HEDGEHOG: Automatic Verification of Design Patterns in Java

Alex Blewitt

Doctor of Philosophy
School of Informatics
University of Edinburgh
2006

Abstract

Design patterns are widely used by designers and developers for building complex systems in object-oriented programming languages such as Java. However, systems evolve over time, increasing the chance that the pattern in its original form will be broken.

To verify that a design pattern has not been broken involves specifying the original intent of the design pattern. Whilst informal descriptions of patterns exist, no formal specifications are available due to differences in implementations between programming languages.

This thesis shows that many patterns (implemented in Java) can be verified automatically. Patterns are defined in terms of variants, mini-patterns, and artefacts in a pattern description language called SPINE. These specifications are then processed by HEDGEHOG, an automated proof tool that attempts to prove that Java source code meets these specifications.

Acknowledgements

I am indebted to Alan Bundy who has given me the freedom to work on this thesis whilst at the same time guiding me towards the final production and presentation of these results. I not would have been able to achieve this without Alan's support through a sometimes difficult, but always busy part of my life. This project, and especially the production of this thesis, would not have been possible without the care and attention that Alan provided.

Ian Stark has provided invaluable feedback on all aspects of this thesis, from the low-level technical intricacies of Java's design patterns through to the high-level structure of the thesis as a whole. Without doubt, he has caught many of the technical errors and inconsistencies of this thesis; the fact that this work stands as it is can be attributed to his detailed attention to proof reading. Any remaining errors are my own fault in transcribing his feedback of the work.

I would also like to thank my previous supervisors, Richard Boulton and Helen Lowe, who helped me during the early stages of this research project before other commitments took them aside. In addition, Andrew Ireland helped with some very early aspects, and in conjunction with Alan Bundy allowed me to realise that my ideas were worth investigating, and convinced me to start down the long path of a PhD. Jon Whittle gave me support and a path to follow as I started out on this thesis.

I would not have been able to do this thesis without the support of Adrian Jackson and others from International Object Solutions Limited, whose support allowed me to work in conjunction with EPSRC to create HEDGEHOG.

My thanks are also due to the proof readers who gave me pages of feedback on this thesis, in alphabetical order: Derek Blewitt, Robert Blewitt, Tony Brookes, Adrian Jackson and Gareth Webber. I would also like to thank some of the particularly influential people for encouraging me to achieve my full potential; Ian Nussey of IBM UK and 'Doc' Misell of Epsom College. My thanks also go to Koos van Tubergen of IBM NL for supporting me through the early stages of my PhD.

Lastly, I would especially like to thank my loving wife Amy, for putting up with me over the past few years with all the late nights (and sometimes early mornings) in working on this project.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Alex Blewitt*)

This thesis is dedicated to the memory of our son Max,
who was born and died during the production of this thesis.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Design patterns, or simply *pattern*s, are an integral part of designing object-oriented software. A pattern is a well-known solution to a common problem, which allows a designer to capture a set of requirements using a unique name.

Patterns have been described in pattern catalogues such as [GHJV95, Bus96, Vli98] and act as a reference for designers and developers alike. These patterns are described in abstract terms as, although they apply to object-oriented languages, there are different ways of solving the same problem in different languages.

The abstract specification of patterns can lead to the introduction of programming errors, from misinterpreting the pattern's requirements through to incorrect implementations. Although formal works have been applied to programming languages before [Mey99, WK03], they have not tended to be applied to patterns because of their abstract specification.

This thesis presents a way of representing and verifying design patterns by focusing on the language-specific implementation; specifically patterns implemented in the Java programming language [GJS96]. By focusing on the implementation, rather than the abstract specification, it is possible to recognise and verify a number of common design patterns.

A pattern representation language called SPINE is presented, along with an automated proof engine called HEDGEHOG that can parse and process Java source files. This allows patterns to be verified, and provides greater confidence that the design pattern will not become accidentally broken later in the development process.

A number of standard patterns are defined in SPINE, using common patterns from catalogues [GHJV95, Bus96]. These patterns are defined according to their implementation, and since it is often possible to realise a pattern in a number of different ways, each pattern has a number of variants. This allows the proof system to verify the existence of a pattern if it takes one of

the known variant forms. The proof results that HEDGEHOG generates are therefore relative to these pattern definitions. It is possible that a pattern could be realised in a way unknown to HEDGEHOG and therefore generate a negative answer. However, although there are many patterns, there are often only a small number of different ways of solving the same problem, and the pattern library provided in Appendix B attempts to cover the standard realisations of these patterns.

In the case of proof failure (HEDGEHOG cannot prove that a pattern exists, or prove that a pattern does not exist), HEDGEHOG flags to the user that the implementation requires further investigation. It may be that the pattern is realised correctly, but in a way that HEDGEHOG does not recognise, and therefore no further action is required. It is possible for the user to extend HEDGEHOG's capabilities and provide a new pattern variant so that the pattern will be subsequently recognised. The primitives for building patterns are described in Chapter 5 and Appendix C.

Although HEDGEHOG is based on an automated theorem prover, it does not assume that the users know anything about automated proofs. It therefore hides the complexity of the proof operations to ensure that the system can work in a fully automated mode and provide a yes/no/unknown answer.

The system is designed to be used by an automated process (such as JUnit [GB], an automated unit testing framework) and provide results to give confidence that a pattern is realised correctly. In the case of a pattern failing to be realised correctly, a suitable message is created and given to the user, and the state of the proof tree is used to generate an English-language reason for why the proof failed.

A way of generating suitable messages from the proof tree state (after automated proof has terminated) is shown in Chapter 7 that gives the user a textual explanation of what the proof system has achieved without having to reveal the existence of a proof tree internally.

1.1 Structure of the thesis

The thesis consists of 11 chapters as follows:

Chapter 1, Introduction provides an overview of the thesis, the structure, and contributions made

Chapter 2, Literature survey lists the related works and references to other design pattern-oriented projects

Chapter 3, System architecture describes HEDGEHOG's architecture and explains the design choices used in implementing the system

Chapter 4, Design patterns gives an overview of design patterns and describes the important properties that design patterns have

Chapter 5, The SPINE language defines the SPINE language and gives examples of patterns defined in SPINE

Chapter 6, The HEDGEHOG proof engine describes the structure of the HEDGEHOG proof system and how it processes SPINE pattern definitions to verify correctly realised design patterns

Chapter 7, Generating error messages describes how error messages in the proof tree are translated and displayed to the end user

Chapter 8, Worked examples gives an in-depth example of HEDGEHOG to prove common design pattern realisations

Chapter 9, Results catalogues the results of the proof system, categorised by pattern type

Chapter 10, Related work discusses similar work to HEDGEHOG and highlights similarities and differences between them

Chapter 11, Further work and Conclusion gives additional ideas outside the scope of this thesis for future work with HEDGEHOG and provides a conclusion of the work presented

Appendix A, Glossary provides a glossary of related items

Appendix B, List of design patterns provides a number of standard design patterns that are referred to throughout this thesis

Appendix C, List of SPINE functions and predicates is a reference of the built-in functions and predicates

1.2 Hypothesis and contributions

This thesis aims to prove the hypothesis that it is possible to represent patterns as a set of constraints on the implementation of one or more Java classes, such that it is possible to verify whether they realise a pattern correctly.

A novel way of representing design patterns is presented, by defining the pattern relative to its implementation rather than its behaviour. This demonstrates that it is possible to treat an implementation of a design pattern as sufficient that it will behave according to catalogues of known design patterns. A catalogue of pattern definitions, along with their variants, is provided in the appendix.

Design patterns and their implementation are investigated in more detail, and are cross-referenced against well-known implementations in existing open-source code [GJS96] and Java design pattern [SM01] books. This approach is amenable not only to traditional design patterns, but also to lower-level idioms that would not be big enough on their own to be considered a design pattern, but still a remarkably common way of implementing a solution to a particular problem.

The results shown in Chapter 9 that this approach works well for structural design patterns and can be used with a number of behavioural and creational design patterns catalogued by [GHJV95].

Chapter 2

Literature survey

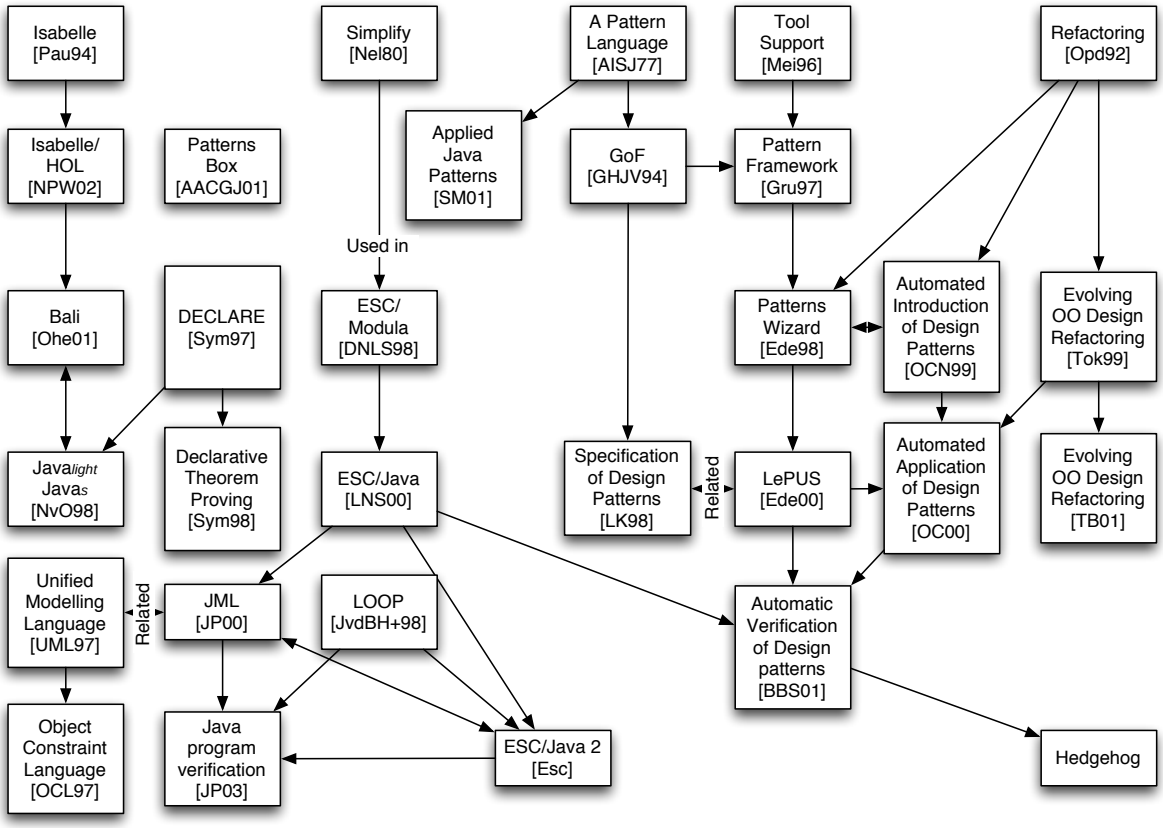
This chapter provides a literature survey of works surrounding Java and research into design patterns; Figure 2.1 shows graphically how they are related to each other. A survey of proof systems, Java language semantics and design pattern representations are covered to show how these works are related to each other and how they have developed. This chapter does not aim to provide a detailed comparison with HEDGEHOG; instead comparisons are made where appropriate in the body of the thesis, and a detailed comparison of how these works are related to HEDGEHOG and SPINE is given in Chapter 10.

