at compile—time; we get static type safety offered by interfaces with runtime protection of critical operations. On the plus side is that `CallerClass` only use reflection to acquire structural information, not to invoke methods or constructors. The caller functionality is through the `CallerClass` and `Caller` types only, no direct use of reflection. The Singleton implementation is more robust in that the constructors in question are already declared protected and thus requires inheritance to even being considered invoked. This is not the case for public methods. The State implementation does not utilise caller information, but could do so analogous to the Memento implementation.

The Bridge implementation uses caller information to determine if a given method is being invoked as part of the super—class initialisation process or because or ordinary usage. The `GuardedSequenceMemento<E>` class also inquires about super—class invocation based on caller information as illustrated in line 15-16 in listing **7.16** above. The `bridge.SequenceAbstraction<E>` class allows its implementation in form of a `bridge.SequenceValueGenerator<? extends E>` type to be set *after* constructor initialisation, but *before* actual use. The internal state of the sequence abstraction is calculated based on the implementation used, but if no generator is set, the state cannot be determined. Hence, the `SequenceAbstraction.state()` method must throw an exception if invoked before a state can be calculated because sequences by design always have a value and state. However, it is perfectly valid to copy an uninitialised sequence abstraction. The `state()` method should therefore not fail in case a generator has not been set when the `state()` method is used in the super—class copy constructor process. This principle could also be applied in Template Method if it delegates control to primitive or hook operations during initialisation.

Conclusion — Class literals have a number of usages, primarily runtime type safety. All pattern implementations in this evaluation that use class literals also use generic types, and all but one use generic methods as well. They are very useful for container—like Behavioural patterns operating on numerous types to identify a particular type of interest. Class literals form the base for type literals, and are indirectly used to create new instances. Without class literals, it would not be possible to implement reflective Creational patterns. Combined with stack trace information, class literals also allow runtime access checks. Using class literals for reflective purposes will influence the design of pattern participants.

7.1.2.2. Type Literals

Unlike class literals, *type literals* are capable of describing actual generic type parameters in certain situations, making the information available at runtime. While `java.lang.Class<T>` describes the type `T`, there is no way to acquire an exact class of `T` if it is a generic type without applying in an unchecked cast. Class literals only represent raw types. Type literals are **not** a built—in mechanism like class literals but reusable components based on an idea by Neal Gafter called *Gafter's Gadget* [Gafter06]. The implementation here is more advanced, though. Type literals are represented by the abstract `meta.reflect.TypeLiteral<T>` class describing the type `T` in a type safe manner, where `T` may be generic. Since abstract, only sub—classing allows a type literal to be created. Using anonymous sub—classes, dynamic usage of `T` becomes static through inheritance because an explicit type of `T` is required to create a new instance of the sub—class. `T` can thus be accessed at runtime via reflection because it is not erased. This is actually close to the use of templates in C++ in the sense that a new type is created per type literal instance. The abstract `meta.reflect.InstantiableTypeLiteral<T>` class is an example of standard sub—classing of `TypeLiteral<T>`. It provides functionality to create new instances of `T` in a type safe manner, even if `T` is generic. Powerful stuff, indeed, but runtime errors may occur in case `T` represents an interface or abstract class, or if the actual sub—class of `InstantiableTypeLiteral<T>` is generic as well (in which case `T` is still used dynamically). However, that defies the whole purpose of type literals and is as such classified as a programmer error. However, unchecked warnings may have to be suppressed while working with type literals, even though the logic ensures conforming types.

Type literals are used for two purposes: for creational purposes and acquisition of a precise generic type. Abstract Factory and Factory Method use type literals to create products, while Prototype, Proxy, Singleton, and Visitor use them to acquire generic type information. While type literals are designed to describe generic types class literals cannot, Prototype, Proxy, and Visitor only use them because they operate on generic model classes. Advanced type literal usage is illustrated in the Singleton implementation in listing **7.28** on page 124.

Inspired by Gamma et al., the Factory Method implementation in this evaluation illustrates how reflection can be used in the creation process. Abstract Factory provides examples of reflective creation as well, but also illustrates how the Prototype pattern can be used as an alternative in many situations. Gamma et al. illustrate how C++ templates in Factory Method can be used to create new products [Gamma95, p.113]: `new` is simply invoked directly on the actual type of the type parameter within the template. Such a scenario is not possible in Java. C++ templates and Java generics are fundamentally different in that a C++ template creates a new type for each unique type parameter usage and allow template specialisation [Stroustrup91, p.596-597], while Java uses the same raw type regardless of type parameter [Bracha04, p.14]. There are two alternatives in Java: a) specify a generic bound that identifies a type acting as a prototype or delivering a factory method, or b) use reflection. Regarding a), Gamma et al. provide a similar example in their Command implementation using C++ function pointers [Gamma95, p.240-241], but also in the Strategy implementation [Gamma95, p.319]. In this evaluation, the `prototype.StrictCopyable<T>` interface defines prototypical behaviour and is thus a prime candidate as a generic upper bound. Abstract Factory utilises this by implementing a simple stand—alone factory that works by copying prototypes. A prototypical registry that can create numerous product types based on class literals is also defined. This is illustrated in listing 7.17 below.

Listing 7.17 — Reusable prototypical factories in Abstract Factory

```java
public class PrototypicalFactory<T extends StrictCopyable<T>> { // generic bound ensures a copy() method is available
  private final T prototype;

  public PrototypicalFactory(T prototype) {
    this.prototype = prototype.copy(); // defensive copy
  }

  public T create() {
    return this.prototype.copy();
  }
}

public class PrototypicalSequenceFactory<E> extends PrototypicalFactory<Sequence<E>> implements SequenceFactory<E,Void> {

  public PrototypicalSequenceFactory(Sequence<E> prototype) {
    super(prototype);
  }

  public Sequence<E> create() { // covariant return type
    return super.create();
  }

  public Sequence<E> createSequence(Void unused) {
    return this.create();
  }
}

public class PrototypicalRegistry { // registry
  private final Map<Class<? extends StrictCopyable<?>>,StrictCopyable<?>> prototypes;
  ..

  public <T extends StrictCopyable<?>> void registerPrototype(Class<T> type, T prototype) {
    if (type == null) {
      throw new NullPointerException("Type cannot be null");
    }
    this.prototypes.put(type, prototype.copy()); // defensive copy
  }

  public <T extends StrictCopyable<?>> T create(Class<T> type) {
    if (type == null) {
      throw new NullPointerException("Type cannot be null");
    }
    StrictCopyable<?> prototype = this.prototypes.get(type);
    if (prototype == null) {
      throw new NullPointerException("Type not found: " + type.getName());
    }
    T object = type.cast(prototype.copy());
    LogFactory.getLog(this).println("Created object from prototype: ", type.getName(), " -> ", object);
    return object;
  }
  ..
}
```

Regarding b), the Factory Method implementation also provides an example of how reflection can be used to construct a generic factory that can create any type of product, providing the product supplies an appropriate constructor. This is illustrated in listing 7.18 below by the abstract `factorymethod.Factory<T>` class. It also illustrates reflective constructor usage. This is used in both Factory Method and Abstract Factory.

Listing 7.18 — A reusable reflective factory in Factory Method

```java
// Meta classes
public abstract class TypeLiteral<T> {..} // represents T, which may be generic

public abstract class InstantiableTypeLiteral<T> extends TypeLiteral<T> { // represents T and a given constructor

  private final Constructor<?> constructor;

  protected InstantiableTypeLiteral(Class<?>... parameterTypes) throws NoSuchMethodException {
    this.constructor = getConstructor(this.type, parameterTypes); // type, i.e. T, set in super constructor
  }
```

Listing 7.18 — A reusable reflective factory in Factory Method

```

  public T newInstance(Object... arguments) throws Exception { // factory method
    return (T)this.constructor.newInstance(arguments); // cast is known to be safe, but warnings must be suppressed!
  }
  ..
}

// Factory (abstract) class
public abstract class Factory<T> extends InstantiableTypeLiteral<T> { // inherits all functionality from Meta classes

  protected Factory(Class<?>... parameterTypes) throws NoSuchMethodException {
    super(parameterTypes);
  }
}
```

As illustrated, `Factory<T>` extends the `meta.reflect.InstantiableTypeLiteral<T>` class that provides the actual functionality. The `Factory<T>` class could have added additional functionality, for example caching products to support creation of singleton types, but this is not the case here. `Factory<T>` thus functions as an instantiable type literal and by declaring it abstract, concrete sub—classes must supply the actual type parameter used as already described. It can therefore be used to create generic types such as `meta.model.Sequence<java.lang.Integer>`, which is not possible using class literals. The usage is interesting enough to warrant listing **7.19**, which illustrates usage of concrete `Factory<T>` objects from the `abstractfactory.Main` test class in the Abstract Factory implementation. While runtime errors may occur if used incorrectly, for example if `T` represents a non—instantiable interface, the usage of type literals help enforce type safety at compile—time, even for generic types.

Listing 7.19 — Using reflective factories in Abstract Factory

```
// constructor used: int x int (or Integer)
Factory<SequenceValueRange> fsvr =
  new Factory<SequenceValueRange>(Integer.class, Integer.class){/*magic*/}; // anonymous sub-class

// constructor used: no-arg
Factory<MemorizableSequenceAbstraction<Integer>> fmsa =
  new Factory<MemorizableSequenceAbstraction<Integer>>(){/*magic*/}; // anonymous sub-class

// constructor used: BigInteger
Factory<FibonacciSequence> ffs = new Factory<FibonacciSequence>(BigInteger.class){/*magic*/}; // anonymous sub-class

// create products of proper type, even if generic, and assemble
SequenceValueRange svr = fsvr.newInstance(100, 105);
MemorizableSequenceAbstraction<Integer> msa = fmsa.newInstance();
msa.setGenerator(svr);

FibonacciSequence fs = ffs.newInstance(BigInteger.ONE);
```

Conclusion — Type literals are **not** a built—in mechanism, but a very useful technique to represent generic types and to create new instances of the represented type in a compile—time type safe manner. Though type literals rely heavily on reflection and inheritance, reflection is only used in the type identification and creation processes. Creational patterns, except Singleton, can benefit from their usage, but instantiable type literals can act as reusable factories on their own. Using type literals will influence the design of pattern participants.

7.1.2.3. Constructors and Methods

The use of constructors and methods is closely related to class and type literals. Reflectively invoking constructors and in particular methods form the very basis for dynamic proxies as described in the next section. Most usage is implicitly covered in sections describing these issues. Method and constructors have object representations that can be treated like any other object except for explicit creation ("second—class objects").

This information is used in the evaluation to identify callers of a given constructor or method, as already described in section 7.1.1.5 and illustrated in listing 7.15 on page 104.

Abstract Factory, Factory Method, Prototype, and Proxy use constructors reflectively. Only Prototype uses constructors explicitly as illustrated in listing 7.21, all other usage is through instantiable type literals. Type literals hide the complexity in getting and executing constructors reflectively, and they provide a uniform interface to create instances of a given type. Usage is illustrated in listing 7.18 and listing 7.19.

Observer, Prototype, Proxy, Singleton, and Visitor use methods reflectively. Observer and Singleton use annotations to identify methods to execute as notification and singleton methods, respectively. The use of annotations is described in section 7.1.2.5, and listing 7.23 on page 115 illustrates how methods are acquired and invoked reflectively in the Singleton implementation. Prototype and Proxy reflectively execute methods as part of *invocation handlers* used to determine the behaviour of dynamic proxies as described in the next section. This is also illustrated in listing 7.21. Combined with decoration and adaptation, Visitor uses reflective methods as a means to make visitation and double—dispatching easier. The Visitor implementation revolves around visitable (composite) `meta.model.Sequence<E>` types that can be visited based on sequence type and/or on the type of values delivered by the sequence (`E`); the latter is the case discussed here. Instead of having to define new visitable elements continually, e.g. new visitable `Sequence<E>` classes, a visitable decorator is used to decorate sequence instances to become visitable by using reflection to identify and invoke the actual visitation method on the decorated sequence. The decorator is thus adapted to become visitable, but at the same time supplies the implementation for the `accept(..)` method once and for all. The decorator class in question is the `visitor.ReflectiveVisitableSequence<E>` class, and the principle in its usage is illustrated in listing 7.20 below. To identify the proper visitation method, the `ReflectiveVisitableSequence<E>` class relies on naming convention for visitation methods. This is error prone, and will not find methods for sequences delivering unknown types of values; but such visitable sequences can in any case only be visited using the `SequenceVisitor.visitUnknown(..)` visitation method. Instead of throwing exceptions in such a scenario, the decorator will also use `visitUnknown(..)` for robustness. An even more robust alternative could be to use annotations to identify the visitation methods, which would render naming conventions obsolete. As already stated, section 7.1.2.5 describes the use of annotations. In general, reflective invocation of methods is handled by the `meta.reflect.Reflection` class as not to clutter pattern implementations with verbose reflection code, but as an exception to this rule, listing 7.23 on page 115 illustrates direct reflective method invocation.

Listing 7.20 — Reflective method invocation in Visitor

```java
public interface ValueVisitableSequence<E> extends Sequence<E> { // element, e.g. (composite) sequences
  public <P> void accept(SequenceValueVisitor<P> visitor, P argument);
}

public interface SequenceValueVisitor<P> extends SequenceVisitor<P> { // visitor
  public void visitIntegerValued(Sequence<Integer> sequence, P argument);
  public void visitDateValued(Sequence<Date> sequence, P argument);
  ..
}

public class ReflectiveVisitableSequence<E> extends AbstractVisitableSequence<E> { // decorator making sequence visitable
  private final Method visitationMethod;
  private final static Method visitUnknown;
  ..
```

Listing 7.20 — Reflective method invocation in Visitor

```java
  public ReflectiveVisitableSequence(Sequence<E> sequence) {
    super(sequence);
    TypeLiteral<?> type = TypeLiteral.create(sequence.getClass()); // get type of E, e.g. Integer, Date, etc.
    Method method = null;
    try {
      Class<?> clazz = type.getRawType();
      String name = "visit" + clazz.getSimpleName() + "Valued"; // visitation name invariant, may fail and default
      method = SequenceValueVisitor.class.getMethod(name, Sequence.class, Object.class);
    } catch (Exception e) {
      ..
      method = visitUnknown; // default
    }
    this.visitationMethod = method;
  }

  public <P> void accept(SequenceValueVisitor<P> visitor, P argument) {
    Reflection.invoke(visitor, this.visitationMethod, this.getSequence(false), argument); // P visitor argument
  }
  ..
}
```

If the security manager allows it, it is possible to access even private fields, constructors, and methods of a class. This can be used to emulate friendship, but is not recommended, as anybody can become a friend in this manner. The normal visibility rules prevent the non—accessible part of a type to be hidden at compile—time, so compile—time safety is gone. As an example, the Observer implementation can make use of even private notification methods.

Conclusion — Reflectively using methods and constructors are part of the foundation for Java's reflective capabilities. Any pattern implementation using advanced runtime features like those described by Gamma et al. have no choice but to use them, directly or indirectly. This might clutter pattern implementations, but much of the functionality can be used in a component—like fashion, such as the type literal Meta classes defined in this project. Behavioural patterns with functionality related to dispatch of messages to one or more targets at runtime can all use methods reflectively to deliver the messages, like Chain of Responsibility, Command, Mediator, Observer, Strategy, and Visitor. In Java, Structural patterns like Adapter and Proxy implemented using dynamic proxies rely heavily on the use of reflective methods. Excluding Singleton, Creational patterns can use constructors reflectively to create new products. This type of reflective usage only involves reflection in the creation process, where after the created products are accessed through their normal type.

7.1.2.4. Dynamic Proxies

A powerful feature in Java is *dynamic proxies*. Dynamic proxies allow Java to exhibit behaviour traditionally ascribed to prototype—based languages, such as modifying the type of an object at runtime or even changing the implementation of methods at runtime. No byte—code manipulation is involved. Compile—time safety is still adhered to as all access to proxy functionality is through compile—time known interfaces, while runtime exceptions may occur at the time of invocation. Their usage is not just of theoretical interest, albeit we admit that we rarely have used them in production environments. For example, annotation classes acquired at runtime are implemented using dynamic proxies in Java. In realistic environments, runtime errors originating because of proxy usage are rare, because they mostly represent programmer errors. Errors originating in non—proxy objects will still occur in proxied objects, naturally, but this is unavoidable. Dynamic proxies are subject to a rather large number of semantic rules defined by the `java.lang.reflect.Proxy` class, some of the most important ones being that they all inherit `Proxy`; can only implement interface behaviour; and follow the standard rules of the `instanceof` operator. Please refer to Sun's JavaDoc for further information. An implication is that they cannot always be used in a given pattern implementation, for example Creational patterns using covariant return types like the Builder implementation from listing **7.12** on page 99. This is because Builder in certain cases has to use covariant return types to return concrete types that do not have a defining interface.

As illustrated in table **7.1**, Adapter, Prototype, Proxy, and State use dynamic proxies in this evaluation, but dynamic proxies could have been applied in numerous other patterns as well. Examples include Bridge in order to change the implementation transparently; changing the strategy by changing the behaviour of the Strategy type (proxy) at runtime; or in Chain of Responsibility to forward requests not understood. Below, listing **7.21** illustrates how the Prototype implementation uses dynamic proxies to allow any type implementing a copy constructor to become `prototype.Copyable<T>` at compile–time, if not already. The `prototype.StrictCopyable<T>` type cannot be used here because `T` cannot be guaranteed to be copyable. Copying is performed by supplying the proxied instance as an argument to its own copy constructor when the `copy()` method is invoked. The call to `copy()` is automatically intercepted (line 18-20) by an *invocation handler* because the context will access the dynamic proxy, not the proxied instance. All other methods are forwarded to the proxied object (line 22), somewhat akin to the Gamma et al. examples utilising `doesNotUnderstand` in Smalltalk. This corresponds to adaptation, but static class–based adapters (proxies) can often be a viable alternative for simple dynamic proxies, especially if combined with decorators.

Listing 7.21 — Dynamic proxies in Prototype

```
01 public class PrototypeFactory {
02   private final static ProxyFactory factory = new ProxyFactory(); // factory creating java.lang.reflect.Proxy instances
03   private final Method copy; // reference to the Copyable.copy() method
04   ..
05   public <T> T getPrototypicalObject(Class<T> type, final T object) { // must assign proxy to an interface!!!
06     if (object instanceof Copyable) { // already copyable?
07       return object;
08     }
09     final Constructor<T> constructor = Reflection.getCopyConstructor(type); // get copy constructor (closure usage)
10     if (constructor == null) {
11       return null;
12     }
13     Set<Class<?>> interfaces = new LinkedHashSet<Class<?>>();
14     interfaces.add(Copyable.class);                         // add Copyable interface first...
15     Reflection.getInterfaces(type, interfaces);             // ...and then all interfaces implemented by type
16
17     InvocationHandler handler = new InvocationHandler() {
18       public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
19         if (PrototypeFactory.this.copy.equals(method)) { // intercept call to copy()...
20           return constructor.newInstance(object);        // ...and use copy constructor instead
21         }
22         return Reflection.invoke(object, method, args);  // forward all other calls to "object"
23       }
24     };
25     return factory.getProxy(object, handler, interfaces.toArray(new Class<?>[interfaces.size()])); // create proxy
26   }
27   ..
28 }
```

The Proxy implementation uses complex dynamic proxies to comply with most proxy functionality described by Gamma et al. Four different types of proxies are described in the Proxy description, namely *remote proxies*, *virtual proxies*, *protection proxies*, and *smart references* [Gamma95, p.208-209]. The evaluation implements all but remote proxies. A virtual proxy allows a `meta.model.Sequence<E>` to be created only when the first method defined in `Sequence<E>` interface is invoked, and a protection proxy will deny access to certain `Sequence<E>` methods. More realistically, dynamic proxies are used for smart references to force synchronisation on access to a proxied object, and to allow sharing of objects until a *mutator method* is invoked. The latter corresponds to the well—known C++ Handle/Body idiom described by Coplien and paraphrased by Gamma et al. in the Bridge pattern [Gamma95, p.155-156]. Though it has probably been done before, we have never seen a transparent and reusable implementation of Handle/Body in Java[11]. The `proxy.Main` class in the Proxy implementation provides numerous examples of proxy usage. The `meta.reflect.proxy` package forms the base of all the functionality just described, for example the `meta.reflect.proxy.Reference<T>` class.

State uses dynamic proxies to imitate dynamic inheritance as described by Gamma et al. [Gamma95, p.309]. The `state.StepSequence` is an interface that publicly is only implemented as a dynamic proxy. The sequence (proxy) will change its implementation when the sequence value goes from *even* to *odd* or vice versa. These two states are represented by two different sequence implementations, `state.EvenSequence` and `state.OddSequence`, and an instance of the appropriate sequence will be set as the target object for all `meta.model.Sequence<E>` methods that are executed reflectively, but transparently, by the dynamic proxy. The same super type is used here, but it is not required. Hence, this technique is the foundation for *duck typing* in Java 6: "*If it walks like a duck, if it quacks like a duck, it must be a duck*" [Orme05].

Conclusion — Java has built—in support for the Proxy pattern via dynamic proxies. Dynamic proxies depend heavily on reflection, especially reflectively invoking methods. The evaluation shows that dynamic proxies can be used to implement much of the alternative pattern implementations described by Gamma et al., but usage may change the implementation of the pattern participants considerably and cause unforeseen consequences. Structural patterns such as Adapter, Proxy, and to a lesser extent Bridge and Decorator, can benefit from dynamic proxies. Behavioural patterns can also benefit from dynamic proxy usage because it allows the behaviour to change at runtime as clearly illustrated by the State implementation.

7.1.2.5. Annotations

Annotations are used in the implementations of Abstract Factory, Builder, Factory Method, Interpreter, Observer, Singleton, Template Method, and Visitor, but are versatile and useful enough to be applied to practically all pattern implementations. The evaluation uses annotations for three different purposes:

1. As compiler hints: `@Override` and `@SuppressWarnings` annotations from the `java.lang` package;
2. For runtime functionality combined with reflection, for example the `singleton.@Singleton` annotation from the Singleton implementation; and
3. For (static) documentation purposes in order to identify pattern participants described by Gamma et al., for example the `@Participant` annotation from the `meta` package. This is illustrated in listing **7.23**, but this usage is **not** included in table **7.1**. JavaDoc elements can also be considered annotations; all developed source code is fully documented using JavaDoc, including non—public members.

[11] There *are* a few issues with the current Handle/Body implementation, e.g. with `equals(Object)`. This is because the check for the specific type will fail for proxies; the type is `java.lang.reflect.Proxy`, not the actual proxied type.

Regarding item one, patterns relying on sub—classing, especially with Class scope, can benefit from the `@Override` annotation. This is not related to reflection usage, but we include it here for completeness. The compiler will generate an error if the annotation type is used to annotate a method that does not override a super—class method, including abstract methods[12]. As a design evolves, classes and methods change. The `@Override` method can signal that a method intended to override a super—class method no longer can or does so. This could lead to unexpected behaviour if the overridden method was a hook operation with default behaviour not appropriate for the sub—class in question. Abstract Factory, Builder, Factory Method, Interpreter, Memento, Template Method, and Visitor use this annotation and listing **7.22** illustrates the usage in Builder.

Listing 7.22 — Annotations as compiler hints in Builder

```java
public class StandardExpressionBuilder<E> extends AbstractExpressionBuilder<E> implements ExpressionBuilder<E> {

  public Expression<Boolean> buildOrExpression(Expression<Boolean> first, Expression<Boolean> second) {
    Expression<Boolean> expression = new OrExpression(true, first, second);
    LogFactory.getLog(this).println("Building OR (||) expression: ", expression);
    return expression;
  }
  ..
}

public class CountingExpressionBuilder<E> extends StandardExpressionBuilder<E> {

  protected <V,W extends Expression<V>> W count(W expression) {..}

  @Override // will generate compiler error if this method does not override a method from a super-class
  public Expression<Boolean> buildOrExpression(Expression<Boolean> first, Expression<Boolean> second) {
    return this.count(super.buildOrExpression(first, second));
  }
  ..
}
```

The `@SuppressWarnings` annotation is used to suppress compiler warnings, typically unchecked casts in connection with generics. The warning does not aid the robustness of the pattern implementations, but its usage may indicate potential problems. However, its presence is not necessarily a sign of trouble as unchecked casts cannot always be avoided [Langer06, p.270]. Listing **7.14** on page 103 provides an example of its usage.

Two specific annotation types have been created that use a runtime retention policy and can thus be queried reflectively (item two): the `@Singleton` and `@Executor` annotations defined in the `singleton` and `meta.reflect` packages, respectively. They are used to identify a given method to invoke reflectively, but in different ways. `@Singleton` is applicable to types only, e.g. classes, interfaces, annotations, and enumerations, where `@Executor` is applicable only to constructors and methods. `@Singleton` is used to identify the static singleton method to invoke to acquire the singleton instance for the annotated singleton type. As the Singleton pattern is instance, and thus type, centric, the logical design choice is to make `@Singleton` applicable to types only. The annotation effectively identifies the annotated type as a singleton type as well. This can enhance the documentation of the singleton type and clarify pattern intent, but the information can also be used runtime. `@Executor` is for general usage and can therefore identify any type of method or constructor, even private ones if the security manager allows it. It can also identify several methods and/or

[12] In Java 5, the `@Override` annotation can only be applied to a method overriding a method from a super class, while as of Java 6, the annotation can also be used on methods implementing a method declared in an interface [Bloch08, p.178]. We only use the annotation on (non—abstract) methods overriding a (non—abstract) method from a super class.

constructors in the same type, possibly used by several different contexts; it can be seen as a generalisation of the `@Singleton` annotation.

The `singleton.StatelessSingletonRegistry<T>` class is a stateless singleton registry that deliver singleton instances using their own singleton method identified via the `@Singleton` annotation; forwarding via reflection. All actual singleton types in the Singleton implementation are annotated with `@Singleton` and the registry can thus handle unrelated singleton types, regardless of how they are implemented and initialised. For example, `singleton.SimpsonsFamilySequence` is a singleton class using its static `getFamily()` method to return the singleton instance, while `singleton.DanishAlphabetSequence` is a singleton implemented using enumerations and therefore inherits the static `java.lang.Enum.valueOf(java.lang.String)` method. They can both be acquired in a uniform way using a `StatelessSingletonRegistry<T>` instance because `@Singleton` is used to specify the different singleton methods for the two types. This is illustrated in listing **7.23** below. For the Singleton implementations in this evaluation, an annotation is preferred over an interface, because different singleton types in general offer no common functionality except for conceptual behaviour in form of only a single instance available. The singleton method must in any case be static, and interfaces cannot declare static methods. More so, if a common super–type is required for all singleton types used by a registry, the registry is generic with an upper bound on such a type for this very purpose (`java.lang.Object` handles all). Hence, annotation functionality can be combined with standard use of interfaces and generics.

Listing 7.23 — Annotation usage in Singleton

```java
@Participant("Singleton")
@Singleton("getFamily")
public class SimpsonsFamilySequence extends ArraySequence<String> {

  public static SimpsonsFamilySequence getFamily() {..}
  ..
  private SimpsonsFamilySequence() {..}
  ..
}

@Participant("Singleton")
@Singleton(value = "valueOf", arguments = "Instance") // valueOf inherited from java.lang.Enum<DanishAlphabetSequence>
public enum DanishAlphabetSequence implements Sequence<String> {

  Instance; // single constant
  ..
  private DanishAlphabetSequence() {..}
  ..
}

public class StatelessSingletonRegistry<T> implements SingletonRegistry<T> {

  public <S extends T> S getInstance(Class<S> type) {
    Singleton singleton = type.getAnnotation(Singleton.class);
    if (singleton == null) {
      throw new SingletonException(type); // potential runtime problems
    }
    try {
      String singletonMethod = singleton.value(); // fetch singleton method name
      Object[] arguments = singleton.arguments(); // fetch fixed string arguments, if any
      ..
      Method method = type.getDeclaredMethod(singletonMethod, parameterTypes); // arguments.length * String
      S instance = type.cast(method.invoke(null, arguments)); // static, no target
      return instance;
    } catch (Exception e) { // potential runtime problems
      throw new SingletonException(e);
    }
  }
  ..
}
```

The Observer implementation makes use of the `@Executor` annotation to identify notification methods for Observer participants to be stored in an `observer.ObserverManager` instance. Observers do not need to implement a common interface. As always, reflection can cause numerous runtime errors. The `ObserverManager` class uses a pluggable error handling strategy to determine how errors should be processed in case of notification errors. To improve type safety, the `ObserverManager` could be used via composition, wrapped behind methods to add, remove, and especially to notify observers. The abstract `observer.AnnotatedObserversSequence<E>` class shows an example of this. Another plus is that a generic interface cannot be implemented more than once with a different type parameter because of type erasure. For example, a sequence observer type cannot implement `observer.SequenceObserver<java.lang.String>` and `SequenceObserver<java.lang.Integer>`. Annotations can solve this. By letting the class implement standard overloaded methods with proper types without the use of generics, annotations can identify two, or more, different notification methods in the same class.

Though not implemented during the evaluation, several other patterns have been identified where annotations could be applied in a manner similar to Observer (and Singleton). The Chain of Responsibility implementation uses regular interfaces to represent handlers, but could have used annotations. The two implementations can thus be considered representations of two separate approaches to implement the container—like pattern functionality, or they can be combined as shown in the Singleton implementation. Other examples include the Visitor and Prototype implementations. Visitor avoids a static compile—time double—dispatch relationship in the `visitor.ReflectiveVisitableSequence<E>` class by using reflection based on naming conventions for visitation methods, but `@Executor` can be used as an alternative to identify the visitation method. The Prototype implementation uses dynamic proxies to allow any type implementing a copy constructor to become `prototype.Copyable<T>`, but `@Executor` can be used to specify any method or constructor with a matching signature.

Finally, though not part of the evaluation as such, the usage of annotations to annotate pattern participants has proven itself useful in our opinion (item three); the annotations are defined in the `meta` package. The two singleton types from listing **7.23** above illustrate example usage of the `@Participant` annotation, identifying the types as representing the Singleton participant from the Singleton pattern (implicitly in form of the `singleton` package). The annotations are all annotated with the `@Documented` annotation from `java.lang.annotation`, which prompts their precise points of application to become visible in generated JavaDoc. Identifying the individual pattern participants is therefore easy. This is important because Gamma et al. emphasise the need for maintainability: dynamic software is hard to understand, but clear identification of pattern participants can aid in the understanding (see section **2.1.2** on page 18). The defined annotations are all retained at runtime, which is currently not used, though. The original idea was to prime Sun's *Annotation Processing Tool* (APT) with a handler that would traverse the source code statically to gather information about participants, thereby generating a summary of pattern usage, perhaps even compiling code that could be utilised at runtime to enhance pattern support. This has not been completed, though.

Conclusion — Annotation usage help express pattern intent, and can be used in several different and interesting ways. The evaluation shows that annotations can be used statically to identify pattern participants, thereby clarifying the intent of the implementation. They can even enforce the semantics of patterns relying on inheritance and method overriding. At runtime, annotations combined with reflection can be a flexible augmentation and/or alternative to using interfaces if polymorphism among instances is not required; the cost is moderately degraded performance and, if not used wisely, the possibility of runtime errors. Creational and especially Behavioural patterns are the best suited match for runtime annotation usage.

7.1.2.6. Summary

We have illustrated and exemplified how the pattern behaviour described in the introduction of section 7.1.2 can be implemented in Java 6. Java's reflective capabilities allow pattern implementations to use features the canonical C++ implementations cannot while still maintaining compile–time safety. Class literals is the feature most used because of its wide applicability pertaining to type safety, structural Meta Data, and class loading, while type literals are used for type safety involving generic types. Annotations offer exiting new possibilities that the Java community has only just begun to explore, and we have shown how annotations can influence many of the "Gang of Four" patterns, especially Behavioural ones. However, many of the features could have been used in practically all implementations, but the evaluation only supplies example usage in selected pattern implementations as reported in table 7.1 on page 82. The evaluation also provides heavy use of the Meta classes developed, especially the `meta.reflect` and `meta.reflect.proxy` packages.

7.1.3. Special Language Mechanisms

This section describes special language mechanisms available in Java used in the evaluation that can influence the pattern application. The features include synchronisation, serialization, cloning, class loaders, and weak references.

7.1.3.1. Synchronisation

In general, the "Gang of Four" patterns do not address concurrency issues. C++ has no similar construct as the `synchronized` statement in Java[13] [Gosling05, p.395]. The `java.util.concurrent` and sub–packages furthermore provide extensive library support for semaphores, concurrent collection types, threaded execution (which at least Observer could benefit from), and much more. The evaluation only uses the `synchronized` statement and finds it overkill to use the `java.util.concurrent` utilities. A general discussion about writing concurrent programs is **not** undertaken as this is a monstrous task by it self; the second volume of the "POSA" patterns, for example, is solely dedicated to patterns pertaining to concurrent behaviour [Schmidt00].

In our view, well–made designs should express synchronisation by choice, not by design unless absolutely necessary. In real–life systems, when *not* to synchronise can be just as important as *when* to synchronise. The evaluation tries to convey this. Synchronisation implies an overhead that in most cases probably can be avoided; Bloch estimates an overhead from anywhere between five to twenty percent [Bloch01, p.199]. There is no reason, for example, to declare a general container type as synchronised. This is surely one of the reasons the

[13] Synchronisation is an implementation issue. Because of this, JavaDoc does **not** document if a given method is synchronised or not (internal locks are often preferred anyways). The only way to tell if an implementation is thread–safe is to read the human–written (JavaDoc) documentation, if provided.

`java.util.Vector<E>` and `java.util.Hashtable<K,V>` classes from Java 1.1 are practically deprecated in favour of newer non—synchronised container (collection) types such as `java.util.List<E>` and `java.util.HashMap<K,V>`, respectively. Instead, the `java.util.Collections` class provides factory methods to create synchronised (thread—safe) versions of various collections supplied as arguments. Incidentally, this corresponds to the Proxy pattern, where the thread—safe collection types are *smart reference* proxies. Analogous to this, there is no reason to declare container—like pattern participants synchronised unless necessary.

The Bridge, Chain of Responsibility, Facade, Factory Method, Flyweight, Memento, Proxy, Singleton, and Template Method implementations use some form of realistic and/or illustrative use of synchronisation. Bridge and Proxy use synchronisation to create specialisations with the purpose of enforcing thread—safe behaviour. The `bridge.SynchronisedSequenceAbstraction<E>` class is a refined thread—safe abstraction, while the `proxy.SynchronisedSequence<E>` proxy is a *smart reference* that allows any sequence to be used in a thread—safe manner. Synchronisation is thus a design choice that defines the actual behaviour conveyed by the pattern. The use of synchronisation in the Bridge implementation to express pattern functionality is illustrated below in listing **7.24**. Synchronisation proxies, including dynamic ones, can specify the object to use as the lock to avoid synchronising on the actual proxy instance it self. This can be useful to help prevent dead—locks as no other object can synchronise on the lock in question, providing the context using the proxies supply appropriate lock objects.

Listing 7.24 — Synchronisation expressing pattern functionality in Bridge

```java
public class SequenceAbstraction<E> extends AbstractSequence<E> implements Sequence<E> {
  ..
  public SequenceAbstraction<E> setGenerator(SequenceValueGenerator<? extends E> generator) {
    ..
    this.generator = generator;
    this.current = this.generator.first();
    this.state = State.START; // valid state
    return this;
  }

  public E next() {
    ..
    this.current = this.generator.get();
    ..
    return this.current;
  }
  ..
}

public class SynchronisedSequenceAbstraction<E> extends SequenceAbstraction<E> { // abstraction is thread-safe!
  ..
  public synchronized SequenceAbstraction<E> setGenerator(SequenceValueGenerator<? extends E> generator) {
    return super.setGenerator(generator);
  }

  public synchronized E next() {
    return super.next();
  }
  ..
}
```

Chain of Responsibility and Memento synchronise when internal state is updated. This is a design choice, which could have been made differently, keeping the above discussion in mind. The rest of the patterns using synchronisation illustrate that Java's built—in synchronisation mechanism is a powerful tool that can aid the overall design and make the pattern implementations more robust at little cost development—wise in simple implementations. Facade ensures that the sub—system classes it uses are created and accessed correctly. This is vital as all access to sub—systems go through it, but even more so if it is implemented as a Singleton, which is usually the case (including here) according to Gamma et al. [Gamma95, p.193]. Below, listing 7.25 illustrates the use of synchronisation in the Facade implementation.

Listing 7.25 — Synchronisation in Façade

```java
public class MathFacade {
  ..
  public synchronized int getNthPrimeNumber(int n) {
    ..
    // lazy initialisation requires synchronisation
    if (this.primes == null) {
      this.primes = new ReversiblePrimeSequence(100);
    }
    // usage, which may "grow" the prime sequence by creating a new sequence
    ..
  }

  public int getAckermannNumber(int m, int n) {
    ..
    AckermannSequence as = null;
    // lazy initialisation requires synchronisation
    synchronized (this.ackermann) {
      if ((as = this.ackermann.get(m)) == null) {
        this.ackermann.put(m, as = new AckermannSequence(m));
      }
    }
    synchronized (as) {
      // usage, synchronised per sequence instance
      ..
    }
  }

  public BigInteger getNthFibonnaciNumber(int n) {..} // synchronisation not required
  ..
}
```

Flyweight has to synchronise in the `flyweight.CharacterFactory` class in order to ensure proper sharing of flyweight objects as this goes to the very core of the pattern functionality, but synchronisation is only enforced when absolutely necessary, on a per method basis. The Factory and Template Method patterns applied in the `meta.log.LogManager` class use synchronisation to ensure a given log is only created once. The `meta.reflect.proxy.ReferenceHandler<T>` class is a smart reference proxy used to share objects aided by the `meta.reflect.proxy.ProxyFactory` class (Handle/Body idiom). Synchronisation has to be enforced to ensure that the reference count is correct. The semantics of the `java.lang.Object.wait()` method forces the `singleton.StatefullSingletonRegistry<T>` to acquire the lock on the object on which it wishes to invoke `wait()`; this is related to Java, not to the core pattern functionality.

The standard Template Method implementation, however, does not synchronise calls to ConcreteClass participants. Primitive operations and hooks are *alien methods* that the AbstractClass participant has no control over: invoking them from a synchronised context is potentially dangerous [Bloch01, p.196]. Schmidt et al. offer a variant of the Singleton pattern that uses the Double—Checked Locking Optimisation [Schmidt00, p.353] pattern to allow lazy initialisation of Singleton instances, which could not be implemented in Java prior to version 5 [Bacon; Bloch08, p.283-284]. The `dk.rode.thesis.singleton.SmileySequence` class show an example implementation using synchronisation and the `volatile` modifier to achieve double—checked locking. Generally in Java, fortunately, lazy initialisation of singletons can be achieved in most cases without the use of synchronisation, for example as illustrated in listing 7.9 or listing 7.15. Observer does not use synchronisation, but the general `observer.ObserverManager` class is designed primarily for aggregate usage. It could be made thread—safe by the Proxy implementation, or by using inner class adapters.

Conclusion — Writing concurrent programs is never an easy task, but Java has great built–in support to aid correct concurrent behaviour. For the simple canonical pattern implementations presented by Gamma et al., we believe that similar Java implementations can all be made thread–safe using the `synchronized` statement without the need for additional concurrency libraries. The need for realistic thread–safe pattern behaviour will depend on the context at hand. Patterns that do not rely on inheritance or on heavy use of composition are easier to implement because they have complete control over all code executed within critical regions.