HEDGEHOG aims to prove that design patterns can be represented in such a way as to allow an automated proof system to decide whether or not a class (or set of classes) realises a design pattern in Java. In order to do this, we must be able to:

- Formally describe design patterns
- Analyse a method's implementation
- Devise a proof system capable of showing the formalised design patterns meet the Java implementation

The chapter is broken down as follows: Section 2.1 presents work related to Java byte-code (Section 2.1.1), semantics (Section 2.1.2) and modelling (Section 2.1.3); Section 2.2 describes work in design pattern specification (Section 2.2.1), detection (Section 2.2.2) and refactoring (Section 2.2.3). The chapter is summarised in Section 2.3.

Figure 2.1: Related work to HEDGEHOG



2.1 Java

Java [GJS96] is a general purpose object-oriented programming language developed by Sun Microsystems. The style of the language is mostly based on C [KR88] syntax, but owes much of its object-oriented features to Smalltalk [GR83]. Notably, Java is compiled into an intermediate *byte-code* which is then executed by the Java Virtual Machine or *JVM*. What makes it appealing for research is that both the source code language and the byte-code language are well defined, and research projects have focussed on Java at both levels.

Java is also used heavily in industry and is now considered to be the default programming language of choice by enterprises after a relatively short gestation period. Of course, Java's arrival at the same time as the world wide web and subsequent dot-com boom helped to bring the language to many people at the right time.

Java has become the focus for research into tools and techniques for improving the quality of software. Formal methods have been applied to determine whether Java as a language has been designed correctly, and have even been used to discover holes in the specification of the Java language itself [Ohe01].

2.1.1 Byte-code

Java source files are compiled into byte-code, which is essentially an assembly language for operating on objects. Unlike processor-specific assembly, the Java byte-code does not deal with arbitrary jumps to locations in memory or change the value of memory locations directly; instead, the byte-code is object-oriented and routines are called by invoking methods on object instances. In deference to performance, the Java byte-code also allows the representation of integer and floating point values as *primitives*, which allows the JVM to process them more efficiently.

The JVM [LY96] is responsible for executing the byte-code generated from a compilation stage. Contrary to common belief, the byte-code does not have to be interpreted but can be executed in a manner of the JVM's choosing. For example, on mobile phones [Rig03], the JVM is most likely to interpret the byte-code and use platform-specific hints, whereas on an AS/400 the byte-code can be transformed prior to execution time into a format suitable for faster execution [MLMN99]. Most PC implementations use a mix of interpretation (for faster startup time) and translation (known as *Just In Time* compilation or *JIT*).

Additionally, the byte-code processed by the JVM does not have to be compiled from Java source; it could have been created from a different programming language (as an example,

NetRexx [HHF+97]) or dynamically from a library that allows creation of byte-code directly (such as the Apache Byte-Code Engineering Library or BCEL [ASF]).

As a result, there has been interest in proving whether the JVM does what it is supposed to at the low-level. For example, the byte-code verifier (which validates whether or not the byte-code has been produced in accordance with the rules of the JVM) has been shown to be sound with a representation in Isabelle/HOL. The approach taken by Pusch [Pus99] was to formalise the JVM in Isabelle/HOL [Pau94, NPW02] covering the main parts of object orientation and the structure of classes stored within the JVM at execution. A specification for a byte-code verifier (not necessarily the one that comes with a given JVM), is given in [Pus99].

In showing that the byte-code verifier was sound, Pusch et al. showed that it is possible to represent the type information of a JVM and that considering stack-based and frame-based execution is possible in Isabelle/HOL. However, it should be noted that their primary goal (and indeed, others based on Isabelle/HOL described below) was to show that the typing system was sound rather than considering the actual values of the JVM's objects at run-time.

Other approaches have created JVMs specifically for the purpose of investigating byte-code (and more specifically, with the aim of preventing rogue byte-code). Richard Cohen's Defensive Java Virtual Machine [Coh97] aims to manually verify at each stage that the type-safety of the executing byte-code still holds, instead of relying on the standard JVM's behaviour.

The dJVM is implemented in ACL2 [KMM00] to allow interpretation of byte-code and maintaining the heap and stack space, whilst at the same time defensively executing instructions in the JVM. For example, after execution of a routine that is declared as resulting in an `int` on the stack, the dJVM will explicitly ensure that the return result is an `int`.

Similarly (but more recently), the Verificard [NvOP02] project aims to show that the JVM built into JavaCard[1] is sound. This is also implemented in Isabelle/HOL, which is to be expected since the same group worked on the specification of Isabelle/HOL and the Java semantics (see below).

However, since most Java design patterns are not visible at the byte-code level[2] it is sensible to consider the Java semantics above the byte-code level. Additionally, a design pattern is not a concept that is normally concerned with properties of the executable code, but rather one of source management and reuse. Therefore, if source is available it should not be necessary to use the compiled code; and if source is not available, then there are no changes that can be made

[1] JavaCard allows Java programs to be installed on to smartcard processors.

[2] It may be possible to convert pattern specifications from recognising source-code patterns to recognising compiled byte-code patterns; however, due to optimising compilers and different ways of compiling the same source, it is likely that there will be many more variations in the compiled code from the same source pattern.

should the design pattern be broken in some way. As a result, it does not make sense to work with a high-level source concept such as design patterns at the byte-code level; instead, patterns should be analysed at the source level.

2.1.2 Java language semantics

One of the main requirements of creating a Java language semantics is the need to analyse and represent Java code in order to formally define or process design pattern realisations in Java. Two key project groups have researched into the Java language execution: the "Sound Languages Underpin Reliable Programming" group (or SLURP [SLU]) of Imperial College, London in conjunction with the University of Nijmegen worked towards a Java semantics that would prove type safety in the Java language. The other key project revolved around extended static checking of Java programs, which was created by the Compaq/Digital systems research center [LNS00].

Having a semantics for the Java language would be useful for the formal anaylsis of design patterns, because a design pattern has both structural (inheritance, relationships) and semantic (correct operation of methods) requirements. Clearly the former can be derived from the structure of the classes, but in order to ensure that methods perform the correct task requires some kind of analysis of the method's implementation. Although a full formal semantics would allow the properties of the method to be completely determined, it is in fact possible to determine some aspects of a method by static analysis. For example, a full semantics is not required if it is necessary to show that one method delegates to another; the possible call-graph from one method to another can be computed by static analysis alone. This is covered in more detail in Section 5.5.

2.1.2.1 $Java_s$ and $Java_{light}$

The SLURP group initiated a project to show that Java was type-safe. They started with a subset of Java, referred to as $Java_s$, which modelled the type system of Java in Isabelle/HOL. Using this subset, they managed to show that it was *probably* type-safe [DE97], although several assumptions about the types meant that the proof was not sound at that time. For example, they did not consider exceptions[3] other than special cases like `java.lang.OutOfMemoryError`, native methods, static initialisers and so forth. However, it was used as the starting point for

[3]These issues have been revisited in later reports [DV00, DVE00] to introduce exceptions to the $Java_s$ subset.

defining $Java_{light}$ used by Bali (below). The program structure of $Java_s$ is shown in Figure 2.2, expressions in Figure 2.3 and types in Figure 2.4.

Figure 2.2: Program structure in $Java_s$

```
prog   = class_1;...;class_n
class  = C extends C_sup implements I_1;...;I_n {
                 field_1;...;field_n
                 method_1;...;method_n
         }
field  = type fieldName
method = exprType method(type x_1;...;type x_n) {
                 stmt_1;...;stmt_n
                 return expr?
         }
stmt   = if expr then expr else expr
       | var := expr
       | {stmt_1;...;stmt_n;}
       | expr
```

Figure 2.3: Expressions and variables in $Java_s$

```
var = id                      (local variable)
    | expr.fieldName          (object field)
    | expr[expr]              (array element)
var = prim                    (literal value)
    | var                     (dereference)
    | expr.method(expr+)      (method call)
    | new C                   (object creation)
    | new type[expr]+[]*      (array creation)
```

The language does not summarise all of Java by any means; e.g. the lack of exceptions has been noted already. However, not all of the Java language is available for processing; and in particular, there are a few notable omissions:

Assignment is an expression Although $Java_s$ treats assignment as a statement, it is actually an expression in the Java language. It is commonly used in idioms such as looping over an array, where the index is declared and initialised in the loop itself. It's also possible to nest assignments, as in $a = b = expr$.