7.1.3.2. Serialization

Three patterns use Java's built–in serialization mechanism [Sierra06, p.443-457], namely Bridge, Memento, as well as Singleton when implemented using enumerations. The Bridge implementation employs serialisation in a somewhat unorthodox manner, while Memento and Singleton usage is straightforward. Only Memento and Singleton can be considered proper use, but serialization in Singleton is not core pattern functionality. It is included to illustrate that singleton types can be made serializable in Java. Note, however, that additional singleton instances may be created during deserialisation of normal singleton types, which must naturally be discarded by the singleton class. Thus, the context will never see more than a single instance.

Instead of explicitly using the `memento.SequenceMemento<E>` class, the Memento implementation uses serialization as an alternative way to save the state of `memento.RangeSequence` objects. This is illustrated in listing **7.26** below. The advantage of using the serialization framework is that it provides a standard way of representing object state. It also includes support for JavaDoc, but this is because the serialized representation becomes part of the exported interface. The downside is that serialization semantics is not as trivial as many people think. Bloch dedicates an entire chapter in [Bloch01] to problematic issues related to serialization. Regarding Memento functionality, the main problem is that it creates new objects and does not update the internal state of an existing object. Using the Bridge pattern can circumvent this by only serializing the Implementation participant. Validation of deserialized data must still be explicitly enforced as deserialization can be considered equivalent to a constructor [Bloch01, p.228]. Hence, both serialization and memento objects must explicitly enforce their internal state invariants, but because Memento updates an existing object, it must always enforce the invariants *before* it updates its state or *failure atomicity* cannot be guaranteed. This is important, as any form of serialisation mechanism is error prone, i.e. unexpected versions and/or types, illegal formatting, I/O errors, unexpected deserialized values, etc.

Listing 7.26 — Serialization in Memento

```java
public class RangeSequence extends AbstractSequence<Integer>
  implements ReversibleSequence<Integer>, Serializable, MemorizableSequence<Integer> {
  ..

  public RangeSequence(int start, int end) {
    super(false);
    validate(this.start = start, this.end = end, this.sequence = this.start); // validate
    ..
  }

  private static void validate(int start, int end, int sequence) {..} // throws exception if illegal range

  public SequenceMemento<Integer> getMemorizableState() { // create memento
    return new GuardedSequenceMemento<Integer>(this);
  }

  public void setMemorizableState(SequenceMemento<Integer> memento) throws MemorizableException { // use memento
    try {
      RangeSequence sequence = (RangeSequence)memento.getSequence(); // acquire memento state
```

Listing 7.26 – Serialization in Memento

```java
      // enforce internal state invariants for failure atomicity
      if (sequence.sequence < this.start) {
        throw new MemorizableException(memento);
      } else if (this.bounded()) && (sequence.sequence > this.end)) {
        throw new MemorizableException(memento);
      } else if {..}
      ..
      // update this instance
      this.sequence = sequence.sequence;
      this.state = sequence.state;
      ..
    } catch (Exception e) {
      throw toThrowableType(e, MemorizableException.class);
    }
  }

  private void writeObject(ObjectOutputStream out) throws IOException { // serialize
    out.writeInt(this.start);
    out.writeInt(this.end);
    ..
  }

  private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException { // deserialize
    // implicitly create a new instance and validate
    validate(this.start = in.readInt(), this.end = in.readInt(), this.sequence = in.readInt());
    ..
  }
}
```

According to Gamma et al., a Bridge implementation will use aggregation between the Abstraction and Implementation participants, with Abstraction as the *aggregator* and Implementation as the *aggregatee* [Gamma95, p.153] (see table 2.2 on page 15). This implies a copy of the Abstraction will imply a copy of the Implementation. The Abstraction participant is represented by the `bridge.SequenceAbstraction<E>` class and the Implementor by the `bridge.SequenceValueGenerator<E>` interface. The abstraction represents an abstraction of the `meta.model.Sequence<E>` type, which is prototypical and can be copied. The `bridge.SequenceValueCollection<E extends Serializable,C extends Collection<E>>` class is a specialisation of `SequenceValueGenerator<E>` that delivers values of type `E` stored in a specific `java.util.Collection<E>` class specified by the type parameter `C`. By demanding that the value type is serializable, deep copying becomes easy for any sub–class of `SequenceValueCollection<E,C>` because all collection classes in the `java.util` package implements `java.io.Serializable`. When a `SequenceValueCollection<E,C>` instance is copied, the internal collection of type `C` is simply serialized, then deserialized to yield a new `C` instance containing new `E` instances as well. The structure of the values in the original collection is adhered to as serialization maintains relative object identity in serialized object graphs [Sierra06, p.446]. Enumerations and arrays are also handled automatically [Gosling05, p.288]. The choice to use `java.io.Serializable` as the bound for `E` as opposed to `java.lang.Cloneable` is because most types in the Java API are serializable, while few are cloneable. Using `Cloneable` would not allow sequence value types such as `java.lang.Integer`, `java.lang.Long`, and `java.util.Date` since they are not cloneable, and the clone operations for the collections in `java.util` do not perform deep copy of the collection values. Serialization is not cheap, but a deep copy in any case requires a traversal of all collection values. According to Bloch, however, serialization should be used with caution for a myriad of reasons such as design maintainability and security issues [Bloch01, p.214-218] and we seriously doubt this usage falls within the approved category. We would not release a real–life system utilising serialization in this manner, but it nonetheless illustrates how powerful serialization can be. It also illustrates a subtle difference between Memento and serialization in ordinary use: Memento objects are handled in memory by the Caretaker participant, for example as an undo mechanism in Command as illustrated in this evaluation, where serialization is a mechanism used for persistence, storing bytes and not objects.

Conclusion — In this thesis, serialization is only used for core pattern behaviour in the Memento implementation. Serialization has *external* Object scope, while Memento has *internal* Object scope in that it updates existing objects. Memento focuses on in—memory state, where serialization focuses on externalised, persistent state. Whether or not this is an acceptable design change is entirely dependent on the system at hand, but if object identity is paramount, it cannot directly be used in place of Memento.

7.1.3.3. Cloning

Even though Java has built—in support for the Prototype pattern via the `java.lang.Cloneable` interface with associated runtime *extra—linguistic* creation procedures (no constructor is called [Bloch01, p.45]), we have rarely used it. This evaluation turned out no different. This is because the `Cloneable` interface is a marker interface that does not define an actual clone method, but still demands that cloneable types must override the protected `clone()` method from `java.lang.Object` publicly. Cloning a polymorphic collection of `Cloneable` objects, for example, can therefore only be performed reflectively (and might still fail). More so, cloneable objects cannot be used in Creational patterns such as Abstract Factory and Builder without knowing the explicit type.

The Prototype implementation defines the `prototype.Copyable<T>` interface that can be implemented by prototypical types, or the `prototype.StrictCopyable<T>` sub—type that requires that `T` is copyable itself. The interface defines a `copy()` method that must return a copy of the instance in question. Prototypical types will in their `copy()` declaration or implementation specify the actual implementing type as the return type, `T`, and sub—classes of the given type will refine the type even further using covariant return types (as described in section **7.1.1.6** on page 97). The copying has to be done manually, but by consistently supplying copy constructors, prototyping becomes easy as the `copy()` method simply creates and returns a copy using the copy constructor. As illustrated in listing **7.27** below, the `prototype.SymbolSequence` class is both copyable and cloneable. All sequence types implemented in the evaluation are inherently copyable as `meta.model.Sequence<E>` extends `StrictCopyable<Sequence<E>>`, but not cloneable unless explicitly declared so. This poses a "small" problem with singleton sequence types, because copying must naturally be disallowed for singleton types. The solution used is to return the same singleton instance, but returning null could have been an alternative, or throwing an exception (like `java.lang.CloneNotSupportedException` from `clone()`). Returning the same instance can lead to unpredictable results if the context rely on object identity. In our view, this illustrates a general design problem with Singleton. We believe that Singleton is commonly overused, especially within frameworks. Relying on identity in favour of equality should rarely be the case. Who cares if the system only has one printer spooler as long as the system provides a way to acquire a fully functional spooler? Singleton or otherwise, the spooler object is still used as any other object once acquired. This is a context dependent design preference.

The non—singleton class `singleton.MutatedSimpsonsFamilySequence` is cloneable even though it inherits the `singleton.SimpsonsFamilySequence` singleton that is not cloneable.

Listing 7.27 — Copyable and cloneable behaviour in Prototype

```java
public class SymbolSequence extends ArraySequence<CharSequence> implements Cloneable {
  ..
  public SymbolSequence(SymbolSequence sequence) { // copy constructor
    super(sequence); // call super-class copy constructor
  }

  public SymbolSequence copy() { // specified by interface
    return new SymbolSequence(this);
  }

  public SymbolSequence clone() { // not specified by interface
    try {
      SymbolSequence sequence = (SymbolSequence)super.clone(); // extra-linguistic creation
      sequence.index = this.index;
      sequence.state = this.state;
      return sequence;
    } catch (CloneNotSupportedException cnse) {
      throw new InternalError(cnse.getMessage());
    }
  }
  ..
}
```

Prototypical objects are used extensively throughout the evaluation, not only for `Sequence<E>` types. The Command and Interpreter implementations also rely on prototypical behaviour, and more importantly rely on it in a polymorphic fashion through the `StrictCopyable<T>` interface. Creational patterns also benefit from the use of a real interface compared to a marker interface. Using `java.lang.Cloneable` is therefore **not** a viable alternative, as it would have made several pattern implementations more difficult.

Conclusion — The evaluation shows that it is preferable to implement prototypical behaviour by design rather than relying on Java's built—in support for `Cloneable`. This ensures that pattern participants can access prototypical functionality directly through interfaces as opposed to using reflection or concrete types.

7.1.3.4. Class Loader and Weak References

Several "Gang of Four" pattern descriptions discuss *ownership* of objects, primarily for Behavioural container—type like patterns such as Composite [Gamma95, p.169], Flyweight [Gamma95, p.200], Iterator [Gamma95, p.266], and Observer [Gamma95, p. 297]. Focus is on memory management, but also on aggregation and composition in general. As Java supports garbage collection, ownership is generally less important compared to C++, but should still not be neglected. By using weak references, a container can declare that it does **not** own contained types. Weak references allow an object, say X, to keep a reference to another object, Y, which will not prevent Y from being garbage collected. Java's Collections Framework has direct support for weak references via the `java.util.WeakHashMap<K,V>` class, which is map storing keys of type `K` as weak references associated with a value stored as a hard reference, `V`.

Weak references are used directly in the Chain of Responsibility, Observer, and Proxy implementations, but also in Meta classes. The Proxy implementation stores proxied objects as weak references in the proxy factory. This ensures that if the context no longer references the proxy, the proxied object may be garbage collected (internally, proxies store a reference to the proxied object). This is analogous to Observer, where the `observer.ObserverManager` class only stores observers as weak references. The implementation automatically handles if an observer goes out of scope, but it also implies that anonymous observers (adapters) created on—the—fly, for example as an argument when the observer is registered, should not be used because they are eligible for garbage collection at any time. Using weak references implies more work on the part of the developer, and can cause surprising behaviour in clients using the patterns as explained. In the Chain of Responsibility implementation, the `chainofresponsibility.WeakHandlerChain<R>` component illustrates similar functionality as it stores handlers as weak references. An example where weak references should not be used is in the `meta.log.LogManager` class, which is implemented using Factory and Template Method. Logs are often not stored as member attributes in the class(es) they are used by, but acquired via static methods before use, e.g. `LogManager.getLog(java.lang.Class)`. Hence, acquisition of a log associated with a given class is likely to cause a new log to be created frequently because immediately after usage, the log is eligible for garbage collection. File system loggers could thus represent a serious performance problem. An alternative is to use cache—like behaviour with timeout functionality to close and discard inactive logs. The problem is that the log manager has no way of knowing how clients internally store the logs, if at all. Keeping only weak references to created logs therefore seems troublesome.

Conversely, in the Singleton implementation, the `singleton.StatefullSingletonRegistry<T>` allows singleton types to stay in memory even after the class loader that created the class has been garbage collected. Classes can be loaded without explicitly using a class loader via class literals. This can be of utmost importance because any class is only a singleton per class loader (see JavaDoc for `java.lang.ClassLoader`). This is not just of theoretical interest as for example servlet containers often use more than one class loader. To cope with such situations, the `singleton.LoadableSingletonRegistry<T>` is a state full registry that allows singleton types to be dynamically loaded using the same class loader, while still enforcing the generic bound `T` using type literals. This is illustrated in listing **7.28** below.

Listing 7.28 — Dynamic class loading in Singleton

```java
public abstract class LoadableSingletonRegistry<T> extends StatefullSingletonRegistry<T> { // abstract to capture T
  private final ClassLoader classLoader;
  ..

  protected LoadableSingletonRegistry(ClassLoader classLoader, SingletonRegistry<T> registry) { // aggregate registry
    super(registry);
    if ((this.classLoader = classLoader) == null) {
      throw new NullPointerException("Class loader cannot be null");
    }
  }

  @SuppressWarnings("unchecked")
  public T getInstance(String className) throws ClassNotFoundException {
    Class<?> clazz = this.classLoader.loadClass(className); // class literal representing the class just loaded
    TypeLiteral<?> type = TypeLiteral.create((Type)clazz);  // type literal representing "clazz"

    TypeLiteral<?> handledTypes = TypeLiteral.create(this.getClass()); // type literal representing T

    handledTypes.asType(type); // if this succeeds without a class cast exception, the types conform
    LogFactory.getLog(this).println("Loaded singleton class: ", className);
    return this.getInstance((Class<T>)clazz); // now safe to cast, but must suppress warnings
  }
}
```

The Proxy implementation also allows a specific class loader to create dynamic proxies for explicit control of proxy creation. Finally, creating classes reflectively will inherently use the class loader of the class used in the creation process.

Conclusion — Explicit usage of class loaders and/or weak references are advanced features useful in special situations only, which can easily make the pattern implementations more complex and lead to unexpected results. Usage is closely related to patterns that require detailed knowledge about and/or control over pattern participant lifespan such as Singleton. Usage should be documented via JavaDoc, at least, so clients will know the pattern behaviour.

7.1.3.5. Summary

As reported in table 7.1 on page 82, the evaluation shows that of the special language mechanisms investigated, only synchronisation is commonly used. It is used in nine pattern implementations as well as in Meta classes. The other investigated mechanisms have their (specialised) uses, but in this evaluation, they are not shown to be widely applicable in conjunction with the "Gang of Four" patterns. However, even though the use of dynamic class loading is only illustrated in the Singleton pattern, this is a mechanism widely used in real—life systems. Structural and in particular Creational patterns can benefit from its usage. Weak references should only be used with extreme care, especially in Behavioural patterns. The use of Java's cloning facility is discouraged because it relies on conventions rather than interfaces. Prototypical behaviour is better achieved through an explicit application of the Prototype pattern that relies on interfaces. Serialization is a useful mechanism for saving and recreating serializable objects, but it has different focus and scope compared to Memento. If working with in—memory objects, Memento is preferable.

7.1.4. Feature Observations

This section gives an overview of how several unique C++ features applied in the "Gang of Four" patterns are "translated" to (dynamic) Java 6 features used in the evaluation. The Java 6 alternatives are all dynamic in nature compared to the static features found in C++. This influences the pattern implementations because they yield more dynamic and parameterised program code, which according to Gamma et al. as explained in section 2.1.2 on page 18 is harder to understand compared to static software [Gamma95, p.21]. Rephrasing the pattern descriptions for Java 6 could yield descriptions that are more verbose as well. On the other hand, several of the utilised static features in Java 6 have similar constructs in C++, but they are not employed by Gamma et al. Examples include inner classes, covariant return types, and varargs. Their usage is not discussed further.

7.1.4.1. Static vs. Runtime Protection

The canonical Iterator, Memento, and State implementations discuss or use C++ friends to achieve two levels of static protection that defines how pivotal pattern participants interrelate. Publicly, a *narrow* interface is exposed, while privileged pattern participants have access to a *wide* interface that grants access to private information. The interface (e.g. Memento participant) is one and the same, using different levels of access modifiers and friends. This is possible because friends in C++ has the same access rights to data and operations of a befriended class as the class itself [Stroustrup91, p.566-568]. Thus, at compile—time, the privileged class (e.g. Originator) has access to the full interface functionality. In Java, a *class* may declare and implement private operations, but there is no way to convey this information to clients without relying on conventions. There is no way to declare that an *interface* defines private operations, though an interface may be private. All operations declared in an interface are public by nature, but protected access can be achieved through abstract classes. Java does not support private implementation inheritance. However, abstract classes are in general not a viable alternative since Java only supports single inheritance.

The evaluation offers two ways to solve this:

1. Define a single type describing the public (narrow) functionality and rely on conventions for the private functionality exposed in the wide interface in C++. Inner classes can act as "friends" locally and through inheritance, but reflection must otherwise be used to access the information outside the package. The cost is compile—time type safety.
2. Define a type describing the narrow (public) functionality and a type describing the private parts of the wide (private) interface. The information exposed by the latter interface has to be public but can be accessed in a compile—time safe manner. The cost is exposing otherwise private information publicly, but exception stack traces and class literals can be used to identify the caller and only grant access to approved "friends".

Regarding item 1), a (protected) inner class may define protected behaviour sub—classes can access but it relies on inheritance once again. This corresponds to the Iterator functionality described by Gamma et al. [Gamma95, p.262]. An example of this functionality is the `meta.reflect.CallerClass.CallerIterator<C>` inner class. Using reflection to access private information is error prone at runtime. It also violates information hiding. Regarding compile—time type safety, it is not a good solution for API's or frameworks, but will work fine in proprietary applications because the use of reflection is more controlled. However, proprietary applications may not be concerned with the same stringent requirements to protect information and may be satisfied by using public types as described in item 2). The split between a public type and accessing the private parts via reflection is not used in the evaluation, though reflection is used in several pattern implementations to access potentially private operations as an alternative to using a single interface and/or private inheritance.

The strategy explained in item 2) is the alternative used in the evaluation to implement Memento and State, but seems applicable in any similar scenario. The Memento implementation enforces access rules at runtime that can be performed statically in C++. It is not correct simply to supply the caller as an object to the methods in the wide interface, because such an object can be forged; there is no way to guarantee that it represents the actual caller. It also clutter the signatures of the methods defined in the interface. Instead, the access rules are enforced using the `CallerClass` type. Providing the JVM supplies a trustworthy stack trace, the caller cannot be faked. It also ensures that the implementation of the methods in the wide interface do not use reflection directly, but access the `CallerClass` type, which is used as any other object. The State implementation does not enforce caller checks, but could easily have done so. When the wide interface is implemented by a private (inner) class, there is no need to enforce caller checks.

7.1.4.2. Multiple Inheritance vs. Interfaces and Composition

In all fairness, Gamma et al. do not utilise multiple inheritance much. The pattern implementations in this evaluation utilise multiple interface implementation to a much larger degree than the "Gang of Four" implementations. This is because the scope of the evaluations here are more realistic, but also because inheritance is not utilised as much in Java. Private (functional) inheritance is a variant of the scenario described in the previous section. It may be used alone without the use of other (public) inherited classes, e.g. single inheritance, but is often used with other inherited types. The canonical Adapter implementation uses both public and private inheritance. In Java, the private inherited type must be used through composition. In case it is necessary to use a public interface to represent the functionality (as a type), access rules can be enforced in a similar manner as described in the previous section.

Public multiple inheritance, as used in the canonical Observer implementation, must be implemented using interfaces with direct implementation or using composition. This is required or the type information is lost. C++ allows the implementation to be inherited as well. Composition indicates more dynamic program structure, but this is not the case here. The interfaces representing the classes otherwise inherited in C++ are fixed at compile—time. In all likelihood the corresponding implementations and/or components will be fixed as well (declared final). The evaluation shows that this is the case here.

7.1.4.3. Templates vs. Generics

All template functionality used in the canonical C++ implementations can be replaced by generics using upper bounds. The requirement for upper bounds is to ensure that all realised type parameters conform to a given interface that determines the functionality, which could otherwise be specified solely through matching types in C++. This enforces type safety. Operators cannot be invoked directly on the type parameter, so more work may be required. Furthermore, because class literals cannot express generic types, type literals may be used to augment the compile—time type safety.

7.1.4.4. Templates vs. Annotations

The use of C++ templates in the canonical examples to ensure matching signatures of function pointers can be replaced by annotations without the need for generic bounds. The gain is increased flexibility at the expense of compile—time safety. The matching signatures can be identified by annotations. The evaluation shows that implementing components to find and execute such signatures is straight—forward.

7.1.4.5. Overloaded Operators vs. Dynamic Proxies

Overloaded operators are not supported in Java 6. The only solution is to use normal methods, which cannot express semantics like the C++ -> operator utilised in the canonical Proxy implementation, or to use (dynamic) proxies. The latter requires extensive factory usage to hide the complexity in the proxy creation process.

7.2. Pattern Relationships

Table **3.4** on page 45 introduced the "Gang of Four" pattern system and explained the relationships between different patterns as described by Gamma et al. Below, table **7.2** compares the described relationships with the actual relationships expressed in this evaluation. The expressed relationships are subjective: a different evaluation will probably produce different relationships. ~~Crossed out~~ relationships are not expressed, while (parenthesised relationships) are expressed indirectly or could be expressed with minor changes, such as for example writing anonymous adapter classes. Relationships prefixed with a plus (+) are not described directly by Gamma et al., but expressed in this evaluation. Some patterns are expressed more than once, such as Factory Method. The relationship may also be expressed via API or Meta classes and not only in the different pattern implementations. These issues are not described in detail as the focus is on expressed pattern relationships.

Table 7.2 — Pattern relationships in the "Gang of Four" pattern implementations

Name	Expressed Relationships	Remarks
Creational Patterns		
Abstract Factory	⇒ creates Bridge ⇒ ~~alternative to Builder~~ ⇒ ~~collaborates with or alternative to Facade~~ ⇒ uses Factory Method ⇒ uses ~~or alternative to~~ Prototype ⇒ ~~is a Singleton~~	Implementation creates `bridge.SequenceAbstraction<E>` and `bridge.SequenceValueGenerator<E>` objects, but prototypical and reflective factories can create various product types.
Builder	⇒ ~~alternative to Abstract Factory~~ ⇒ ~~creates Bridge~~ ⇒ creates Composite ⇒ ~~is a Singleton~~	Not an alternative to Abstract Factory as different products are created. Created `interpreter.Expression<E>` objects can be composite.
Factory Method	⇒ used by Abstract Factory ⇒ used by Iterator ⇒ alternative to Prototype ⇒ used by Template Method	Creates `command.Command<E>` objects, which are prototypes. Iterator types used are not polymorphic, but `meta.reflect.CallerClass.CallerIterator<C>` returns polymorphic types using Factory Method.
Prototype	⇒ ~~used by or alternative to Abstract Factory~~ ⇒ implemented by Command ⇒ collaborates with Decorator ⇒ alternative to Factory Method ⇒ ~~is a Singleton~~ ⇒ ~~collaborates with Template Method~~	Both `command.Command<E>` and `decorator.SequenceDecorator<E>` objects act as prototypes. Alternative to Factory Method as the implementation creates commands. All Singleton types disallow copy. Prototypes are not created from Template Method(s).
Singleton	⇒ ~~implemented by Abstract Factory~~ ⇒ ~~implemented by Builder~~ ⇒ implemented by Facade ⇒ ~~implemented by Mediator~~ ⇒ (implemented by Prototype) ⇒ ~~implemented by Observer~~ ⇒ (implemented by State) ⇒ + uses Adapter	The State implementation uses enumeration constants to implement different states that are singletons per design. The `prototype.PrototypeFactory` class is not a Singleton, but uses a local singleton instance of the `meta.reflect.proxy.ProxyFactory` class. The `singleton.DanishAlphabetSequence` class uses adaptation to create an anonymous delegate sequence.
Structural Patterns		
Adapter	⇒ (alternative to Bridge) ⇒ alternative to Decorator ⇒ (alternative to Proxy) ⇒ + is a Strategy ⇒ + used by Singleton	An `adapter.SequenceAdapter<E,E>` is an alternative to `decorator.SequenceDecorator<E>`. The adapter delegates defined in `adapter.AdapterStrategy` class are applications of the Strategy pattern.
Bridge	⇒ created by Abstract Factory ⇒ ~~alternative to Adapter~~ ⇒ ~~created by Builder~~	The Bridge implementation cannot use a `meta.model.Sequence<E>` type as the implementation.
Composite	⇒ created by Builder ⇒ collaborates with Chain of Responsibility	The Builder implementation can create and the Interpreter implementation can evaluate composite `interpreter.Expression<E>` objects.

Table 7.2 — Pattern relationships in the "Gang of Four" pattern implementations

Name	Expressed Relationships	Remarks
	⇒ collaborates with Decorator ⇒ collaborates with Flyweight ⇒ used by Interpreter ⇒ uses ~~or collaborates with~~ Iterator ⇒ collaborates with Visitor ⇒ + uses Strategy	The Chain of Responsibility, Flyweight, and Visitor implementations all operate on (different) composite elements. The `decorator.SequenceDecorator<E>` class decorates `composite.CompositeSequence<E>` instances. A specific traversal strategy can be used to traverse the composite structure, for example breath—first.
Decorator	⇒ alternative to Adapter ⇒ collaborates with Prototype ⇒ collaborates with Composite ⇒ ~~alternative to Strategy~~	Decorator is used by Adapter, Prototype, and Composite.
Facade	⇒ ~~collaborates or alternative to Abstract Factory~~ ⇒ ~~alternative to Mediator~~ ⇒ is a Singleton	Implemented as a Singleton.
Flyweight	⇒ collaborates with Composite ⇒ ~~used by Interpreter~~ ⇒ implemented ~~or used~~ by State ⇒ implemented by Strategy ⇒ + uses Factory Method ⇒ + creates Proxy	`flyweight.Sentence`, `flyweight.Word`, and `flyweight.Character` objects form a composite—like structure. Factory Method is used to create flyweight objects. Returned collections of flyweights are guarded by *protection proxies* in form of unmodifiable collections.
Proxy	⇒ alternative to Adapter ⇒ alternative to Decorator ⇒ + is a Decorator	Dynamic proxies can simulate any adapter or decorator. Decorators are inherited to supply default behaviour.
Behavioural Patterns		
Chain of Responsibility	⇒ collaborates with Composite	The `chainofresponsibility.HandlerChain<R>` type maintains handlers in a composite structure.
Command	⇒ is a Composite ⇒ uses Memento ⇒ is a Prototype ⇒ + is a Template Method	The `command.CompositeCommand<E>` class represents a composite (macro) command. The undo mechanism for `command.SequenceCommand<E>` sub—classes uses mementos, if possible, and is implemented as a Template Method. Any command acts as a prototype.
Interpreter	⇒ uses Composite ⇒ ~~uses Flyweight~~ ⇒ uses Iterator ⇒ ~~uses Visitor~~ ⇒ + uses Prototype	The `interpreter.FlowExpression<E>` class is composite, and can be traversed using iterators. No specific visitor interface is defined and the `interpreter.Interpreter<T>` class traverses the syntax—tree based on the normal `interpreter.Expression<E>` interface. Expressions are prototypical.
Iterator	⇒ used by ~~or collaborates with~~ Composite ⇒ uses Factory Method ⇒ used by Interpreter ⇒ ~~uses or alternative to Memento~~ ⇒ + uses Decorator ⇒ + uses Strategy	The internal iterator class `iterator.ProcessableSequence<E>` uses composition instead of inheritance. Any sequence can become iterable using a given strategy to process the sequence values.
Mediator	⇒ ~~alternative to Facade~~ ⇒ ~~collaborates with Observer~~	Not implemented (but evaluated).

Table 7.2 – Pattern relationships in the "Gang of Four" pattern implementations

Name	Expressed Relationships	Remarks
	⇒ ~~is a Singleton~~	
Memento	⇒ used by Command ⇒ ~~used by or alternative to Iterator~~ ⇒ + uses Prototype	The `memento.SequenceMemento<E>` class uses `meta.model.Sequence<E>` prototypical behaviour to save the sequence state.
Observer	⇒ ~~collaborates with Mediator~~ ⇒ ~~is a Singleton~~ ⇒ + is a Composite ⇒ + is a Template Method ⇒ + uses Decorator ⇒ + creates Proxy ⇒ + uses Adapter	Observers can form a composite structure. To ensure `meta.model.Sequence<E>` objects supplied as arguments in notification methods cannot be modified, adaptation (of type parameters) followed by proxying is employed (immutable). Decorator can make any sequence observable.
State	⇒ is a ~~or uses~~ Flyweight ⇒ is a Singleton ⇒ + <u>uses Abstract Factory</u>	The State implementation uses enumeration constants to implement different states that are singletons and only operate on extrinsic data. Abstract Factory is used to create *target objects* (dynamic proxies).
Strategy	⇒ alternative to Decorator ⇒ is a Flyweight ⇒ ~~alternative to Template Method~~	Strategy implementations have no internal state and are unique.
Template Method	⇒ uses Factory Method ⇒ ~~collaborates with Prototype~~ ⇒ ~~alternative to Strategy~~ ⇒ + implemented by Decorator ⇒ + creates Proxy	The `templatemethod.SequenceTemplate<K,E>` uses Factory Method to generate keys for sequence values. Creates protection proxies to control access to constructed sequence values.
Visitor	⇒ collaborates with Composite ⇒ ~~used by Interpreter~~ ⇒ + uses Strategy	The `visitor.VisitableCompositeSequence` class is a composite sequence supporting visitors. A specific traversal strategy can be used to traverse the visitable structure, for example depth–first.

As table 7.2 illustrates, there is a fair overlap between the expressed relationships and the relationships identified by Gamma et al. The question is thus what caused the changes: the design and/or the language? As the remarks explain, the primary difference is a matter of design. This is truly an indication of versatility of the "Gang of Four" design patterns. There are only two relationships that we can attribute *almost solely* to Java 6 features, namely the compositions Singleton using Adapter and State using Abstract Factory (<u>underlined</u> above).

Adaptation is used in the `singleton.DanishAlphabetSequence` singleton type, which is implemented using the *Singleton–as–Single–Constant* idiom defined in this thesis. As Java does not support multiple inheritance, `DanishAlphabetSequence` cannot inherit `meta.model.ArraySequence<E>`, which supplies all the needed functionality, because it already extends `java.lang.Enum<DanishAlphabetSequence>`. To avoid duplicate code, we use composition and delegate the actual sequence functionality of `DanishAlphabetSequence` to an anonymous sub–class of `ArraySequence<java.lang.String>` stored as an aggregate member. The anonymous sub–class is an adaptation to the `meta.model.Sequence<java.lang.String>` type.

The State implementation uses Abstract Factory to create `state.StepSequence` instances. This is because there is no single class defining the functionality described by `StepSequence`. Instead, the factory returns a dynamic proxy *representing* the functionality. The dynamic proxy imitates dynamic inheritance [Gamma95, p.309] by changing the *target object* of the methods being invoked reflectively via the proxy. This effectively changes the implementation of the methods being invoked and thus of the `StepSequence` proxy instance.

There is a rather high coherency between the different pattern implementations. This is because they basically operate on the same model classes, primarily `meta.model.Sequence<E>` and derived types such as `prototype.StrictCopyable<T>`, `interpreter.Expression<E>`, and `command.Command<E>` that one way or the other end up manipulating sequences. On the other hand, several implementations are fixed because of this, and cannot easily, if at all, be componentized. For example, the Bridge implementation must use an explicit type of `bridge.SequenceValueGenerator<E>` and cannot easily act as an alternative to a general adapter. Componentization has not been a primary goal of the implementations, but a welcome addition (as for example the Chain of Responsibility, Factory Method, Observer, and Singleton Registry implementations as listed in the next section).

Alternative relationships between applied patterns are primarily expressed in form of application of the Strategy, Decorator, Adapter, and Proxy patterns, which all are somewhat similar in purpose and highly dynamic. This is expected and understandable. Using Prototype for the Memento pattern and Composite for the Observer pattern is nothing novel either, but it illustrates the versatility in the pattern application.

7.3. Implementation Level

The classification of implementation level as defined by Norvig is applied to the pattern implementations and listed below in table 7.3 (see table 4.1 on page 55).

Table 7.3 – Implementation level of the "Gang of Four" patterns in Java 6

Name	Level	Type	Description
Creational Patterns			
Abstract Factory	Formal	Evaluation component, Language support	Reflection, using prototypes and/or, `factorymethod.Factory<T>`.
Builder	Informal	Problem specific, Language support	Could use reflection and/or `meta.reflect.InstantiableTypeLiteral<T>`.
Factory Method	Formal	Evaluation component, Language support	Reflection, using `InstantiableTypeLiteral<T>`, and/or `Factory<T>`.
Prototype	Formal	API, Language support	`java.lang.Cloneable` with specific language support.
Singleton	Invisible	Evaluation component, Language support	Singleton registries reusable. Enumerations support creational semantics.
Structural Patterns			
Adapter	Invisible	Problem specific,	Anonymous inner classes (closures), dynamic proxies.

Table 7.3 — Implementation level of the "Gang of Four" patterns in Java 6

Name	Level	Type	Description
		Language support	
Bridge	Informal	Problem specific	
Composite	Informal	Possible component	`instanceof` operator can help differentiate between Composites and Leafs.
Decorator	Informal	Problem specific	
Facade	Informal	Problem specific, Language support	Packages and access modifiers.
Flyweight	Informal	Problem specific, Language support	Synchronisation to ensure proper creation of flyweights.
Proxy	Formal	Evaluation component, API, Language support	`java.lang.reflect.Proxy` with specific language support, and `meta.reflect.proxy` package.
Behavioural Patterns			
Chain of Responsibility	Formal	Evaluation component	`HandlerChain<R>`, `Handler<R>`, and `HandlerLink<R>` in the `chainofresponsibility` package (including implementations).
Command	Informal	Possible component	Candidates are `command.Command<E>` and `command.CommandProcessor`.
Interpreter	Informal	Problem specific, Possible component	Candidates are `Expression<E>` (sub—)types, `Context`, and `Interpreter<T>` in the `interpreter` package.
Iterator	Invisible	API, Language support	Java *for—each* loop, `java.util.Iterator<E>` and `java.lang.Iterable<T>` with standard implementations.
Mediator	Informal	Problem specific	
Memento	Informal	Problem specific	`java.io.Serializable` targets different problem.
Observer	Formal	API, Evaluation component	`java.util.Observer` and `java.util.Observable`, rarely used. `observer.ObserverManager` and `meta.reflect.@Executor` are reusable.
State	Informal	Problem specific, Language support	Can be implemented using enumerations. Dynamic proxies can alter behaviour at runtime.
Strategy	Invisible	Problem specific, Language support	Anonymous inner classes (closures).
Template Method	Informal	Problem specific, Language support	Abstract classes, abstract methods, varargs, and covariant return types.
Visitor	Informal	Problem specific	Only single—dispatch, but can use reflection.
Meta Classes	Formal	Evaluation Components	`meta.reflect`, `meta.reflect.proxy`, and most of `meta.util` packages reusable.

We consider Adapter, Iterator, Singleton, and Strategy to be Invisible. The use of inner classes to adapt to a given interface is so routinely used it merits the Invisible classification. The same is true for Strategy. Iterator even more so, because of the built—in *for—each* loop as of Java 5. Singleton is considered Invisible if implemented using enumerations, because the implementation will not differ from normal enumeration usage.

No special tricks are required; the only difference between normal enumerations and singletons is that only a single constant is present in the singleton type.

Abstract Factory, Chain of Responsibility, Factory Method, Observer, Prototype, and Proxy are considered Formal. In the implementations, all but Prototype offer reusable components, either directly or in form of Meta classes found in `meta.reflect`, `meta.reflect.proxy`, or `meta.util`. In fact, we claim the key classes from the pattern implementations can be used "as is", or at least with few adjustments or adaptation. Prototype is classified as Formal because of Java's built—in cloning mechanism, though the evaluation advise against using it. It is the usage of reflection that makes the Creational patterns componentizable, while it is the easily recognisable container—like structure that aids the Behavioural patterns identified as Formal. Proxy exploits the built—in support for dynamic proxies. We consider componentized patterns a good source for possible Java 6 idioms. The patterns classified as Invisible above could also be classified as "universally known" Java 6 idioms, but applying a pattern or idiom in our view indicates more work than using Invisible features.

The rest of the patterns are deemed Informal, though it is a judgement call. Memento, for example, can exploit serialization in a manner similar to the cloning mechanism, which is classified as Formal. The reason is that we believe serialization does not adequately describe the Memento abstraction in that it creates new objects instead of updating existing ones. Command, Composite, and Interpreter are borderline, but still not classified as Formal; we estimate that reusable components could be made fairly easy. The Command Processor variant is especially interesting because of its very flexible use of commands. Even though many Informal patterns are considered problem specific, and thus requires careful application, reflection could aid in creating reusable components based on conventions. Builder, for example, is not classified as Formal since the evaluation did not provide at least a partial reusable component. Visitor uses reflection to provide reuse based on convention, but it is still too specific to be promoted to Formal.

7.4. Summary

Below, we list and summarise the key issues from the comparative evaluation of the "Gang of Four" patterns:

– The evaluation shows that **Java's mixture of static and dynamic features is well suited for the abstractions described by the "Gang of Four" patterns**. The **static features aid the robustness**, **pattern intent**, **and reusability** of the patterns, while the **dynamic features allow for much greater flexibility**.

– The evaluation revealed **possible approaches on how to implement pattern functionality described with intrinsic C++ features in Java 6**. The approaches are closely related to identified **high—lights**.

– The evaluation illustrates that **the "Gang of Four" patterns are truly versatile** because they can **cooperate in numerous different ways**, many of which are not even described by Gamma et al.

– **We consider four "Gang of Four" patterns to have an implementation level of Invisible**, while an additional **six are considered Formal with associated reusable components**.

The evaluation of **core language features** includes (abstract) classes, interfaces, inheritance, nested and anonymous classes, generic types and methods, enumerations, exception handling, covariant return types, and varargs. The features are **mostly static** by nature, and **form the foundation for any Java 6 program**. All features are **widely used in the pattern implementations**, but especially so for **interfaces**, **inheritance**, and **generic types** and **methods**. Aided by core language features, the implementations express the "**program to an interface, not an implementation**" and "**favour object composition over class inheritance**" themes described by Gamma et al. Inheritance does not exclude composition and the two are often used hand—in—hand, combined with interface usage. The core language features **promote robustness**, **pattern intent**, and **reusability**.

The evaluation of **reflective features** includes class literals, type literals, constructors, methods, dynamic proxies, and annotations. The features are **mostly dynamic** by nature and **Creational and Behavioural patterns benefit most from their usage**. Within the realm of this evaluation, **Java 6 can express all reflective pattern behaviour described by Gamma et al**. Java's reflective capabilities offer **several new**, **flexible**, and generally **interesting approaches** on how to implement pattern functionality, **especially when combined with core language features**. **Annotations are especially interesting**. The evaluation of **special language mechanisms** includes synchronisation, serialization, cloning, class loader, and weak references. While the features offer **interesting possibilities**, only synchronisation is commonly used in the evaluation.

By utilising the evaluation features, several **pattern and implementation high—lights have been identified**. **Abstract Factory and Factory Method utilise type literals** for generic factories capable of creating even generic types in a type—safe manner. **Memento uses class literals and stack trace information** to identify callers to simulate C++ friends while still providing compile—time type safety. **Observer uses annotations** to identify observers and notification methods without coercing a common observer type. **Proxy and State use dynamic proxies** to mimic duck typing and dynamic inheritance. **Singleton uses class literals and stack trace information** to allow sub—class specialisation, while the ***Singleton—as—Single—Constant*** idiom defined in this thesis **justifies that Java 6 has built—in support for the Singleton pattern**. These high—lights are described in detail in section 9.3.