Figure 2.4: Types in $Java_s$

primitiveType	=	`bool` | `char` | `short` | `int` | `long` | `float` | `double`
simpleRefType	=	*className*|*interfaceName*
componentType	=	*simpleRefType*|*primitiveType*
arrayType	=	$componentType[]^n$
referenceType	=	*simpleRefType*|*arrayType*| `nullType`
type	=	*primitiveType*|*referenceType*
exprType	=	*type*| `void`
argType	=	`list of` *type*
methodType	=	$argType \rightarrow exprType$

Constructors are not represented Although it shows a class as containing methods and fields, constructors are not the same as methods. Specifically, they are invoked when a `new` operator is invoked, and arguments can be passed into constructors. As a corollary, the only initialisation operator in $Java_s$ is the `new` keyword, which only takes a class type, and no arguments. Given constructors play a part in some design patterns, the omission of constructors is not possible in HEDGEHOG.

Visibility of members is not considered There is no concept of modifiers such as `public`, `protected` or `private` which limits what patterns may be represented in $Java_s$. As an example, the **Abstract Factory** specifically requires an `abstract` modifier for both method and class types.

The subset is shown to be well-formed, and the language rules enforce type checking (for example, a method overridden in a subclass must have the same return type and same typed arguments) as well as compilation rules (methods defined in an interface that is implemented by a concrete class must be defined). These are rules that a Java compiler will enforce itself.

However, the important part of the work is that the run-time semantics is modelled as rewrite rules. Each step of execution (only considering a single thread) is represented as a tuple of steps remaining in a program and the current state. In essence, the execution engine is a state machine, and the act of executing a statement is a state transition. State is represented (as in the VM itself) as a *heap* of object instances and a *stack*, which contains variables defined in method calls.

The work on $Java_s$ is based on earlier work that used the C programming language, using a proof engine called DECLARE [Sym98]. This allowed proofs to be specified declaratively, and "results that are both machine checkable and human readable" [Sym97, page 33], although

it was heavily influenced by other proof systems such as PVS [ORS92] and Isabelle/HOL [NPW02]. The DECLARE proof engine was also used to prove $Java_s$ was sound, but as an experiment into declarative proof engines rather than specifically focussing on the properties of $Java_s$ itself.

The DECLARE proof engine was used to show that the rules formalising the language were sound and type-safe. The proof was built by repeated refinement of a set of original goals, and the interactive nature of DECLARE helped to discover lemmas that could be used in future parts of the proof. By the end, the subset of $Java_s$ was shown to be type-safe at both compile-time and run-time.

2.1.2.2 Bali

The Bali project is concerned with the formalisation of aspects of Java in the theorem prover Isabelle/HOL. As such, it shares a very similar purpose to the work done by the SLURP group, but with the distinction of using a different proof engine to do the work. The development of a second formalisation of the Java language started off with $Java_s$ to create $Java_{light}$, and proved that $Java_{light}$ (very similar to $Java_s$) was definitely type-safe [NvO98]. Bali used Isabelle/HOL [NPW02, Pau94] to represent the Java language and to determine that the $Java_{light}$ language was type-safe.

The purpose Bali and SLURP was to prove the type safety of Java as a language, rather than proving specific features about the programs written in the Java language. HEDGEHOG's architecture (discussed more in Chapter 3) also uses a built-in proof engine to reason about the implementation of Java design patterns. However, the Isabelle/HOL and Bali semantics were not suitable for representing patterns, since they only focussed on the run-time execution and not the relationship between classes.

2.1.2.3 LOOP

The LOOP tool (for Logic of Object-Oriented Programming) is a tool for reasoning about object-oriented programs. It uses an internal specification language called Coalgebraic Class Specification Language or *CCSL* [HHJT98] to define the program in terms of a *coalgebra*, which provides a hidden state space and a set of functions on that space. Whilst CCSL can be generated "by hand", it can be generated automatically from either C++ or Java programs. The CCSL statements are then translated into a set of statements for the PVS theorem prover, which allows it to reason about certain properties of the program. Unlike Bali, this not only allows

it to reason about the type of Java programs, but also about the execution flow and state of the class at a specific point.

The class is represented as a coalgebra, which is a set of functions that translate the state of an instance from one value to another. The application of each method therefore returns a new state, as well as a possible return value; the return value may either be a successful return code or an exception. In the case of an exception being raised, it is treated as a secondary flow and normal code flow is not resumed until it is caught by an appropriate handler.

The purpose of LOOP is to allow the user to prove certain statements about the Java program itself. In order for it to operate, it needs to translate each class required into a set of PVS theories, and questions about the program can be answered by presenting them as goals for the proof system to solve. It can also be used to prove invariants about a class by showing that they hold before and after a method's execution.

2.1.3 Java modelling and constraints

As an object-oriented language, Java is amenable to modelling using standard OO tools and techniques. The most common of these at present is the Unified Modelling Language or *UML* [FS03]. This is a graphical language for representing relationships between classes and objects. Other modelling languages exist, and are covered individually in this section.

Models in UML may be static (showing relationships between classes such as inheritance or navigability) or dynamic (showing how the sequence of messages flows between classes in a system). As a result, it is a useful way of graphically describing a concrete system, both from a structural and semantic viewpoint.

2.1.3.1 UML and OCL

UML is the standard modelling language for object-oriented systems, having evolved over the years from separate modelling techniques. UML itself is the distilled commonality between a number of separate precursor modelling mechanisms such as Booch, Rumbaugh, and Jacobsen.

Design patterns are often described with UML in various pattern books [GHJV95, Vli98, Cop95, Vli96]. However, since design patterns are implicitly abstract, and are supposed to be a template for multiple concrete systems, patterns described in UML often take the form of a concrete example, from which the reader is expected to interpret the key features. For example, the **Command** pattern [GHJV95, page 236] is often represented with an abstract superclass `Command` with an abstract `execute` method. However, this does not imply that all instances of

the **Command** pattern should use these exact names; it is equally correct to use `Action` and `perform` for these names. Since UML does not provide meta-names, it is not possible to use UML to represent meta-descriptions of patterns, but only show concrete examples.

By itself, UML does not provide semantic information of how methods should behave. To a certain extent, information can be shown with object interaction diagrams (such as sequence or collaboration diagrams). However, this information is limited to showing the order in which messages are sent, and does not really deal with conditional branching or constraints on the object data.

To solve this problem, a constraint language called Object Constraint Language or *OCL* [WK03] was created as part of the UML. Its purpose is to fill the semantic gaps left by UML's static and dynamic diagrams, so that semantic constraints on the object's data can be expressed. Expressions in OCL are often used to specify invariants for object state data, and are side-effect free; they cannot change the state of an existing system. They are also used to specify pre-conditions and post-conditions for method execution, and to describe conditions for guards or other looping sequences. An example of OCL constraining an `Account` class is shown in Figure 2.5.

Figure 2.5: Object Constraint Language example

```
context Account
  inv: overdraft < 0 and balance - overdraft >= 0

context Account::withdraw(amount:Integer)
  pre:  amount > 0 and amount < balance - overdraft
  post: balance = balance@pre - amount@pre
```

Although constraint-based programming is a very useful way of specifying a program's behaviour,[4] it has not yet reached mainstream acceptance. In part, this is a lack of understanding of how contracts are written or what purpose they serve. To some extent, their use (in Java) has been negated by the addition of assertions to the Java language and the increased use of test-driven development [Bec02] and automated unit testing [GB]. These processes both add

[4]Design by contract [Mey02] was popularised by languages like Eiffel [Mey99] and has subsequently been made available in Java with implementations such as iContract [Kra98]

test code to the system (internally in the case of assertions, externally in the case of test-driven development) to exercise it in normal, abnormal or edge[5] cases. There is also an interesting trend towards Aspect Oriented Programming [CHWC05], in which code is inserted at the entry and exit points of a method call. These are most commonly used to insert logging points, but also potentially to check pre-conditions and post-conditions of a method as well.

Ironically, whilst OCL is a much more precise (and complete) way of specifying constraints than testing code, the latter seems to have much more acceptance in developer communities. The adoption of this testing crutch may mean that languages such as OCL do not get used in mainstream Java developments, whilst code that can be formally verified may use more formal languages such as Z or Eiffel.

2.1.3.2 ESC/Java

Rather than verify assertions at run-time, other approaches that verify the semantic execution of code are possible. The HP SRC Classic[6] researched into proving operational checking using source code, with the Extended Static Checker (ESC) [Lei97, DLNS98] which discussed a mechanism for extended static checking (validating constraints using only statically available code), and originally focussed on Modula-3.

The idea of extended static checking is to verify properties about code based on the source code and a semantics for the programming language. Although ESC will be able to prove less stringent results than a fully formalised proof system, it can be used to prove properties about code that would be useful to others. In this respect, it is similar to HEDGEHOG's use of weak semantics; they are not capable of proving everything about a language, but they are capable of proving certain items of interest. For example, ESC is capable of proving whether there are any array index errors, NIL-dereferences (`null` in Java) and so forth. Further, these are errors which would not be picked up by a compiler but are only noticeable during run-time, and would not be picked up by a purely static type check of the code.

ESC works by generating a set of verification constraints from the source code, which it then feeds into a proof engine called Simplify [Nel80]. This is a refutation-based proof engine; given a requirement *P*, it tries to derive a contradiction from the negation of *P*. It works by automatically trying to prove a first order formula as input, and then determine whether it is valid or not. If it is invalid, it prints out something which it believes to negate the requirement.

[5] An edge case is one that exercises values close to special 'magic' values of the system, such as testing arrays with accessing elements number -1, 0 and 1 as well as n-1, n and n+1

[6] Formerly known as the Compaq Systems Research Center[7]

[7] Formerly known as the NEC/Digital Systems Research Center

The proof engine is conservative; it never claims that an invalid requirement is valid, but can sometimes fail to prove a formula that is valid.

A subsequent version, ESC/Java [LNS00, FLN+02], was created to get a larger user base by parsing Java source code instead of Modula-3 source code. This would have been an ideal base to use for HEDGEHOG, but unfortunately was not available until the end of this project. The operation of ESC/Java works by translating Java source files into a set of axioms that can be fed into the Simplify proof engine. It distils properties about the Java source (such as inheritance relationships, type definitions, and so forth) and converts these into axioms. It also translates methods into an intermediate form, so that program assertions are automatically added to the code (for example, asserting that the index used in an array dereference is positive and less than the size of the array itself). These expressions are then translated into verification conditions such that the proof engine can attempt to provide a counterexample.