The implementations **express many of the pattern relationships identified by Gamma et al.** Several relationships are not expressed, while **numerous others not described by Gamma et al. have materialised**. The primary cause of **difference is related to the design**, not to the language. This is a strong indication of **how versatile the "Gang of Four" patterns are** and the **quality and familiarity they contribute to any design** where applied properly. Only **two relationships can be ascribed solely to the use of Java 6**, namely the **Singleton using Adapter** and **State using Abstract Factory** compositions. Singleton using Adapter is caused by usage of the *Singleton—as—Single—Constant* idiom, while State using Abstract Factory is because dynamic proxies are used to simulate dynamic inheritance. On the other hand, **Java 6 makes it easy to express many relationships using its powerful static and runtime features**.

Based on the evaluation, **we consider Adapter**, **Iterator**, **Singleton**, and **Strategy** to have an **implementation level corresponding to Invisible**. The use of inner classes to adapt to a given interface is so routinely used it merits the Invisible classification, which is also true for Strategy. Iterator even more so because of direct language support. Singleton is considered Invisible if implemented using the *Singleton—as—Single—Constant* idiom. **We consider Abstract Factory**, **Chain of Responsibility**, **Factory Method**, **Observer**, **Prototype**, and **Proxy** to have an **implementation level corresponding to Formal**. All but Prototype offer reusable components, either directly or in form of Meta classes, but the Prototype interface functionality can still be reused. The **rest of the patterns are considered Informal**, but the **Meta classes also aid greatly in their implementations.**

8. Detailed Evaluation

A language that does not affect the way
you think about programming, is not worth knowing.
— Alan J. Perlis

This chapter presents the individual "Gang of Four" pattern implementations and corresponding evaluations. The patterns are presented in the order defined by Gamma et al. To keep this thesis as short as possible, the detailed pattern evaluations are kept to a minimum using the evaluation format defined in the evaluation approach. Each pattern has a dedicated section that presents its Intent, Structure, Participants, and Implementation, using familiar "Gang of Four" pattern element names. We present the pattern structure as a detailed UML class diagram that includes information about relevant attributes and operations. The UML diagrams also identifies the pattern participants, which are described thereafter, linked to the developed Java classes. An Implementation section addresses all functionality described by Gamma et al. in the Implementation and Sample Code elements for a given pattern, and illustrates whether or not the functionality can be implemented in Java 6. The paragraph headings in the Implementation sections are taken from the corresponding topics discussed by Gamma et al. in the pattern descriptions. A brief summary concludes this chapter.

8.1. Creational Patterns

8.1.1. Abstract Factory

Abstract Factory is a Creational pattern with Object scope. The implementation is located in the `dk.rode.thesis.abstractfactory` package.

8.1.1.1. Intent

Provide an interface for creating families of related or dependent objects without specifying their concrete classes [Gamma95, p.87].

8.1.1.2. Structure

Figure 8.1 illustrates the Abstract Factory implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

8.1.1.3. Participants

Table 8.1 describes the Abstract Factory participants in the words of Gamma et al. [Gamma95, p.89] and lists the corresponding implementations developed in the evaluation.

8.1.1.4. Implementation

Factories as singletons — Design choice. **Creating the products** — Factory Method is used in all implementations of `SequenceFactory<E,P>`. Prototype usage is illustrated in `PrototypicalFactory<T>`. A prototypical registry capable of creating any number of prototypical products is implemented as `PrototypicalRegistry`. Type literals are preferred over class literals to create products because they support generic types, for example using `factorymethod.Factory<T>`. **Defining extensible factories** — Parameterised product creation is illustrated in `factorymethod.CommandCreator<E,T>` and `GeneratorFactory<E,P>`. The first illustrates use of arguments to *determine* the product type, while the latter use arguments required *by* the product type. The `PrototypicalRegistry` class shows how to create unrelated product types in a type safe manner that does not require an unsafe down–cast as in C++.

Figure 8.1 — Abstract Factory UML Class diagram

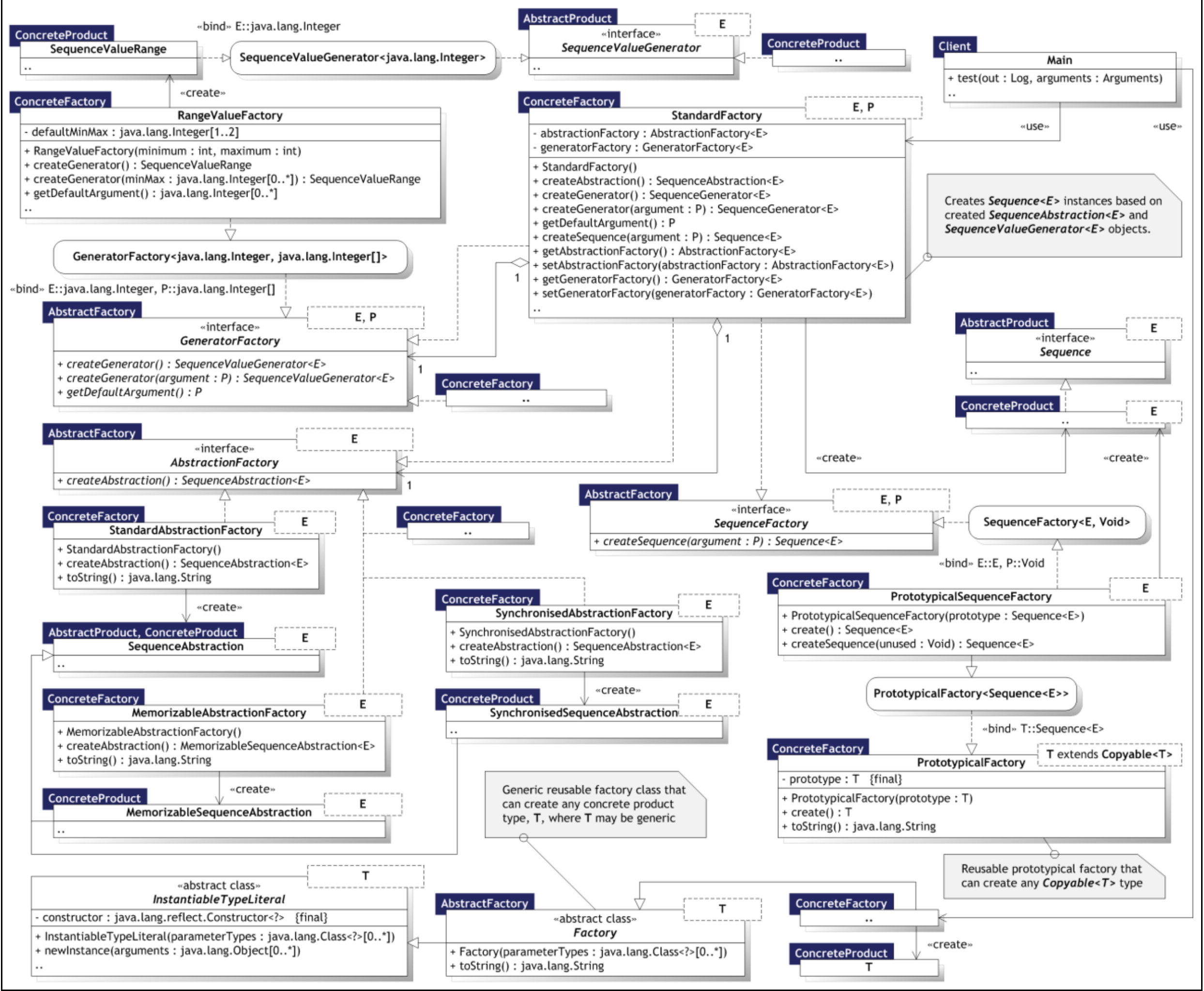

ConcreteProduct
SequenceValueRange
«bind» E::java.lang.Integer
SequenceValueGenerator<java.lang.Integer>
AbstractProduct
«interface»
SequenceValueGenerator
E
ConcreteProduct
Client
Main
+ test(out : Log, arguments : Arguments)
ConcreteFactory
RangeValueFactory
«create»
- defaultMinMax : java.lang.Integer[1..2]
+ RangeValueFactory(minimum : int, maximum : int)
+ createGenerator() : SequenceValueRange
+ createGenerator(minMax : java.lang.Integer[0..*]) : SequenceValueRange
+ getDefaultArgument() : java.lang.Integer[0..*]
ConcreteFactory
StandardFactory
E, P
- abstractionFactory : AbstractionFactory<E>
- generatorFactory : GeneratorFactory<E>
+ StandardFactory()
+ createAbstraction() : SequenceAbstraction<E>
+ createGenerator() : SequenceGenerator<E>
+ createGenerator(argument : P) : SequenceGenerator<E>
+ getDefaultArgument() : P
+ createSequence(argument : P) : Sequence<E>
+ getAbstractionFactory() : AbstractionFactory<E>
+ setAbstractionFactory(abstractionFactory : AbstractionFactory<E>)
+ getGeneratorFactory() : GeneratorFactory<E>
+ setGeneratorFactory(generatorFactory : GeneratorFactory<E>)
«use»
«use»
Creates Sequence<E> instances based on created SequenceAbstraction<E> and SequenceValueGenerator<E> objects.
GeneratorFactory<java.lang.Integer, java.lang.Integer[]>
«bind» E::java.lang.Integer, P::java.lang.Integer[]
AbstractFactory
«interface»
GeneratorFactory
E, P
+ createGenerator() : SequenceValueGenerator<E>
+ createGenerator(argument : P) : SequenceValueGenerator<E>
+ getDefaultArgument() : P
ConcreteFactory
AbstractProduct
«interface»
Sequence
E
ConcreteProduct
E
AbstractFactory
«interface»
AbstractionFactory
E
+ createAbstraction() : SequenceAbstraction<E>
«create»
«create»
AbstractFactory
«interface»
SequenceFactory
E, P
+ createSequence(argument : P) : Sequence<E>
SequenceFactory<E, Void>
«bind» E::E, P::Void
ConcreteFactory
StandardAbstractionFactory
E
+ StandardAbstractionFactory()
+ createAbstraction() : SequenceAbstraction<E>
+ toString() : java.lang.String
ConcreteFactory
«create»
AbstractProduct, ConcreteProduct
SequenceAbstraction
E
ConcreteFactory
SynchronisedAbstractionFactory
E
+ SynchronisedAbstractionFactory()
+ createAbstraction() : SequenceAbstraction<E>
+ toString() : java.lang.String
ConcreteFactory
PrototypicalSequenceFactory
E
+ PrototypicalSequenceFactory(prototype : Sequence<E>)
+ create() : Sequence<E>
+ createSequence(unused : Void) : Sequence<E>
ConcreteFactory
MemorizableAbstractionFactory
E
+ MemorizableAbstractionFactory()
+ createAbstraction() : MemorizableSequenceAbstraction<E>
+ toString() : java.lang.String
«create»
ConcreteProduct
SynchronisedSequenceAbstraction
E
PrototypicalFactory<Sequence<E>>
«bind» T::Sequence<E>
ConcreteFactory
PrototypicalFactory
T extends Copyable<T>
- prototype : T {final}
+ PrototypicalFactory(prototype : T)
+ create() : T
+ toString() : java.lang.String
«create»
ConcreteProduct
MemorizableSequenceAbstraction
E
Generic reusable factory class that can create any concrete product type, T, where T may be generic
Reusable prototypical factory that can create any Copyable<T> type
«abstract class»
InstantiableTypeLiteral
T
- constructor : java.lang.reflect.Constructor<?> {final}
+ InstantiableTypeLiteral(parameterTypes : java.lang.Class<?>[0..*])
+ newInstance(arguments : java.lang.Object[0..*])
AbstractFactory
«abstract class»
Factory
T
+ Factory(parameterTypes : java.lang.Class<?>[0..*])
+ toString() : java.lang.String
ConcreteFactory
ConcreteProduct
T
«create»
1
1
1
1

Table 8.1 — Abstract Factory participants

Participant	Description	Implementation
AbstractFactory	– Declares an interface for operations that create abstract product objects.	`AbstractionFactory<E>`, `GeneratorFactory<E,P>`, `SequenceFactory<E,P>`, `PrototypicalFactory<T>`, `factorymethod.Factory<T>`
ConcreteFactory	– Implements the operations to create concrete product types.	`PrototypicalRegistry`, `StandardFactory<E,P>`, `StandardAbstractionFactory<E>`,

Table 8.1 — Abstract Factory participants

Participant	Description	Implementation
		`SynchronisedAbstractionFactory<E>`, `MemorizableAbstractionFactory<E>`, `PrototypicalAbstractionFactory<E>`, `CollectionValueFactory<E>`, `RangeValueFactory`, `PrototypicalSequenceFactory<E>`, sub–classes of `Factory<T>` and `PrototypicalFactory<T>`
AbstractProduct	– Declares an interface for a type of product object.	`bridge.SequenceAbstraction<E>`, `bridge.SequenceValueGenerator<E>`, `meta.model.Sequence<E>`
ConcreteProduct	– Defines a product object to be created by the corresponding concrete factory, and implements the **AbstractProduct** interface.	`facade.FibonacciSequence`, and all `bridge.SequenceAbstraction<E>` and `bridge.SequenceValueGenerator<E>` implementations
Client	– Uses only interfaces declared by **AbstractFactory** and **AbstractProduct** types.	`Main`

8.1.2. Builder

Builder is a Creational pattern with Object scope. The `dk.rode.thesis.builder` package contains the implementation.

8.1.2.1. Intent

Separate the construction of a complex object from its representation so that the same construction process can create different representations [Gamma95, p.97].

8.1.2.2. Structure

Figure 8.2 illustrates the Builder implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

8.1.2.3. Participants

Table 8.2 describes the Builder participants in the words of Gamma et al. [Gamma95, p.98-99] and lists the corresponding implementations developed in the evaluation.

8.1.2.4. Implementation

Assembly and construction interface — Design choice. The `ExpressionBuilder<E>` type builds products in a tree–like structure. Various build methods accept already constructed expressions, while some constructed expressions can be manipulated directly, e.g. `interpreter.FlowExpression<E>`. **Why no abstract class for products?** — Design choice. The Builder implementation use covariant return types to express specific types when required. **Empty methods as default in builder** — Design choice. In our view, however, empty methods constitute a poor design choice that will inevitable lead to problems in form of runtime errors, e.g. null pointers. Empty methods are must better suited for Template Method with primitive operations that do not return a value the context depends on.

Figure 8.2 — Builder UML Class diagram

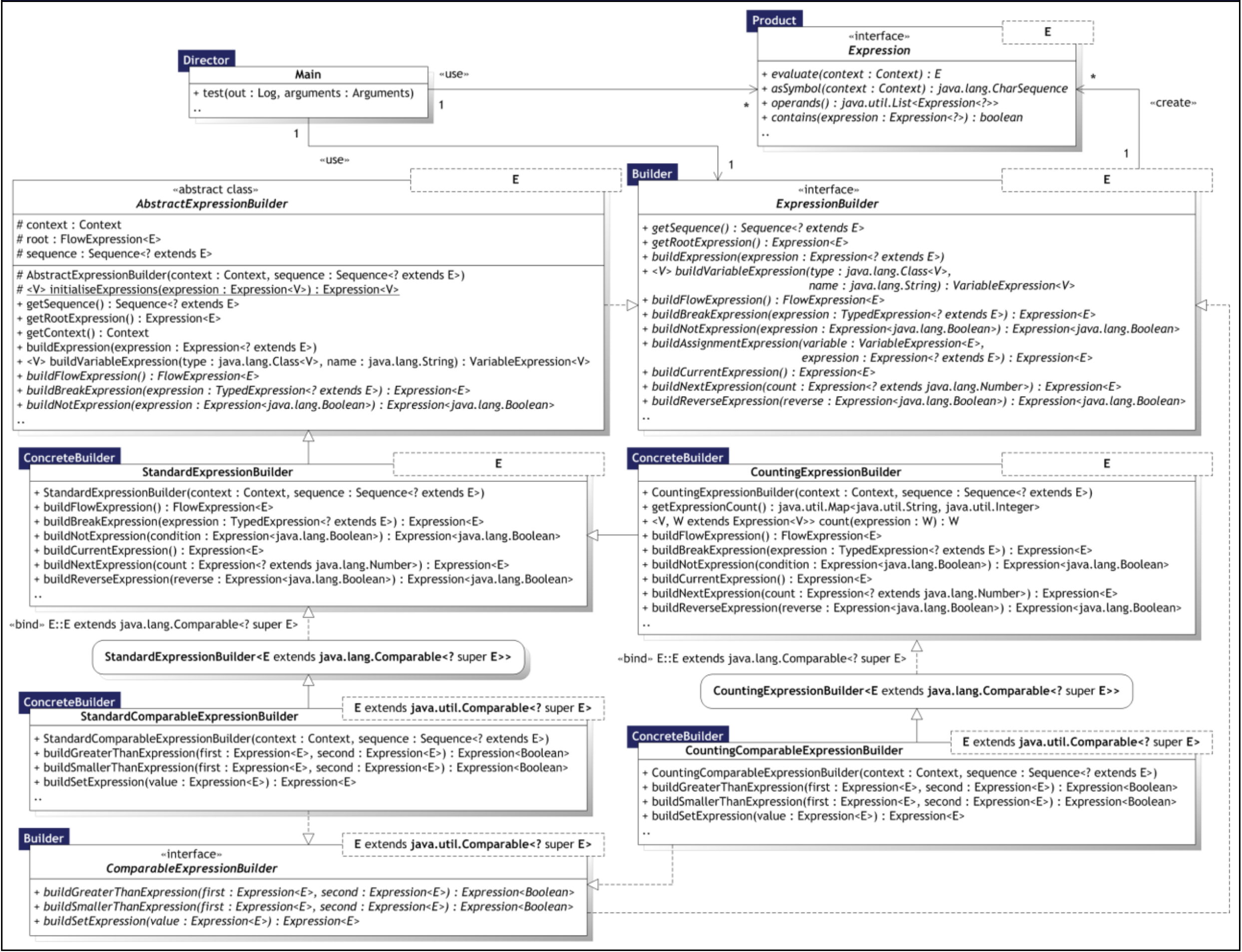

Product
«interface»
Expression
Director
Main
+ test(out : Log, arguments : Arguments)
«use»
«create»
Builder
«abstract class»
AbstractExpressionBuilder
«interface»
ExpressionBuilder
ConcreteBuilder
StandardExpressionBuilder
CountingExpressionBuilder
«bind» E::E extends java.lang.Comparable<? super E>
StandardExpressionBuilder<E extends java.lang.Comparable<? super E>>
CountingExpressionBuilder<E extends java.lang.Comparable<? super E>>
StandardComparableExpressionBuilder
CountingComparableExpressionBuilder
E extends java.util.Comparable<? super E>
«interface»
ComparableExpressionBuilder

Table 8.2 — Builder participants

Participant	Description	Implementation
Builder	– Specifies an abstract interface for creating parts of a **Product** object.	`ExpressionBuilder<E>`, `ComparableExpressionBuilder<E>`
ConcreteBuilder	– Constructs and assembles parts of the product by implementing the **Builder** interface. – Defines and keeps track of the representation it creates. – Provides an interface for retrieving the product.	`StandardExpressionBuilder<E>`, `StandardComparableExpressionBuilder<E>`, `CountingExpressionBuilder<E>`, `CountingComparableExpressionBuilder<E>`, `TypedExpressionBuilder<E>`, `TypedComparableExpressionBuilder<E>`
Director	– Constructs an object using the **Builder** interface.	`Main`
Product	– Represents the complex object under construction. **ConcreteBuilder** builds the product's internal representation and defines the process by which it is assembled. – Includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.	All `interpreter.Expression<E>` implementations, including `interpreter.TypedExpression<E>` types

8.1.3. Factory Method

Factory Method is a Creational pattern with Class scope. The `dk.rode.thesis.factorymethod` package contains the implementation.

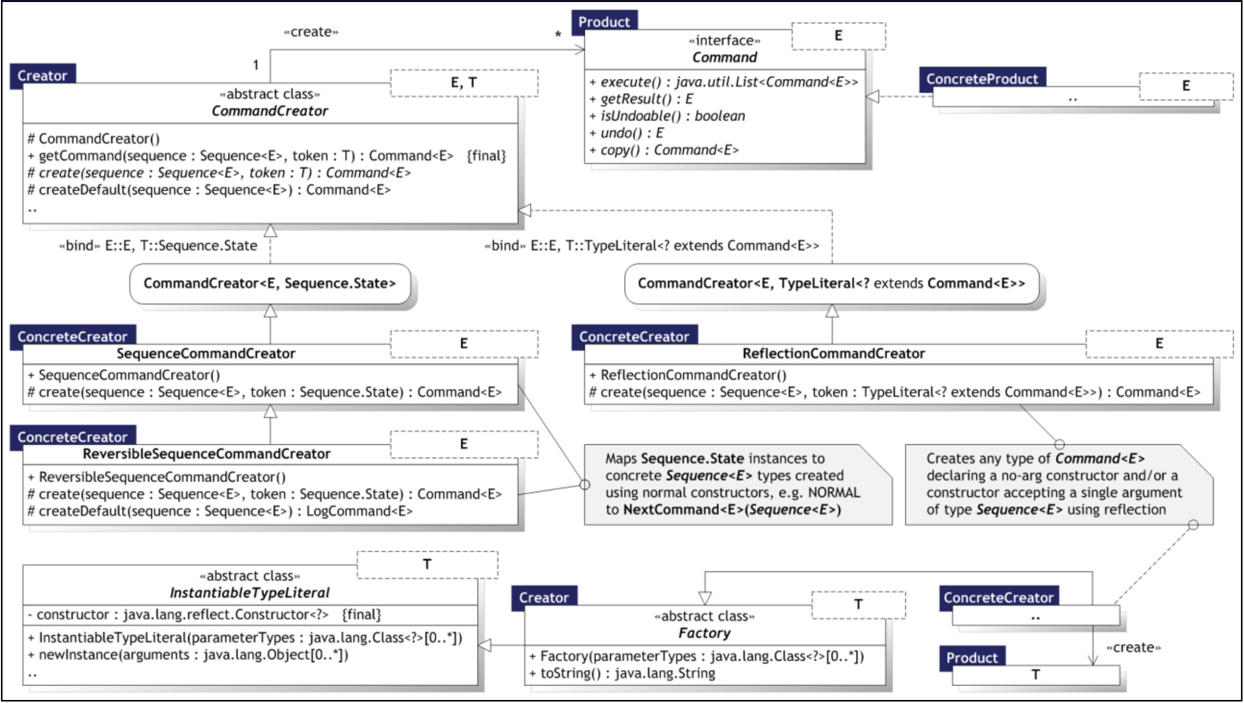
8.1.3.1. Intent

Separate the construction of a complex object from its representation so that the same construction process can create different representations [Gamma95, p.97].

8.1.3.2. Structure

Figure 8.3 illustrates the Factory Method implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

Figure 8.3 – Factory Method UML Class diagram


«create»
1
*
Product
«interface»
Command
E
+ execute() : java.util.List<Command<E>>
+ getResult() : E
+ isUndoable() : boolean
+ undo() : E
+ copy() : Command<E>
ConcreteProduct
E
Creator
«abstract class»
CommandCreator
E, T
# CommandCreator()
+ getCommand(sequence : Sequence<E>, token : T) : Command<E> {final}
# create(sequence : Sequence<E>, token : T) : Command<E>
# createDefault(sequence : Sequence<E>) : Command<E>
«bind» E::E, T::Sequence.State
«bind» E::E, T::TypeLiteral<? extends Command<E>>
CommandCreator<E, Sequence.State>
CommandCreator<E, TypeLiteral<? extends Command<E>>
ConcreteCreator
SequenceCommandCreator
E
+ SequenceCommandCreator()
# create(sequence : Sequence<E>, token : Sequence.State) : Command<E>
ConcreteCreator
ReflectionCommandCreator
E
+ ReflectionCommandCreator()
# create(sequence : Sequence<E>, token : TypeLiteral<? extends Command<E>>) : Command<E>
ConcreteCreator
ReversibleSequenceCommandCreator
E
+ ReversibleSequenceCommandCreator()
# create(sequence : Sequence<E>, token : Sequence.State) : Command<E>
# createDefault(sequence : Sequence<E>) : LogCommand<E>
Maps Sequence.State instances to concrete Sequence<E> types created using normal constructors, e.g. NORMAL to NextCommand<E>(Sequence<E>)
Creates any type of Command<E> declaring a no-arg constructor and/or a constructor accepting a single argument of type Sequence<E> using reflection
«abstract class»
InstantiableTypeLiteral
T
- constructor : java.lang.reflect.Constructor<?> {final}
+ InstantiableTypeLiteral(parameterTypes : java.lang.Class<?>[0..*])
+ newInstance(arguments : java.lang.Object[0..*])
Creator
«abstract class»
Factory
T
+ Factory(parameterTypes : java.lang.Class<?>[0..*])
+ toString() : java.lang.String
ConcreteCreator
«create»
Product
T

8.1.3.3. Participants

Table 8.3 describes the Factory Method participants in the words of Gamma et al. [Gamma95, p.108-109] and lists the corresponding implementations developed in the evaluation.

Table 8.3 – Factory Method participants

Participant	Description	Implementation
Product	– Defines the interface of objects the factory method creates.	`command.Command<E>`
ConcreteProduct	– Implements the **Product** interface.	All `command.Command<E>` implementations

Table 8.3 — Factory Method participants

Participant	Description	Implementation
Creator	– Declares the factory method, which returns an object of type **Product**. Creator may also define a default implementation of the factory method that returns a default **ConcreteProduct** object. – May call the factory method to create a Product object.	`CommandCreator<E,T>`, `Factory<T>`, `TypedFactory<T,P>`
ConcreteCreator	– Overrides the factory method to return an instance of a **ConcreteProduct**.	`SequenceCommandCreator<E>`, `ReversibleSequenceCommandCreator<E>`, `ReflectiveCommandCreator<E>`, `EvilSequenceCommandCreator<E>`, sub–classes of `Factory<T>` or `TypedFactory<T,P>`

8.1.3.4. Implementation

Two major varieties — The `CommandCreator<E,T>` class provide a default implementation, but requires sub–classes to implement the abstract factory method. This is design choice; sub–classes could simply return null from the implemented factory method and the default would always be used. **Parameterised factory methods** — `CommandCreator<E,T>` illustrates parameterised factory methods, including call to super (default) implementations. `T` is the token used to determine the product type (command) to create. **Language specific variants and issues** — Type literals are preferred over class literals to create products because they support generic types, for example using `Factory<T>`. The `abstractfactory.PrototypicalRegistry` class stores prototypes based on their classes. In Template Method, `templatemethod.SequenceTemplate<K,E>` illustrates that, unlike C++, it is possible to invoke sub–class hooks from the constructor. Lazy initialisation is used in Facade, which only creates sequences used for calculation on demand in a thread–safe manner. **Using templates to avoid sub—classing** — The `Factory<T>` class can create any type in a type–safe fashion, including generic types, providing `T` supplies an applicable constructor. The `abstractfactory.PrototypicalRegistry` uses upper bounded type parameters to ensure that all type parameters are copyable since the `new` operator cannot be invoked on type parameters because of erasure. **Naming conventions** — Design choice, but important if overloaded factory methods are used. Java cannot overload on generic types, only raw types. The Visitor implementation shows how naming can allow visitation based on type parameters.

8.1.4. Prototype

Prototype is a Creational pattern with Object scope. The `dk.rode.thesis.prototype` package contains the implementation.

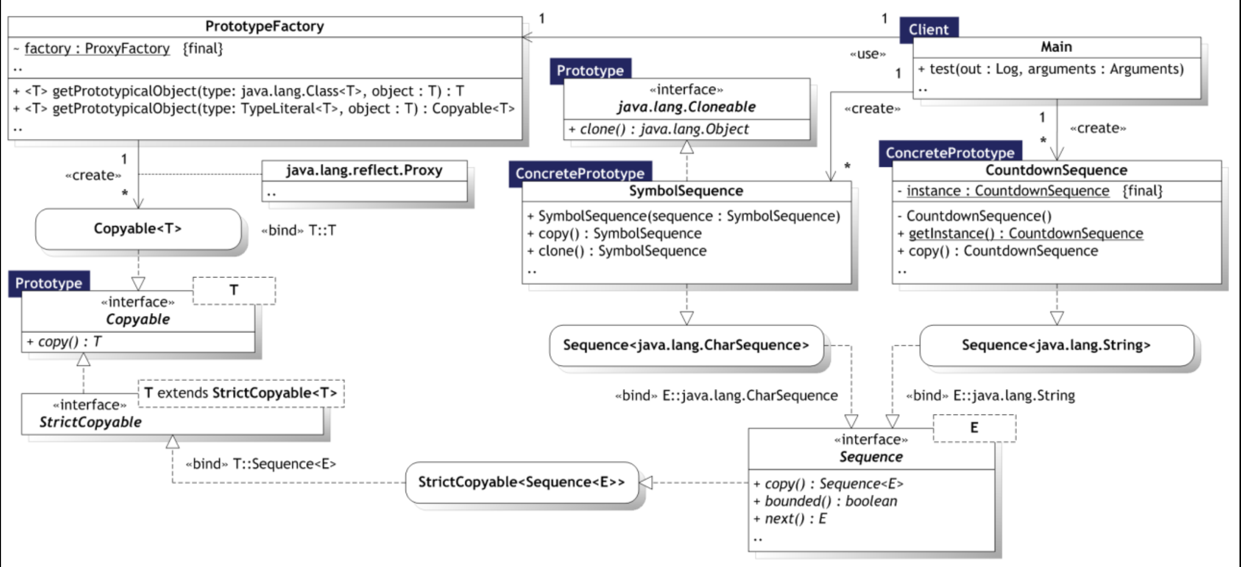
8.1.4.1. Intent

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype [Gamma95, p.117].

8.1.4.2. Structure

Figure 8.4 illustrates the Prototype implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

Figure 8.4 — Prototype UML Class diagram


PrototypeFactory
~ factory : ProxyFactory {final}
+ <T> getPrototypicalObject(type: java.lang.Class<T>, object : T) : T
+ <T> getPrototypicalObject(type: TypeLiteral<T>, object : T) : Copyable<T>
«create»
java.lang.reflect.Proxy
Copyable<T>
«bind» T::T
Prototype
«interface» Copyable
+ copy() : T
T
«interface» StrictCopyable
T extends StrictCopyable<T>
«bind» T::Sequence<E>
StrictCopyable<Sequence<E>>
Prototype
«interface» java.lang.Cloneable
+ clone() : java.lang.Object
Client
Main
+ test(out : Log, arguments : Arguments)
«use»
«create»
ConcretePrototype
SymbolSequence
+ SymbolSequence(sequence : SymbolSequence)
+ copy() : SymbolSequence
+ clone() : SymbolSequence
ConcretePrototype
CountdownSequence
- instance : CountdownSequence {final}
- CountdownSequence()
+ getInstance() : CountdownSequence
+ copy() : CountdownSequence
Sequence<java.lang.CharSequence>
Sequence<java.lang.String>
«bind» E::java.lang.CharSequence
«bind» E::java.lang.String
E
«interface» Sequence
+ copy() : Sequence<E>
+ bounded() : boolean
+ next() : E

8.1.4.3. Participants

Table 8.4 describes the Prototype participants in the words of Gamma et al. [Gamma95, p.119] and lists the corresponding implementations developed in the evaluation.

Table 8.4 — Prototype participants

Participant	Description	Implementation
Prototype	– Declares an interface for cloning itself.	`Copyable<T>`, `StrictCopyable<T>`
ConcretePrototype	– Implements the operation for cloning itself.	`SymbolSequence` and `CountdownSequence`, but also all other `meta.model.Sequence<E>` implementations
Client	– Creates a new object by asking a prototype to clone itself.	`Main`

8.1.4.4. Implementation

Using a prototype manager — The `abstractfactory.PrototypicalRegistry` class is a prototype manager. It also covers the Smalltalk example supplied in the Sample Code element [Gamma95, p.125]. **Implementing the *clone* operation** — The `SymbolSequence` class uses Java's built–in clone facility as well as the prototypical functionality offered by `StrictCopyable.copy()`. The `copy()` method used in `interpreter.Expression<E>` objects may encounter unmanageable cyclic references in expression trees. However, the `Expression.asSymbol(Context)` method illustrates that Java easily can handle cyclic references in graph traversals, so it is simply a matter of implementation. All `Copyable<T>` objects supply copy constructors by convention to perform deep–copy. **Initialising clones** — The `Copyable.copy()` method is parameter–less, but could employ arguments in a manner similar to `factorymethod.Factory<T>`. It is a design choice we do not recommend because it shifts focus more towards factories, but at flag to indicate deep copying may be appropriate. A copy of an uninitialised `interpreter.FlowExpression<E>` instance will have to be initialised before use.

8.1.5. Singleton

Singleton is a Creational pattern with Object scope. The `dk.rode.thesis.singleton` package contains the implementation.

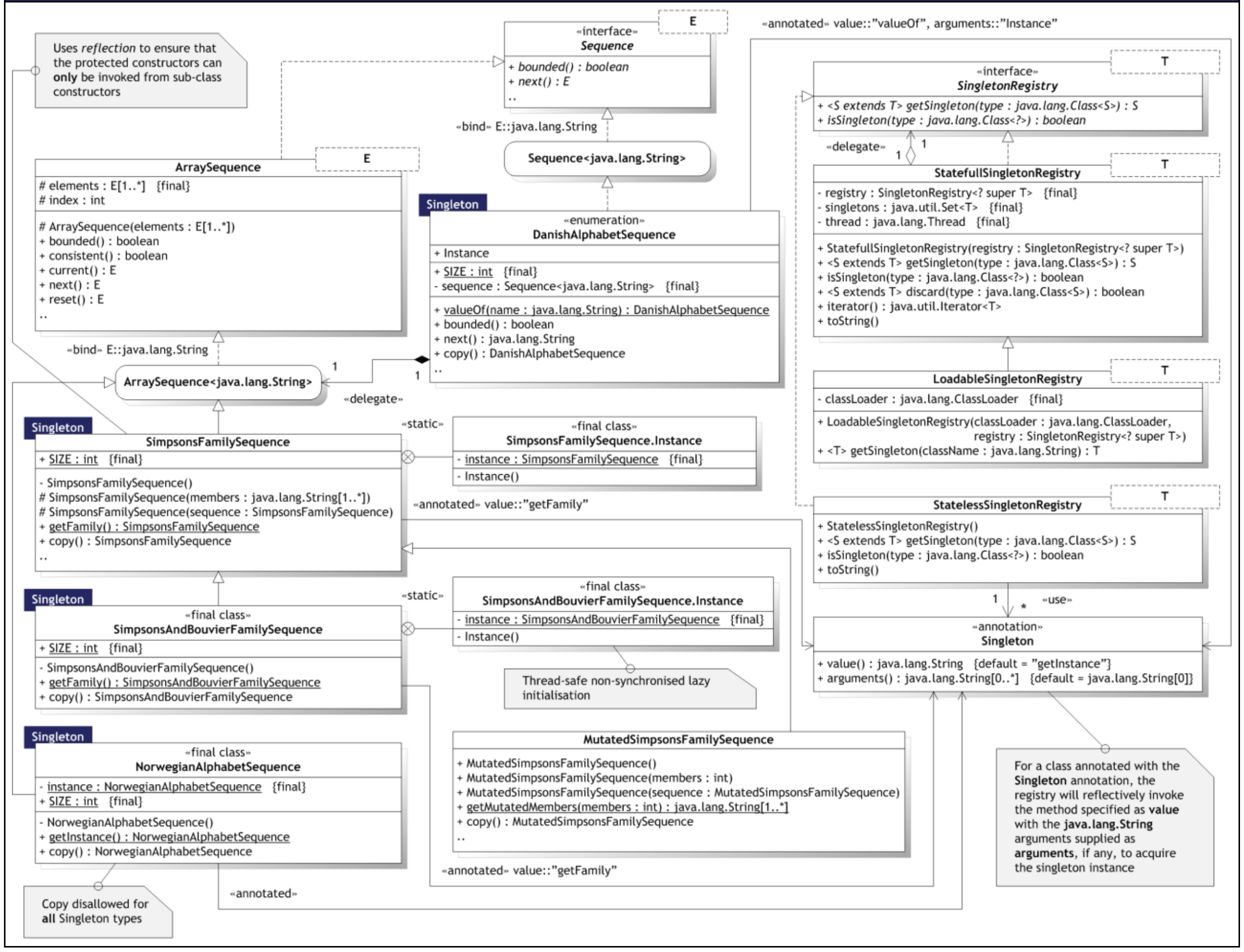
8.1.5.1. Intent

Ensure a class only has one instance, and provide a global point of access to it [Gamma95, p.127].

8.1.5.2. Structure

Figure 8.5 illustrates the Singleton implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

Figure 8.5 — Singleton UML Class diagram


Uses *reflection* to ensure that the protected constructors can **only** be invoked from sub-class constructors
«interface» Sequence
+ bounded() : boolean
+ next() : E
«bind» E::java.lang.String
Sequence<java.lang.String>
ArraySequence
# elements : E[1..*] {final}
# index : int
# ArraySequence(elements : E[1..*])
+ bounded() : boolean
+ consistent() : boolean
+ current() : E
+ next() : E
+ reset() : E
ArraySequence<java.lang.String>
«delegate»
Singleton
«enumeration» DanishAlphabetSequence
+ Instance
+ SIZE : int {final}
- sequence : Sequence<java.lang.String> {final}
+ valueOf(name : java.lang.String) : DanishAlphabetSequence
+ bounded() : boolean
+ next() : java.lang.String
+ copy() : DanishAlphabetSequence
«annotated» value::"valueOf", arguments::"Instance"
«interface» SingletonRegistry
+ <S extends T> getSingleton(type : java.lang.Class<S>) : S
+ isSingleton(type : java.lang.Class<?>) : boolean
«delegate»
StatefullSingletonRegistry
- registry : SingletonRegistry<? super T> {final}
- singletons : java.util.Set<T> {final}
- thread : java.lang.Thread {final}
+ StatefullSingletonRegistry(registry : SingletonRegistry<? super T>)
+ <S extends T> getSingleton(type : java.lang.Class<S>) : S
+ isSingleton(type : java.lang.Class<?>) : boolean
+ <S extends T> discard(type : java.lang.Class<S>) : boolean
+ iterator() : java.util.Iterator<T>
+ toString()
LoadableSingletonRegistry
- classLoader : java.lang.ClassLoader {final}
+ LoadableSingletonRegistry(classLoader : java.lang.ClassLoader, registry : SingletonRegistry<? super T>)
+ <T> getSingleton(className : java.lang.String) : T
StatelessSingletonRegistry
+ StatelessSingletonRegistry()
+ <S extends T> getSingleton(type : java.lang.Class<S>) : S
+ isSingleton(type : java.lang.Class<?>) : boolean
+ toString()
«use»
«annotation» Singleton
+ value() : java.lang.String {default = "getInstance"}
+ arguments() : java.lang.String[0..*] {default = java.lang.String[0]}
SimpsonsFamilySequence
+ SIZE : int {final}
- SimpsonsFamilySequence()
# SimpsonsFamilySequence(members : java.lang.String[1..*])
# SimpsonsFamilySequence(sequence : SimpsonsFamilySequence)
+ getFamily() : SimpsonsFamilySequence
+ copy() : SimpsonsFamilySequence
«static»
«final class» SimpsonsFamilySequence.Instance
- instance : SimpsonsFamilySequence {final}
- Instance()
«annotated» value::"getFamily"
«final class» SimpsonsAndBouvierFamilySequence
+ SIZE : int {final}
- SimpsonsAndBouvierFamilySequence()
+ getFamily() : SimpsonsAndBouvierFamilySequence
+ copy() : SimpsonsAndBouvierFamilySequence
«final class» SimpsonsAndBouvierFamilySequence.Instance
- instance : SimpsonsAndBouvierFamilySequence {final}
- Instance()
Thread-safe non-synchronised lazy initialisation
«final class» NorwegianAlphabetSequence
- instance : NorwegianAlphabetSequence {final}
+ SIZE : int {final}
- NorwegianAlphabetSequence()
+ getInstance() : NorwegianAlphabetSequence
+ copy() : NorwegianAlphabetSequence
MutatedSimpsonsFamilySequence
+ MutatedSimpsonsFamilySequence()
+ MutatedSimpsonsFamilySequence(members : int)
+ MutatedSimpsonsFamilySequence(sequence : MutatedSimpsonsFamilySequence)
+ getMutatedMembers(members : int) : java.lang.String[1..*]
+ copy() : MutatedSimpsonsFamilySequence
«annotated» value::"getFamily"
«annotated»
For a class annotated with the **Singleton** annotation, the registry will reflectively invoke the method specified as **value** with the **java.lang.String** arguments supplied as **arguments**, if any, to acquire the singleton instance
Copy disallowed for **all** Singleton types

8.1.5.3. Participants

Table 8.5 describes the Singleton participants in the words of Gamma et al. [Gamma95, p.119] and lists the corresponding implementations developed in the evaluation.

Table 8.5 — Singleton participants

Participant	Description	Implementation
Singleton	– Defines an *instance* operation that lets clients access its unique instance. *instance* is a class operation. – May be responsible for creating its own unique instance.	`DanishAlphabetSequence`, `NorwegianAlphabetSequence`, `SimpsonsFamilySequence`, `SimpsonsAndBouvierFamilySequence`, `MutatedSimpsonsFamilySequence`, `SmileySequence`

Though not described as a participant, Gamma et al. thoroughly describe and illustrate the use of *singleton registries* [Gamma95, p.130-132] in the Implementation element. Hence, singleton registries are also implemented in the evaluation. The name and minimal description of the "pseudo—participants" as listed in table 8.6 below are not defined by Gamma et al.

Table 8.6 — Additional Singleton entities

Name	Description	Implementation

SingletonRegistry	– Defines an operation to return **Singleton** instances.
	– May defer creation of a **Singleton** instance to the singleton type itself.
		`SingletonRegistry<T>`
		`@Singleton`

ConcreteSingletonRegistry	– Implements the ***SingletonRegistry*** interface. – May be a **Singleton**.	`StatelessSingletonRegistry<T>`, `StatefullSingletonRegistry<T>`, `LoadableSingletonRegistry<T>`

Notice, that we do not demand that a registry must store the different singleton types.

8.1.5.4. Implementation

Ensuring a unique instance — The `DanishAlphabetSequence` is implemented as a single enumeration constant. The `SimpsonsFamilySequence` uses a static method to lazily create the singleton instance in a thread—safe manner that does not require synchronisation. Both rely on automatic initialisation, but only when requested. Of the three problematic issues related to this in C++ [Gamma95, p.129-130], only parameterised singleton methods cannot be done in this fashion. They must be implemented as static synchronised methods. Though a parameterised singleton method is a design choice, we do not recommend it as it obfuscates the Singleton purpose. What happens if the singleton method is invoked again with a different value, for example. Overriding `new` as in Smalltalk is not possible in Java. **Sub—classing the Singleton class** — The `meta.log.LogFactory` class uses a system property to determine the type of log to use. Using conditional statements to decide the class is trivial. Java has no separation between header and object files, so compile—time linking to a different implementation is changing the entire class. The `SimpsonsFamilySequence` allows for sub—classing of the actual Singleton type, which to our knowledge is a novel approach. It does so by identifying the caller to simulate C++ friends, but in a type—safe fashion. A *Singleton Registry* as described by Gamma et al. [Gamma95, p.130-132] is defined via the `SingletonRegistry<T>` interface. The

`LoadableSingletonRegistry<T>` is an implementation that allows for type—safe dynamic loading of singleton types based on class names. The actual singleton creation is deferred to the singleton types themselves and no registration is required. This is clearly more flexible than the canonical implementation.

8.2. Structural Patterns

8.2.1. Adapter

Adapter is a Structural pattern with both Class and Object scope. The implementation is located in the `dk.rode.thesis.adapter` package.

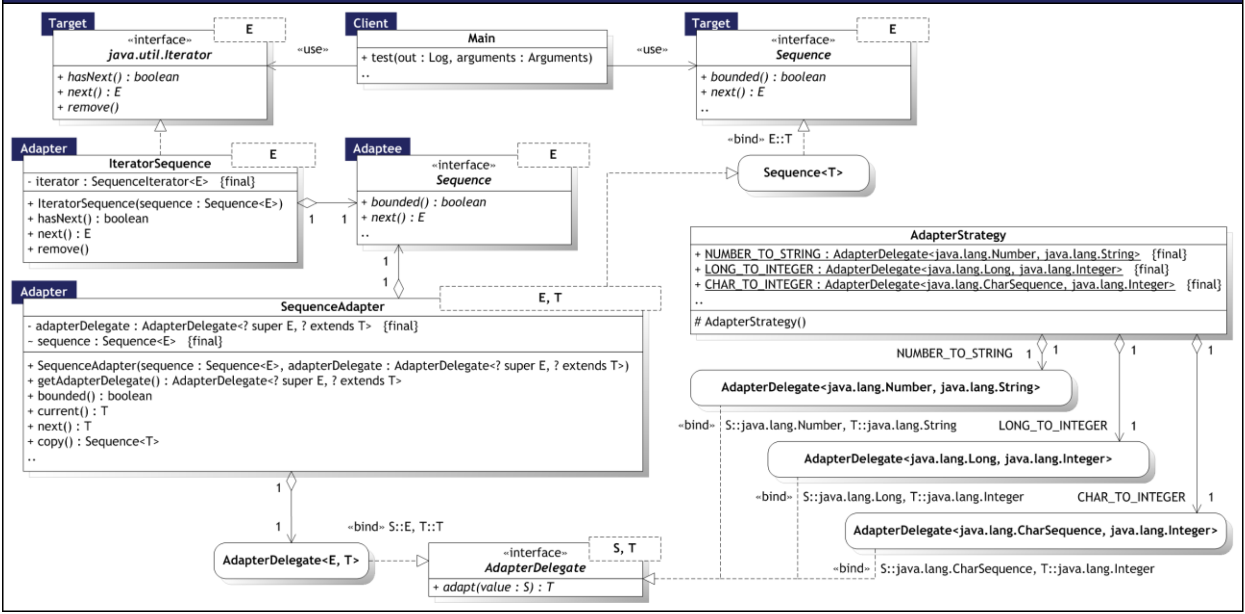
8.2.1.1. Intent

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces [Gamma95, p.139].

8.2.1.2. Structure

Figure 8.6 illustrates the Adapter implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

Figure 8.6 — Adapter UML Class diagram


Target
E
«interface»
java.util.Iterator
+ hasNext() : boolean
+ next() : E
+ remove()
«use»
Client
Main
+ test(out : Log, arguments : Arguments)
«use»
Target
E
«interface»
Sequence
+ bounded() : boolean
+ next() : E
«bind» E::T
Sequence<T>
Adapter
IteratorSequence
E
- iterator : SequenceIterator<E> {final}
+ IteratorSequence(sequence : Sequence<E>)
+ hasNext() : boolean
+ next() : E
+ remove()
Adaptee
E
«interface»
Sequence
+ bounded() : boolean
+ next() : E
1
1
1
1
Adapter
SequenceAdapter
E, T
- adapterDelegate : AdapterDelegate<? super E, ? extends T> {final}
- sequence : Sequence<E> {final}
+ SequenceAdapter(sequence : Sequence<E>, adapterDelegate : AdapterDelegate<? super E, ? extends T>)
+ getAdapterDelegate() : AdapterDelegate<? super E, ? extends T>
+ bounded() : boolean
+ current() : T
+ next() : T
+ copy() : Sequence<T>
AdapterStrategy
+ NUMBER_TO_STRING : AdapterDelegate<java.lang.Number, java.lang.String> {final}
+ LONG_TO_INTEGER : AdapterDelegate<java.lang.Long, java.lang.Integer> {final}
+ CHAR_TO_INTEGER : AdapterDelegate<java.lang.CharSequence, java.lang.Integer> {final}
# AdapterStrategy()
NUMBER_TO_STRING 1
1
1
1
AdapterDelegate<java.lang.Number, java.lang.String>
«bind» S::java.lang.Number, T::java.lang.String
LONG_TO_INTEGER 1
AdapterDelegate<java.lang.Long, java.lang.Integer>
«bind» S::java.lang.Long, T::java.lang.Integer
CHAR_TO_INTEGER 1
AdapterDelegate<java.lang.CharSequence, java.lang.Integer>
«bind» S::java.lang.CharSequence, T::java.lang.Integer
1
1
«bind» S::E, T::T
AdapterDelegate<E, T>
S, T
«interface»
AdapterDelegate
+ adapt(value : S) : T

8.2.1.3. Participants

Table 8.7 describes the Adapter participants in the words of Gamma et al. [Gamma95, p.141] and lists the corresponding implementations developed in the evaluation.

Table 8.7 — Adapter participants

Participant	Description	Implementation
Target	– Defines the domain–specific interface that **Client** use.	`meta.model.Sequence<E>`, `java.util.Iterator<E>`
Client	– Collaborates with objects conforming to the **Target** interface.	`Main`
Adaptee	– Defines an existing interface that needs adapting.	Any `Sequence<E>` implementation
Adapter	– Adapts the interface of **Adaptee** to the **Target** interface.	`SequenceAdapter<E,T>` (using `AdapterDelegate<S,T>`), `IteratorSequence<E>`

8.2.1.4. Implementation

Implementing class adapters in C++ — Java does not support multiple inheritance and thus not class adapters. Dynamic proxies can be used to simulate private implementation as illustrated in the `meta.reflect.ProxyFactory` class, but it still requires composition as in object adapters. **Pluggable adapters** — a) The set of abstract operations to use is a design choice. Sub–class implementation for adapter functionality is used by any adapter that utilises inner classes such as the `proxy.SequenceProxyFactory` class. b) The `SequenceAdapter<E,T>` class uses composition and forwards requests to the adaptee (sequence) stored internally. c) The `SequenceAdapter<E,T>` is parameterised and uses `AdapterDelegate<S,T>` instances.

8.2.2. Bridge

Bridge is a Structural pattern with Object scope. The `dk.rode.thesis.bridge` package contains the implementation.

8.2.2.1. Intent

Decouple an abstraction from its implementation so that the two can vary independently [Gamma95, p.151].

8.2.2.2. Structure

Figure 8.7 illustrates the Bridge implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

8.2.2.3. Participants

Table 8.8 describes the Bridge participants in the words of Gamma et al. [Gamma95, p.154] and lists the corresponding implementations developed in the evaluation.

8.2.2.4. Implementation

Only one implementor — Use of a single Implementor is a design choice. Java scope rules allow implementations to be hidden from clients or by using anonymous adapters for the Implementor implementation. **Creating the right Implementor object** — Design choice. **Sharing implementors** — The `Main` (test) class in the Bridge implementation utilises the `meta.reflect.ProxyFactory` to manage shared implementations. The sharing is handled via the Handle/Body idiom implemented in Java using dynamic proxies.

Using multiple inheritance — Java does not support multiple inheritance, but as Gamma et al. also note, this binds the implementation to the abstraction at compile–time. Composition can be used instead as illustrated by the `SequenceAbstraction<E>` class, which can be either fixed or changeable.

Figure 8.7 — Bridge UML Class diagram

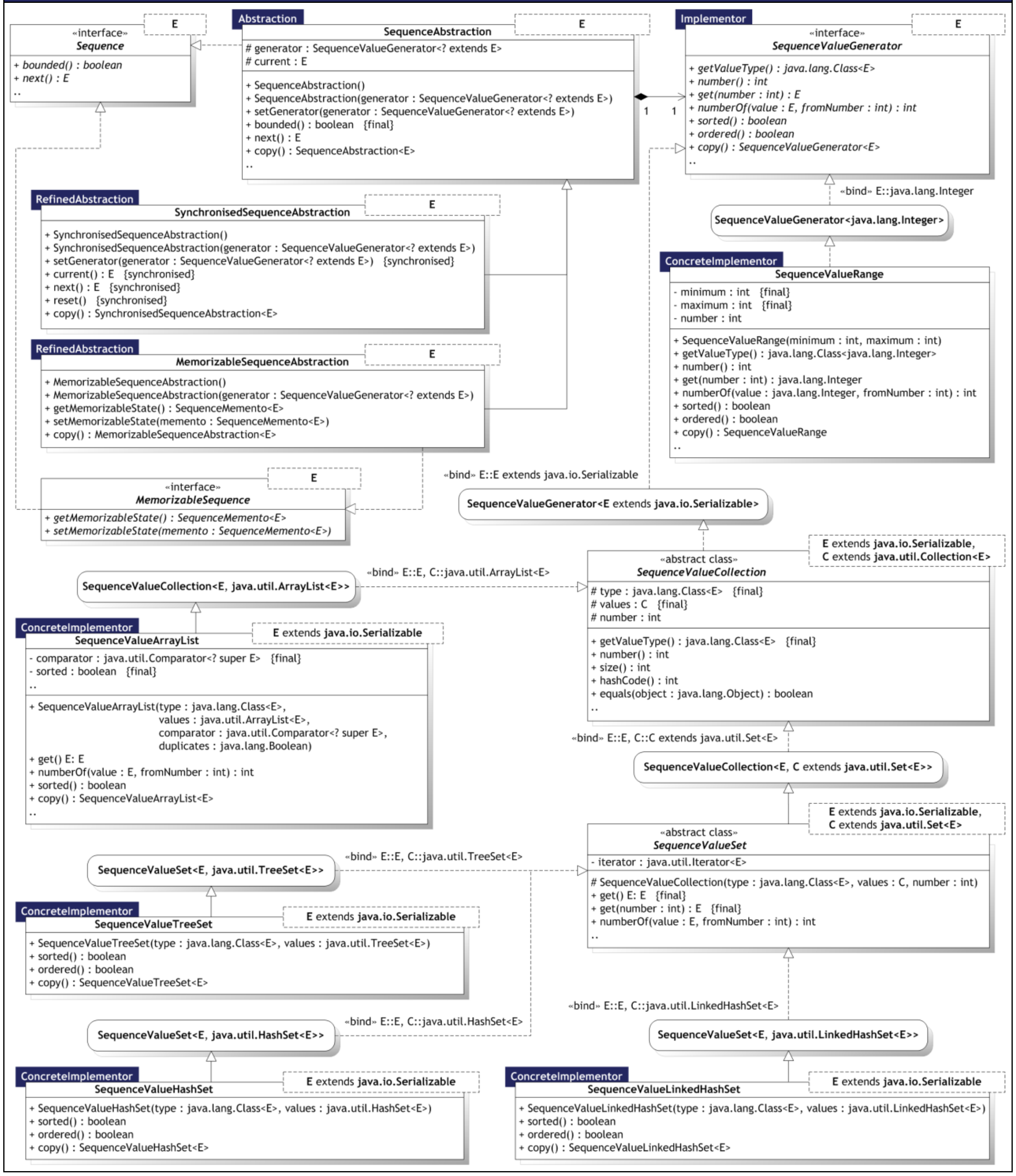

«interface» Sequence
E
+ bounded() : boolean
+ next() : E
Abstraction
SequenceAbstraction
# generator : SequenceValueGenerator<? extends E>
# current : E
+ SequenceAbstraction()
+ SequenceAbstraction(generator : SequenceValueGenerator<? extends E>)
+ setGenerator(generator : SequenceValueGenerator<? extends E>)
+ bounded() : boolean {final}
+ next() : E
+ copy() : SequenceAbstraction<E>
1
1
Implementor
«interface» SequenceValueGenerator
+ getValueType() : java.lang.Class<E>
+ number() : int
+ get(number : int) : E
+ numberOf(value : E, fromNumber : int) : int
+ sorted() : boolean
+ ordered() : boolean
+ copy() : SequenceValueGenerator<E>
«bind» E::java.lang.Integer
SequenceValueGenerator<java.lang.Integer>
RefinedAbstraction
SynchronisedSequenceAbstraction
+ SynchronisedSequenceAbstraction()
+ SynchronisedSequenceAbstraction(generator : SequenceValueGenerator<? extends E>)
+ setGenerator(generator : SequenceValueGenerator<? extends E>) {synchronised}
+ current() : E {synchronised}
+ next() : E {synchronised}
+ reset() {synchronised}
+ copy() : SynchronisedSequenceAbstraction<E>
ConcreteImplementor
SequenceValueRange
- minimum : int {final}
- maximum : int {final}
- number : int
+ SequenceValueRange(minimum : int, maximum : int)
+ getValueType() : java.lang.Class<java.lang.Integer>
+ number() : int
+ get(number : int) : java.lang.Integer
+ numberOf(value : java.lang.Integer, fromNumber : int) : int
+ sorted() : boolean
+ ordered() : boolean
+ copy() : SequenceValueRange
RefinedAbstraction
MemorizableSequenceAbstraction
+ MemorizableSequenceAbstraction()
+ MemorizableSequenceAbstraction(generator : SequenceValueGenerator<? extends E>)
+ getMemorizableState() : SequenceMemento<E>
+ setMemorizableState(memento : SequenceMemento<E>)
+ copy() : MemorizableSequenceAbstraction<E>
«interface» MemorizableSequence
+ getMemorizableState() : SequenceMemento<E>
+ setMemorizableState(memento : SequenceMemento<E>)
«bind» E::E extends java.io.Serializable
SequenceValueGenerator<E extends java.io.Serializable>
«abstract class» SequenceValueCollection
E extends java.io.Serializable, C extends java.util.Collection<E>
# type : java.lang.Class<E> {final}
# values : C {final}
# number : int
+ getValueType() : java.lang.Class<E> {final}
+ number() : int
+ size() : int
+ hashCode() : int
+ equals(object : java.lang.Object) : boolean
«bind» E::E, C::java.util.ArrayList<E>
SequenceValueCollection<E, java.util.ArrayList<E>>
ConcreteImplementor
SequenceValueArrayList
E extends java.io.Serializable
- comparator : java.util.Comparator<? super E> {final}
- sorted : boolean {final}
+ SequenceValueArrayList(type : java.lang.Class<E>, values : java.util.ArrayList<E>, comparator : java.util.Comparator<? super E>, duplicates : java.lang.Boolean)
+ get() E: E
+ numberOf(value : E, fromNumber : int) : int
+ sorted() : boolean
+ copy() : SequenceValueArrayList<E>
«bind» E::E, C::C extends java.util.Set<E>
SequenceValueCollection<E, C extends java.util.Set<E>>
«abstract class» SequenceValueSet
E extends java.io.Serializable, C extends java.util.Set<E>
- iterator : java.util.Iterator<E>
# SequenceValueCollection(type : java.lang.Class<E>, values : C, number : int)
+ get() E: E {final}
+ get(number : int) : E {final}
+ numberOf(value : E, fromNumber : int) : int
«bind» E::E, C::java.util.TreeSet<E>
SequenceValueSet<E, java.util.TreeSet<E>>
ConcreteImplementor
SequenceValueTreeSet
E extends java.io.Serializable
+ SequenceValueTreeSet(type : java.lang.Class<E>, values : java.util.TreeSet<E>)
+ sorted() : boolean
+ ordered() : boolean
+ copy() : SequenceValueTreeSet<E>
«bind» E::E, C::java.util.HashSet<E>
SequenceValueSet<E, java.util.HashSet<E>>
«bind» E::E, C::java.util.LinkedHashSet<E>
SequenceValueSet<E, java.util.LinkedHashSet<E>>
ConcreteImplementor
SequenceValueHashSet
E extends java.io.Serializable
+ SequenceValueHashSet(type : java.lang.Class<E>, values : java.util.HashSet<E>)
+ sorted() : boolean
+ ordered() : boolean
+ copy() : SequenceValueHashSet<E>
ConcreteImplementor
SequenceValueLinkedHashSet
E extends java.io.Serializable
+ SequenceValueLinkedHashSet(type : java.lang.Class<E>, values : java.util.LinkedHashSet<E>)
+ sorted() : boolean
+ ordered() : boolean
+ copy() : SequenceValueLinkedHashSet<E>

Table 8.8 — Bridge participants

Participant	Description	Implementation
Abstraction	– Defines the abstraction's interface. – Maintains a reference to an object of type **Implementor**.	`SequenceAbstraction<E>`
RefinedAbstraction	– Extends the interface defined by **Abstraction**.	`SynchronisedSequenceAbstraction<E>`, `MemorizableSequenceAbstraction<E>`

Implementor	– Defines the interface for implementation classes. This interface does not have to correspond exactly to **Abstraction**'s interface; in fact, the two interfaces can be quite different. Typically, the **Implementor** interface provides only primitive operations, and **Abstraction** defines higher–level operations based on these primitives.	`SequenceValueGenerator<E>`
		`SequenceValueCollection<E,C>`, `SequenceValueSet<E,C>`

ConcreteImplementor	– Implements the **Implementor** interface and defines its concrete implementation.	`SequenceValueArrayList<E>`, `SequenceValueHashSet<E>`, `SequenceValueLinkedHashSet<E>`, `SequenceValueTreeSet<E>`, `SequenceValueRange`

8.2.3. Composite

Composite is a Structural pattern with Object scope. The `dk.rode.thesis.composite` package contains the implementation.

8.2.3.1. Intent

Compose objects into tree structures to represent part–whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly [Gamma95, p.163].

8.2.3.2. Structure

Figure 8.8 illustrates the Composite implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

8.2.3.3. Participants

Table 8.9 describes the Composite participants in the words of Gamma et al. [Gamma95, p.165] and lists the corresponding implementations developed in the evaluation.

8.2.3.4. Implementation

Explicit parent references — Design choice; not implemented here. **Sharing components** — Design choice. **Maximizing the Component interface** — Design choice. **Declaring the child management operations** — The Composite implementation opts for type safety over transparency as discussed by Gamma et al. [Gamma95, p.167-168]. Child management is defined in the Composite (`CompositeSequence<E>`) participant because it can never make sense for Leaf components (`Sequence<E>`). This makes sense in Java because unlike C++, the `instanceof` operator provides a safe way to cast to a given type. **Should Component implement a list of Components?** — Children are stored in the Composite participant. **Child ordering** — The

`CompositeStrategy` enumeration defines strategies for traversing the composite structure, such as depth–first and breath–first. **Caching to improve performance** — Design choice. The strategies defined here allow retrieved lists to be reused without the need to traverse the composite structure again. **Who should delete components?** — Java employs garbage collection. **What is the best data structure for storing components?** — This is a design choice. `AbstractCompositeSequence<E>` uses a `java.util.RandomAccess` list for fast traversal.

Figure 8.8 — Composite UML Class diagram

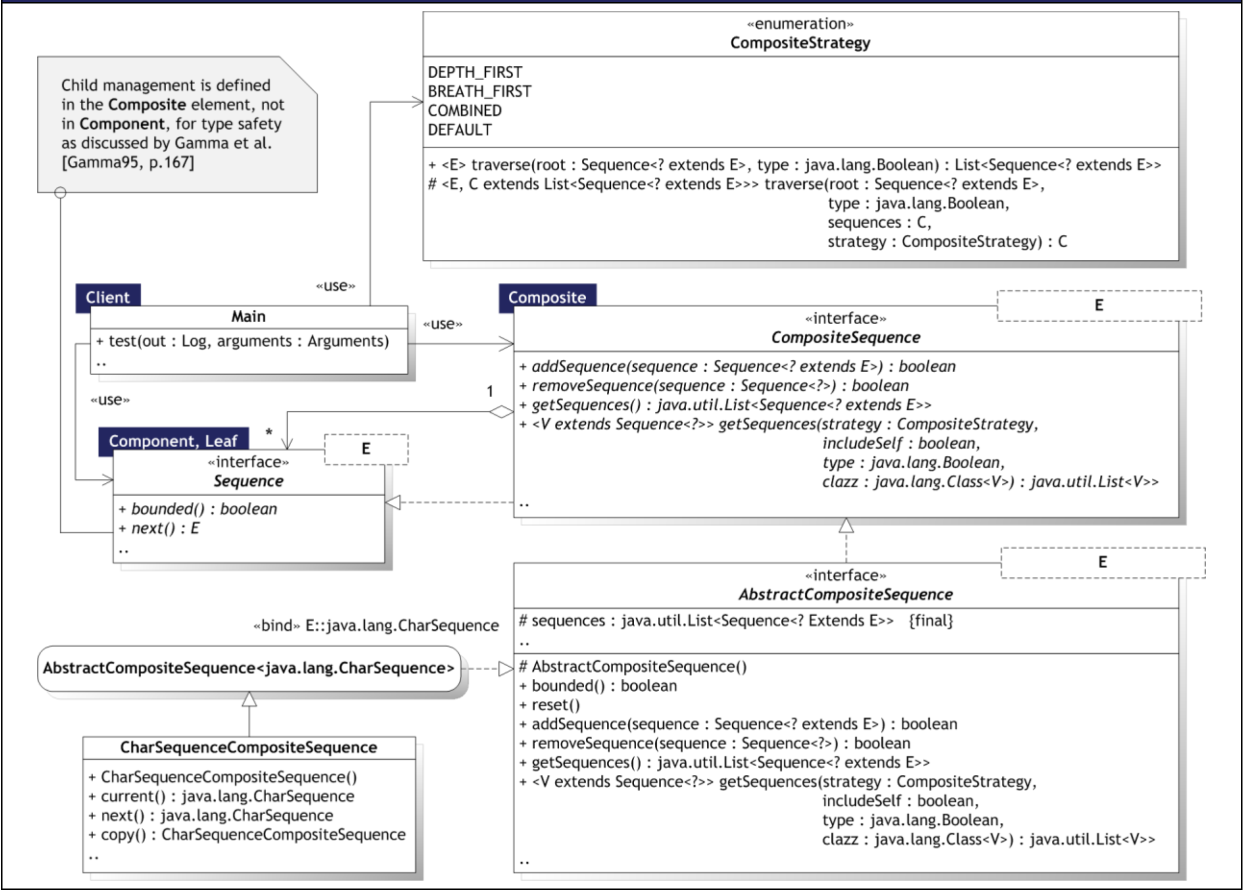

Child management is defined in the Composite element, not in Component, for type safety as discussed by Gamma et al. [Gamma95, p.167]
«enumeration»
CompositeStrategy
DEPTH_FIRST
BREATH_FIRST
COMBINED
DEFAULT
+ <E> traverse(root : Sequence<? extends E>, type : java.lang.Boolean) : List<Sequence<? extends E>>
# <E, C extends List<Sequence<? extends E>>> traverse(root : Sequence<? extends E>, type : java.lang.Boolean, sequences : C, strategy : CompositeStrategy) : C
«use»
Client
Main
+ test(out : Log, arguments : Arguments)
..
«use»
Composite
«interface»
CompositeSequence
E
+ addSequence(sequence : Sequence<? extends E>) : boolean
+ removeSequence(sequence : Sequence<?>) : boolean
+ getSequences() : java.util.List<Sequence<? extends E>>
+ <V extends Sequence<?>> getSequences(strategy : CompositeStrategy, includeSelf : boolean, type : java.lang.Boolean, clazz : java.lang.Class<V>) : java.util.List<V>>
..
«use»
1
*
Component, Leaf
«interface»
Sequence
E
+ bounded() : boolean
+ next() : E
..
«interface»
AbstractCompositeSequence
E
# sequences : java.util.List<Sequence<? Extends E>> {final}
..
# AbstractCompositeSequence()
+ bounded() : boolean
+ reset()
+ addSequence(sequence : Sequence<? extends E>) : boolean
+ removeSequence(sequence : Sequence<?>) : boolean
+ getSequences() : java.util.List<Sequence<? extends E>>
+ <V extends Sequence<?>> getSequences(strategy : CompositeStrategy, includeSelf : boolean, type : java.lang.Boolean, clazz : java.lang.Class<V>) : java.util.List<V>>
..
«bind» E::java.lang.CharSequence
AbstractCompositeSequence<java.lang.CharSequence>
CharSequenceCompositeSequence
+ CharSequenceCompositeSequence()
+ current() : java.lang.CharSequence
+ next() : java.lang.CharSequence
+ copy() : CharSequenceCompositeSequence
..

Table 8.9 — Composite participants

Participant	Description	Implementation
Component	– Declares the interface for objects in the composition. – Implements default behaviour for the interface common to all classes, as appropriate. – Declares an interface for accessing and managing its child components. – Defines an interface for accessing a component's parent in the recursive structure, and implements it if that is appropriate (optional).	`meta.model.Sequence<E>`
Leaf	– Represents a leaf object in the composition. A leaf	Any `Sequence<E>` implementation

Table 8.9 — Composite participants

Participant	Description	Implementation
	has no children. – Defines behaviour for primitive objects in the composition.	

Composite	– Defines behaviour for components having children.
	– Stores child components.
	– Implements child—related operations in the **Component** interface.
		`CompositeSequence<E>`, `AbstractCompositeSequence<E>`, `CharSequenceCompositeSequence`
		`CompositeStrategy`

Client	– Manipulates objects in the composition through the **Component** interface.	`Main`

8.2.4. Decorator

Decorator is a Structural pattern with Object scope. The `dk.rode.thesis.decorator` package contains the implementation.

8.2.4.1. Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub—classing for extending functionality [Gamma95, p.175].

8.2.4.2. Structure

Figure 8.9 illustrates the Decorator implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

8.2.4.3. Participants

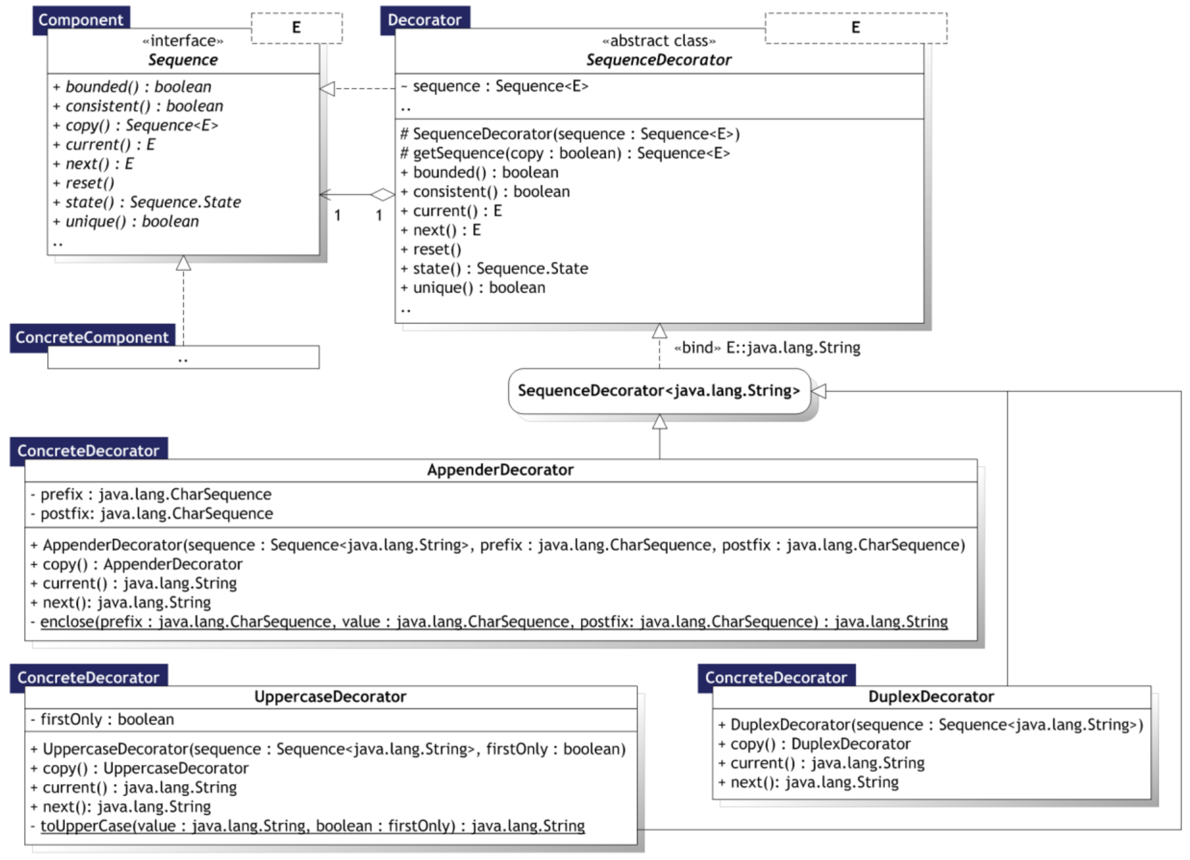
Table 8.10 describes the Decorator participants in the words of Gamma et al. [Gamma95, p.177] and lists the corresponding implementations developed in the evaluation.

Table 8.10 — Decorator participants

Participant	Description	Implementation
Component	– Defines the interface for objects that can have responsibilities added to them dynamically.	`meta.model.Sequence<E>`
ConcreteComponent	– Defines an object to which additional responsibilities can be attached.	Any `Sequence<E>` implementation
Decorator	– Maintains a reference to a **Component** object and defines an interface that conforms to **Component**'s interface.	`SequenceDecorator<E>`
ConcreteDecorator	– Adds responsibilities to the **Component**.	`AppenderDecorator`, `DuplexDecorator`, `UppercaseDecorator`

The `SequenceDecorator<E>` class is used in several other pattern implementations as well, including Iterator, Proxy, and Visitor.

Figure 8.9 — Decorator UML Class diagram


Component
E
«interface»
Sequence
+ bounded() : boolean
+ consistent() : boolean
+ copy() : Sequence<E>
+ current() : E
+ next() : E
+ reset()
+ state() : Sequence.State
+ unique() : boolean
..
Decorator
E
«abstract class»
SequenceDecorator
~ sequence : Sequence<E>
..
# SequenceDecorator(sequence : Sequence<E>)
# getSequence(copy : boolean) : Sequence<E>
+ bounded() : boolean
+ consistent() : boolean
+ current() : E
+ next() : E
+ reset()
+ state() : Sequence.State
+ unique() : boolean
..
1
1
ConcreteComponent
..
«bind» E::java.lang.String
SequenceDecorator<java.lang.String>
ConcreteDecorator
AppenderDecorator
- prefix : java.lang.CharSequence
- postfix: java.lang.CharSequence
+ AppenderDecorator(sequence : Sequence<java.lang.String>, prefix : java.lang.CharSequence, postfix : java.lang.CharSequence)
+ copy() : AppenderDecorator
+ current() : java.lang.String
+ next(): java.lang.String
- enclose(prefix : java.lang.CharSequence, value : java.lang.CharSequence, postfix: java.lang.CharSequence) : java.lang.String
ConcreteDecorator
UppercaseDecorator
- firstOnly : boolean
+ UppercaseDecorator(sequence : Sequence<java.lang.String>, firstOnly : boolean)
+ copy() : UppercaseDecorator
+ current() : java.lang.String
+ next(): java.lang.String
- toUpperCase(value : java.lang.String, boolean : firstOnly) : java.lang.String
ConcreteDecorator
DuplexDecorator
+ DuplexDecorator(sequence : Sequence<java.lang.String>)
+ copy() : DuplexDecorator
+ current() : java.lang.String
+ next(): java.lang.String

8.2.4.4. Implementation

Interface conformance — Implementing a common interface is sufficient in Java; inheritance is not required. `SequenceDecorator<E>` implements `Sequence<E>` and inherits `meta.model.AbstractSequence<E>`, but can decorate any sequence type. **Omitting the abstract decorator class** — Design choice. **Keeping Component classes lightweight** — Design choice, but one that cannot always be controlled in case API classes, for example, have to be decorated. **Changing the skin of an object vs. changing the guts** — Design choice. Abstract decorator types can make it easier to implement multiple concrete decorator types as illustrated with the abstract `SequenceDecorator<E>` class.

8.2.5. Facade

Facade is Structural with Object scope. The `dk.rode.thesis.facade` package contains the implementation.

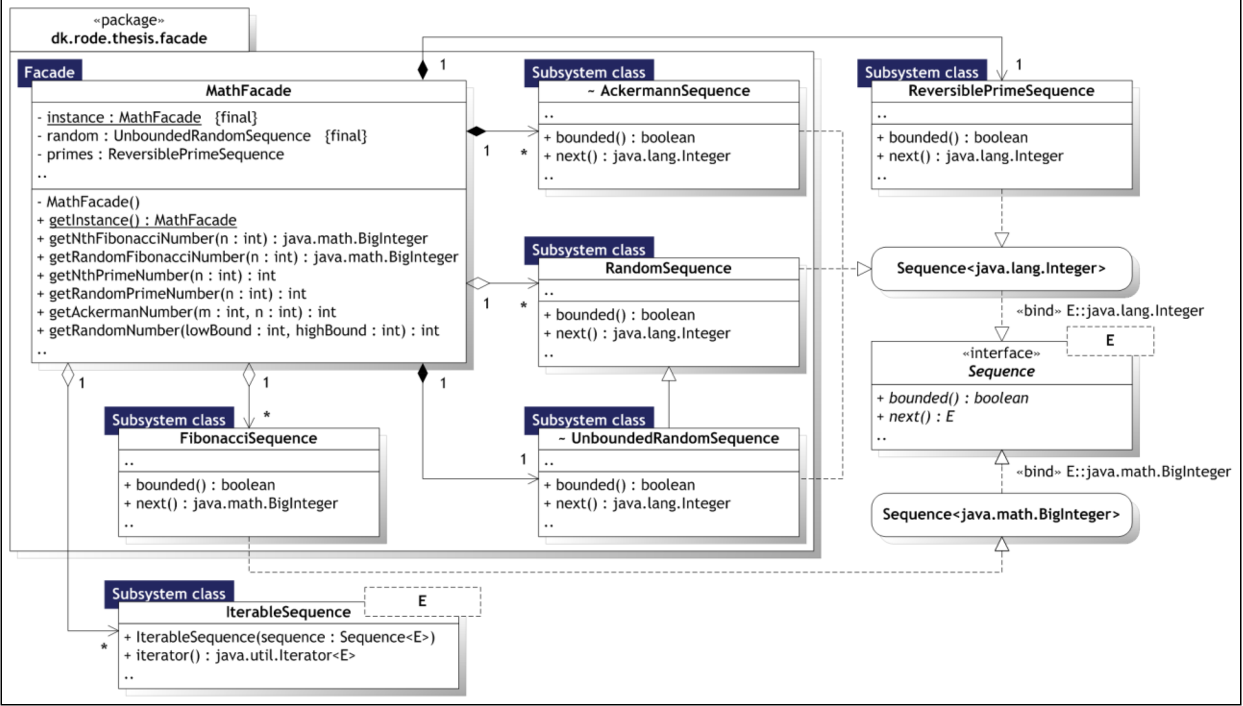
8.2.5.1. Intent

Provide a unified interface to a set of interfaces in a sub—system. Facade defines a higher—level interface that makes the sub—system easier to use [Gamma95, p.185].