Since the requirements are translated from source into the intermediate assertions, then into proof statements for Simplify, the authors acknowledge that this process is unsound. However, the goal of ESC/Java is to show that certain types of errors are not present, as opposed to determining that there are no errors present. By tackling (and successfully solving) a much simpler problem, they argue that the use of ESC/Java will allow some otherwise hidden/run-time problems to be uncovered, even if it cannot catch all of them.

Since the creation of ESC/Java, a new version has been forked called ESC/Java 2. Although similar in concept, the second version has been created by groups unconnected to the original development. It was created by the group responsible for LOOP [JvdBH+98] verification. In particular, it has been amended so that instead of requiring the original ESC syntax for marking up requirements, a new specification language called the Java Modelling Language or *JML* [JP00] is used to annotate the source. In principle, JML is similar to OCL, except that it is specifically aimed at Java instead of generic OO languages. It can therefore take advantage of both Java syntax and common Java constructs, such as lists and sets. An example, similar to the one given for OCL in Figure 2.5, is presented in Figure 2.6.

Additionally, since JML and its related tools are open-source, it encourages the ongoing development of the project. It is thus slightly more likely to be used outside of a research field than the previous version of ESC/Java. It should be noted that the ESC/Java 2 still uses the Simplify proof engine for proving the constructs under the covers; essentially, it wraps a different (and more extensive) grammar for writing the pre- and post-conditions, as well as invariants, of methods in the code. Other tools are being developed to support JML assertions, including plug-ins for IDEs such as Eclipse.

Figure 2.6: Example of JML

```
public class Account {
  /*@ public normal_behaviour
    @   requires amount >= 0;
    @   assignable balance;
    @   ensures balance.equals(\old(balance-amount))
    @*/
  public void withdraw(int amount) {
    balance -= amount;
  }
  // ...
}
```

2.2 Design patterns

Design patterns have been used by the object-oriented community for many years; the seminal catalogue is [GHJV95]. One of the key features of a design pattern is that it is an abstraction from the underlying implementation, which means that a pattern is by definition an intangible product. As a result, a specification of a design pattern needs to work at the abstract level rather than the concrete.

There are several works that specify or reason about design patterns. Each has a different way of representing patterns, depending on the nature of how the patterns are used. A brief description of each is given below. This thesis presents a more detailed review of design patterns in Chapter 4, and how they may be specified in SPINE in Chapter 5.

2.2.1 Specification

The specification of patterns has become an interesting question since the adoption of pattern catalogues such as [GHJV95, Vli98, Cop95, Vli96] and so forth. As noted earlier, patterns are often specified by example using UML as a notation for presenting a concrete example. However, since the pattern itself is abstract, and the example concrete, the reader needs to infer the important parts of the pattern in order to use it.

As a result, there has been work towards a specification for design patterns that allows the pattern to be represented as an abstract concept, rather than exemplified. Some specifications are also aimed at the automated processing or refactoring of design patterns.

2.2.1.1 Tool support

Several pieces of work aimed at the specification of design patterns for use within tools have been done at Utrecht University. The first describes tool support for object-oriented patterns [Mei96, FMvW97] in which patterns are integrated into the development process. The fragment tool provides templates from which design patterns can be instantiated in code; in essence, these templates form the specification of the pattern itself. The goal is to allow patterns to be treated as "first-class citizens in an integrated object-oriented development environment" [FMvW97, page 2].

LOOP focusses on the Smalltalk [GR83] programming language, and has an immediate benefit in that the Smalltalk programming environment itself is written in Smalltalk. As such, the program can be easily parsed and processed using tools inherent in the Smalltalk operating environment. Additionally, the tool is built upon the Smalltalk Refactoring Browser [RBJ97] which allows classes and methods to be refactored. A more detailed view of refactoring and design patterns is given below.

Patterns are represented as declarative constraints on a set of classes. These declarative constraints can be combined to form pattern specifications, which can then be applied to existing classes in the system. This leads to two definitions:

Pattern fragment An object-oriented artefact that specifies part of a pattern (a method call, required signature etc.)

Pattern constraint A requirement that a pattern must do a particular action, such as calling a method or instantiating a class

The pattern specifications can also include meta-information about the pattern itself, such as documentation or intent of the pattern. These are used within the tool to present information to the user when selecting or instantiating a pattern.

As well as using the tool for instantiating patterns, it can be used to refactor and repair patterns. Thus, names of the classes implementing the patterns may be refactored, but the fragments defining the pattern will still be associated with the code; and hence will still be documented as implementing the pattern. If the pattern is broken during the refactoring, then the fragments can be used to repair the break; for example, if a singleton class is split, the fragment can be detached and then re-attached to the correct class; and from there, the pattern can be re-instantiated.

This was followed up with a framework for representing patterns [Gru97] which defined a way of extending the definition of patterns so that they can be broken down into groupings. In particular, it describes patterns not in terms of structural relations, but in terms of the participants of those patterns. It also highlights a number of very useful observations:

- Some design patterns may share similarities with others, both in terms of their intent and implementation. If this is the case, then the common features can be described as *mini-pattern*s, which may then provide building blocks as a way to integrate other patterns in the future.
- Structure is not enough to describe the pattern. The problem is that the structure is only part of the solution; how the classes are related in intent can be just as important as how they are implemented (see the discussion of **Command** earlier).
- Participants are always present in a pattern. Patterns are only useful when they are used, but it is not always easy to see how the patterns are connected. For example, they may be directly connected via an instance variable reference, or they may be indirectly connected by many list-like data structures.
- Designing an abstract pattern language for transformation into several languages makes sense if there are more than three languages. However, the abstract pattern language would have to encompass every feature from every language, and would have to have emulated features for those languages which did not support it. Additionally, a number of specific features in a language may require tailoring to a specific solution. The author concludes: "All in all, it is practically impossible to provide readable translations between two arbitrary object-oriented languages without loss of meaning" [Gru97, page 85].

2.2.1.2 Pattern specification

Amnon Eden has worked towards a representation for design patterns. He started with the concept of the Patterns Wizard [EGY97] in which patterns are processed by a tool and in which patterns are represented in the "Pattern Specification Language" or *PSL*. This described patterns in terms of Smalltalk metaprograms, which when run, would generate an instance of a design pattern. Interestingly, whilst the pattern specification language was implemented in Smalltalk, the objects being modified were Eiffel.

The definitions of patterns in PSL are represented as *tricks* that the wizard can use to apply a pattern to an existing class (or set of classes). When the trick is applied to a class, features of

that pattern are instantiated. As such, the trick encodes the pattern in an imperative way, which is primarily useful when adding an instance of a pattern to a class.

Although not declarative (and thus not easy for use in detecting patterns) PSL did provide some key features that were used later on, and echoes similar observations made by others. For example, it introduced the idea of *micro-patterns*, by observing that there are some common features in design patterns such as [GHJV95], and that by representing these as individual units it is possible to build up a more fine-grained level of pattern library.

This was followed by a graphical (and thus declarative) representation of design patterns called the "LanguagE for Patterns' Uniform Specification" or *LePUS* [Ede98]. A graphical representation is worth a thousand words (as the saying goes) because of the clarity of the information which can be conveyed in a few well-known symbols. UML is used for designing object-oriented systems for exactly this reason, and LePUS aims to provide a language which could be used to represent design patterns and their interrelations. An example of LePUS can be seen in Figure 10.4 in Section 10.4.

Since graphical constructs are not easy to reason with directly, a textual syntax was created for LePUS [Ede00] which identified ground-rule relations between *participant*s (classes that work together to form a pattern) and their *collaboration*s which represent the interrelationships between the participants. These are referred to as a pattern's *artefact*s, and this term is used in HEDGEHOG's description to indicate both collaborations and participants.

Given that UML is a graphical modelling notation for object oriented systems, one may ask what LePUS provides over and above UML. One key difference is that LePUS provides the ability to reason about meta-classes or groups of classes as a single unit. For example, it's possible with LePUS to add a constraint to a family of methods that may be scattered throughout many classes. It's not so much that UML can't represent this (you could have one graphic for each of the classes, and stereotypes for each of the methods) but with LePUS it can be compacted into a couple of figures. Also, UML is limited to a textual stereotype name with a single graphic notation, whereas LePUS uses different graphical notations for specific meanings, so the LePUS diagram is very much more condensed than a UML representation. Unfortunately, this is also one of the weaknesses of LePUS – the diagrams can be so compact that it can be difficult to interpret.

PatternsBox [AACGJ01] provides a way of instantiating design patterns. It uses a library of design patterns and a set of requirements; for example, a composite pattern can be created by instantiating a `new Composite()` and subsequent operations could be added to it dynamically. The result of this processing is a set of Java source files that realised the design pattern. This is

a one-way process; if a new instantiation is required then the existing code is thrown away and new code is generated.

A library of patterns defined in the Pattern Definition Language (PDL), was created for the PatternsBox tool. PDL defined a number of classes used to represent the design pattern, so a superclass `Pattern` represents a generic pattern, and specific sub-types such as `Composite` provided mechanisms to be able to represent and create such patterns.

PDL is very similar[7] in intent to Eden's approach of a meta-language to instantiate design patterns [Ede00]. The implementation allows patterns to be created from a library, but is not suitable for verification or analysis.

HEDGEHOG initially used an object-oriented representation to define patterns, but switched to an external text-based format in order to provide easily extensible functionality for users who would not be familiar with HEDGEHOG's internals.

The ideas for presenting a pattern as a set of constraints, described as a declarative set of Prolog-like statements were presented in my paper [BBS01] and a set of patterns using this syntax defined in a technical report [Ble00] earlier; both of these are further developed in this thesis. The pattern definition language is described in more detail in Chapter 5, and a full list of pattern definitions is available in Appendix B.

2.2.2 Detection

Kyle Brown investigated automated design pattern detection in Smalltalk [Bro96] with a tool called KT. Its aim was to detect where patterns are used in existing Smalltalk code, with the intention of being able to detect potential patterns and allow the user to make the patterns more explicit.

Since patterns are often composed of inheritance, aggregation/association and messaging information, these can be used as hints to find where a pattern may be present. (These hints are also used in the refactoring of design patterns; [OC00] refers to this as a *precursor*.) Since patterns will often have some key features that are identifiable, a scan of existing code looking for these key features highlights potential uses of a particular pattern. Patterns that have multiple key features may be easier to detect; or at least, the tool can give a higher confidence that a pattern is present.

However, [Bro96] identifies that not all patterns are detectable:

[7] Indeed, it uses the term *leitmotif* which is attributed to [Ede00]

> Many design patterns, such as **Interpreter**, rely upon general design principles that cannot be directly represented as design diagram fragments. In cases like this, a general statement such as "Build a parse tree for a language" can be interpreted and implemented in many ways. This reliance upon semantics results in so many possible interpretations as to make the pattern impossible to detect.

It also highlights the issue with identifying a design pattern and being able to tell it distinctly from others:

> In general, a pattern is detectable if its template solution is both *distinctive* and *unambiguous*. If a particular solution is distinctive, then it may be represented by a unique diagram that is not likely to be generated in a design that does not utilize that pattern. If a solution is unambiguous, then the particular solution must be representable in only one way. ... The basic solution has the **Adapter** class subclass the "target" class, and contain an instance of the "adaptee" class to which it forwards messages. This solution is unambiguous; there are not many other ways to do this. However, it is not distinctive; the structure of the diagram is not unique enough to be positively identified as an instance of **Adapter**. If it were taken to be so, then every class that descended from another class and contained instance variables could be construed as being an **Adapter**!

The KT tool focussed on searching for **Composite**, **Decorator**, **Template Method** and **Chain of Responsibility**. It noted that **Strategy**, **State** and **Command** would potentially be detectable, but that they would be ambiguous; so much so, that it would potentially be easy to obtain a false positive.

More recently, Plezbert and Cytron described [PC00] concepts behind pattern recognition and verification. The paper and related work seems to be a work-in-progress at this time, with no other papers published since.

The paper is primarily aimed at being able to detect patterns from existing systems, with a view to also verifying the pattern's existence. Conversely, HEDGEHOG's approach is to focus on the verification of the design pattern, but as noted in the further work in Chapter 11, one possible use of a verification system is for detection. Using a verification system as a pattern detection mechanism is quadratic since the brute-force method is to verify each class against each pattern. This is noted by [PC00], and is just as applicable to HEDGEHOG.

Brute force attempts to scan for patterns have also been postulated by searching for a specific feature (or combination of features) in an object-oriented class. For example, [Ban98] suggests that design patterns can be identified in code by searching through code for a specific pattern. This has been tried in the form of regular expressions as well as language-specific searches; in essence, certain tell-tale markers (such as a `static final` variable) are used to identify where

a pattern may be present in a system. However, the article notes that such brute-force searches are by necessity time consuming in large systems and, because structure alone does not capture intent, may fire up many false positives.

2.2.3 Refactoring

Refactoring is the ability to change a computer system in such a way as to leave the behaviour unchanged. Refactoring was introduced [Opd92] to formally explain how behaviour-preserving transformations[8] can be made on existing code. It was made into a tool in the Smalltalk Refactoring Browser [RBJ97], and now refactoring is a key component in Integrated Development Environments (*IDE*s). More recently, the catalogue has been presented as a set of examples in [Fow00].

Since refactoring changes the design of code, but does not change its functionality, it is often used to improve the design of code in order to make it more flexible, more elegant or more maintainable. It is therefore possible to use a sequence of refactorings to introduce a design pattern into a system. Refactoring to instantiate patterns has been investigated in work such as [TB01, Tok99] and introducing design patterns [OCN99, OC00].

Refactoring to introduce design patterns is a way of changing code in order to make a pattern present. For example, in order to implement the **Command** pattern, it is necessary to have an abstract class (representing a generic command) and then subclasses that provide the required behaviour. It is possible to use low-level refactorings to introduce a **Command** pattern manually; for example, one could be achieved as follows:

1. Create a new target class to act as the placeholder for the command
2. Select the source class containing the method that needs to be encapsulated as a command
3. Create an instance of the command in the source class
4. Move the method from the source class to the target class (and replace it with an instance call)
5. Declare a concrete subclass of the command class
6. Push down the definition of the method into the subclass
7. Make the parent class abstract

[8] An example of a simple refactoring would be to replace all variables in a given scope called ‘`x`’ with a variable called ‘`y`’.

However, this requires two assumptions; that the user of the refactoring tool knows what the **Command** pattern looks like, and that they can devise a list of steps to transform it from the current source to the target. It may also be desirable to introduce other patterns, such as **Singleton** or **Flyweight** which are not shown in this refactoring sequence.

Thus the evolution of design patterns through refactoring may help simplify these stages, because the steps to instantiate a pattern can be well defined. Importantly, because refactoring tools exist and have a rich set of built-in primitive refactorings, it is possible to create a tool to evolve design patterns based on these primitive refactorings. Obviously, the actual refactorings themselves will be specific to a given target language, but also to the refactoring tool used; although the principle should map to other languages and refactoring tools.

The automated application of design patterns by refactoring was investigated in [OCN99, OC00], which also introduced ways of representing design patterns via transformations.

Consideration is given to the ability to automate the application of design patterns; and in order to ensure that the pattern transformation is valid, the pre-conditions and post-conditions of each primitive refactoring are combined to ensure that the whole overall refactoring is valid. This can be used to chain refactorings of any length, and not just ones aimed at pattern application.

Patterns can be automatically applied by representing them as a sequence of refactorings that instantiates the pattern. In order to do this, a number of refactoring *transformation*s are created, in which the steps are applied. When analysing patterns at this low-level, it becomes clear that there may be shared functionality or structure between patterns; and hence *mini-pattern*s can be discovered along with the *mini-transformation*s that are used to instantiate them.

It follows that the pattern itself is encoded in the transformations that generate it. It is also the case that the pattern itself will always have the same structure, given that the transformations that caused it to exist will have been the same each way. It should be possible to create different variations of a pattern using this method, but they would have to be represented as different top-level transformations. However, in any pattern where there are similarities, these may well have been represented as mini-patterns anyway, and hence a different ‘flavour’ of design pattern may be created.

Whilst the pattern’s pre-conditions allow the tool to determine whether the transformation can be applied, the implicit post-conditions assure that the pattern is indeed present. However, it is possible for the code to be further modified such that the post-conditions no longer hold, which is where the benefit of a verification tool such as HEDGEHOG comes in.

2.3 Summary

As one of the higher profile object-oriented languages in use today, Java is attracting a number of research projects. Most of the semantic projects relating to Java are focussed on the issues surrounding the Java language, rather than the use of the Java language.

Although design patterns have been used in object-oriented systems for many years, there is still relatively little work on reasoning about design patterns. Some works focus on generic design patterns across all languages, whereas others focus on a specific object-oriented language (notably Smalltalk or Java). Work in this area is still in its preliminary stages, and HEDGEHOG's approach of using a language-specific representation builds on the abstract representation and extended static checking to provide a system which is not only useful for research, but also may be applied to real systems.

Chapter 3

System architecture

This chapter presents an overview of HEDGEHOG and sets the context for later chapters.

3.1 HEDGEHOG

HEDGEHOG is a tool for automatically verifying the existence of design patterns in Java source code. In order to operate on Java source files, they must first be parsed into a format that HEDGEHOG can understand. Once parsed into a suitable representation, the classes can then be processed with the internal proof engine.

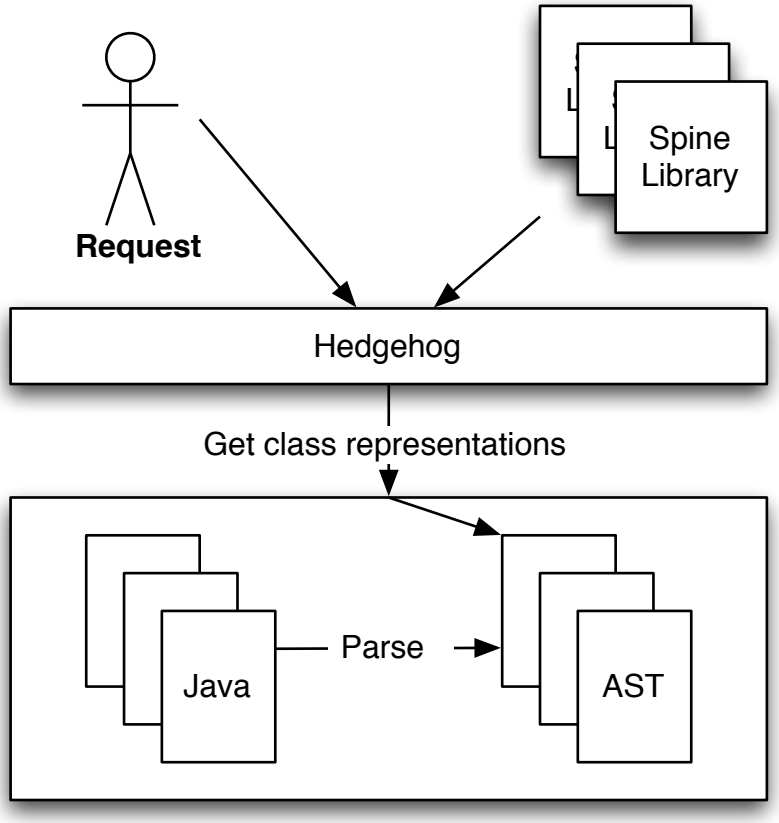
The design patterns, which are discussed in more detail in Chapter 4, are represented in a specification language called SPINE, which is discussed in Chapter 5. This allows a library of patterns to be used for verification of the source code and provides a means of extending HEDGEHOG's capabilities at a later stage.

When starting up, HEDGEHOG reads in the pattern definitions from the library. When the user requests verification that a class (or set of classes) realises a pattern, HEDGEHOG parses each required Java source file into an Abstract Syntax Tree or *AST*, and caches them for future use. The request is then passed to the proof engine, which uses the pattern definitions to show that the Java class(es) realise the given pattern.