8.2.5.2. Structure

Figure 8.10 illustrates the Facade implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

Figure 8.10 — Facade UML Class diagram


«package» dk.rode.thesis.facade
Facade
MathFacade
- instance : MathFacade {final}
- random : UnboundedRandomSequence {final}
- primes : ReversiblePrimeSequence
- MathFacade()
+ getInstance() : MathFacade
+ getNthFibonacciNumber(n : int) : java.math.BigInteger
+ getRandomFibonacciNumber(n : int) : java.math.BigInteger
+ getNthPrimeNumber(n : int) : int
+ getRandomPrimeNumber(n : int) : int
+ getAckermanNumber(m : int, n : int) : int
+ getRandomNumber(lowBound : int, highBound : int) : int
Subsystem class
~ AckermannSequence
+ bounded() : boolean
+ next() : java.lang.Integer
Subsystem class
ReversiblePrimeSequence
+ bounded() : boolean
+ next() : java.lang.Integer
Subsystem class
RandomSequence
+ bounded() : boolean
+ next() : java.lang.Integer
Sequence<java.lang.Integer>
«bind» E::java.lang.Integer
E
«interface» Sequence
+ bounded() : boolean
+ next() : E
«bind» E::java.math.BigInteger
Sequence<java.math.BigInteger>
Subsystem class
FibonacciSequence
+ bounded() : boolean
+ next() : java.math.BigInteger
Subsystem class
~ UnboundedRandomSequence
+ bounded() : boolean
+ next() : java.lang.Integer
Subsystem class
IterableSequence
E
+ IterableSequence(sequence : Sequence<E>)
+ iterator() : java.util.Iterator<E>
1
*

8.2.5.3. Participants

Table 8.11 describes the Facade participants in the words of Gamma et al. [Gamma95, p.187] and lists the corresponding implementations developed in the evaluation.

Table 8.11 — Facade participants

Participant	Description	Implementation
Facade	– Knows which sub—system classes are responsible for a request. – Delegates client requests to appropriate sub—system objects.	`MathFacade`
Subsystem Class	– Implement sub—system functionality. – Handle work assigned by **Facade** object. – Have no knowledge of the facade; that is, they keep no reference to it.	`AckermannSequence`, `FibonacciSequence`, `RandomSequence`, `UnboundedRandomSequence`, `iterator.IterableSequence<E>`, `state.ReversiblePrimeSequence`

8.2.5.4. Implementation

Reducing client/sub—system coupling — Design choice, but may require factory creation. The `MathFacade` class is implemented as a Singleton. **Public vs. private sub—system classes** — Design choice. Packages in Java can provide encapsulation and implicitly information hiding (access modifiers) for sub—systems. `MathFacade`, `FibonacciSequence`, and `RandomSequence` represent public classes, while

`AckermannSequence` and `UnboundedRandomSequence` are private sub—system classes and are as such declared package private. Sub—system classes from other packages must be public to be utilised.

8.2.6. Flyweight

Flyweight is a Structural pattern with Object scope. The `dk.rode.thesis.flyweight` package contains the implementation.

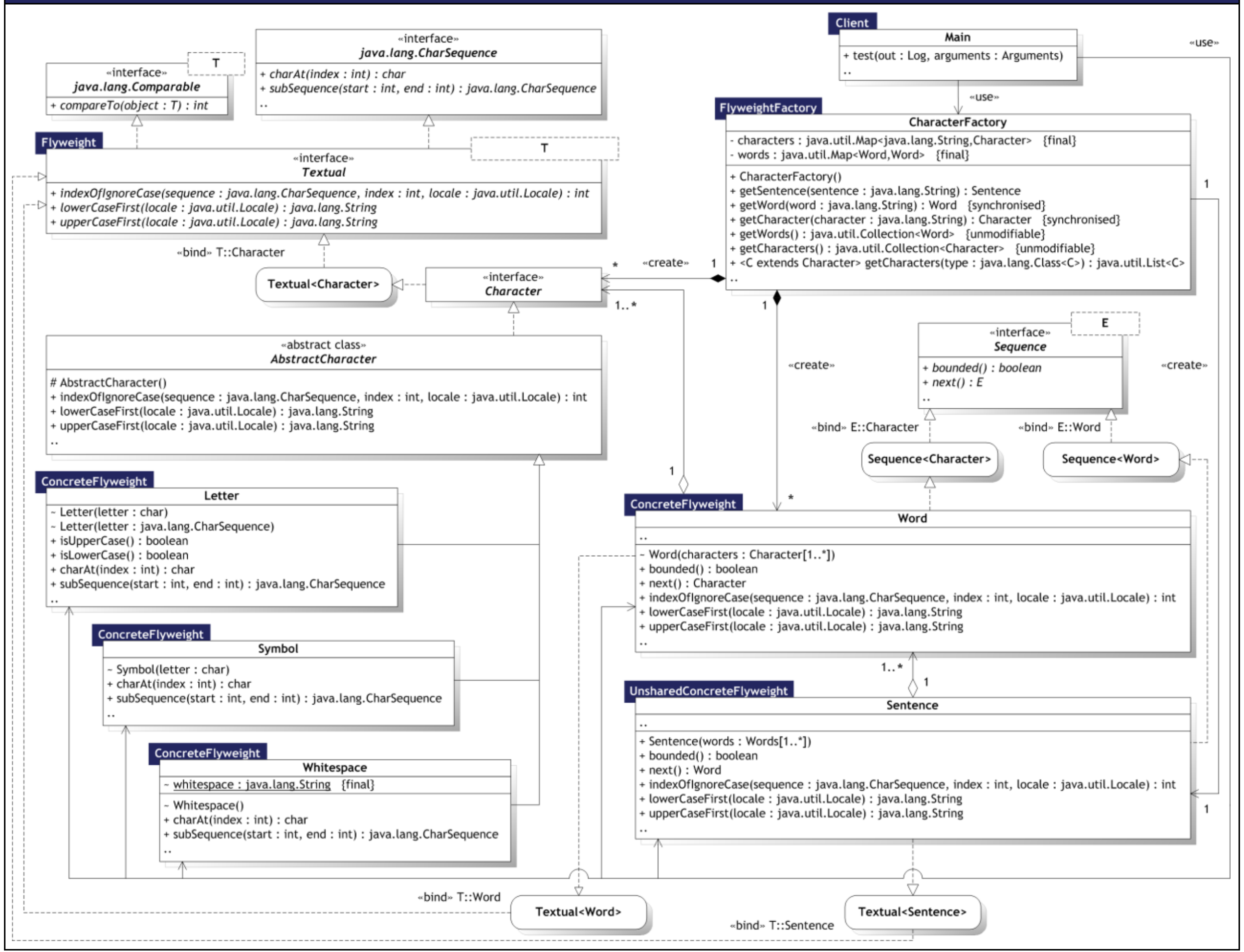
8.2.6.1. Intent

Use sharing to support large numbers of fine—grained objects efficiently [Gamma95, p.195].

8.2.6.2. Structure

Figure 8.11 illustrates the Flyweight implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

Figure 8.11 — Flyweight UML Class diagram


Client
Main
+ test(out : Log, arguments : Arguments)
«use»
«use»
FlyweightFactory
CharacterFactory
- characters : java.util.Map<java.lang.String,Character> {final}
- words : java.util.Map<Word,Word> {final}
+ CharacterFactory()
+ getSentence(sentence : java.lang.String) : Sentence
+ getWord(word : java.lang.String) : Word {synchronised}
+ getCharacter(character : java.lang.String) : Character {synchronised}
+ getWords() : java.util.Collection<Word> {unmodifiable}
+ getCharacters() : java.util.Collection<Character> {unmodifiable}
+ <C extends Character> getCharacters(type : java.lang.Class<C>) : java.util.List<C>
«interface» java.lang.Comparable
T
+ compareTo(object : T) : int
«interface» java.lang.CharSequence
+ charAt(index : int) : char
+ subSequence(start : int, end : int) : java.lang.CharSequence
Flyweight
«interface» Textual
T
+ indexOfIgnoreCase(sequence : java.lang.CharSequence, index : int, locale : java.util.Locale) : int
+ lowerCaseFirst(locale : java.util.Locale) : java.lang.String
+ upperCaseFirst(locale : java.util.Locale) : java.lang.String
«bind» T::Character
Textual<Character>
«interface» Character
«create»
*
1
1..*
1
«abstract class» AbstractCharacter
# AbstractCharacter()
+ indexOfIgnoreCase(sequence : java.lang.CharSequence, index : int, locale : java.util.Locale) : int
+ lowerCaseFirst(locale : java.util.Locale) : java.lang.String
+ upperCaseFirst(locale : java.util.Locale) : java.lang.String
«create»
«interface» Sequence
E
+ bounded() : boolean
+ next() : E
«create»
«bind» E::Character
«bind» E::Word
Sequence<Character>
Sequence<Word>
ConcreteFlyweight
Letter
- Letter(letter : char)
- Letter(letter : java.lang.CharSequence)
+ isUpperCase() : boolean
+ isLowerCase() : boolean
+ charAt(index : int) : char
+ subSequence(start : int, end : int) : java.lang.CharSequence
ConcreteFlyweight
Word
- Word(characters : Character[1..*])
+ bounded() : boolean
+ next() : Character
+ indexOfIgnoreCase(sequence : java.lang.CharSequence, index : int, locale : java.util.Locale) : int
+ lowerCaseFirst(locale : java.util.Locale) : java.lang.String
+ upperCaseFirst(locale : java.util.Locale) : java.lang.String
ConcreteFlyweight
Symbol
- Symbol(letter : char)
+ charAt(index : int) : char
+ subSequence(start : int, end : int) : java.lang.CharSequence
1..*
1
UnsharedConcreteFlyweight
Sentence
+ Sentence(words : Words[1..*])
+ bounded() : boolean
+ next() : Word
+ indexOfIgnoreCase(sequence : java.lang.CharSequence, index : int, locale : java.util.Locale) : int
+ lowerCaseFirst(locale : java.util.Locale) : java.lang.String
+ upperCaseFirst(locale : java.util.Locale) : java.lang.String
1
ConcreteFlyweight
Whitespace
- whitespace : java.lang.String {final}
- Whitespace()
+ charAt(index : int) : char
+ subSequence(start : int, end : int) : java.lang.CharSequence
«bind» T::Word
Textual<Word>
Textual<Sentence>
«bind» T::Sentence

8.2.6.3. Participants

Table 8.12 describes the Flyweight participants in the words of Gamma et al. [Gamma95, p.198-199] and lists the corresponding implementations developed in the evaluation.

Table 8.12 — Flyweight participants

Participant	Description	Implementation
Flyweight	– Declares an interface through which flyweights can receive and act on extrinsic state.	`Textual<T>` `Character`
ConcreteFlyweight	– Implements the **Flyweight** interface and adds storage for intrinsic state, if any. A **ConcreteFlyweight** object must be sharable. Any state it stores must be intrinsic; that is, it must be independent of the **ConcreteFlyweight** object's context.	`Word`, `AbstractCharacter`, `Letter`, `Symbol`, `Whitespace`
UnsharedConcreteFlyweight	– Not all **Flyweight** sub–classes need to be shared. The **Flyweight** interface *enables* sharing; it does not enforce it. It is common for **UnsharedConcreteFlyweight** objects to have **ConcreteFlyweight** objects as children at some level in the flyweight object structure.	`Sentence`
FlyweightFactory	– Creates and manages flyweight objects. – Ensures that flyweights are shared properly. When a client requests a flyweight, the **FlyweightFactory** object supplies an existing instance or creates one, if none exists.	`CharacterFactory`
Client	– Maintains a reference to flyweight(s). – Computes or stores the extrinsic state of flyweight(s).	`Main`

8.2.6.4. Implementation

Removing extrinsic state — Design choice. Here, `java.util.Locale` represents extrinsic state for sentence structures, because it rarely makes sense to store a locale with each character, word, or even sentence. **Managing shared objects** — The `CharacterFactory` class uses a `java.util.HashMap<K,V>` to store flyweights, which are created on demand only in a thread–safe manner. The `java.lang.Object.hashCode()` and `java.lang.Object.equals(Object)` methods makes it very easy to handle flyweights using the Java Collections framework. Purging of old flyweights is a design choice.

8.2.7. Proxy

Proxy is a Structural pattern with Object scope. The implementation is located in the `dk.rode.thesis.proxy` package.

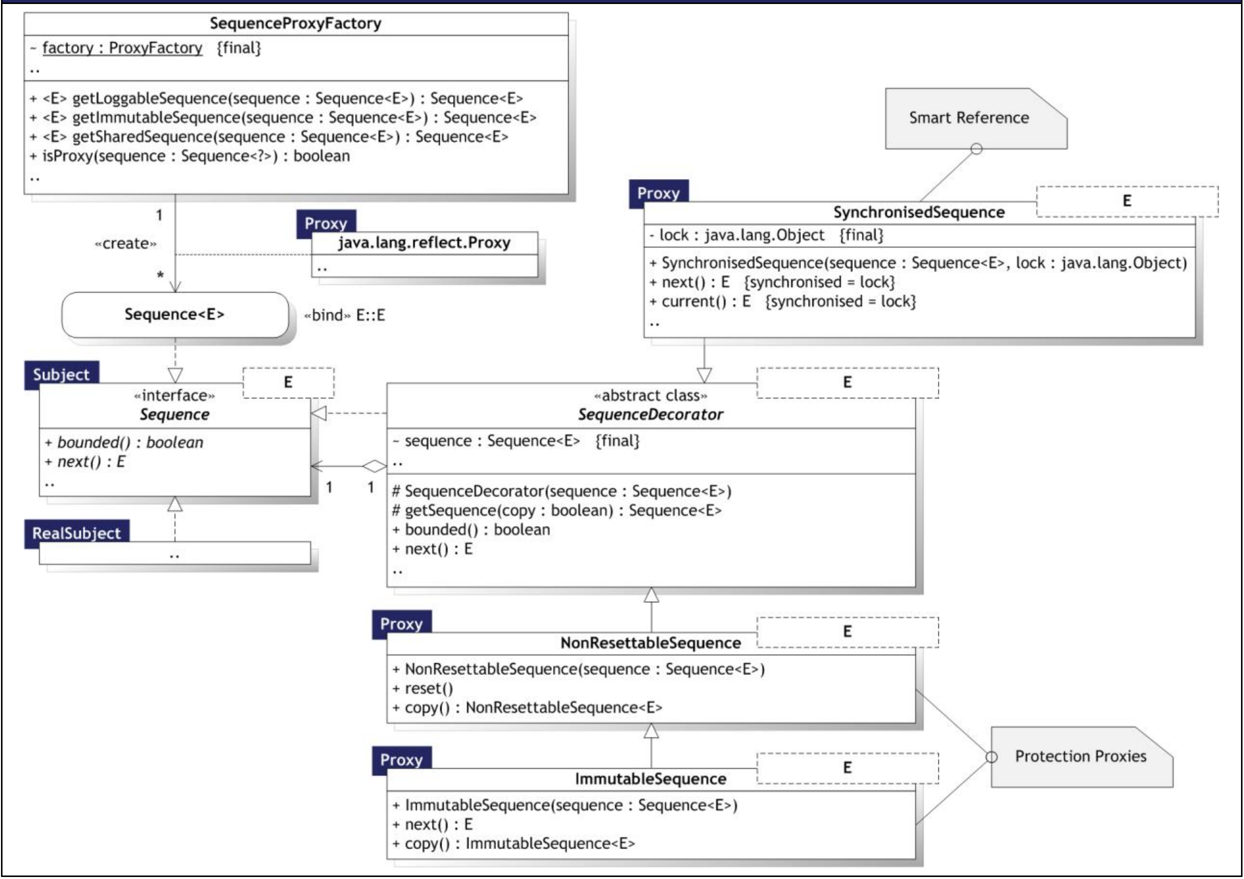
8.2.7.1. Intent

Provide a surrogate placeholder for another object to control access to it [Gamma95, p.207].

8.2.7.2. Structure

Figure 8.12 illustrates the Proxy implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

Figure 8.12 — Proxy UML Class diagram


SequenceProxyFactory
~ factory : ProxyFactory {final}
+ <E> getLoggableSequence(sequence : Sequence<E>) : Sequence<E>
+ <E> getImmutableSequence(sequence : Sequence<E>) : Sequence<E>
+ <E> getSharedSequence(sequence : Sequence<E>) : Sequence<E>
+ isProxy(sequence : Sequence<?>) : boolean
1
«create»
*
Proxy
java.lang.reflect.Proxy
Sequence<E>
«bind» E::E
Smart Reference
Proxy
SynchronisedSequence
E
- lock : java.lang.Object {final}
+ SynchronisedSequence(sequence : Sequence<E>, lock : java.lang.Object)
+ next() : E {synchronised = lock}
+ current() : E {synchronised = lock}
Subject
«interface»
Sequence
E
+ bounded() : boolean
+ next() : E
1
1
«abstract class»
SequenceDecorator
E
~ sequence : Sequence<E> {final}
# SequenceDecorator(sequence : Sequence<E>)
# getSequence(copy : boolean) : Sequence<E>
+ bounded() : boolean
+ next() : E
RealSubject
Proxy
NonResettableSequence
E
+ NonResettableSequence(sequence : Sequence<E>)
+ reset()
+ copy() : NonResettableSequence<E>
Protection Proxies
Proxy
ImmutableSequence
E
+ ImmutableSequence(sequence : Sequence<E>)
+ next() : E
+ copy() : ImmutableSequence<E>

8.2.7.3. Participants

Table 8.13 describes the Proxy participants in the words of Gamma et al. [Gamma95, p.209-210] and lists the corresponding implementations developed in the evaluation.

Table 8.13 — Proxy participants

Participant	Description	Implementation
Proxy	– Maintains a reference that lets the proxy access the real subject. **Proxy** may refer to a **Subject** if the **RealSubject** and **Subject** interfaces are the same. – Provides an interface identical to **Subject**'s so that a proxy can be substituted for the real subject. – Controls access to the real subject and may be responsible for creating and deleting it. – Other responsibilities depend on the kind of proxy (*remote proxy*, *virtual proxy*, *protection proxy*, or *smart reference*).	`SynchronisedSequence<E>`, `NonResettableSequence<E>`, `ImmutableSequence<E>`, `java.lang.reflect.Proxy` objects (`SequenceProxyFactory`, `meta.reflect.ProxyFactory`)
Subject	– Defines the common interface for **RealSubject** and **Proxy** so that a **Proxy** can be used anywhere a **RealSubject** is expected.	`meta.model.Sequence<E>`
RealSubject	– Defines the real object that the proxy represents.	Any `Sequence<E>` implementation

8.2.7.4. Implementation

Overloading the member access operator in C++ — Java does not support operator overloading, but dynamic proxies can simulate the behaviour. The `SequenceProxyFactory.getVirtualSequence(..)` method returns a virtual (dynamic) proxy that will not create an actual `Sequence<E>` instance until a method declared in the `Sequence<E>` interface is invoked. Methods declared in other types can still be invoked, for example `java.lang.Object.toString()`. However, the actual class of the proxied sequence cannot be used for casts or `instanceof` tests using the proxy, since the class is `java.lang.reflect.Proxy`. This can be a severe limitation on dynamic proxy usage since the client will not (necessarily) know a proxy is accessed in place of the real object. As `java.lang.Object.equals(Object)`, for example, utilises `instanceof`, this can be a problem for object comparison and collection usage. This also affects the Java Handle/Body idiom used to manage shared objects acquired from `meta.reflect.ProxyFactory`. **Using `doesNotUnderstand` in Smalltalk** — Not possible in Java 6, since the signature of a method to be invoked is determined at compile—time, while only the actual type of the (polymorphic) object is determined at runtime [Sierra06, p.111]. Reflection does not allow creation of new methods at runtime, so only compile—time known methods can be invoked. However, method names (strings) can be used to identify methods, but will require an elaborate "framework" to fetch methods, dispatch if found, and error handling if not found ("does not understand"). This principle is used in several dynamic proxy implementations. **Proxy does not always have to know the type of the RealSubject** — The virtual sequence described above uses type literals to supply the generic type to be created. The protection proxies defined operate on interfaces, for example `ImmutableSequence<E>` that makes any `Sequence<E>` type immutable.

8.3. Behavioural Patterns

8.3.1. Chain of Responsibility

Chain of Responsibility is a Behavioural pattern with Object scope. The implementation is located in the `dk.rode.thesis.chainofresponsibility` package.

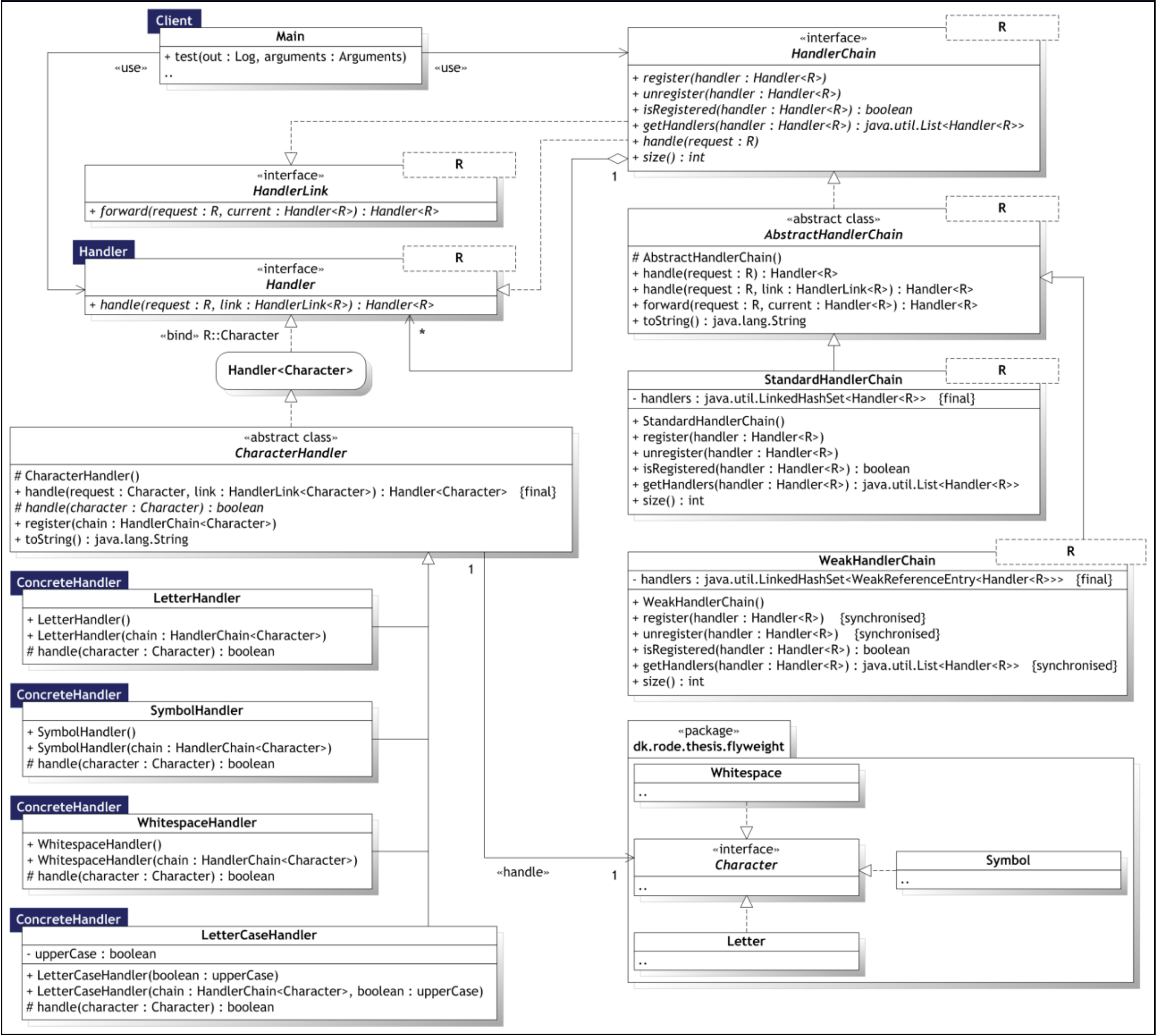
8.3.1.1. Intent

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it [Gamma95, p.223].

8.3.1.2. Structure

Figure 8.13 illustrates the Chain of Responsibility implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

Figure 8.13 — Chain of Responsibility UML Class diagram


Client
Main
+ test(out : Log, arguments : Arguments)
..
«use»
«use»
«interface»
HandlerChain
R
+ register(handler : Handler<R>)
+ unregister(handler : Handler<R>)
+ isRegistered(handler : Handler<R>) : boolean
+ getHandlers(handler : Handler<R>) : java.util.List<Handler<R>>
+ handle(request : R)
+ size() : int
1
«interface»
HandlerLink
R
+ forward(request : R, current : Handler<R>) : Handler<R>
Handler
«interface»
Handler
R
+ handle(request : R, link : HandlerLink<R>) : Handler<R>
*
«bind» R::Character
Handler<Character>
«abstract class»
AbstractHandlerChain
R
# AbstractHandlerChain()
+ handle(request : R) : Handler<R>
+ handle(request : R, link : HandlerLink<R>) : Handler<R>
+ forward(request : R, current : Handler<R>) : Handler<R>
+ toString() : java.lang.String
StandardHandlerChain
R
- handlers : java.util.LinkedHashSet<Handler<R>> {final}
+ StandardHandlerChain()
+ register(handler : Handler<R>)
+ unregister(handler : Handler<R>)
+ isRegistered(handler : Handler<R>) : boolean
+ getHandlers(handler : Handler<R>) : java.util.List<Handler<R>>
+ size() : int
«abstract class»
CharacterHandler
# CharacterHandler()
+ handle(request : Character, link : HandlerLink<Character>) : Handler<Character> {final}
# handle(character : Character) : boolean
+ register(chain : HandlerChain<Character>)
+ toString() : java.lang.String
1
WeakHandlerChain
R
- handlers : java.util.LinkedHashSet<WeakReferenceEntry<Handler<R>>> {final}
+ WeakHandlerChain()
+ register(handler : Handler<R>) {synchronised}
+ unregister(handler : Handler<R>) {synchronised}
+ isRegistered(handler : Handler<R>) : boolean
+ getHandlers(handler : Handler<R>) : java.util.List<Handler<R>> {synchronised}
+ size() : int
ConcreteHandler
LetterHandler
+ LetterHandler()
+ LetterHandler(chain : HandlerChain<Character>)
# handle(character : Character) : boolean
ConcreteHandler
SymbolHandler
+ SymbolHandler()
+ SymbolHandler(chain : HandlerChain<Character>)
# handle(character : Character) : boolean
ConcreteHandler
WhitespaceHandler
+ WhitespaceHandler()
+ WhitespaceHandler(chain : HandlerChain<Character>)
# handle(character : Character) : boolean
ConcreteHandler
LetterCaseHandler
- upperCase : boolean
+ LetterCaseHandler(boolean : upperCase)
+ LetterCaseHandler(chain : HandlerChain<Character>, boolean : upperCase)
# handle(character : Character) : boolean
«package»
dk.rode.thesis.flyweight
Whitespace
..
«interface»
Character
..
Symbol
..
Letter
..
«handle»
1

8.3.1.3. Participants

Table 8.14 describes the Chain of Responsibility participants in the words of Gamma et al. [Gamma95, p.225-226] and lists the corresponding implementations developed in the evaluation.

Table 8.14 — Chain of Responsibility participants

Participant	Description	Implementation
Handler	– Defines an interface for handling requests. – Implements the successor link (optional).	`Handler<R>`

ConcreteHandler	– Handles requests it is responsible for.
	– Can access its successor.
	– If the **ConcreteHandler** can handle the request, it does so; otherwise it forwards the request to its successor.
		`CharacterHandler`
		
		`LetterHandler, LetterCaseHandler, SymbolHandler, WhitespaceHandler`

Table 8.14 — Chain of Responsibility participants

Participant	Description	Implementation
Client	– Initiates the request to a **ConcreteHandler** object on the chain.	`Main`

The implementation makes the chain explicit, represented by the `HandlerChain<R>` class. Ergo, the name and minimal description of the "pseudo—participants" as listed in table 8.15 below are not defined by Gamma et al.

Table 8.15 — Additional Chain of Responsibility entities

Name	Description	Implementation

HandlerChain	– Maintains a chain of **ConcreteHandler** objects, which may be altered at any time.
	– Manages the forwarding of reqests to **ConcreteHandler** objects until a matching handler is found, if any.
	– Declares operations for accessing and removing handlers.
	– Doubles as a ***HandlerLink***.
		`HandlerChain<R>`
		
		`AbstractHandlerChain<R>`, `StandardHandlerChain<R>`, `WeakHandlerChain<R>`

HandlerLink	– Forwards an unhandled request to the next **ConcreteHandler** represented by the link. The link may represent a **ConcreteHandler** or a ***HandlerChain***.	`HandlerLink<R>`

8.3.1.4. Implementation

Implementing the successor chain — The implementation differs from the canonical implementation by Gamma et al. in that it makes the chain explicit, represented by the `HandlerChain<R>` type. This means a concrete handler can only access its successor through the chain as it does not store a reference to it itself. This reduces coupling between handlers; handlers can participate in several chains; and allow a chain to be altered at any time by adding or removing handlers to and from it. **Connecting successors** — The `HandlerLink<R>` type represents a link to the next handler in the chain, if any. **Representing requests** — The type of request is specified using generics in the `Handler<R>` class, where `R` is the type of request. This is compile—time type—safe and flexible. **Automatic forwarding in Smalltalk** — Not supported in Java (but see section 8.2.7.4).

8.3.2. Command

Command is a Behavioural pattern with Object scope. The `dk.rode.thesis.command` package contains the implementation.

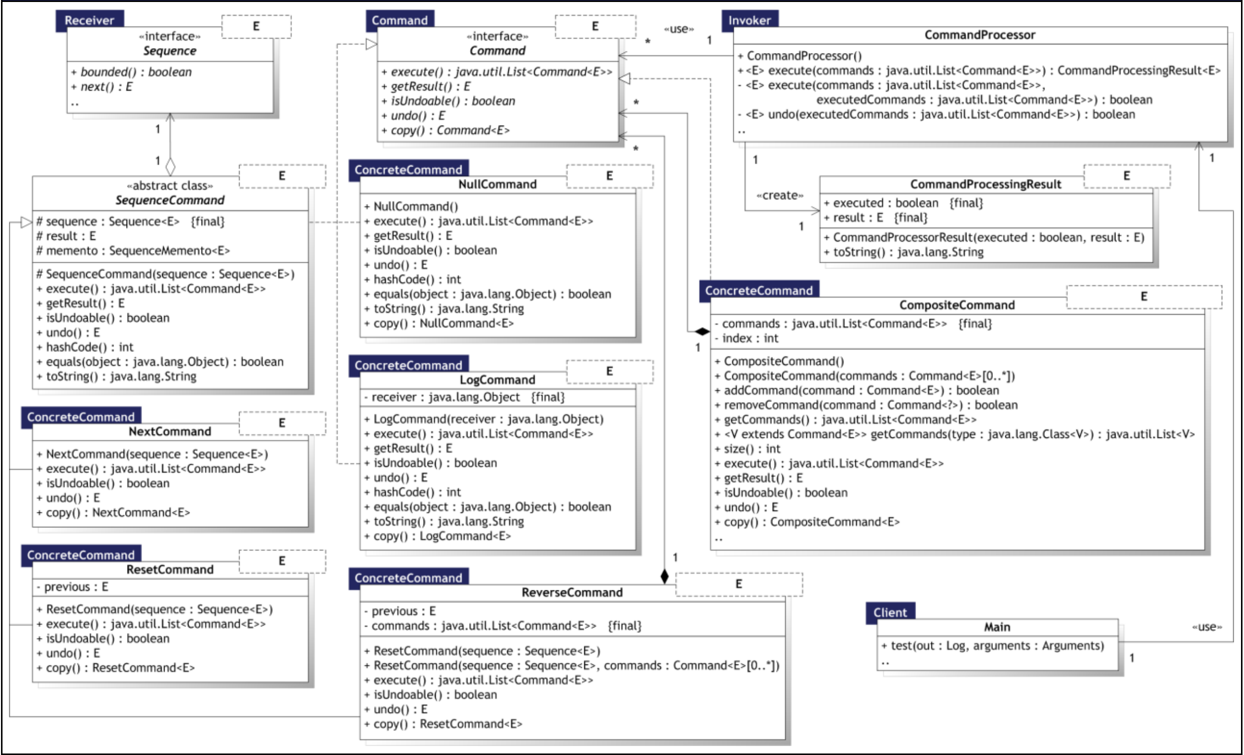
8.3.2.1. Intent

Encapsulate a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations [Gamma95, p.233].

8.3.2.2. Structure

Figure 8.14 illustrates the Command implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

Figure 8.14 — Command UML Class diagram


Receiver
E
«interface»
Sequence
+ bounded() : boolean
+ next() : E
..
Command
E
«interface»
Command
+ execute() : java.util.List<Command<E>>
+ getResult() : E
+ isUndoable() : boolean
+ undo() : E
+ copy() : Command<E>
«use»
Invoker
CommandProcessor
+ CommandProcessor()
+ <E> execute(commands : java.util.List<Command<E>>) : CommandProcessingResult<E>
- <E> execute(commands : java.util.List<Command<E>>, executedCommands : java.util.List<Command<E>>) : boolean
- <E> undo(executedCommands : java.util.List<Command<E>>) : boolean
..
«abstract class»
SequenceCommand
# sequence : Sequence<E> {final}
# result : E
# memento : SequenceMemento<E>
# SequenceCommand(sequence : Sequence<E>)
+ execute() : java.util.List<Command<E>>
+ getResult() : E
+ isUndoable() : boolean
+ undo() : E
+ hashCode() : int
+ equals(object : java.lang.Object) : boolean
+ toString() : java.lang.String
ConcreteCommand
NullCommand
+ NullCommand()
+ execute() : java.util.List<Command<E>>
+ getResult() : E
+ isUndoable() : boolean
+ undo() : E
+ hashCode() : int
+ equals(object : java.lang.Object) : boolean
+ toString() : java.lang.String
+ copy() : NullCommand<E>
«create»
CommandProcessingResult
+ executed : boolean {final}
+ result : E {final}
+ CommandProcessorResult(executed : boolean, result : E)
+ toString() : java.lang.String
ConcreteCommand
CompositeCommand
- commands : java.util.List<Command<E>> {final}
- index : int
+ CompositeCommand()
+ CompositeCommand(commands : Command<E>[0..*])
+ addCommand(command : Command<E>) : boolean
+ removeCommand(command : Command<?>) : boolean
+ getCommands() : java.util.List<Command<E>>
+ <V extends Command<E>> getCommands(type : java.lang.Class<V>) : java.util.List<V>
+ size() : int
+ execute() : java.util.List<Command<E>>
+ getResult() : E
+ isUndoable() : boolean
+ undo() : E
+ copy() : CompositeCommand<E>
..
ConcreteCommand
LogCommand
- receiver : java.lang.Object {final}
+ LogCommand(receiver : java.lang.Object)
+ execute() : java.util.List<Command<E>>
+ getResult() : E
+ isUndoable() : boolean
+ undo() : E
+ hashCode() : int
+ equals(object : java.lang.Object) : boolean
+ toString() : java.lang.String
+ copy() : LogCommand<E>
ConcreteCommand
NextCommand
+ NextCommand(sequence : Sequence<E>)
+ execute() : java.util.List<Command<E>>
+ isUndoable() : boolean
+ undo() : E
+ copy() : NextCommand<E>
ConcreteCommand
ResetCommand
- previous : E
+ ResetCommand(sequence : Sequence<E>)
+ execute() : java.util.List<Command<E>>
+ isUndoable() : boolean
+ undo() : E
+ copy() : ResetCommand<E>
ConcreteCommand
ReverseCommand
- previous : E
- commands : java.util.List<Command<E>> {final}
+ ResetCommand(sequence : Sequence<E>)
+ ResetCommand(sequence : Sequence<E>, commands : Command<E>[0..*])
+ execute() : java.util.List<Command<E>>
+ isUndoable() : boolean
+ undo() : E
+ copy() : ResetCommand<E>
Client
Main
+ test(out : Log, arguments : Arguments)
..
«use»

8.3.2.3. Participants

Table 8.16 describes the Command participants in the words of Gamma et al. [Gamma95, p.236-237] and lists the corresponding implementations developed in the evaluation.

Table 8.16 — Command participants

Participant	Description	Implementation

Command	– Declares an interface for executing an operation.	`Command<E>`
		
		`SequenceCommand<E>`

ConcreteCommand	– Defines a binding between a **Receiver** object and an action. – Implements *execute* by invoking the corresponding operation(s) on **Receiver**.	`CompositeCommand<E>`, `NextCommand<E>`, `ResetCommand<E>`, `ReverseCommand<E>`, `LogCommand<E>`, `NullCommand<E>`
Client	– Creates a **ConcreteCommand** object and sets its receiver.	`Main`
Invoker	– Asks the command to carry out the request.	`CommandProcessor`
Receiver	– Knows how to perform the operations associated with carrying out a request. Any class may serve as a **Receiver**.	Any `Sequence<E>` implementation

The implementation differs from the canonical implementation by Gamma et al. in that it uses a Command Processor to maintain, execute, and possibly undo commands, corresponding to a simple variant of the "POSA"

Command Processor pattern [Buschmann96, p.277]. The processor thus functions as the Invoker participant, but it will be handed the commands to execute by the Client. Hence, the name and minimal description of the "pseudo—participant" as listed in table 8.17 below are not defined by Gamma et al., but by Buschmann et al. [Buschmann96, p.280].

Table 8.17 — Additional Command entities

Name	Description	Implementation
CommandProcessor	– Activates **Command** execution, including commands spawned by an executed **ConcreteCommand**. – Maintains **Command** objects. – Provides additional services related to command execution.	`CommandProcessor`, `CommandProcessingResult<E>`

The execution of a `Command<E>` may spawn new commands to be executed immediately by the `CommandProcessor` in a depth—first manner. As far as we know, this is a novel approach to command execution. This lessen need for macro commands considerably and allow more control of command execution as composite (macro) commands no longer handle the execution of contained commands. Undo of spawned commands is possible in several different ways.

8.3.2.4. Implementation

How intelligent should the command be? — Design choice. **Supporting undo and redo** — The `SequenceCommand<E>` class uses mementos for undo, while sub—classes also employ other means of undo as necessary. The `CommandProcessor` class implicit uses a history list in form of a collection of commands passed to it for execution. **Avoiding error accumulation in the undo process** — By using the `CommandProcessor` class, the history list, undo, and error handling is made explicit and hence not up to the individual command. Error handling supports spawned commands. The implementation throws an exception in case undo fails, but different error handling strategies can be applied because of centralised control. **Using C++ templates** — This corresponds to using generics with an upper bound as described in section 8.1.3.4.

8.3.3. Interpreter

Interpreter is a Behavioural pattern with Class scope. The `dk.rode.thesis.interpreter` package contains the implementation.