Requests are read in from the user either directly (through an interactive GUI) or indirectly (through an automated process). The pattern verification process is intended to be automatic (thereby hiding the complexities of proof systems from the end user) and the interactive GUI is an interface for a read/prove/print loop.

Figure 3.1 shows the way in which each of these components are joined together.

Figure 3.1: Overview of HEDGEHOG

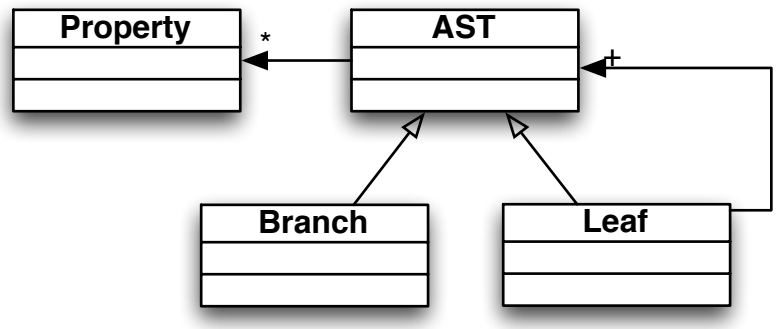


3.2 Parsing Java source files

In order to process Java source files, they are parsed into an internal AST that the HEDGEHOG system can deal with. The parser used is *antlr* [Par], which is a generic parser written in Java and capable of building many language-specific parsers. [Antlr is a Java version of 'lex' and 'yacc' type systems commonly found on UNIX systems.]

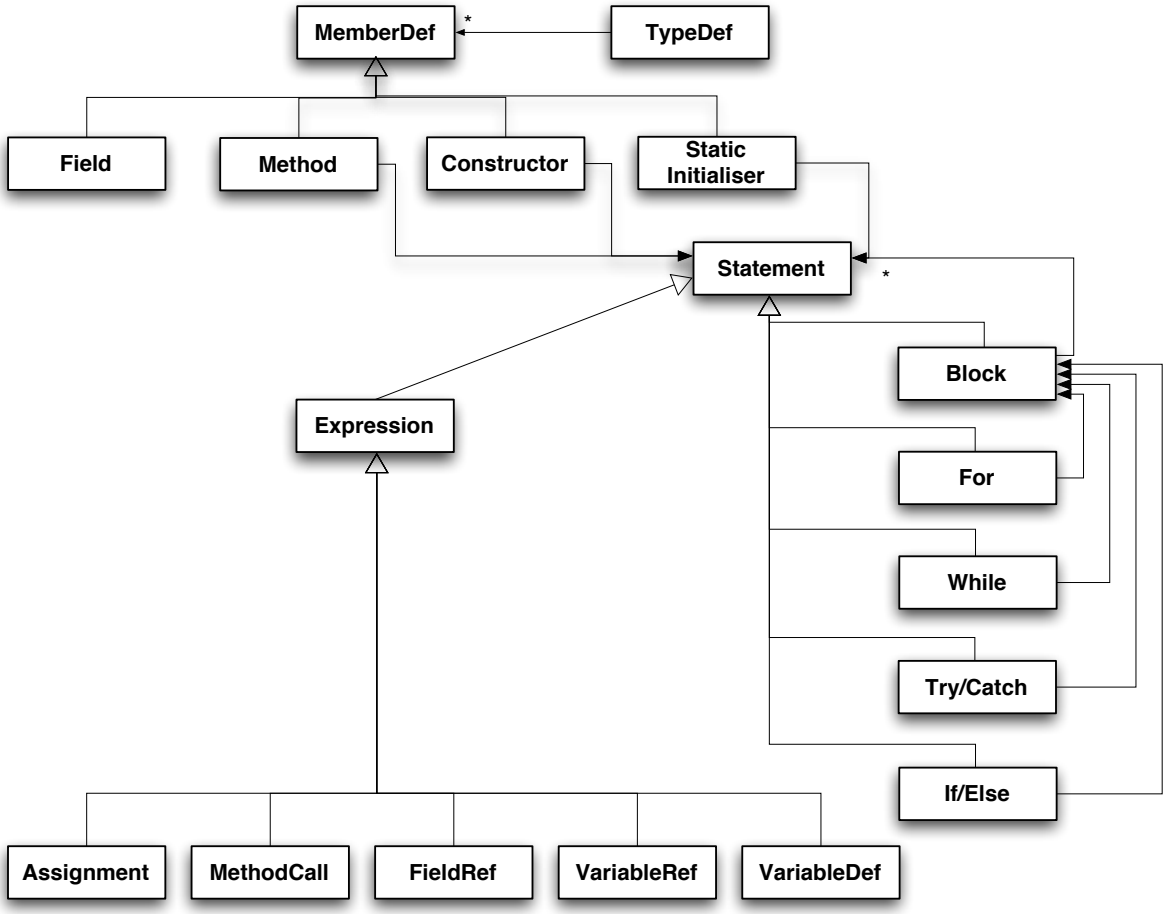
Antlr also comes with a built-in definition of the Java language, so it is capable of parsing the Java source file directly into an AST (Figure 3.2). From this, the HEDGEHOG AST is created (Figure 3.3). [Antlr provides a generic tree-type architecture that is neither Java keyword specific nor type specific. For ease of implementing some of the internal routines in the proof system, another AST is wrapped around which is type-safe and provides methods for processing that are not directly available for the underlying antlr AST.]

Figure 3.2: Antlr parse tree



The proof engine uses these ASTs when processing Java source code to determine certain properties. For example, the `Constructor` representation in the AST declares whether or not the constructor is `public`, `protected` or `private`; this information is then used by the proof engine in certain proofs.

Figure 3.3: HEDGEHOG AST



3.3 Parsing and processing SPINE files

All of the patterns are defined in SPINE, a declarative language similar to Prolog, to represent each pattern's requirements. (Spine is covered in detail in Chapter 5.) These patterns are stored in the patterns library, which is loaded at startup through the `init.sp` file. Through the use of the meta-`load` command, pattern definitions may be spread over a number of SPINE files (see Figure 3.4).

Figure 3.4: Example SPINE file

```
load('Creational.sp').                (* Load the creational
                                         patterns *)
load('Structural.sp').                (* Load the structural
                                         patterns *)
realises('Singleton',[C]) :-          (* Class C is a
                                         singleton if *)
  or([                                (* either *)
    realises('LazySingleton',[C]),    (* C is a lazy
                                         singleton, or *)
    realises('PublicSingleton',[C])   (* C is a public
                                         singleton *)
    realises('PrivateSingleton',[C])  (* C is a private
                                         singleton *)
  ]).
```

The SPINE statements are parsed into *rule*s which are then stored in memory. Order of rules is important; rules loaded earlier have precedence over rules loaded later. The rules are used by the HEDGEHOG proof engine (described in Chapter 6) when it needs to prove whether or not a class meets a particular design pattern.

3.4 Interacting with the user

The purpose of HEDGEHOG is to provide an automated system which therefore needs very little input other than the initial request (to prove that a class or set of classes meets a particular design pattern). Once this goal has been given to HEDGEHOG, the proof system is automatic until a result can be given. The internal proof tree is not shown directly to the user; instead, the result of the proof is shown as a yes/no/unknown answer. In the case of proof failure, the proof tree

can be translated to an error message to give more information about the problems that caused it to fail, and the error messages are described in more detail in Chapter 7.

Although a primitive user interface has been developed for executing HEDGEHOG, it is expected that a more integrated approach will be taken to integrate it with automated build tools like Ant and XDoclet, and automated testing tools like JUnit. This, and other ideas to extend HEDGEHOG, are discussed in Chapter 11.

3.5 Summary

HEDGEHOG uses a built-in Antlr-based parser to be able to create a representation of the Java source file in memory. This representation is used by the proof engine to ascertain statements about the Java classes during the proof process. Proofs are initiated by user request, which then starts the automated proof process, using SPINE definitions of patterns to verify correctness. Errors generated during the proof are translated into human-readable error messages and shown to the user, thereby hiding the complexities of the proof.

Chapter 4

Design patterns

This chapter starts by introducing patterns, and relating their history and incorporation into software products as design patterns. The basic content of a pattern is presented, along with references to several catalogues of patterns in the software industry.

Some terminology used in this thesis in relation to patterns is presented in Section 4.2. Section 4.3 then discusses how patterns can be represented. Several approaches are analysed and the advantages and disadvantages of each investigated. The patterns themselves are then analysed in Section 4.4 as a basis for what the key features in a pattern are.

4.1 The history of patterns

The term *pattern* was coined by an architect, Christopher Alexander [AISJ77]. He noted that when a building was designed, several key features were reproduced similar to previous designs.

The advantage of reusing an architectural design is obvious; once a design has been shown to solve a particular problem (such as the simple arch, introduced by the RomansfootnoteYes, but apart from arches, what have the Romans ever done for us? [Joh99]), then it can be reused in other buildings for a fraction of the effort taken to design the first instance.

Christopher Alexander noted that several elements were repeatedly reused, and termed these ‘patterns’. He catalogued a set of patterns that could be used to recreate building designs easily, and for each pattern, gave it a name (so that it could be referenced easily), described the key features of it (so that it could be reproduced in different situations), and described its advantages and disadvantages (to allow architects to choose between different patterns). Importantly, these patterns could be used in conjunction with each other to provide a complete solution, rather than as separate components.

His work on architectural designs was adopted by object-oriented pioneers to reformulate *design pattern*s to be used in software systems. Catalogues of design patterns appeared such as "Pattern Oriented Software Architecture" [Bus96] and also "Patterns: Elements of reusable software" [GHJV95], colloquially known as "Gang of Four" or "GoF" after the four authors who wrote it.

4.1.1 Software design patterns

Design patterns in software applications follow the same goal as Alexander's design patterns. Each software pattern is named (e.g. **Visitor**, **Singleton**, **Bridge**) in order to provide a common vocabulary between software engineers; key features are demonstrated (with code examples); and a set of advantages and disadvantages are given when considering using the design pattern. Additionally, patterns often have other embellishments; aliases (some patterns are known by different names: for example, **Listener** and **Observer** are the same pattern), relationships with other patterns (to allow choices between different solutions) and other comments indicating the cost or performance benefits between the two. Figure 4.1 shows an example of a pattern.