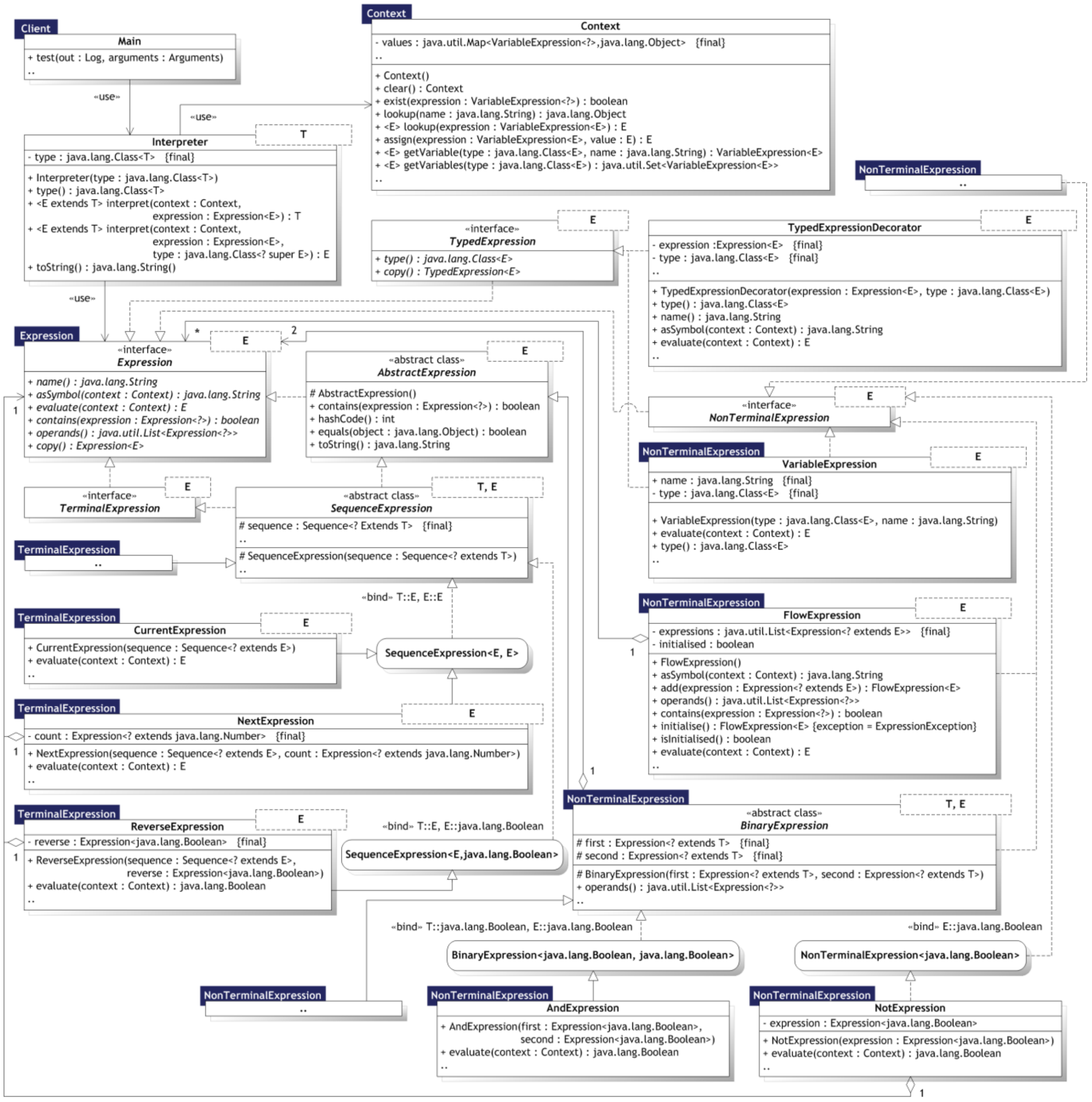
8.3.3.1. Intent

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language [Gamma95, p.243].

8.3.3.2. Structure

Figure 8.15 illustrates the Interpreter implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

Figure 8.15 — Interpreter UML Class diagram


Client
Main
+ test(out : Log, arguments : Arguments)
Context
Context
- values : java.util.Map<VariableExpression<?>,java.lang.Object> {final}
+ Context()
+ clear() : Context
+ exist(expression : VariableExpression<?>) : boolean
+ lookup(name : java.lang.String) : java.lang.Object
+ <E> lookup(expression : VariableExpression<E>) : E
+ assign(expression : VariableExpression<E>, value : E) : E
+ <E> getVariable(type : java.lang.Class<E>, name : java.lang.String) : VariableExpression<E>
+ <E> getVariables(type : java.lang.Class<E>) : java.util.Set<VariableExpression<E>>
«use»
Interpreter
T
- type : java.lang.Class<T> {final}
+ Interpreter(type : java.lang.Class<T>)
+ type() : java.lang.Class<T>
+ <E extends T> interpret(context : Context, expression : Expression<E>) : T
+ <E extends T> interpret(context : Context, expression : Expression<E>, type : java.lang.Class<? super E>) : E
+ toString() : java.lang.String()
NonTerminalExpression
«interface» TypedExpression
+ type() : java.lang.Class<E>
+ copy() : TypedExpression<E>
TypedExpressionDecorator
- expression :Expression<E> {final}
- type : java.lang.Class<E> {final}
+ TypedExpressionDecorator(expression : Expression<E>, type : java.lang.Class<E>)
+ type() : java.lang.Class<E>
+ name() : java.lang.String
+ asSymbol(context : Context) : java.lang.String
+ evaluate(context : Context) : E
Expression
«interface» Expression
+ name() : java.lang.String
+ asSymbol(context : Context) : java.lang.String
+ evaluate(context : Context) : E
+ contains(expression : Expression<?>) : boolean
+ operands() : java.util.List<Expression<?>>
+ copy() : Expression<E>
«abstract class» AbstractExpression
# AbstractExpression()
+ contains(expression : Expression<?>) : boolean
+ hashCode() : int
+ equals(object : java.lang.Object) : boolean
+ toString() : java.lang.String
«interface» NonTerminalExpression
NonTerminalExpression
VariableExpression
+ name : java.lang.String {final}
- type : java.lang.Class<E> {final}
+ VariableExpression(type : java.lang.Class<E>, name : java.lang.String)
+ evaluate(context : Context) : E
+ type() : java.lang.Class<E>
«interface» TerminalExpression
«abstract class» SequenceExpression
# sequence : Sequence<? Extends T> {final}
# SequenceExpression(sequence : Sequence<? extends T>)
TerminalExpression
«bind» T::E, E::E
TerminalExpression
CurrentExpression
+ CurrentExpression(sequence : Sequence<? extends E>)
+ evaluate(context : Context) : E
SequenceExpression<E, E>
NonTerminalExpression
FlowExpression
- expressions : java.util.List<Expression<? extends E>> {final}
- initialised : boolean
+ FlowExpression()
+ asSymbol(context : Context) : java.lang.String
+ add(expression : Expression<? extends E>) : FlowExpression<E>
+ operands() : java.util.List<Expression<?>>
+ contains(expression : Expression<?>) : boolean
+ initialise() : FlowExpression<E> {exception = ExpressionException}
+ isInitialised() : boolean
+ evaluate(context : Context) : E
TerminalExpression
NextExpression
- count : Expression<? extends java.lang.Number> {final}
+ NextExpression(sequence : Sequence<? extends E>, count : Expression<? extends java.lang.Number>)
+ evaluate(context : Context) : E
TerminalExpression
ReverseExpression
- reverse : Expression<java.lang.Boolean> {final}
+ ReverseExpression(sequence : Sequence<? extends E>, reverse : Expression<java.lang.Boolean>)
+ evaluate(context : Context) : java.lang.Boolean
«bind» T::E, E::java.lang.Boolean
SequenceExpression<E,java.lang.Boolean>
NonTerminalExpression
«abstract class» BinaryExpression
# first : Expression<? extends T> {final}
# second : Expression<? extends T> {final}
# BinaryExpression(first : Expression<? extends T>, second : Expression<? extends T>)
+ operands() : java.util.List<Expression<?>>
«bind» T::java.lang.Boolean, E::java.lang.Boolean
«bind» E::java.lang.Boolean
BinaryExpression<java.lang.Boolean, java.lang.Boolean>
NonTerminalExpression<java.lang.Boolean>
NonTerminalExpression
NonTerminalExpression
AndExpression
+ AndExpression(first : Expression<java.lang.Boolean>, second : Expression<java.lang.Boolean>)
+ evaluate(context : Context) : java.lang.Boolean
NonTerminalExpression
NotExpression
- expression : Expression<java.lang.Boolean>
+ NotExpression(expression : Expression<java.lang.Boolean>)
+ evaluate(context : Context) : java.lang.Boolean

8.3.3.3. Participants

Table 8.18 describes the Interpreter participants in the words of Gamma et al. [Gamma95, p.245-246] and lists the corresponding implementations developed in the evaluation.

Table 8.18 — Interpreter participants

Participant	Description	Implementation

AbstractExpression	– Declares an abstract *interpret* operation that is common to all nodes in the abstract syntax tree.	`Expression<E>`
		
		`TypedExpression<E>`, `InitialisableExpression<E>`

Table 8.18 — Interpreter participants

Participant	Description	Implementation
TerminalExpression	– Implements an *interpret* operation associated with terminal symbols in the grammar. – An instance is required for every terminal symbol in a sentence.	`TerminalExpression<E>`, `SequenceExpression<T,E>`, `CurrentExpression<E>`, `NextExpression<E>`, `ResetExpression<E>`, `ReverseExpression<E>`, `SetExpression<E>`
NonTerminalExpression	– One such class is required for every rule $R ::= R_1R_2...R_n$ in the grammar. – Maintains instance variables of type **AbstractExpression** for each of the symbols R_1 through R_n. – Implements an *interpret* operation for nonterminal symbols in the grammar. *interpret* typically calls itself recursively on the variables representing R_1 through R_n.	`NonTerminalExpression<E>`, `BinaryExpression<T,E>`, `AndExpression`, `AssignmentExpression<E>`, `BreakExpression<E>`, `CompareExpression<E>`, `ConditionalExpression<E>`, `ConstantExpression<E>`, `EqualExpression`, `FlowExpression<E>`, `NotExpression`, `OrExpression`, `VariableExpression<E>`
Context	– Contains information that is global to the interpreter.	`Context`
Client	– Builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the **NonTerminalExpression** and **TerminalExpression** classes. – Invokes the *interpret* operation.	`Main`

Unlike Gamma et al., the implementation makes the interpreter explicit. This allows for much better error handling and transfer of evaluation control. Hence, the name and minimal description of the "pseudo—participant" as listed in table 8.19 below are not defined by Gamma et al.

Table 8.19 — Additional Interpreter entities

Name	Description	Implementation
Interpreter	– Invokes the *interpret* operation on the **Expression** supplied by the Client. – Is used by the **Client** in favour of directly invoking *interpret* on **Expression** objects. – May provide additional services related to expression interpretation.	`Interpreter<T>`

8.3.3.4. Implementation

Creating the abstract syntax tree — Design choice. Here, the Builder implementation is used to build expressions that can be assembled. **Defining the *interpret* operation** — The `Expression<E>` type defines the `evaluate(Context)` method, but the `Interpreter<T>` class defines the *interpret* method, performs error handling, and flow control. **Sharing terminal symbols with the Flyweight pattern** — Design choice.

8.3.4. Iterator

Iterator is a Behavioural pattern with Object scope. The `dk.rode.thesis.iterator` package contains the implementation.

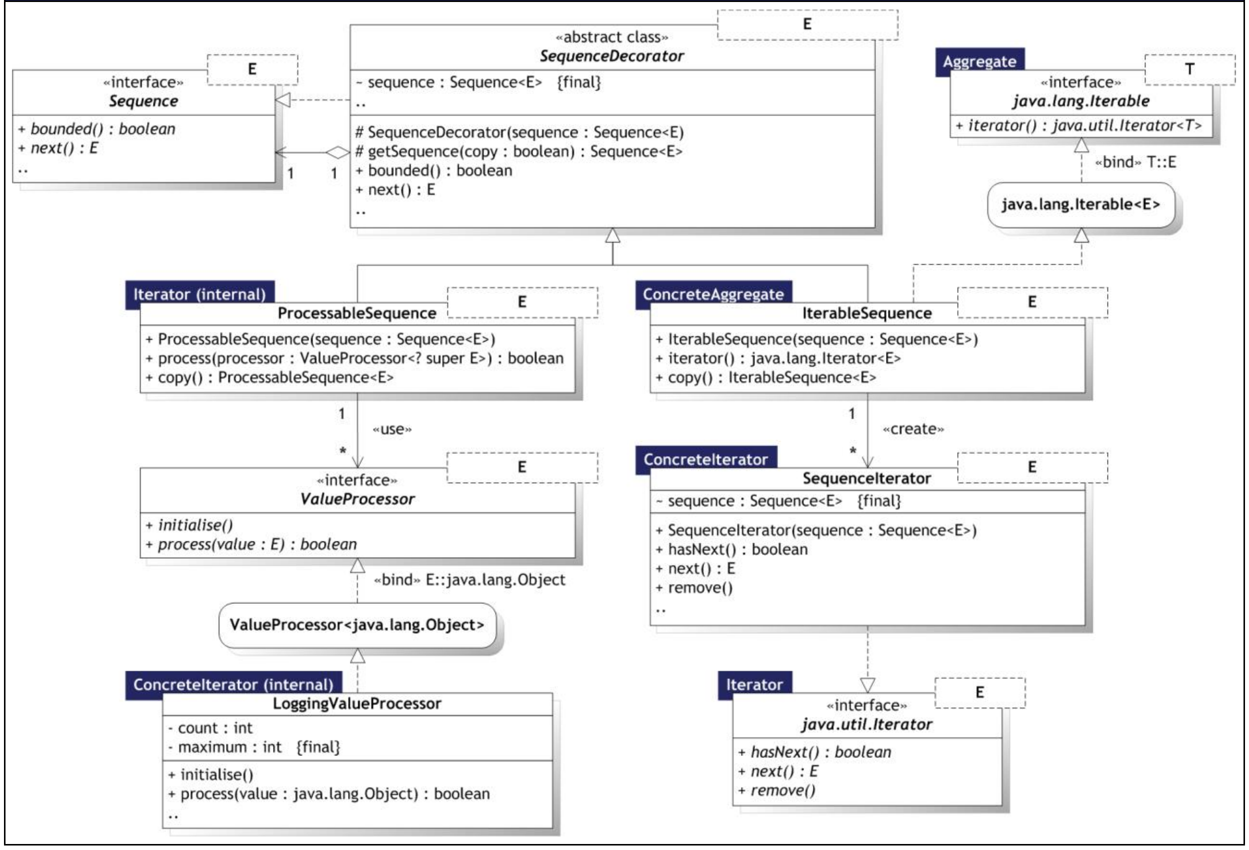
8.3.4.1. Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation [Gamma95, p.257].

8.3.4.2. Structure

Figure **8.16** illustrates the Iterator implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

Figure 8.16 — Iterator UML Class diagram


«interface» Sequence E
+ bounded() : boolean
+ next() : E
..
«abstract class» SequenceDecorator E
~ sequence : Sequence<E> {final}
# SequenceDecorator(sequence : Sequence<E>)
# getSequence(copy : boolean) : Sequence<E>
+ bounded() : boolean
+ next() : E
1
1
Aggregate
«interface» java.lang.Iterable T
+ iterator() : java.util.Iterator<T>
«bind» T::E
java.lang.Iterable<E>
Iterator (internal)
ProcessableSequence E
+ ProcessableSequence(sequence : Sequence<E>)
+ process(processor : ValueProcessor<? super E>) : boolean
+ copy() : ProcessableSequence<E>
ConcreteAggregate
IterableSequence E
+ IterableSequence(sequence : Sequence<E>)
+ iterator() : java.lang.Iterator<E>
+ copy() : IterableSequence<E>
«use»
«create»
«interface» ValueProcessor E
+ initialise()
+ process(value : E) : boolean
«bind» E::java.lang.Object
ValueProcessor<java.lang.Object>
ConcreteIterator
SequenceIterator E
~ sequence : Sequence<E> {final}
+ SequenceIterator(sequence : Sequence<E>)
+ hasNext() : boolean
+ next() : E
+ remove()
ConcreteIterator (internal)
LoggingValueProcessor
- count : int
- maximum : int {final}
+ initialise()
+ process(value : java.lang.Object) : boolean
Iterator
«interface» java.util.Iterator E
+ hasNext() : boolean
+ next() : E
+ remove()

8.3.4.3. Participants

Table **8.20** describes the Iterator participants in the words of Gamma et al. [Gamma95, p.259] and lists the corresponding implementations developed in the evaluation.

Table 8.20 — Iterator participants

Participant	Description	Implementation
Iterator	– Defines an interface for accessing and traversing elements.	`java.util.Iterator<E>` (external), `ProcessableSequence<E>`, `ValueProcessor<E>` (internal)
ConcreteIterator	– Implements the **Iterator** interface. – Keeps track of the current position in the traversal of the aggregate.	`SequenceIterator<E>` (external), `LoggingValueProcessor<E>` (internal)
Aggregate	– Defines an interface for creating an **Iterator** object.	`java.lang.Iterable<T>` (external)
ConcreteAggregate	– Implements the **Iterator** creation interface (**Aggregate**) to return an instance of the proper **ConcreteIterator**.	`IterableSequence<E>` (external)

No classes have been defined to represent Aggregate and ConcreteAggregate for internal iterators. Java 6 defines no standard interface for internal iterators. The creation is left at the discretion of the context using internal iterators.

8.3.4.4. Implementation

Java has built—in API and language support for the Iterator pattern. **Who controls the iteration?** — The `SequenceIterator<E>` class represents an external iterator, while the `ProcessableSequence<E>` represents an internal iterator. **Who defines the traversal algorithm?** — The `composite.CompositeStrategy` utilises package private access to ensure encapsulation and information hiding is not violated when the traversal algorithm is external. **How robust is the iterator?** — Design choice. Standard iterators in Java are *fail—fast*, and fail immediately in case of concurrent modification. **Additional Iterator operations** — Design choice. **Using polymorphic iterators in C++** — `java.lang.Iterable<T>` defines a factory method to return a `java.util.Iterator<E>` instance and the usage is illustrated in the `IterableSequence<E>` class. **Iterators may have privileged access** — Illustrated in the `meta.reflect.CallerClass.CallerIterator<C>` inner class. **Iterators for composites** — The `composite.CompositeStrategy` determines how the composite structure should be traversed. **Null iterators** — Trivial, though note that making a null iterator for the `meta.model.Sequence<E>` type is not possible, because sequence semantics require that a sequence always have at least a single value.

8.3.5. Mediator

Mediator is a Behavioural pattern with Object scope. This pattern is not implemented due to a lack of applicability because of its abstraction and granularity level. It offers no new information compared to the rest of the "Gang of Four" patterns with regards to language functionality.

8.3.5.1. Intent

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently [Gamma95, p.273].

8.3.5.2. Participants

Table 8.21 describes the Mediator participants in the words of Gamma et al. [Gamma95, p.277].

Table 8.21 — Mediator participants

Participant	Description	Implementation
Mediator	– Defines an interface for communicating with **Colleague** objects.	-
ConcreteMediator	– Implements cooperative behaviour by coordinating **Colleague** objects. – Knows and maintains its colleagues.	-
Colleague Class	– Each **Colleague** class knows its **Mediator** object. – Each **Colleague** communicates with its mediator whenever it would have otherwise communicated with another colleague.	-

8.3.5.3. Implementation

As stated, the Mediator pattern has not been implemented because it only addresses general design issues. We still address the Implementation element, since it only discuss two items. **Omitting the abstract Mediator class** — Design choice. **Colleague/Mediator communication** — Design choice. Using the Observer pattern is possible as the means to communicate, for example the stand—alone `observer.ObserverManager` class that could handle different kind of colleagues without requiring them to implement a common super type. Passing itself as an argument is a common approach with trivial implementation. Also supported by `ObserverManager`.

8.3.6. Memento

Memento is Behavioural with Object scope. The implementation is located in the `dk.rode.thesis.memento` package.

8.3.6.1. Intent

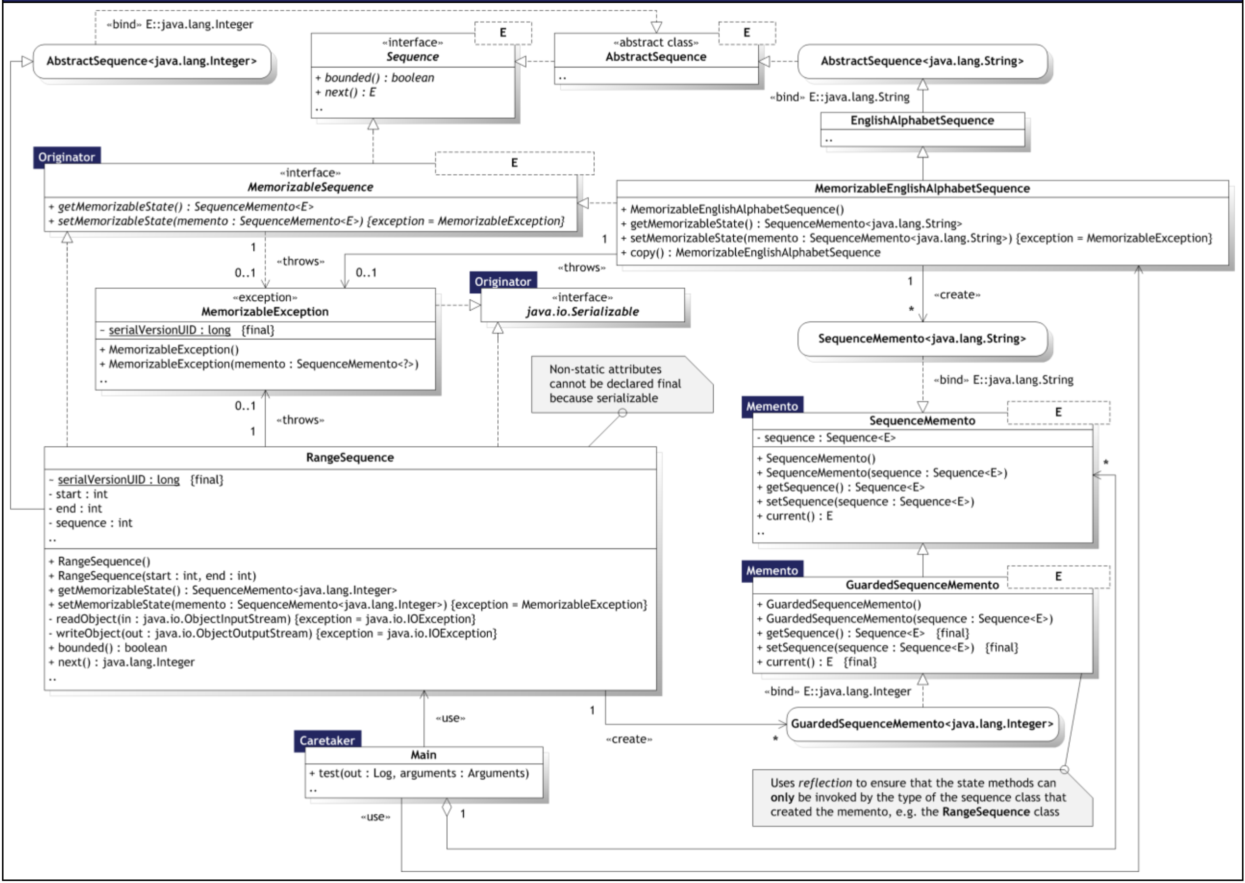
Without violating encapsulation, capture and externalise an objects internal state so that the object can be restored to this state later [Gamma95, p.283].

Note that according to our separation of encapsulation and information hiding into two distinct concepts as described in section 2.1.1, encapsulation as described above refers to both concepts. Encapsulation must not be broken in the sense that the state must still be localised in the object, which must be hidden to shield it from unwanted access.

8.3.6.2. Structure

Figure 8.17 illustrates the Memento implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

Figure 8.17 — Memento UML Class diagram



8.3.6.3. Participants

Table 8.22 describes the Memento participants in the words of Gamma et al. [Gamma95, p.285] and lists the corresponding implementations developed in the evaluation.

Table 8.22 — Memento participants

Participant	Description	Implementation
Memento	– Stores internal state of the **Originator** object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion. – Protects against access by objects other than the originator. Mementos have effectively two interfaces. **Caretaker** sees a *narrow* interface to the **Memento** - it can only pass the memento to other objects. **Originator**, in contrast, sees a *wide* interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produced the memento would be permitted to access the memento's internal state.	`SequenceMemento<E>`, `GuardedSequenceMemento<E>`
Originator	– Creates a memento containing a snapshot of its	Any `MemorizableSequence<E>` implementation like `RangeSequence` and

Table 8.22 – Memento participants

Participant	Description	Implementation
	current internal state. – Use the memento to restore its internal state.	`MemorizableEnglishAlphabetSequence`
Caretaker	– Is responsible for the memento's safekeeping. – Never operates on or examines the contents of a memento.	`Main`

8.3.6.4. Implementation

Language support — Java does not support *narrow* (public) and *wide* (private) interface functionality because all methods declared in an interface must be public. A class may define and implement private methods, but they are inaccessible to other types. The `GuardedSequenceMemento<E>` class illustrates how private methods par design have to be made public for compile–time safety, but are guarded at runtime to ensure that only legal callers are allowed (see section 7.1.4.1). **Storing incremental changes** — Design choice. The `command.SequenceCommand<E>` class use mementos to store the complete state.

8.3.7. Observer

Observer is Behavioural with Object scope. The implementation is located in the `dk.rode.thesis.observer` package.

8.3.7.1. Intent

Define a one–to–many dependency between objects so that when one object changes state, all dependants are notified and updated automatically [Gamma95, p.293].

8.3.7.2. Structure

Figure 8.18 illustrates the Observer implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

8.3.7.3. Participants

Table 8.23 describes the Observer participants in the words of Gamma et al. [Gamma95, p.295] and lists the corresponding implementations developed in the evaluation.

Figure 8.18 — Observer UML Class diagram

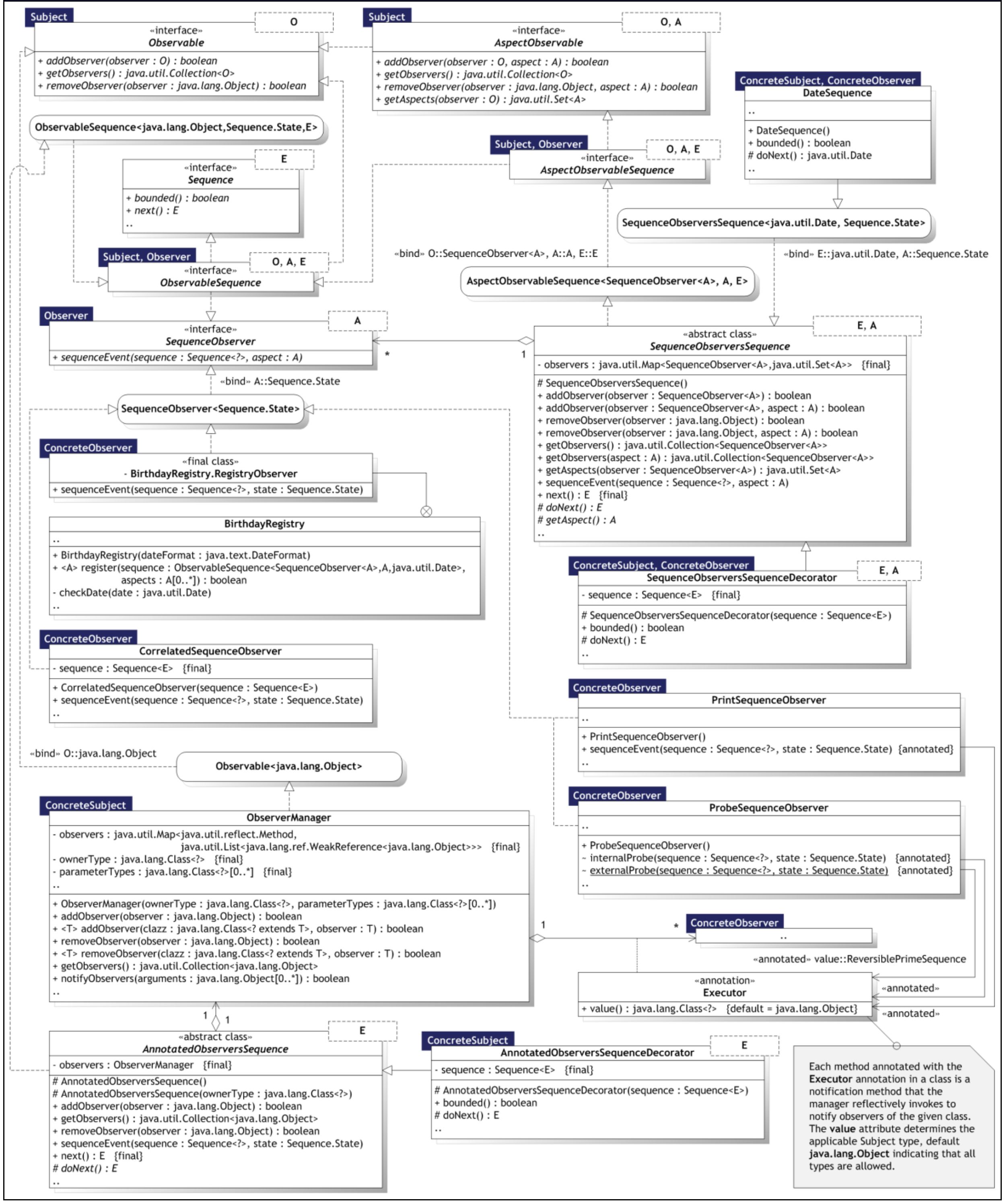

Subject
O
«interface»
Observable
+ addObserver(observer : O) : boolean
+ getObservers() : java.util.Collection<O>
+ removeObserver(observer : java.lang.Object) : boolean
Subject
O, A
«interface»
AspectObservable
+ addObserver(observer : O, aspect : A) : boolean
+ getObservers() : java.util.Collection<O>
+ removeObserver(observer : java.lang.Object, aspect : A) : boolean
+ getAspects(observer : O) : java.util.Set<A>
ConcreteSubject, ConcreteObserver
DateSequence
+ DateSequence()
+ bounded() : boolean
# doNext() : java.util.Date
ObservableSequence<java.lang.Object,Sequence.State,E>
E
«interface»
Sequence
+ bounded() : boolean
+ next() : E
Subject, Observer
O, A, E
«interface»
AspectObservableSequence
SequenceObserversSequence<java.util.Date, Sequence.State>
«bind» O::SequenceObserver<A>, A::A, E::E
«bind» E::java.util.Date, A::Sequence.State
Subject, Observer
O, A, E
«interface»
ObservableSequence
AspectObservableSequence<SequenceObserver<A>, A, E>
Observer
A
«interface»
SequenceObserver
+ sequenceEvent(sequence : Sequence<?>, aspect : A)
*
1
E, A
«abstract class»
SequenceObserversSequence
- observers : java.util.Map<SequenceObserver<A>,java.util.Set<A>> {final}
# SequenceObserversSequence()
+ addObserver(observer : SequenceObserver<A>) : boolean
+ addObserver(observer : SequenceObserver<A>, aspect : A) : boolean
+ removeObserver(observer : java.lang.Object) : boolean
+ removeObserver(observer : java.lang.Object, aspect : A) : boolean
+ getObservers() : java.util.Collection<SequenceObserver<A>>
+ getObservers(aspect : A) : java.util.Collection<SequenceObserver<A>>
+ getAspects(observer : SequenceObserver<A>) : java.util.Set<A>
+ sequenceEvent(sequence : Sequence<?>, aspect : A)
+ next() : E {final}
# doNext() : E
# getAspect() : A
«bind» A::Sequence.State
SequenceObserver<Sequence.State>
ConcreteObserver
«final class»
- BirthdayRegistry.RegistryObserver
+ sequenceEvent(sequence : Sequence<?>, state : Sequence.State)
BirthdayRegistry
+ BirthdayRegistry(dateFormat : java.text.DateFormat)
+ <A> register(sequence : ObservableSequence<SequenceObserver<A>,A,java.util.Date>, aspects : A[0..*]) : boolean
- checkDate(date : java.util.Date)
ConcreteSubject, ConcreteObserver
E, A
SequenceObserversSequenceDecorator
- sequence : Sequence<E> {final}
# SequenceObserversSequenceDecorator(sequence : Sequence<E>)
+ bounded() : boolean
# doNext() : E
ConcreteObserver
CorrelatedSequenceObserver
- sequence : Sequence<E> {final}
+ CorrelatedSequenceObserver(sequence : Sequence<E>)
+ sequenceEvent(sequence : Sequence<?>, state : Sequence.State)
ConcreteObserver
PrintSequenceObserver
+ PrintSequenceObserver()
+ sequenceEvent(sequence : Sequence<?>, state : Sequence.State) {annotated}
«bind» O::java.lang.Object
Observable<java.lang.Object>
ConcreteSubject
ObserverManager
- observers : java.util.Map<java.util.reflect.Method, java.util.List<java.lang.ref.WeakReference<java.lang.Object>>> {final}
- ownerType : java.lang.Class<?> {final}
- parameterTypes : java.lang.Class<?>[0..*] {final}
+ ObserverManager(ownerType : java.lang.Class<?>, parameterTypes : java.lang.Class<?>[0..*])
+ addObserver(observer : java.lang.Object) : boolean
+ <T> addObserver(clazz : java.lang.Class<? extends T>, observer : T) : boolean
+ removeObserver(observer : java.lang.Object) : boolean
+ <T> removeObserver(clazz : java.lang.Class<? extends T>, observer : T) : boolean
+ getObservers() : java.util.Collection<java.lang.Object>
+ notifyObservers(arguments : java.lang.Object[0..*]) : boolean
ConcreteObserver
ProbeSequenceObserver
+ ProbeSequenceObserver()
- internalProbe(sequence : Sequence<?>, state : Sequence.State) {annotated}
- externalProbe(sequence : Sequence<?>, state : Sequence.State) {annotated}
1
*
ConcreteObserver
«annotated» value::ReversiblePrimeSequence
«annotation»
Executor
+ value() : java.lang.Class<?> {default = java.lang.Object}
«annotated»
«annotated»
1
1
E
«abstract class»
AnnotatedObserversSequence
- observers : ObserverManager {final}
# AnnotatedObserversSequence()
# AnnotatedObserversSequence(ownerType : java.lang.Class<?>)
+ addObserver(observer : java.lang.Object) : boolean
+ getObservers() : java.util.Collection<java.lang.Object>
+ removeObserver(observer : java.lang.Object) : boolean
+ sequenceEvent(sequence : Sequence<?>, state : Sequence.State)
+ next() : E {final}
# doNext() : E
ConcreteSubject
E
AnnotatedObserversSequenceDecorator
- sequence : Sequence<E> {final}
# AnnotatedObserversSequenceDecorator(sequence : Sequence<E>)
+ bounded() : boolean
# doNext() : E
Each method annotated with the Executor annotation in a class is a notification method that the manager reflectively invokes to notify observers of the given class. The value attribute determines the applicable Subject type, default java.lang.Object indicating that all types are allowed.

Table 8.23 — Observer participants

Participant	Description	Implementation

Subject	– Knows its observers. Any number of **Observer** objects may observe a subject.
	– Provides an interface for attaching and detaching **Observer** objects.
		`Observable<O>`, `AspectObservable<O,A>`
		
		`ObservableSequence<O,A,E>`, `AspectObservableSequence<O,A,E>`
		
		`SequenceObserversSequence<E,A>`, `AnnotatedObserversSequence<E>`

Observer	– Defines an updating interface for objects that should be notified of changes in a subject.	`SequenceObserver<A>` and any `java.lang.Object` annotated with the `meta.reflect.@Executor` annotation when used with `ObserverManager` or a sub–class of `AnnotatedObserversSequence<E>`

ConcreteSubject	– Stores state of interest to **ConcreteObserver** objects.
	– Sends a notification to its observers when its state changes.
		`ObserverManager`, `DateSequence`
		
		`SequenceObserversSequenceDecorator<E,A>`, `AnnotatedObserversSequenceDecorator<E>`

ConcreteObserver	– Maintains a reference to a **ConcreteSubject** object. – Stores state that should be consistent with the subject's. – Implements the Observer updating interface to keep its state consistent with the subject's.	`CorrelatedSequenceObserver`, `PrintSequenceObserver`, `ProbeSequenceObserver`, `BirthdayRegistry`

8.3.7.4. Implementation

Mapping subjects to their observers — `java.util.Map<K,V>` is used to store observers for fast lookup. **Observing more than one subject** — The stand–alone `ObserverManager` class can manage and notify observers, typically used as a delegate by another object in a composition. It can store observers with different types, but using the same signature to notify the observers. Notification methods are specified via annotations, Its `notifyObservers(java.lang.Object…)` method uses varargs and it is up to the client to specify the signature of the actual observer notification methods and to supply the proper arguments. Hence, whether or not the subject is passed to the notification methods is a design choice based on the signature of the notification methods used. The `AnnotatedObserversSequence<E>` uses `ObserverManager` as a delegate and passes itself as the first argument to `notifyObservers(Object…)` when notification is performed. Observers can thus observe several subjects. **Who triggers the update?** — Design choice. `ObservableSequence<O,A,E>` implementations issue notifications when their internal state changes, for example on invocation on `next()` and `reset()`. **Dangling references to deleted subjects** — Subjects storing observers as hard references will prevent observers from being garbage collected. On the other hand, the `ObserverManager` class stores observers as weak references and if there are no other references to a given observer, it will be garbage collected. The manager handles this and purges weak references transparently when the observer has been garbage collected. **Making sure Subject state is self—consistent before notification** — Template Method is used in the `SequenceObserversSequence<E,A>` and `AnnotatedObserversSequence<E>` classes to handle the notification in a state–consistent manner. **Avoiding observer—specific update protocols** — The `ObserverManager` component can be used for both *push* and *pull* semantics because the type of observers and notification methods are configurable. **Specifying modifications of interest explicitly** — Implementations of

`AspectObservable<O,A>` allow observers of type `O` to subscribe to specific aspects of type `A`. **Encapsulating complex update semantics** — The `ObserverManager` class can be considered a *Change Manager*. Sub—classes could simply override the notification method for different notification strategies.

8.3.8. State

State is a Behavioural pattern with Object scope. The `dk.rode.thesis.state` package contains the implementation.

8.3.8.1. Intent

Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class [Gamma95, p.305].

8.3.8.2. Structure

Figure 8.19 illustrates the State implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

8.3.8.3. Participants

Table 8.24 describes the State participants in the words of Gamma et al. [Gamma95, p.306-307] and lists the corresponding implementations developed in the evaluation.

Table 8.24 — State participants

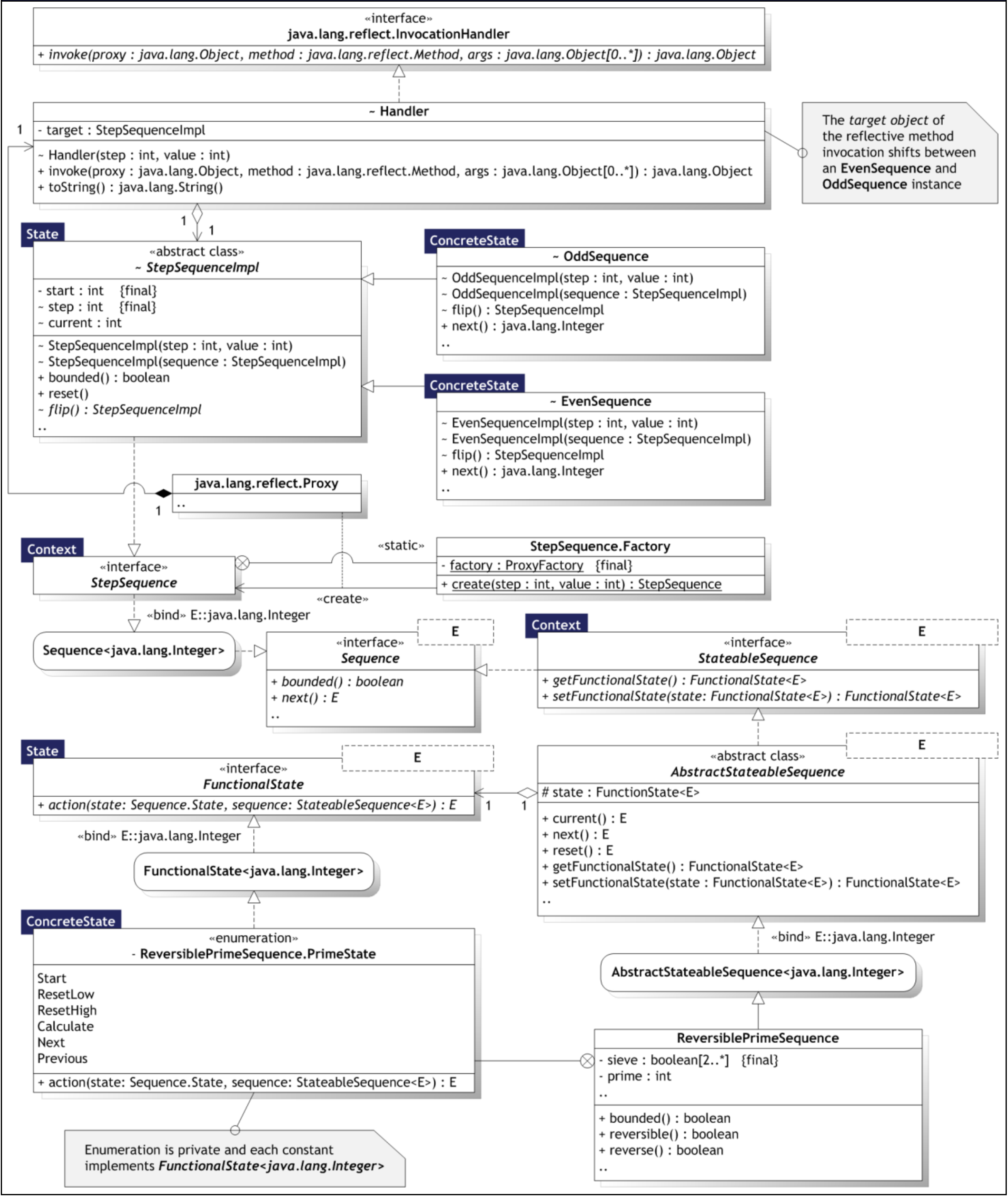
Participant	Description	Implementation

Context	– Defines the interface of interest to clients.
	– Maintains an instance of a **ConcreteState** sub—class that defines the current state.
		`StateableSequence<E>`, `AbstractStateableSequence<E>`, `ReversiblePrimeSequence`
		
		`StepSequence`

State	– Defines an interface for encapsulating the behaviour associated with a particular state of the **Context**.	`FunctionalState<E>`
		
		`StepSequenceImpl`

ConcreteState	– Each sub—class implements a behaviour associated with a state of the **Context**.	Each constant in the `ReversiblePrimeSequence.PrimeState` enumeration `EvenSequence` and `OddSequence`

Figure 8.19 – State UML Class diagram


«interface»
java.lang.reflect.InvocationHandler
+ invoke(proxy : java.lang.Object, method : java.lang.reflect.Method, args : java.lang.Object[0..*]) : java.lang.Object
~ Handler
- target : StepSequenceImpl
~ Handler(step : int, value : int)
+ invoke(proxy : java.lang.Object, method : java.lang.reflect.Method, args : java.lang.Object[0..*]) : java.lang.Object
+ toString() : java.lang.String()
The target object of the reflective method invocation shifts between an EvenSequence and OddSequence instance
State
«abstract class»
~ StepSequenceImpl
- start : int {final}
~ step : int {final}
~ current : int
~ StepSequenceImpl(step : int, value : int)
~ StepSequenceImpl(sequence : StepSequenceImpl)
+ bounded() : boolean
+ reset()
~ flip() : StepSequenceImpl
ConcreteState
~ OddSequence
~ OddSequenceImpl(step : int, value : int)
~ OddSequenceImpl(sequence : StepSequenceImpl)
~ flip() : StepSequenceImpl
+ next() : java.lang.Integer
ConcreteState
~ EvenSequence
~ EvenSequenceImpl(step : int, value : int)
~ EvenSequenceImpl(sequence : StepSequenceImpl)
~ flip() : StepSequenceImpl
+ next() : java.lang.Integer
java.lang.reflect.Proxy
«static»
StepSequence.Factory
- factory : ProxyFactory {final}
+ create(step : int, value : int) : StepSequence
Context
«interface»
StepSequence
«create»
«bind» E::java.lang.Integer
Sequence<java.lang.Integer>
«interface»
Sequence
+ bounded() : boolean
+ next() : E
Context
«interface»
StateableSequence
+ getFunctionalState() : FunctionalState<E>
+ setFunctionalState(state: FunctionalState<E>) : FunctionalState<E>
State
«interface»
FunctionalState
+ action(state: Sequence.State, sequence: StateableSequence<E>) : E
«abstract class»
AbstractStateableSequence
# state : FunctionState<E>
+ current() : E
+ next() : E
+ reset() : E
+ getFunctionalState() : FunctionalState<E>
+ setFunctionalState(state : FunctionalState<E>) : FunctionalState<E>
«bind» E::java.lang.Integer
FunctionalState<java.lang.Integer>
«bind» E::java.lang.Integer
AbstractStateableSequence<java.lang.Integer>
ConcreteState
«enumeration»
- ReversiblePrimeSequence.PrimeState
Start
ResetLow
ResetHigh
Calculate
Next
Previous
+ action(state: Sequence.State, sequence: StateableSequence<E>) : E
ReversiblePrimeSequence
- sieve : boolean[2..*] {final}
- prime : int
+ bounded() : boolean
+ reversible() : boolean
+ reverse() : boolean
Enumeration is private and each constant implements FunctionalState<java.lang.Integer>

8.3.8.4. Implementation

Who defines the state transitions? — Design choice. Transitions are primarily handled by concrete states, such as the `EvenSequence` class and the `ReversiblePrimeSequence.PrimeState` constants, but also by `ReversiblePrimeSequence` when reversed. **A table—based alternative** — Design choice. Trivial, not implemented. **Creating and destroying State objects** — `ReversiblePrimeSequence.PrimeState` is an enumeration and all states are thus known beforehand. The states are stateless, small, and cheap to create. Creation (and destruction) is handled automatically and singleton behaviour for each state is guaranteed by the compiler. `EvenSequence` and `OddSequence` states are created on demand and garbage collected when no longer used. **Using dynamic inheritance** — Not supported directly by Java, but dynamic proxies can simulate the behaviour. The invocation handler used by the proxy can change the ***target object*** of a given method (signature) to supply different implementations at runtime. This is illustrated in the `StepSequence` dynamic proxy that uses instances of the `EvenSequence` and `OddSequence` classes as two different implementations.

8.3.9. Strategy

Strategy is a Behavioural pattern with Object scope. Also known as Policy. The implementation is located in the `dk.rode.thesis.strategy` package.

8.3.9.1. Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it [Gamma95, p.315].

8.3.9.2. Structure

Figure 8.20 illustrates the Strategy implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

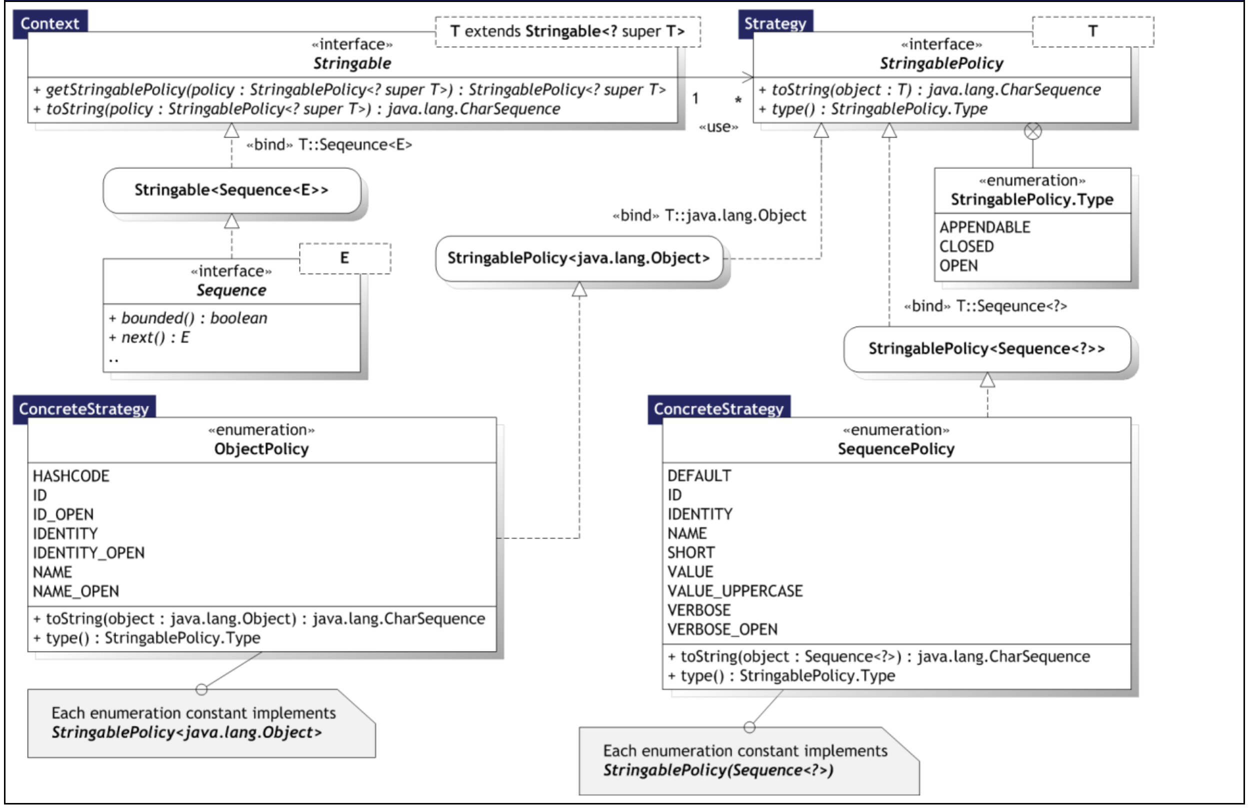
8.3.9.3. Participants

Table 8.25 describes the Strategy participants in the words of Gamma et al. [Gamma95, p.317] and lists the corresponding implementations developed in the evaluation.

Table 8.25 — Strategy participants

Participant	Description	Implementation
Strategy	– Declares an interface common to all supported algorithms. **Context** uses this interface to call the algorithm defined by a **ConcreteStrategy**.	`StringablePolicy<T>`
ConcreteStrategy	– Implements the algorithm using the **Strategy** interface.	Each `SequencePolicy` enumeration constant, and each `ObjectPolicy` enumeration constant
Context	– Is configured with a **ConcreteStrategy**. – Maintains a reference to a **Strategy** object. – May define an interface that lets **Strategy** access its data.	Any `Stringable<T>` implementation, including sequences as they implement the interface

Figure 8.20 — Strategy UML Class diagram


Context
T extends Stringable<? super T>
«interface»
Stringable
+ getStringablePolicy(policy : StringablePolicy<? super T>) : StringablePolicy<? super T>
+ toString(policy : StringablePolicy<? super T>) : java.lang.CharSequence
1
*
«use»
Strategy
T
«interface»
StringablePolicy
+ toString(object : T) : java.lang.CharSequence
+ type() : StringablePolicy.Type
«bind» T::Seqeunce<E>
Stringable<Sequence<E>>
«bind» T::java.lang.Object
«enumeration»
StringablePolicy.Type
APPENDABLE
CLOSED
OPEN
E
«interface»
Sequence
+ bounded() : boolean
+ next() : E
..
StringablePolicy<java.lang.Object>
«bind» T::Seqeunce<?>
StringablePolicy<Sequence<?>>
ConcreteStrategy
«enumeration»
ObjectPolicy
HASHCODE
ID
ID_OPEN
IDENTITY
IDENTITY_OPEN
NAME
NAME_OPEN
+ toString(object : java.lang.Object) : java.lang.CharSequence
+ type() : StringablePolicy.Type
ConcreteStrategy
«enumeration»
SequencePolicy
DEFAULT
ID
IDENTITY
NAME
SHORT
VALUE
VALUE_UPPERCASE
VERBOSE
VERBOSE_OPEN
+ toString(object : Sequence<?>) : java.lang.CharSequence
+ type() : StringablePolicy.Type
Each enumeration constant implements StringablePolicy<java.lang.Object>
Each enumeration constant implements StringablePolicy(Sequence<?>)

8.3.9.4. Implementation

Defining the Strategy and Context interfaces — Design choice. The `Stringable<T>` interface pass itself to a concrete strategy of type `StringablePolicy<T>`, i.e. delegation is used. **Strategies as template parameters** — Generics with an upper bound can be used in place of templates, where the upper bound identifies the strategy functionality. This is analogous to the `abstractfactory.PrototypicalRegistry` functionality that uses an upper bound of `StrictCopyable<?>` to ensure that the actual type has a `copy()` method. **Making Strategy objects optional** — Trivial design choice. This is illustrated in the `Stringable<T>` interface that will apply a default strategy if none is supplied.

8.3.10. Template Method

Template Method is a Behavioural pattern with Class scope. The `dk.rode.thesis.templatemethod` package contains the implementation.

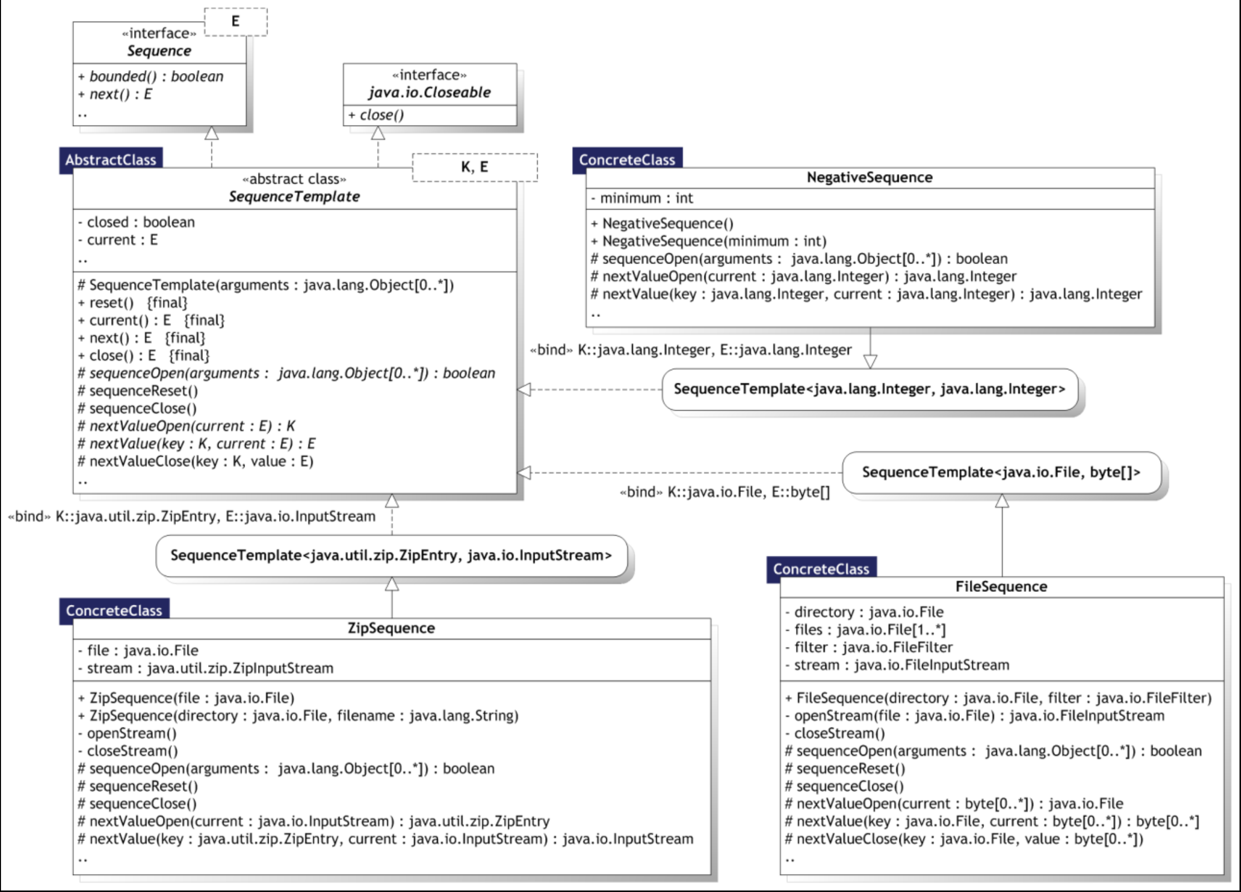
8.3.10.1. Intent

Define the skeleton of an algorithm in an operation, deferring some steps to sub–classes. Template Method lets sub–classes redefine certain steps of an algorithm without changing the algorithm's structure [Gamma95, p.325].

8.3.10.2. Structure

Figure 8.21 illustrates the Template Method implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

Figure 8.21 — Template Method UML Class diagram


E
«interface»
Sequence
+ bounded() : boolean
+ next() : E
..
«interface»
java.io.Closeable
+ close()
AbstractClass
K, E
«abstract class»
SequenceTemplate
- closed : boolean
- current : E
..
# SequenceTemplate(arguments : java.lang.Object[0..*])
+ reset() {final}
+ current() : E {final}
+ next() : E {final}
+ close() : E {final}
# sequenceOpen(arguments : java.lang.Object[0..*]) : boolean
# sequenceReset()
# sequenceClose()
# nextValueOpen(current : E) : K
# nextValue(key : K, current : E) : E
# nextValueClose(key : K, value : E)
..
ConcreteClass
NegativeSequence
- minimum : int
+ NegativeSequence()
+ NegativeSequence(minimum : int)
# sequenceOpen(arguments : java.lang.Object[0..*]) : boolean
# nextValueOpen(current : java.lang.Integer) : java.lang.Integer
# nextValue(key : java.lang.Integer, current : java.lang.Integer) : java.lang.Integer
..
«bind» K::java.lang.Integer, E::java.lang.Integer
SequenceTemplate<java.lang.Integer, java.lang.Integer>
SequenceTemplate<java.io.File, byte[]>
«bind» K::java.io.File, E::byte[]
«bind» K::java.util.zip.ZipEntry, E::java.io.InputStream
SequenceTemplate<java.util.zip.ZipEntry, java.io.InputStream>
ConcreteClass
ZipSequence
- file : java.io.File
- stream : java.util.zip.ZipInputStream
+ ZipSequence(file : java.io.File)
+ ZipSequence(directory : java.io.File, filename : java.lang.String)
- openStream()
- closeStream()
# sequenceOpen(arguments : java.lang.Object[0..*]) : boolean
# sequenceReset()
# sequenceClose()
# nextValueOpen(current : java.io.InputStream) : java.util.zip.ZipEntry
# nextValue(key : java.util.zip.ZipEntry, current : java.io.InputStream) : java.io.InputStream
..
ConcreteClass
FileSequence
- directory : java.io.File
- files : java.io.File[1..*]
- filter : java.io.FileFilter
- stream : java.io.FileInputStream
+ FileSequence(directory : java.io.File, filter : java.io.FileFilter)
- openStream(file : java.io.File) : java.io.FileInputStream
- closeStream()
# sequenceOpen(arguments : java.lang.Object[0..*]) : boolean
# sequenceReset()
# sequenceClose()
# nextValueOpen(current : byte[0..*]) : java.io.File
# nextValue(key : java.io.File, current : byte[0..*]) : byte[0..*]
# nextValueClose(key : java.io.File, value : byte[0..*])
..

8.3.10.3. Participants

Table 8.26 describes the Template Method participants in the words of Gamma et al. [Gamma95, p.327] and lists the corresponding implementations developed in the evaluation.

Table 8.26 — Template Method participants

Participant	Description	Implementation
AbstractClass	– Defines abstract primitive operations that concrete sub–classes define to implement the steps of an algorithm. – Implements a template method defining the skeleton of an algorithm. The template method class primitive operations as well as operations defined in **AbstractClass** or those of other objects.	`SequenceTemplate<K,E>`
ConcreteClass	– Implements the primitive operations to carry out sub–class specific steps of the algorithm.	`ZipSequence`, `FileSequence`, `NegativeSequence`

8.3.10.4. Implementation

Using C++ access control — Primitive operations are declared as protected methods in the `SequenceTemplate<K,E>` abstract class. All sub—classes and classes in the same package can thus execute the primitive operations. Primitive operations that must be implemented are declared abstract, while the template methods have to be declared final to ensure that they cannot be overridden. Unlike C++, primitive operations overridden by a sub—class can be called in the (super) constructor. This is illustrated in `SequenceTemplate<K,E>`. A potential issue is that Java does not support multiple inheritance. Once the abstract class is inherited, no other classes can be inherited. An alternative is to use composition and interface implementation as illustrated in the `builder.TypedExpressionBuilder<E>` class; Builder often rely on sub—classing like Template Method. **Minimizing primitive operations** — This is a design choice. **Naming conventions** — Design choice. `SequenceTemplate<K,E>` utilises meaningful method name prefixes to identify primitive operations. Potentially important for reflective invocation.

8.3.11. Visitor

Visitor is Behavioural with Object scope. The implementation is located in the `dk.rode.thesis.visitor` package.

8.3.11.1. Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates [Gamma95, p.331].

8.3.11.2. Structure

Figure 8.22 illustrates the Visitor implementation as an UML Class diagram, where the pattern participants can also be identified. The pattern participants are described in the next section.

Notice that there are two separate hierarchies of visitors: *sequence value visitors* and *type visitors*. The first performs visitation based on the type of values delivered by a sequence, while the latter performs visitation based on actual sequence type.

8.3.11.3. Participants

Table 8.27 describes the Visitor participants in the words of Gamma et al. [Gamma95, p.334-335] and lists the corresponding implementations developed in the evaluation.

Figure 8.22 — Visitor UML Class diagram

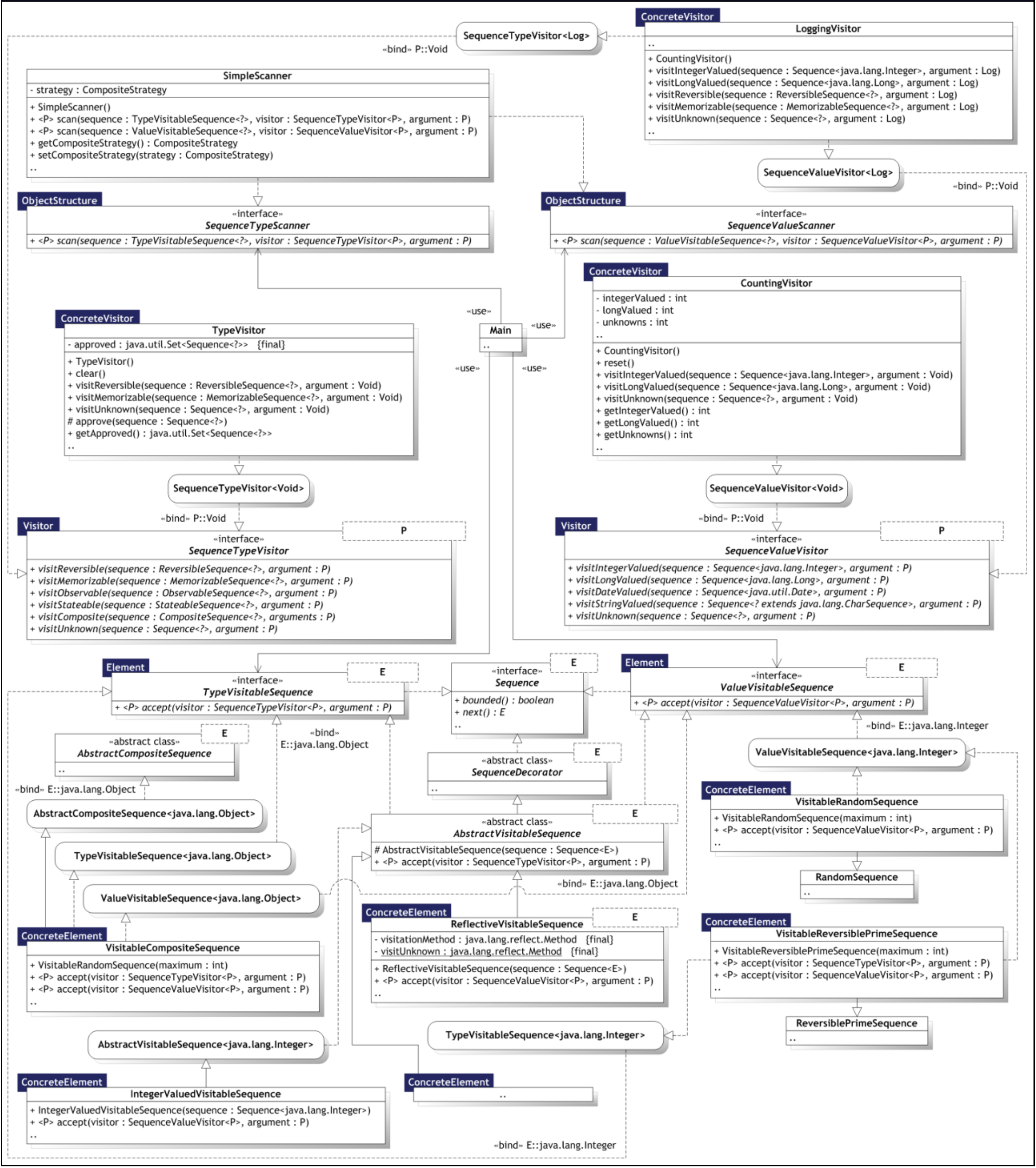


Table 8.27 — Visitor participants

Participant	Description	Implementation
Visitor	– Declares a *visit* operation for each class for **ConcreteElement** in the object structure. The operation's name and signature identifies the class that sends the *visit* request to the visitor. That lets the	`SequenceVisitor<P>` `SequenceTypeVisitor<P>`, `SequenceValueVisitor<P>`

Table 8.27 — Visitor participants

Participant	Description	Implementation
	visitor determine the concrete class of the element being visited. Then the visitor can access the element directly through its particular interface.	
ConcreteVisitor	– Implements each operation declared by **Visitor**. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. **ConcreteVisitor** provides the context for the algorithm and stores its local state. The state often accumulates results during the traversal of the structure.	`TypeVisitor`, `CountingVisitor`, `LoggingVisitor`

Element	– Defines an *accept* operation that takes a visitor as an argument.	`TypeVisitableSequence<E>`, `ValueVisitableSequence<E>`
		
		`AbstractVisitableSequence<E>`

ConcreteElement	– Implements an *accept* operation that takes a visitor as an argument.	`VisitableCompositeSequence`, `VisitableLongSequence`, `VisitableRandomSequence`, `VisitableReversiblePrimeSequence`
		
		`StringValuedVisitableSequence<E>`, `IntegerValuedVisitableSequence`, `DateValuedVisitableSequence`, `ReflectiveVisitableSequence<E>`

ObjectStructure	– Can enumerate its elements.
	– May provide a high—level interface to allow the visitor to visit its elements.
	– May either be a composite or a collection such as a list or a set.
		`SequenceTypeScanner`, `SequenceValueScanner`
		
		`SimpleScanner`

8.3.11.4. Implementation

The `SequenceValueVisitor<P>` cannot overload visitation methods because it performs visitation based on type parameters that are erased at runtime. **Double—dispatch** — As C++, Java only supports single—dispatch, so Visitor is indeed relevant. The `ReflectiveVisitableSequence<E>` is a decorator that uses reflection and naming conventions to avoid a static binding of the actual visitation method. Annotations can also be used as illustrated in `observer.ObserverManager` that uses annotations to determine notification methods. **Who is responsible for traversing the object structure?** — The traversal of the composite sequence structure is made explicit by use of scanners, e.g. the `SequenceTypeScanner` and `SequenceValueScanner` types. Scanners represent the object structure, but only operate on it. Composite sequences do not traverse their children, this is handled by scanners as well. This allows scanners full control over the traversal strategy, for example depth—first or breath—first.

8.4. Summary

Below, we list and summarise the key issues from the detailed evaluation of the "Gang of Four" patterns:

- The evaluation shows that **practically *all* functionality described in the Implementation and Sample Code elements in the "Gang of Four" patterns can be implemented in Java 6**, not just the canonical implementations but ***all* functionality—related issues discussed by Gamma et al**. in these pattern elements.
- The **implementations express "Best Practices" from both Java 6 and design patterns** epitomised by the "Gang of Four" patterns.

The pattern implementations illustrate that **only Adapter with Class scope cannot be done in Java 6**. **All other pattern functionality can be implemented or simulated**, but may require more work. Adapter with Class scope fails because the pattern abstraction and functionality, and not merely implementation issues, is targeted at languages supporting multiple inheritance. In this respect, we conclude that **Adapter with Class scope is in fact a C++ idiom representing the general Adapter abstraction**. Adapter with Object scope is easily implemented in Java 6, and dynamic proxies can furthermore be used to simulate multiple inheritance. However, in general, there are **some problems related to the use of dynamic proxies** that may influence pattern behaviour, such as testing for object equality.

The implementations **express the themes and concepts described by Gamma et al. in a realistic manner**, but **also the "Best Practices" offered by Bloch concerning Java** [Bloch01]. All implementation issues discussed by Gamma et al. in the Implementation and Sample Code elements in the "Gang of Four" patterns are expressed or simulated in the implementations.

9. Evaluation Conclusions

High thoughts must have a high language.
*— **Aristophanes***

The detailed and comparative evaluations disclose much information concerning individual and collective behaviour, including summaries and in—part conclusions where appropriate. This chapter comments on the evaluation as a whole. We determine the compliance level of the developed pattern implementations compared not only to the functionality examined, but also to the concepts and themes described by Gamma et al. Based on the compliance level, we determine how well Java 6 suits the "Gang of Four" patterns from a practical perspective. Several pattern implementations are interesting enough to be classified as *high—lights* of the evaluation. These pattern implementations are briefly described and their functionality, applied techniques, and used features are put into perspective. Finally, for others to examine and judge the evaluation and its results, the evaluation approach must not only be meaningful, but understood. We therefore present an evaluation of the evaluation approach, and thus of the evaluation itself, to put the results into perspective.

9.1. Implementation Compliance

The evaluation goals from section **5.3** have all been achieved. All "Gang of Four" patterns except Mediator have been implemented, while all have been evaluated. All representative pattern functionality described in the Implementation and Sample code elements in the "Gang of Four" pattern descriptions have been addressed. Only Adapter with Class scope fails because the core pattern abstraction and functionality is directly targeted at languages supporting multiple inheritance. All other pattern functionality can be implemented or simulated.

Several patterns include different variants, some close to the canonical implementations offered by Gamma et al., while many utilise Java mechanisms alien to C++, such as annotations, dynamic proxies, bounded generics, etc. Even though the evaluation quite frequently utilises advanced features, the implementations fundamentally rely on OO concepts like types, classes, objects, inheritance, polymorphism, etc., as described by Gamma et al. The specific features used are summarised in section **7.1**. Furthermore, the developed Meta classes also make good use of applying the "Gang of Four" patterns internally. Regardless of mechanisms used, the participants and their functionality as described by Gamma et al. are logically named and clearly identifiable in the source code. Even more so, the individual participants for each pattern is annotated with annotations describing their name and classification down to a level of a specific class, interface, method, or even field. The information is also present in the generated JavaDoc, aiding the identification of participants even further.

The implementations not only express the themes and concepts described by Gamma et al., but also the "Best Practices" described by Bloch for Java applications. This augments the quality of the pattern implementations. We believe that many of the classes produced in the evaluation could be used "as is" in real—life applications. However, the Java 6 implementations are generally more dynamic compared to the canonical implementations. This may conflict with the warning set forth by Gamma et al., which states that highly parameterised, dynamic software may be harder to understand than static software.

9.2. Language Features

Java's core language features promote robustness, pattern intent, and reusability, and form the base of all the pattern implementations. The evaluation confirms the observations regarding dynamic features and reflection made by Norvig and Sullivan examined in chapter 4. We do not claim that Java's dynamic and reflective capabilities make the "Gang of Four" pattern implementations easier, but combined with core features they certainly allow for alternative and more flexible implementations compared to the canonical examples by Gamma et al. Creational and Behavioural patterns especially benefit from such usage. Annotations deserve special mention because they bridge the gap between compile—time and runtime. They can be used to express pattern intent at compile—time as well as runtime, and opens up for many new and interesting ways to express pattern functionality.

The language features examined are all present in Java 5 as well. With minor source code modifications, the code compiles with JDK 1.5.0_12 without errors[14]. Conclusions drawn are therefore also applicable to Java 5.

9.3. High—Lights

On page 82, table 7.1 emphasises what we consider the ***representative high—lights*** discovered during the evaluation. The high—lights are illustrated with dark—blue background, for example the *Abstract Factory* × *Generics* entry. The high—lights represent the pattern functionality expressed using the features. They are representative because the features may be applied in other patterns as well, and hence we put the pattern and feature use in perspective here. The high—lighted features are typically dependent on each other, for example the use of generics, type literals, and constructors for Abstract Factory. Below, the *pattern* × *feature* combinations are summarised in no particular order, and the feature dependencies explained. We believe the most realistic use of high—lights are Abstract Factory, Factory Method, Observer, and Singleton implementations. Memento, Proxy, and State may be too exotic for general use, but still useful in specialised cases. The high—lights are closely related to several of the feature observations described in section 7.1.4.

9.3.1. Abstract Factory and Factory Method

The Factory Method implementation shows that it is possible to create a reusable factory that can create precise generic types, `T`, in a type—safe manner based on *type literals*. The same principle applies to Abstract Factory. The reusable factory component is the `factorymethod.Factory<T>` abstract class. Sub—classing is required to acquire the type of `T` and to specify the constructor to create new instances reflectively. Reflection is only used to determine the type of `T` and to create new instances of `T`, where after `T` instances are accessed normally. This corresponds to minimal reflective usage as described by Bloch [Bloch01, p.159]. The factory is a simple generalisation of the `meta.reflect.InstantiableTypeLiteral<T>` class. Type literals show how patterns can manipulate generic types in a type—safe fashion. This is important as the evaluation indicates a prolific use of generics. Section 7.1.2.2 explains the precise usage of type literals, and the `Factory<T>` class is

[14] The results do **not** apply to Java 1.4. Compiling with Java 1.4 yields 9.400+ errors and 800+ warnings! This is because many of the features were introduced as of Java 5, i.e. generics, annotations, enumerations, covariant return types, and varargs.

illustrated in listing 7.18 on page 108 as well as listing 7.19. Section 8.1.3 provides the detailed description of the Factory Method implementation, while section 8.1.1 provides the description for Abstract Factory. Type literals are also used by Prototype, Proxy, Singleton, and Visitor.

9.3.1.1. Perspective

Central to many designs are Creational patterns such as Abstract Factory and Factory Method. They provide the core business objects the rest of the application depends on; load classes on demand; create other types of factories and interrelated hierarchies; and help promote object composition over class—based inheritance [Gamma95, p.81]. Creational patterns provide flexibility, but often at the expense of more verbose code and little reuse [Gamma95, p.85]. In Java, creating generic product types can also be tricky as this evaluation clearly demonstrates. A reusable component such as `Factory<T>` can remedy many of the pitfalls associated with Creational pattern usage. Its intent is also clear by giving it a meaningful, but general, name. While the component is open for sub—classing for specialised behaviour, e.g. caching, it can be used "as is" for any (non—abstract) product type with an applicable constructor, perhaps as a component in another pattern. However, an exact type must be known at compile—time so it cannot be used when the type is determined at runtime. Still, the Singleton implementation demonstrates how dynamic class loading can be used with type literals to enforce generic bounds in certain situations. These issues certainly have an impact on the design and development process because the designer can now utilise the knowledge and component(s). We believe the functionality provided by `Factory<T>` is needed and general enough to warrant an API inclusion in the long run, but more research and testing is required.

9.3.2. Memento

As the evaluation shows, interface usage is practically unavoidable when applying the patterns using Java, but interface behaviour must be public. As discovered and explained in section 7.1.4.1, the Memento implementation illustrates how the use of friends and interfaces with *narrow* (public) and *wide* (private) methods in the canonical C++ implementations can be transformed to Java 6 in a type—safe manner using *guarded types*. Iterator and State also use the same C++ features. A Java solution is to declare the otherwise private (*wide*) methods as part of the interface. The methods thus become public, but they must then guard access to themselves at runtime. Exception stack traces combined with class literals identify the caller of a given context. Appropriate action can then be taken in case the caller is illegal. The caller can be identified at the level of a class, static initialiser block, constructor, or method. This is essentially an imitation of friends in C++. The cost is potentially error prone runtime access checks enforced by the developer versus compiler guaranteed static checks in C++. Callers are represented by the `meta.reflect.CallerClass` and `meta.reflect.Caller` types and section 7.1.2.1 explains their usage. Memento usage is illustrated in listing 7.16 and listing 7.26. Section 8.3.6 provides the detailed description of the Memento implementation. Caller identification is also used in Singleton and Bridge, but could easily have been applied in other patterns as well.

9.3.2.1. Perspective

Interface implementation is paramount in any Java program, not just in pattern implementations. The demand that all interface methods are public can have serious consequences for any design. It forces verbose types in the sense that the original type will have to implement all interface functionality publicly. It is not uncommon that necessity requires a type to implement an interface expressing functionality that is private by nature. A common way to circumvent this is to use private inner classes to implement, or adapt, functionality as is illustrated in for example the Observer implementation. However, this does not work when communication is required between two different objects as is the case with Memento. Packages can solve the problem locally by declaring otherwise private methods package private or even protected, thereby making all other classes in the same package "friends". It is plausible this is sufficient for pattern usage if the pattern is applied in an Informal way, within the same package and not using interfaces to describe the functionality. If the objects span different packages, however, a public type must be used to establish the contract between the objects to ensure type safety. Memento shows how type safety can be combined with runtime access checks to protect its (otherwise) private state.

The solution promotes type safety for applicable patterns, but may confuse developers because the guarded functionality is callable at development—time. Familiar pattern interfaces may suddenly change, but because of the prolific use of interfaces in Java, the functionality may actually already be expressed publicly. This is the case with the State implementation in this evaluation. The only way to inform users of the intent is to include the information in the documentation, JavaDoc or otherwise. This is not unheard of, however, keeping the `java.lang.UnsupportedOperationException` from the Java Collections Framework in mind. While the principles utilised are sound and directly available in Java, we doubt its usage in real—life systems unless backed by a major player, e.g. Sun. There are too many unknowns and uncertainties connected with its usage, and acquisition of stack traces is not exactly cheap in terms of running time. However, caller identification is already used by Sun in at least the `java.lang.Class` and `java.lang.ClassLoader` classes, using proprietary functionality in `sun.reflect.Reflection.getClassCaller(int)` to fetch the caller. On the other hand, combining guarded types with dynamic proxies and annotations offers new and exiting possibilities. To us, it seems like an interesting candidate for a reusable pattern component, namely a protection proxy. Annotations can document the types and methods guarded, perhaps even declaring the friends, while dynamic proxies can enforce the caller checks on behalf of the (real) guarded type.

9.3.3. Observer

Establishing pattern functionality in the source code is vital for pattern behaviour, naturally, but also to document pattern intent and structure for maintainability and reuse. Annotations bridges the gap between compile—time and runtime, because they allow the developer to augment source code in user—defined ways that can be processed at compile—time **and** at runtime, if so specified. The Observer implementation provides a reusable component that supports different kinds of observers without coercing a common observer type. Coercing a common observer type normally requires extensive adaptation by clients and may violate information hiding, analogous to the interface discussion from the previous section. Instead, `observer.ObserverManager` uses the `meta.reflect.@Executor` annotation to identify both observers **and** their notification methods. It also allows the same type to be notified using several different notification methods. The strategy on how to notify a given observer is thus determined by the observer itself, lessening the need for adaptation. This technique is also applicable to several other Behavioural patterns, at least to Chain of Responsibility, Command, Mediator, Strategy, and Visitor, but at the cost of (reusable) components issuing reflective dispatch of the methods in question. Annotation usage is described and illustrated in section 7.1.2.5. Section 8.3.7 provides the detailed description of the Observer implementation.