In object-oriented systems, such as Java [GJS96], using a design pattern will result in the creation (or modification) of one or more classes. However, software design patterns can still be used in other types of languages; for example, list-based processing in Prolog is a kind of pattern, and recursive function programming in ML is another type of pattern.

Catalogues (such as [GHJV95] define conditions for a successful pattern. They must:

1. have a name
2. have been used in several different situations
3. provide a common solution to a common problem
4. have consequences (benefits and drawbacks) of using the solution

The first requirement is obviously necessary; developers need to have a common way of identifying (and talking about) a particular pattern. As patterns are used more often, developers become much more familiar with the terms used discussing them, and thus communication regarding the pattern can be achieved much more quickly.

The second requirement is necessary because design patterns are meant to be reused in several different situations. It is not until a pattern has been used in several different situations that common parts of the pattern can be identified, and highly specific parts of the pattern can be factored out.

Figure 4.1: **Command** pattern description

Name	**Command**
Aliases	*Action*
See also	Interpreter
Description	The **Command** pattern is used to decouple the source of the request from the object that executes the command.
Overview	An application's operations (open file, find, next document) are often tightly coupled with user interface. However, if these can be decoupled, then it increases the maintainability (and flexibility) of the application. In the **Command** pattern, every action is implemented as a subclass of a designated `Command` class. Additional commands can be added simply by creating a new subclass of `Command`. Additionally, the command is no longer dependent on the class that requests it, which allows the GUI to run the command not from a single place (e.g. a button) but from multiple places in the same application (e.g. a menu item or hyperlink).
Notes	• Maintainability due to separate request/invocation of the command • Decouple the command requester from the command execution • Support security or logging • Support undo commands • Sharing the command instance across the application • Consequence: many discrete classes can result in a more fragmented application, which is more difficult to test

Example	`public interface Command {`
	`  public void execute(Application app);`
	`}`
	
	`public class SaveCommand implements Command {`
	`  public void execute(Application app) {`
	`  app.save();`
	`  }`
	`}`

The third requirement is necessary because if the problem is not a common one, then the pattern will not be used frequently. If it is not frequently used, it will become forgotten and thus not achieve its potential. Patterns are like genes; the fitter the pattern is (the more it is used; i.e. the more generic problem it solves) the more likely it will be to be remembered and reused.[1]

The last requirement is necessary because in order to properly choose a pattern to solve a particular problem, the developer must know what the advantages and disadvantages are of using that pattern.

4.2 Terminology

When discussing, specifying and verifying the correct use of patterns in an existing software codebase, as well as discussing how the patterns themselves may be structured, it is important to define the terms that will be used. To avoid ambiguity with other object-oriented concepts, terms such as *instance* and *instantiate* will apply solely to object instances and the creation of new object instances.

Additionally, since Java uses the terms *implements* and *extends* to indicate a relationship between classes and interfaces, these terms will only be used in the context of Java code.

4.2.1 Realises

If a class is a **Singleton**, then developers may use a number of different terms to indicate this. For example, some may say that "class X implements the **Singleton** pattern" or "class X is an instance of the **Singleton** pattern."

Since the term instance in object-oriented systems is very commonly used when talking about an instance of an object, we need a new term for discussing patterns that exist in code.

This thesis will use the term *realises* to indicate that a class (or set of classes) is a particular pattern. The above example will thus be phrased "class X realises the **Singleton** pattern", or more briefly, "class X realises **Singleton**."

4.2.2 Variant

A design pattern may be realised by several *variant*s. For example, a **Singleton** may be realised using lazy instantiation (see **LazySingleton** in Figure 4.2) or a constant static variable (see

[1] A solution to a single problem is not a pattern; only if it is reused in other situations can it be called a pattern.

PublicSingleton and **PrivateSingleton** in Figure 4.2) . They both behave in a similar way, but there are implementation differences (and therefore some minor variations in performance[2]).

Figure 4.2: **Singleton** variants

```
public class PublicSingleton {
  public static final PublicSingleton instance =
    new PublicSingleton();
  private PublicSingleton() {
  }
}
```

```
public class PrivateSingleton {
  private static final PrivateSingleton instance =
    new PrivateSingleton();
  private PrivateSingleton() {
  }
  public static PrivateSingleton getInstance() {
    return instance;
  }
}
```

```
public class LazySingleton {
  private static LazySingleton instance;
  private LazySingleton() {
  }
  public static LazySingleton getInstance() {
    if (instance == null) {
      instance = new LazySingleton();
    }
    return instance;
  }
}
```

In order to formally define a library of patterns, each pattern was analysed to define variants. It should also be noted that the examples shown in Figure 4.2 above are all ways in which a single type of **Singleton** may be realised. In [GHJV95], there are actually several variants of **Singleton** listed, including the simple **Singleton** (shown above), a subclassable **Singleton** and a registry of **Singleton**s.

There is some debate [OC00, page 97] as to whether a **SubclassableSingleton** is desirable,

[2]Lazy instantiation works by delaying the construction of an object until it is first needed. Thus, for instances that are rarely used, lazy instantiation works very well. However, instances which are often used are better suited for creation during class initialisation, since they will always be instantiated.

since control over creating instances is delegated to subclasses. Furthermore, given that one of the benefits of the **Singleton** pattern is to ensure one globally unique instance (and that other instances cannot be created), it's not clear whether the **SubclassableSingleton** is consistent with this requirement.

In actual fact, a **SubclassableSingleton** can be seen as a combination of two patterns; the **LazySingleton** combined with a **Factory Method** (or **Abstract Factory**) that can be used to (dynamically) determine which class to use. If only the simple **Singleton** variants are allowed, and a developer was to merge the class hierarchies of the **Abstract Factory** and **Singleton** classes, then HEDGEHOG would report this as a violation of the **Singleton** pattern since the non-instantiability requirement would have been falsified. This thesis concentrates only on the simple **Singleton** variants, but there is no reason it could not be extended to allow other variants, including the **SubclassableSingleton**.

There is also a **Registry Of Singletons** in [GHJV95] that is very similar to the idea of the **Flyweight** pattern, or even of a common dictionary-type data structure such as Java's `Map` classes. In this instance, the data structure is used to hold an instance of a class with a given key, and since there can only be one instance associated with a key, we have the uniqueness constraint. However, this does not prevent the same instance being inserted in multiple keys; and in any case, developers may not consider a hashtable of instances to be a realisation of the **Singleton** pattern.[3] Although it may be possible to specify an extra variant to cover this implementation, it has not been included since this **Singleton** variant is not normally seen in code.

It is more common to see variations from the **Singleton** pattern, such as the Java *`Color`* class, which defines a number of **Singleton**-esque instances. In fact, the **Singleton** can be seen as a particular sub-pattern of a notional **Multipleton** or **Enum** pattern that has a size of 1. Additionally, it is possible for some **Multipleton** realisations to be '`open`' as well as '`closed`'; for example, the Java *`Color`* class is an '`open`' **Multipleton**; it defines a number of fixed instances ('`RED, GREEN, BLUE`' etc.) and allows new instances to be defined. Then there could be other instances such as *`CardSuit`* which have a number of fixed instances ('`SPADE,HEART,DIAMOND,CLUB`') but do not allow new instances to be created, and this could be described as a '`closed`' **Multipleton**. However, this thesis will not investigate this further since the purpose is to investigate existing accepted patterns rather than suggest new patterns.

[3] There's also the difficulty of determining in which cases a hashtable is being used as a **Singleton**, and which is just a quick lookup table.

4.2.3 Artefact

Each design pattern places a number of constraints on each class. These may be constraints on the inheritance relationship, or on navigability with other classes, or on the implementation or expected behaviour of certain methods.

The term *artefact* will be used to indicate a single constraint on a class. A design pattern can therefore be defined as a set of artefacts that must be present in a class or a set of classes. This is similar to the idea of a fragment in [Mei96] but is more encompassing; a fragment merely relates the collaborators together in a design pattern. Similarly, [Ede00] defines relations that associate participants in a design pattern, but also includes some aspects of behaviour such as message forwarding. The term artefact is also used in [Bro96] as a way of detecting part of a design pattern's implementation.

Each artefact consists of one or more SPINE statements. These can then be grouped together to form pattern variants, which in turn can be grouped to provide a pattern definition. An artefact is a smaller requirement than a mini-pattern; the former is a constraint on the way one feature must be implemented, but a mini-pattern is a common idiom that may be repeated over several design patterns or other code implementations.

4.2.4 Super-pattern

Since there can be multiple variants for a single pattern, we need to be able to group them appropriately to allow code to refer to the abstract name, rather than a concrete variant. Thus the **Singleton** is actually a grouping of **PublicSingleton**, **PrivateSingleton** and **LazySingleton**. However, although these are three different implementations of the same type of singleton, there are also **SubclassableSingleton** as well as the proposed **Multipleton** patterns.

These can be grouped into a *super-pattern*. As with object-oriented inheritance, a super-pattern defines a super-type, and realisations of the sub-pattern are implicitly realisations of the super-pattern. Thus, in order to show that a class realises a **Singleton** pattern, it is sufficient to show that it realises a **PublicSingleton** variant. As with abstract types in object-oriented systems, the super-pattern need not have any implementation (or requirements) since these can be delegated to the sub-types. In essence, the pattern variants represent the concrete pattern realisations, and they are grouped into super-patterns which are abstract.