9.3.3.1. Perspective

It is worth noting that in our experience, some developers refrain from using annotations, and perhaps more so reflection in general. Primary reasons are missing type safety and performance, but also lack of familiarity. However, several well—known frameworks support annotations, such as EJB3, Hibernate, JBoss Seam, Google Guice, etc., and annotation usage seems to continue to spread. The use of JavaBeans has always relied on naming conventions reminiscent of annotated method usage. As of Java 1.4 reflective access is only twice as slow as normal access [Bloch01, p.158] – which is actually pretty decent – but will still slow any application considerably down if used blindly, for example in critical loops. For design patterns, this is usually not an issue. Type safety is a big issue, however, especially since generics have been introduced to heighten type safety. However, annotations need not be used runtime at all. Nonetheless, in our view, wise (and moderate) use of runtime annotations combined with reflection certainly has its uses in the "Gang of Four" patterns, and we think the evaluation exhibits this.

With proper use of annotations, code inspection becomes much easier. Consider being handed a Java project with tens of thousands lines of code and being able to generate a report on, say, all applications of the Observer and Singleton patterns, including places where the standard Singleton conventions are not used. Singleton shows how the `singleton.@Singleton` annotation is also used to identify Singleton types and behaviour at runtime, but with Sun's Annotation Processing Tool (APT) annotation information can also be utilised at compile—time. The Observer implementation uses annotations in a similar fashion. Report—generation is truly an improvement over "traditional" scattered documentation, especially because annotations are maintained in conjunction with the source code itself. Developers knowing the "Gang of Four" patterns will quickly be able to understand the reasoning behind the solution, because they understand the reasoning behind the patterns **and** have a complete list of the concrete pattern applications generated by annotation processing. If the annotations furthermore express actual pattern functionality, the pattern implementations become very compact.

The evaluation illustrates how simple annotation usage can identify pattern participants and simple functionality, such as identifying notification methods in Observer. Ernst proposes an extension to Java's annotation system (JSR308) that permits annotations to appear on any *use* of a type as opposed to Java 6 where annotations are only applicable to declarations of packages, types, methods, fields, and local variables as well as method parameters [Ernst07, p.1,3]. Ernst suggests usage of annotations such as `@NotNull` and `@ReadOnly` to create custom type qualifiers that provide extra information about a type or a variable, such as `String s = (@NonNull String)o` and `class UnmodifiableList<T> implements @ReadOnly List<@ReadOnly T>` [Ernst07, p.4]. The focus is on static usage, but they can still be used like standard Java 6 annotations because of backward compatibility. Hence, the annotations may be retained at runtime and used reflectively [Ernst07, p.17]. This form of annotations can be used to express pattern participants, properties, and relationships even more detailed. This could be useful as the interactions between participants can be described in detail, for example annotations describing receiver behaviour on methods. For example, the Proxy implementation in this evaluation allows sharing of `meta.model.Sequence<E>` instances via the Handle/Body idiom (dynamic proxies), where a copy of the shared sequence will be created once a *mutator method* is invoked. This is described in section **7.1.2.4**. Mutator methods are specified via their names and fetched reflectively, which is unsafe and error prone. A `@ReadOnly`, or rather a `@Mutator`, annotation could specify this functionality directly in a type—safe fashion on the relevant methods, as in `public E current() @ReadOnly {..}` and `public E next @Mutator {..}`. The use of such low—level annotations combined with dynamic proxies may perhaps allow dynamic implementation of some of the "Gang of Four" patterns, particular Behavioural ones. How much this type of annotations will offer compared to the standard use is questionable and would require extensive research to determine.

As a final comment on annotation usage in patterns, note that several CASE tools already have support for the "Gang of Four" patterns, for example Rational Rose. The NetBeans IDE used as secondary IDE in this thesis has tool support to apply all "Gang of Four" patterns, but with the improved modularity offered by annotations, tooling can be much improved. This goes both ways, i.e. code—generation and reverse engineering. Common standards might even evolve in form of annotations used to describe the individual "Gang of Four" patterns.

9.3.4. Proxy and State

Java has built—in support for the Proxy pattern via dynamic proxies. Dynamic proxies depend heavily on reflection, especially reflectively invoking methods. Gamma et al. describe four different types of proxies, namely *remote proxies*, *virtual proxies*, *protection proxies*, and *smart references* [Gamma95, p.208-209]. All can be implemented using dynamic proxies, and the evaluation provides examples of all but remote proxies, for example Handle/Body idiom as already described (smart reference). The State implementation illustrates how dynamic proxies can be used to simulate dynamic inheritance by changing the target of method invocations at runtime. By changing the target object, the implementation effectively changes. The implementations need not even have the same type, only matching method signatures. Section **7.1.2.4** explains the general usage of dynamic proxies in the pattern implementations. Practical application is illustrated in listing **7.21** on page 112, where Prototype uses Proxy to make objects prototypical. Section **8.2.7** and **8.3.8** provide the detailed descriptions of the Proxy and State implementations, respectively. Section **8.1.4** provides the detailed implementation for Prototype, illustrating the use of Proxy.

9.3.4.1. Perspective

Dynamic proxies are not merely of academic interest. Annotations are runtime represented by dynamic proxies, for example. Annotation and dynamic proxy collaboration is a very interesting area to pursue concerning the pattern functionality just described in section **9.3.3.1**. The evaluation shows that dynamic proxies can be used to implement much of the alternative pattern implementations described by Gamma et al. However, usage may cause unforeseen consequences and change the implementation of the pattern participants considerably. Behavioural patterns can benefit from dynamic proxy usage because it allows the behaviour to change at runtime. This is clearly illustrated in the State implementation. Structural patterns such as Adapter and Decorator can also benefit from dynamic proxies.

Several of the properties associated with dynamic proxies can make them ill suited for pattern implementations. In general, patterns that rely on inheritance as opposed to interface implementation (composition) could be problematic, for example Singleton and Template Method. As with any reflective feature, dynamic proxies should be used with care. They can have tremendous impact on the pattern and application functionality in ways not expected, not even considering the increase in code required to make them work, which may clutter otherwise concise implementations. Such code can be hard to maintain and make pattern implementations harder to understand. This contradicts traditional OO lore and especially forces like Efficiency, Reliability, and Testability as described by Buschmann et al. in section **3.4**. Refactoring and component usage as for example the `meta` packages in this evaluation can help. Consider as part of an API a Creational pattern such as Abstract Factory, Builder, or Factory Method that produces dynamic proxies instead of concrete types (classes and/or interfaces). Creational patterns are often pivotal in any design because they have the responsibility of creating new objects on demand. Dynamic proxies will thus proliferate throughout the application, probably without the client knowing. They will be used in composites, iterators, commands, decorators, etc. Problems related to performance, marshalling, serialization, object identities, testing on and casting to exact classes, etc., could easily arise.

9.3.5. Singleton

The evaluation demonstrates two important, and as far as we know, novel issues regarding the Singleton pattern implemented in Java: 1) As of Java 5, Java has built—in support for Singleton semantics; and 2) sub—classing is possible using *guarded types*.

The *Singleton—as—Single—Constant* idiom defined in this thesis illustrates Java's support for the Singleton pattern. By using enumerations as the mean to enforce Singleton behaviour, it is guaranteed that the Singleton type is non—instantiable and has a single global point of access – the singleton constant. This corresponds to the Singleton requirements set forth by Gamma et al. All the desirable traits of enumerations will inherently be expressed in the Singleton type, such as automatic instantiation; serializable; non—cloneable; known at compile—time; etc. On the other hand, the enumeration limitations will also be expressed. The Singleton constant is accessed directly, without the use of method indirection; arguments cannot be supplied at construction time; and it cannot be sub—classed. Section **7.1.1.4** explains the general usage of enumerations in the pattern implementations, and the *Singleton—as—Single—Constant* idiom is illustrated in listing **7.9** on page 94. Section **8.1.5** provides the detailed description of the Singleton implementation.

Like Memento, sub—classing of Singleton requires the singleton type to be a guarded type that validates the sub—class caller. Combining this with hiding of static methods, the evaluation shows that singleton inheritance is practically feasible, but must be applied in an Informal way for each applicable singleton type. The same restrictions apply as for Memento, though the Singleton sub—class constructor is already declared protected. A certain level of trust is still required, though. Callers are represented by the `meta.reflect.CallerClass` and `meta.reflect.Caller` types and section **7.1.2.1** explains their usage. Singleton sub—classing is illustrated in listing **7.15** on page 104.

9.3.5.1. Perspective

In the discussion at [PPR, p.JavaSingleton], it is suggested that Java should support Singleton behaviour directly via a `single` keyword, i.e. that a class can declare it is a singleton like `public single class SingletonClass`. The use of a keyword clearly states the pattern intent, but Java 6 supports Singleton behaviour via enumerations and can use annotations to clarify the pattern intent. This is illustrated in the evaluation via the `singleton.@Singleton` annotation. More so, if annotation usage where expanded as discussed in section **9.3.3.1**, `@Singleton` could be enforced wherever the Singleton type is used. On the other hand, since enumerations can be utilised, syntactic sugar in form of a `single` keyword seems feasible similar to arrays and varargs usage.

An interesting idea is to combine guarded types and annotations. Annotations can be used to specify "friends", particularly if *usage* of methods can also be annotated.

9.4. Evaluation Approach

To put the experiments and evaluation in perspective, we give a short evaluation of the evaluation approach itself. While the intent of the evaluation approach is to address all pattern functionality in specific elements, it is loosely formulated. Especially the comparative evaluation. This leaves room for interpretation, but it may also collide with the premise that the results can be verified. This is reminiscent of using pattern descriptions. Concrete issues include the functionality to examine, i.e. all information in the Implementation and Sample Code elements, the semantics of the language chosen as the catalyst, and the format of the detailed evaluation. This leaves many variables to the discretion of the evaluator. As the functionality described by Gamma et al. is written in natural language, it is up for interpretation. The features chosen can be an arbitrary sub—set of the language used. How verbose or exacting the written evaluation is is also up to the evaluator. Ergo, while the evaluation approach tries to structure the evaluation, it inherently reflects the lingual form of the patterns it investigates. Even though conclusions can be drawn, they will still be subjective. For example, the choice of language features to investigate and how well the evaluator knows these features will in effect determine the output of the evaluation. This is one of the reasons why we changed focus in this thesis from using several languages on a few patterns to using a single language on all the "Gang of Four" patterns: performing an in—depth evaluation using a language with little to no practical experience does not make sense. Language knowledge and experience is paramount, even more so than experience with the "Gang of Four" patterns. Before this evaluation, there were several patterns we had never implemented "as is", for example Builder, Chain of Responsibility, Interpreter, and Visitor. It is hard to make concrete, but we argue that the implementations at least express the pattern qualities associated with the Implementation and Sample Code elements as listed in table **3.2** on page 41, i.e. Composibility, Equilibrium, Generativity, and Openness. This is because we adhere to the "Gang of Four" pattern descriptions **and** to "Best Practices" in Java as described by Bloch [Bloch01].

Amidst conducting the implementations and evaluations, we realised that the Sample Code element really does not provide new information compared to what is written in the Implementation element; it merely provides

C++ (and Smalltalk) examples of the topics discussed in the Implementation element. Instead, the Applicability and Consequences elements seem better candidates to include in future evaluations for additional information. The patterns describe the interesting forces in the Consequences element that truly defines the pattern behaviour as in for example Strategy, and Visitor even provides source code in the element. Another example is Proxy that describes the different kind of proxies in the Applicability element, but only shows examples of how to implement few of these in the Implementation and Sample Code elements. The Consequences element further describes interesting proxy features such as *copy—on—write* that are not illustrated. Still, as the evaluation shows, it is impossible not to include functionality from (almost all) other elements. The point is that the patterns describe more than the source code, and the source code only makes sense within the pattern context. However, performing a structural evaluation based on these suggested elements as well may be difficult because they focus more on natural language and abstractions than on programming languages and practical examples. The evaluation will not become less subjective, which is illustrated by the pattern qualities we identified and associated with the Abstraction pattern element as listed in table **3.2**.

A question that arises from an evaluation of this kind is whether the evaluation investigates the "Gang of Four" patterns using a given language, or if it evaluates the language by merely using the "Gang of Four" patterns as the showcase. The evaluation approach tries to set the bar high and effectively asks how well suited a given language is to express the "Gang of Four" pattern abstractions and qualities, albeit only using certain elements from the pattern descriptions. The "Gang of Four" pattern concepts, themes, and functionality *are* the focus of attention because the language features are used to express them. Nonetheless, the evaluation is undeniably also a testament to the language used, illustrated especially by the comparative evaluation and the high—lights. Because of the extensive focus on implementation related pattern elements, the investigation also becomes very language specific and technical. The Implementation and Sample Code elements primarily use C++ to illustrate pattern functionality, while dynamic languages, e.g. Smalltalk and Self, are used in the Meta—information in these elements to discuss alternative implementations. Little remarks by Gamma et al. regarding (not illustrated) pattern functionality cause verbose (Java 6) implementations and associated evaluations that have to analyse the solution. This is for example the case with the State implementation using dynamic proxies to simulate dynamic inheritance: a paragraph of eight lines [Gamma95, p.309] yields 1000+ lines of code overall (including documentation and blank lines). The evaluation shows that Java 6 can address practically all issues discussed in the examined elements. In that context, Java 6 offers the designer a "better toolbox" to express the pattern functionality. Of course, in real—life applications there are other considerations to make and forces to consider, for example Efficiency and Reliability as discussed in section **3.4** on page 38. C++ is faster compared to Java 6, and reflection is more error prone than compiled code. Hence, the evaluation does not conclude that Java 6 should be preferred in all contexts. It merely illustrates how well suited the language, e.g. Java 6, is to express the "Gang of Four" abstractions. The rest is up the designer. However, it is important to remember that only parts of the pattern descriptions are investigated. Many of the patterns could justify having a small book dedicated to them, so could many of the Java 6 features. In fact, many of them do.

Another question is how the specific set of features is chosen. The features listed in table **7.1** are selected based on language experience, but also because of prior knowledge regarding the "Gang of Four" patterns.

Other features could have been primitive types, assertions, and more focused analysis of polymorphism. Had reflection not been chosen, the evaluation outcome would have been less impressive concerning supported "Gang of Four" functionality. Still, the evaluation clearly expresses the features evaluated, thereby establishing the language context within to judge the evaluation outcome. As table **7.1** illustrates, some patterns utilise many features, while others utilise fewer. This is a testament to pattern granularity and abstraction as in Facade compared to Singleton, but also indicates that it is almost impossible implementation—wise not to favour patterns supporting multiple (and fun) features.

The division between detailed and comparative evaluation is easier to express in structural literary form than it is to conduct in practice. The process of implementing the patterns in effect took the form of Iterative development as described in chapter **2**, where discoveries in one pattern would lead to the evolution of other similar patterns and so forth. Because of different pattern abstractions and relationships, the time spent in the different implementations has not been identical. Nonetheless, since all implementations express the pattern functionality, we believe they express the qualities and forces described by Gamma et al. as well. Including the Consequences element in future evaluations would merit a better judgement of this, however.

Conclusion — The evaluation approach offers a way to investigate and judge how well a given language can express the "Gang of Four" functionality expressed in the Implementation and Sample Code elements, but it does not draw any conclusions as to whether the language catalyst should be used in a given scenario. This is – of course – left to the discretion of the designer. The conclusions on the practical features used are subjective, but can be verified using a similar evaluation set—up. The evaluation serves as a tool from which practical experience can be drawn, but it is too loosely formulated to be considered a methodology.

10. Conclusion

A question that sometimes drives me hazy:
am I or are the others crazy?
*— **Albert Einstein***

The goals of this thesis have all been reached. We have presented the basic theory necessary to understand the "Gang of Four" patterns with a practical approach focusing on Java 6. We have defined an evaluation approach on how to evaluate the "Gang of Four" patterns using a language as the catalyst. Using this approach, we have implemented the "Gang of Four" design patterns in Java 6, and performed a detailed and a comparative evaluation from the perspective of a practicing designer and/or developer. The evaluations and the conclusions have been presented in this thesis. A short evaluation of the defined evaluation approach has also been offered. This chapter presents the overall thesis results and conclusions from our established perspective. We outline ideas for future work, and end this thesis with a few final remarks regarding the project and work performed.

10.1. Perspective

This thesis is a report on what has been achieved in this project. Its approach is experimental and practical, explanatory in nature. This is because we view the use of design patterns as a practical discipline based on the inclinations of the user. It is all about choices. The choice to use patterns as an aid in the design process, the choice of pattern to use for a given context, the choice of programming language. Patterns are all about interpretation. Interpretation of the pattern description, of the sample code, and ultimately of the pattern functionality in order to apply the pattern. Patterns are all about learning from experience. Utilising the experience of others prevents the need to reinvent the wheel. However, because choices and interpretations must be made, the use of patterns is also a learning process that augments the user's experience. While a design pattern is recorded by its author, the success and eventual applicability is based on the choices, interpretations, and experiences made by the user. The evaluation presented in this thesis is no different. Many choices and decisions have been made during the project, and a lot of valuable experience has been gained from the in—depth work with the "Gang of Four" patterns and Java 6. Several important issues related to their practical application have been learned. This thesis has strengthened our belief that the "Gang of Four" patterns can be a valuable aid in the design process and that they are applicable using languages other than C++ and Smalltalk. By using Java 6 as the programming language we have illustrated that the patterns can be implemented in a variety of ways that might differ greatly from the canonical "Gang of Four" implementations, including as far as we can tell novel approaches in several pattern implementations, while still expressing the pattern intent, qualities, and forces. However, the high level of abstraction combined with practical experience is a key issue as to why we do not believe built—in language support is possible for design patterns in general.

10.2. Results

As stated in the initial goals, the primary objective of this Master's Thesis is for the undersigned to obtain a Master's Degree in Computer Science. This thesis represents a project with a formal workload of 30 ECTS, and it presents the work performed during the project. To obtain the degree, the project and thesis must be approved after an oral examination based on this thesis. We feel the work performed has been educational; we perceive it as comprehensive and satisfactory, rigorously presented, thereby warranting such an approval. The introduction defined a set of sub—goals, I-IV, for the work to be performed during the project. The goals have all been achieved and presented in this thesis:

I. **Theory and Background** — We have performed extensive research and presented theory and background related to OO development and patterns relevant to the implementation and evaluation, but with focus on practical application and Java 6 usage. We have explained how design patterns can be used as a practical tool in the development process regardless of the OO method used, and identified the OO concepts and themes relevant for the "Gang of Four" patterns, including their relation to Java 6. We have introduced the pattern theory on which the "Gang of Four" patterns are founded, and provided numerous examples to illustrate OO and pattern theory from a more practical and Java related view. We have presented the "Gang of Four" pattern system and established an informal classification of their relationships. We have examined and related several studies on how language features can influence the "Gang of Four" design patterns. The studies conclude that dynamic features and reflection can aid the pattern implementation, while static features can promote Structural pattern componentizations. We have shown this is a fine match to Java 6 with its mixture of static and dynamic features. We believe the extensive focus on practical aspects and Java 6 with regards to the theory and the "Gang of Four" patterns provides consistency throughout this thesis.

II. **Evaluation Approach** — We have defined a simple, but structured approach used to conduct the evaluation. The approach is general enough for reuse in similar evaluations, using other languages as catalysts. By specifically utilising pattern elements described by Gamma et al., the approach is closely tied to the "Gang of Four" patterns, but it makes it easier to verify the otherwise subjective results.

III. **Implementation** — We have implemented the "Gang of Four" patterns in Java 6 within the realm of the evaluation approach, fully documented in JavaDoc, with clearly identifiable pattern participants. Several reusable Meta classes help provide realistic pattern functionality. We have been methodical and thorough in merging "Best Practices" from the "Gang of Four" design patterns with that of Java. The quality of the implementations is therefore high and adhere to the concepts and themes defined by Gamma et al., but also to Java—specific issues described by Bloch [Bloch01].

IV. **Evaluation** — We have performed a comparative as well as detailed evaluation of the pattern implementations using the defined approach. The detailed evaluation provides comprehensive UML class diagrams, description of pattern participants, but most importantly illustrates how practically *all* functionality described in the relevant "Gang of Four" pattern elements can be implemented in Java 6. The comparative evaluation provides an in—depth investigation on how Java 6 features and mechanisms are applied across all patterns to identify common traits and relationships. Based on the evaluations, we have identified the following high—lights:

- The ***Singleton—as—Single—Constant* idiom** explains how a single enumeration constant can express the Singleton qualities and forces described by Gamma et al.
- **Reflection and runtime—retained annotations** provide very flexible pattern implementations exhibiting behaviour that normally is ascribed to dynamic languages only. Creational and especially Behavioural patterns can benefit from the usage, for example Observer by annotating the notification methods without requiring a common observer interface.
- ***Type literals*** to enforce type safety for generic types and to act as reusable factories in Creational patterns such as Abstract Factory and Factory Method.
- **Dynamic proxies** can be utilised in especially Structural patterns such as Proxy and Bridge, but also to simulate dynamic inheritance as illustrated in the State implementation.
- ***Guarded types*** allow runtime protection of interface behaviour that has to be declared public in Java for compile—time type safety. This facilitates Singleton inheritance, and at least the Memento, and possibly State, implementations.

We have also evaluated the evaluation approach and determined it to be useful, albeit found it wanting in certain respects, and recommended improvements to make future evaluations even more thorough.

The contributions made by this thesis are:

- We have shown that practically *all* pattern functionality described by Gamma et al. in the Implementation and Sample Code elements can be implemented in Java 6. While Java 6 does not support several of the described language mechanisms, alternative means exist and can be utilised successfully. The pattern implementations illustrate such usage and this thesis explains it. This is a strong indication that Java is well suited to express *all* pattern abstractions described by Gamma et al.
- We have defined a simple evaluation approach that can be utilised by others to determine the level of compliance between the "Gang of Four" patterns and a language catalyst.
- We have presented what we believe are novel approaches on how to implement several of the "Gang of Four" patterns in Java 6, in particular Singleton, Abstract Factory, Factory Method, Memento, Observer, Proxy, and State (high—lights). Many of the principles applied are applicable for other patterns as well, such as *type literals*, *guarded types*, dynamic proxies, and annotation usage in particular.
- We have linked the themes described by Gamma et al. to practical concepts expressed in Java 6.

10.3. Future Work

Before a few final remarks, we list possible future studies related to the work presented in this thesis.

1. Adjust the evaluation approach to include at least the Consequences element in favour Sample Code for a more thorough evaluation, because Sample Code in general did not provide additional information.

2. Provide in—depth studies of the presented high—lights, including studies on possible language support for described patterns. For example, what will be the cost of adding a `singleton` keyword to Java to express Singleton functionality as when implemented using the *Singleton—as—Single—Constant* idiom? Are there patterns or language idioms that could be promoted to language features? Is it worth adding generic factories to the Java API? Is dynamic proxy usage viable in real—life systems? Annotations?

3. Describe new "participants" or functionality expressed in the evaluation. This includes Command spawning new commands, generic factories, *Singleton—as—Single—Constant* idiom, *guarded types*, etc.

4. Using different language catalysts for evaluation. It could be truly interesting to apply the evaluation approach using a prototype—based language and corresponding OO taxonomy. This is no small task as the concepts and themes described by Gamma et al. are clearly targeted at class—based languages.

5. Apply the evaluation approach using different design pattern systems, for example the "POSA" patterns. This will require adaptation to the pattern format used, but is certainly manageable. A bigger task is perhaps to ensure a consistent understanding of the OO concepts and themes. It could be interesting to see if the features and techniques applied in this evaluation are applicable elsewhere.

6. One of the reasons for choosing Java 6 over Java 5 was the support for annotation processing via Sun's Annotation Processing Tool (APT). The original idea was to prime APT with a handler that would traverse the source code and gather information about patterns and participants, thereby generating a summary of pattern usage. Perhaps even compiling code that could be utilised at runtime to enhance pattern support akin to Meta Data expressed with classes. This could also aid the documentation. As explained, this could be a very interesting tool to implement, with many exiting possibilities.

7. Target the implementations for reusable components. The evaluation shows that Creational and especially Behavioural patterns are good candidates for componentization.

8. For completeness, implement Mediator and evaluate the secondary objective of this thesis, to establish if this thesis could be helpful to others pondering with the "Gang of Four" patterns and language issues. This could be a practical survey among co—workers, discussion group, or lectures on the findings, etc.

10.4. Final Remarks

Our initial goals have been reached, though we did not accomplish all we set out to do, in part because we changed focus half way through corresponding to the work description available as [Rode07a]. We believe others may benefit from the experiences and results presented in this thesis, perhaps using it actively and practically. It has been educational to write, and we hope it will be educational to read, thus causing developers and others to think more actively about design patterns.

Bibliography

There is no quote for the bibliography.
— Maz Spork

References post fixed with a date indicates the date they were retrieved, in the format dd.mm.yyyy. Note that the bibliography has been updated to reflect thesis changes made *after* grading.

[Alexander77] **A Pattern Language** – Towns, Buildings, Construction
Christopher Alexander
1977; Oxford University Press; ISBN 0195019199

[Alexander79] **The Timeless Way of Building**
Christopher Alexander
1979; Oxford University Press; ISBN 0195024028

[Alexander05a] **Generative Codes** – The Path to Building Welcoming, Beautiful, Sustainable Neighborhoods
Christopher Alexander, *Randy Schmidt*, *Maggie Moore Alexander*, *Brian Hanson*, and *Michael Mehaffy*
2005; DRAFT, Version 17;
http://www.livingneighborhoods.org/library/generativecodesv10.pdf (29.06.2006)

[Alexander05b] **The Relationships Between Pattern Languages, Sequences, and Generative Codes**
Christopher Alexander
2005; http://www.livingneighborhoods.org/ht-0/patternlanguages.htm (20.05.2007)

[Appleton] **Patterns for Java and Distributed Computing**
Brad Appleton
http://www.cmcrossroads.com/bradapp/javapats.html (26.06.2006)

[Appleton97] **On the Nature of *The Nature of Order***
Brad Appleton
1997; http://www.cmcrossroads.com/bradapp/docs/NoNoO.html (26.06.2006)

[Appleton00] **Patterns and Software** – Essential Concepts and Terminology
Brad Appleton
2000; http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html (26.06.2006)

[Armstrong06] **The Quarks of Object—Oriented Development**
Deborah J. Armstrong
2006; Communications of the ACM, Volume 49, No 2, pages 123-128; ACM Press

[Arnout06] **Pattern componentization: The Factory example**
Karine Arnout and *Bertrand Meyer*
2006; Innovations in Systems and Software Technology (a NASA journal);
http://se.ethz.ch/~meyer/publications/nasa/factory.pdf (30.04.2007)

[AspectJ] **AspectJ Web Site**
http://www.eclipse.org/aspectj/ (21.05.2007)

[Bacon] **The "Double—Checked Locking is Broken" Declaration**
David Bacon, *Joshua Bloch*, *Jeff Bogda*, *Cliff Click*, *Paul Haahr*, *Doug Lea*, *Tom May*, *Jan—Willem Maessen*, *Jeremy Manson*, *John D. Mitchell*, *Kelvin Nilsen*, *Bill Pugh*, and *Emin Gun Sirer*
http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html (13.01.2009)

[Baroni03] **Design Patterns Formalization**
Aline Lúcia Baroni, *Yann-Gaël Guéhéneuc*, and *Hervé Albin-Amiot*
2003; Technical Report 03/03/INFO; Computer Science Department, École des Mines de Nantes;
http://www.iro.umontreal.ca/~ptidej/Publications/Documents/Research+report+Metamodeling+June03.doc.pdf
(17.05.2007)

[Beck87] **Using Pattern Languages for Object—Oriented Programs**
Kent Beck and *Ward Cunningham*
1987; Technical Report CR-87-43, Presented at OOPSLA-87;
Abstract at http://c2.com/doc/oopsla87.html (20.06.2006)

[Bloch01] **Effective Java** - Programming Language Guide
Joshua Bloch
2001; Prentice Hall; ISBN 0201310058

[Bloch08] **Effective Java** - Second Edition
Joshua Bloch
2008; 2nd Edition; Prentice Hall; ISBN 0321356683

[Borchers99] **Pattern Languages in Human—Computer Interaction** - Suite Overview
Jan O. Borchers
1999; http://www.hcipatterns.org/tiki-download_file.php?fileId=10 (25.06.2006)

[Bracha04] **Generics in the Java Programming Language**
Gilad Bracha
2004; http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf (19.10.2006)

[Buschmann96] **Pattern—Oriented Software Architecture** - A System of Patterns, Volume 1
Frank Buschmann, *Regine Meunier*, *Hans Rohnert*, *Peter Sommerlad*, and *Michael Stal*
1996; John Wiley & Sons, Inc.; ISBN 0471958697

[Brown98] **AntiPatterns** - Refactoring Software, Architectures, and Projects in Crisis
William J. Brown, *Raphael C. Malveau*, *Hays W. McCormick*, and *Thomas J. Mowbray*
1998; John Wiley & Sons, Inc.; ISBN 0471197130

[Cockburn01] **Writing Effective Use Cases**
Alistair Cockburn
2001; Addison—Wesley; ISBN 0201702258

[Coplien] **Software Design Patterns: Common Questions and Answers**
James O. Coplien
ftp://st.cs.uiuc.edu/pub/patterns/papers/PatQandA.ps (20.05.2007)

[Coplien91] **Advanced C++ Programming Styles and Idioms**
James O. Coplien
1991; Addison—Wesley; ISBN 0201548550

[Dominus02] **"Design Patterns" Aren't**
Mark Jason Dominus
2002; http://perl.plover.com/yak/design (25.06.2006)

[DeMichiel87] **The Common Lisp Object System: An Overview**
Linda G. DeMichiel and *Richard P. Gabriel*
1987; Lucid, Inc.; http://www.dreamsongs.com/NewFiles/ECOOP.pdf (01.10.2006)

[Eckel03] **Thinking in Patterns** - Problem—Solving Techniques using Java, Revision 0.9
Bruce Eckel
2003; http://www.mindviewinc.com/downloads/TIPatterns-0.9.zip (20.06.2006)

[Eden98] **Giving "The Quality" A Name**
Amnon H. Eden
1998; http://www.eden-study.org/articles/1998/giving_the_quality_a_name.pdf (20.05.2007)

[Eden02] **A Theory of Object—Oriented Design**
Amnon H. Eden
2002; Information Systems Frontiers 4:4, pages 379-391;
http://www.eden-study.org/articles/2002/isf4(4).pdf (18.02.2007)

[Eden04] **LePUS2** - Updates and Challenges
Amnon H. Eden
2004; http://www.eden-study.org/articles/2004/Lepus2-updates-challenges.pdf (21.06.2006)

[Ernst07] **Annotations on Java types** - JSR 308 working document
Michael D. Ernst
2007; http://groups.csail.mit.edu/pag/javari/java-annotation-design.pdf (16.08.2007)

[Fowler97] **Analysis Patterns** - Reusable Object Models
Martin Fowler
1997; Addison—Wesley; ISBN 0201895420

[Fowler03] **Patterns of Enterprise Application Architecture**
Martin Fowler
2003; Addison—Wesley; ISBN 0321127420

[Fowler04] **Is Design Dead?**
Martin Fowler
2004; http://www.martinfowler.com/articles/designDead.html (22.06.2006)

[Fowler05] **CallSuper**
Martin Fowler
2005; http://www.martinfowler.com/bliki/CallSuper.html (16.05.2007)

[Fowler06] **Writing Software Patterns**
Martin Fowler
2006; http://www.martinfowler.com/articles/writingPatterns.html (15.05.2007)

[Gafter06] **Super Type Tokens**
Neal Gafter
2006; http://gafter.blogspot.com/2006/12/super-type-tokens.html (16.05.2007)

[Gamma95] **Design Patterns** - Elements of Reusable Object—Oriented Software
Erich Gamma, *Richard Helm*, *Ralph Johnson*, and *John Vlissides*
1995; Addison—Wesley Longman, Inc.; ISBN 0201633612

[Gosling05] **The Java Language Specification**
James Gosling, *Bill Joy*, *Guy Steele*, and *Gilad Bracha*
2005; 3rd Edition; Addison—Wesley; ISBN 0321246780;
http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf (09.10.2006)

[Graham02] **Revenge of the Nerds**
Paul Graham
2002; http://www.paulgraham.com/icad.html (27.07.2006)

[Grand98] **Patterns in Java** - A Catalog of Reusable Design Patterns Illustrated with UML, Volume 1
Mark Grand
1998; John Wiley & Sons, Inc.; ISBN 0471258393

[Grand99] **Patterns in Java** - A Catalog of Reusable Design Patterns Illustrated with UML, Volume 2
Mark Grand
1999; John Wiley & Sons, Inc.; ISBN 0471258415

[Hacknot07] **Invasion Of The Dynamic Language Weenies**
www.hacknot.info
2007; http://www.hacknot.info/hacknot/action/showPrintableEntry?eid=93 (18.05.2007)

[Halloway07] **Design Patterns are Code Smells**
Stuart Dabbs Halloway
2007; http://relevancellc.com/2007/5/17/design-patterns-are-code-smells.html (04.06.2007)

[Hannemann02] **Design Pattern Implementation in Java and AspectJ**
Jan Hannemann and *Gregor Kiczales*
2002; Proceedings of the 17th Annual ACM conference on OOPSLA; pages 161-173;
http://www.cs.ubc.ca/labs/spl/papers/2002/oopsla02-patterns.pdf (22.05.2007)

[Hillside] **Hillside.net** - Online Pattern Catalog
http://hillside.net/patterns/onlinepatterncatalog.htm (26.06.2006)

[Hohmann98] **Getting Started With Patterns**
Luke Hohmann
1998; Software Development Magazine, February issue;
http://www.lukehohmann.com/paper-gettingstartedwithpatterns.php (26.06.2006)

[Langer06] **Java Generics** - Frequently Asked Questions
Angelika Langer
2007; http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.pdf (03.06.2007)

[Larman04] **Applying UML and Patterns** - An Introduction to Object—Oriented Analysis and Design and Iterative Development
Craig Larman
2004; 3rd Edition; Prentice Hall; ISBN 0131489062; http://authors.phptr.com/larman/uml_ooad/index.html

[Lea93] **Christopher Alexander: An Introduction for Object—Oriented Designers**
Doug Lea
1993; http://g.oswego.edu/dl/ca/ca/ca.html (23.06.2006)

[Lea00] **Patterns—Discussion FAQ**
Doug Lea (maintained by)
2000; http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html (23.06.2006)

[Livshits05] **Turning Eclipse Against Itself: Finding Bugs in Eclipse Code Using Lightweight Static Analysis**
V. Benjamin Livshits
2005; Eclipsecon '05 Research Exchange;
http://suif.stanford.edu/~livshits/papers/pdf/eclipsecon05-checklipse.pdf (14.04.2007)

[Meyer03] **The power of abstraction, reuse and simplicity: an object—oriented library for event—driven design**
Bertrand Meyer
2003; Festschrift in Honor of Ole-Johan Dahl, Lecture Notes in Computer Science 2635;
http://se.ethz.ch/~meyer/publications/lncs/events.pdf (30.04.2007)

[Meyer06] **Componentization: The Visitor Example**
Bertrand Meyer and *Karine Arnout*
2006; Computer (IEEE); http://se.ethz.ch/~meyer/publications/computer/visitor.pdf (29.04.2007)

[McCormick01] **AntiPatterns**
Hays W. McCormick and *Raphael C. Malveau*
2001; Dr. Dobb's Journal, July 22;
http://www.ddj.com/article/printableArticle.jhtml?articleID=184410581&dept_url=/ (22.06.2006)

[MDA] **OMG Model Driven Architecture**
http://www.omg.org/mda/

[Norvig96] **Design Patterns in Dynamic Programming**
Peter Norvig
1996; http://norvig.com/design-patterns/design-patterns.ppt (21.06.2006)

[Orme05] **Java does Duck Typing**
David Orme
2005; http://www.coconut-palm-software.com/the_visual_editor/?p=25 (07.07.2007)

[PPR] **Portland Pattern Repository**
http://c2.com/cgi/wiki (23.06.2006)

[Rode07a] **Evaluating Software Design Patterns** - Thesis Specification
Gunni Rode
2007; http://www.rode.dk/thesis/download/thesis-specification.pdf

[Rode07b] **Evaluating Software Design Patterns** - Thesis Defence
Gunni Rode
2007; http://www.rode.dk/thesis/download/thesis-defence.pps

[Rogers01] **Encapsulation is not information hiding**
Paul Rogers
2001; JavaWorld; http://www.javaworld.com/cgi-bin/mailto/x_java.cgi (03.05.2007)

[RUP] **Rational Unified Process**
http://www-306.ibm.com/software/awdtools/rup/

[Schmidt] **Overview of Object—Oriented Design Principles and Techniques**
Douglas Schmidt
http://www.cs.wustl.edu/~schmidt/PDF/ood-overview4.pdf (19.03.2007)

[Schmidt00] **Pattern—Oriented Software Architecture** - Patterns for Concurrent and Networked Objects, Volume 2
Douglas Schmidt, *Michael Stal*, *Hans Rohnert*, and *Frank Buschmann*
2001; John Wiley & Sons, Inc.; ISBN 0471606952

[SEI] **Carnegie Mellon Software Engineering Institute** - Software Technology Roadmap
http://www.sei.cmu.edu/str/descriptions/ (19.02.2007)

[Sethi96] **Programming Languages** - Concepts & Constructs
Ravi Sethi
1996; 2nd Edition, Addison—Wesley; ISBN 0201590654

[Sierra06] **SCJP Sun Certified Programmer for Java 5** - Study Guide
Kathy Sierra and *Bert Bates*
2006; McGraw—Hill; ISBN 0072253606

[Stroustrup91] **The C++ Programming Language**
Bjarne Stroustrup
1991; 2nd Edition, Addison—Wesley; ISBN 0201539926

[Sullivan02a] **Advanced Programming Language Features for Executable Design Patterns** — Better Patterns Through Reflection
Gregory T. Sullivan
2002; ftp://publications.ai.mit.edu/ai-publications/2002/AIM-2002-005.pdf (18.05.2007)

[Sullivan02b] **Advanced Programming Language Features and Software Engineering: Friend or Foe?**
Gregory T. Sullivan
2002; http://people.csail.mit.edu/gregs/proglangsandsofteng.ppt (18.05.2007)

[Taibi07] **Design Pattern Formalization Techniques**
Toufik Taibi
2007; IGI publishing; ISBN 1599042193

[UML05] **Unified Modelling Language 2.0**
Formal/05-07-04, Formal/05-07-05; ISO/IEC 19501;
2005; Object Management Group (OMG);
http://www.omg.org/technology/documents/formal/uml.htm (20.06.2006)

[Vieiro06] **Singletonitis**
Antonio Vieiro
2006; http://www.antonioshome.net/blog/pivot/entry.php?id=30 (20.05.2007)

[Vlissides97] **Patterns: The Top Ten Misconceptions**
John Vlissides
1997; Object Magazine;
http://www.research.ibm.com/designpatterns/pubs/top10misc.html (20.01.2007)

[WirfsBrock90] **Designing Object—Oriented Software**
Rebecca Wirfs—Brock, *Brian Wilkerson*, and *Lauren Wiener*
1990; Prentice Hall; ISBN 0136298257