Using this definition, it has been possible to find other super-patterns. In fact, it has been possible to create a super-pattern of the **Singleton** pattern called **SingleAccess**. This pattern ensures that all methods can only be executed on a single instance, but as well as the **Singleton**

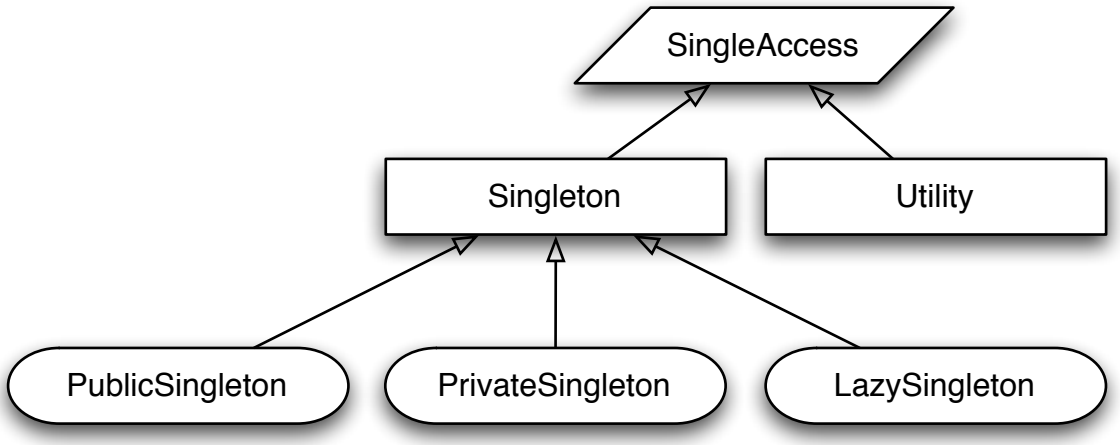
Figure 4.3: **Utility** pattern description

Name	**Utility**
Aliases	*Helper Class*
See also	Singleton
Description	Used to provide a number of global functions
Overview	The **Utility** class provides a number of global functions as `static` methods. This allows any code to refer to those functions using the class methods and does not require an instance to call them (such as the *`java.lang.Math`* class).
Notes	• Provides a set of global functions • Requires no instantiation

Example	`public class Utility {`
	`  // prevent others from instantiation`
	`  private Utility() {`
	`  }`
	`  public static int negate(int arg) {`
	`  return -arg;`
	`  }`
	`  public static String cap(String s) {`
	`  return s.toUpperCase();`
	`  }`
	`}`

pattern, the **Utility** pattern is also a sub-pattern of **SingleAccess**. The **Utility** pattern dictates that all methods are `static`, and as there can only be one instance of a class in a namespace, the methods all share the same access. These relationships are shown graphically in Figure 4.4.

Figure 4.4: Pattern variants and their super-patterns



4.2.5 Mini-pattern

In creating a list of design patterns and their definitions, it becomes clear that there are different types of pattern. Some design patterns, like the ones presented in this chapter, are realised with a single class; other patterns relate multiple classes together (such as **Bridge** and **Visitor**).

Additionally, a number of patterns have similar features. For example, all the variants of **Singleton** have a requirement to prevent other objects instantiating the singleton class. In Java, this is achieved with a `private` constructor. But this feature is so common (see also the **Utility** class) that it is worth naming as a shared feature across these, and other, patterns.

In [OCN99], the authors define the term *mini-pattern*. This term is used to represent these shared features across multiple design patterns and use these to define design patterns. A list of mini-patterns is provided at the end of Appendix B; and an example of the 'non instantiable' mini-pattern is shown in Figure 4.5.

Figure 4.5: Example mini-pattern

NonInstantiable
A type is non instantiable if: • The type is `abstract`, or • The type – Has one (or more) constructors (to avoid default constructor being created), and – All constructors are `private`

4.3 Formally defining patterns

In order to verify whether a pattern has been implemented correctly, it is necessary to be able to define the design pattern in such a way as to allow a proof engine to compare the definition against a suspected realisation of that pattern. Unfortunately, there are no immediately available formal definitions of design patterns, as catalogues tend to describe patterns verbally (such as in Figure 4.1). In some cases, the reader is expected to distill the key points of the design pattern by interpretation of provided example code, as well as understand the basic intent of the design pattern from the description.

Additionally, the choice of target language can dramatically affect the way in which a pattern is realised. Although most object-oriented languages share a large core of similar features (inheritance, overriding) there are a number of features that may be different (static or dynamic typing, static or dynamic method dispatch, single or multiple inheritance). As such, patterns tend to be adapted for the particular language that they are written in, to take advantage of the language's core features, with the result that some patterns are less useful in certain languages. For example, Python has no need for an **Iterator** pattern since the ability to iterate over lists is built into the core language, and this is used exclusively to provide iteration over sets of data.

However, we need to define patterns in order to be able to verify their implementation. There are three approaches which may be used to define patterns; each of which are discussed below.

[42] This page dedicated to the memory of Douglas Adams

4.3.1 Run-time semantic definition

The semantics of programming languages is a widely researched subject [Win93]. One way of specifying a design pattern would be to create a constraint as to how a program would operate if a particular pattern was present.

We can imagine a set of design pattern definitions as a set of Hoare triples [Hoa69], such that given an initial state of the program, a resulting state is guaranteed by the correct existence of a pattern. In order to do this, we need to determine for each pattern what the semantic behaviour needs to be. We could then use tools such as iContract [Kra98] or ESC/Java [LNS00] to determine whether or not the pattern correctly obeyed its specification.

Work such as Bali [NvO98] and LOOP [JvdBH$^+$98] have investigated the semantics of the Java programming language using operational semantics in order to reason about Java programs. Although much of the work in investigating the semantics has been an investigation of the Java language, they provide a foundation to build upon for reasoning about Java programs.

However, reasoning about programs is a hard problem; even for simple program execution flows, proofs can become very complex. The problem is magnified for imperative programming languages; unlike functional or declarative languages (where state is essentially immutable), the combinations of possible state values combined with the possibility of state changing during the program's execution leads to complex proofs. For example, Bali required some 2400 lines of Isabelle/HOL theories in order to prove that $Java_{light}$ was typesafe. Other approaches, such as [Sym97, Sym98] have also been used to prove operational semantics of the Java language.

Should a powerful proof system be available, then it may be feasible to verify whether a design pattern is implemented correctly or not. For example, the **Singleton** pattern (shown in Figure 4.7, taken from [SM01, page 37]), may be specified by encoding its constraints on the run-time behaviour of the program; specifically, that only a single instance of the class is created.[4] A semantic definition of the **Singleton** pattern may look like Figure 4.6.

Figure 4.6: Semantic definition of a **Singleton** pattern

$$\forall x, y. x \in Singleton \wedge y \in Singleton \rightarrow x = y\,.$$

[4] In fact, a large number of realisations of **Singleton** may actually not exhibit this behaviour faithfully when executed on multiprocessor platforms due to obscure unspecified behaviours in the way that the JVM operates. Even if this behaviour could be specified, multithreading may make these type of statements unprovable. However, many developers do not concern themselves with such quirks of multiprocessing systems and would still consider Figure 4.7 a **Singleton**.

Figure 4.7: **Singleton** pattern description

Name	**Singleton**
Aliases	*None*
See also	Utility
Description	To have only one instance of this class in the system, while allowing other classes to get access to this instance.
Overview	The **Singleton** pattern ensures a maximum of one instance is created by the JVM. To ensure control over the instantiation, make the constructor `private`. Since it is impossible to create a new instance, a static accessor method is provided , which creates the instance if it does not exist and returns it. [From [SM01]]
Notes	• The **Singleton** is the only class that can create an instance of itself. You can't create one without using the static method provided. • You don't need to pass this reference to all objects needing access to this **Singleton** • The **Singleton** can present threading problems, depending on the JVM.

Example	`public class HistoryList {`
	`  private static HistoryList instance = new HistoryList();`
	`  private HistoryList() {`
	`  }`
	`  public static HistoryList getInstance() {`
	`  return instance;`
	`  }`
	`}`

There is a further problem with using run-time semantics to specify a design pattern. Two key parts of any design pattern are:

Intent The purpose of the pattern (what it is trying to achieve) may be difficult, or impossible, to constrain in terms of run-time semantics alone.

Implementation Many implementations may provide the same behaviour, including some that would not be recognised (by humans) as realisations of the design pattern. Since a pattern is (by definition) a common solution to a common problem, it needs to be something that is recognisable in order to declare that a pattern is present.

In fact, it is not only possible that there will be multiple implementations that achieve the same goal as a pattern, but likely that these solutions will evolve independently. The goal of refactoring [OJ90] is to transform the structure or implementation of a piece of code to one with a better design, but the same behaviour. Additionally, several works have investigated introducing design patterns through the process of refactoring existing code [OC00, TB01]. In these instances, the goal is to introduce a design pattern into an existing system but without changing the behaviour. This would cause difficulties in expressing a design pattern as run-time semantic constraints only, since both original and the transformed code would have the same semantics.

It would also be problematic to specify how patterns are implemented. For example, the creational patterns from [GHJV95] discuss different strategies for instantiating objects, such as **Abstract Factory** and **Prototype**. In these cases, the desired behaviour is to instantiate an object, but using different mechanisms to obtain a new instance. A semantic constraint on the run-time behaviour may not be able to determine which of these approaches was used (or indeed, whether `new` was just called), since the run-time behaviour would just result in a new instance being present.

In much the same way as structure alone not being enough to represent design patterns [Gru98, page 30], it is not possible to define a pattern solely in terms of its externally observed behaviour, i.e. its semantics.

4.3.2 Metaprogramming definition

Another approach for representing patterns is to create a fragment of code that is capable of instantiating that pattern. This approach tends to be used when the purpose of the pattern definition is to create a pattern, either directly (in tools such as the “Patterns Wizard” [EGY97]

and Marco Meijers' "Tool Support" [FMvW97]) or indirectly (in refactoring approaches such as [OC00] and [TB01]).

In all these cases, a pattern is represented as a set of transformations or refactorings on existing code, with the intent of introducing a pattern into the system. The decision of which pattern to introduce (and where) has already been made by the user, so there is a clearly defined area of where the pattern is desired. It is also expected that the pattern will not be there before, since the purpose of invoking the refactoring in the first place will be to introduce the pattern into the system.

Instantiating a pattern is also likely to be easier from a template, since requirements (such as what to name newly created classes, interfaces, methods or fields) can be automatically assumed, or requested from the user. However, existing patterns are unlikely to match this template exactly, which could prove problematic for using them to recognise patterns directly.

Figure 4.8: Metaprogramming example of a **Singleton** pattern

```
applySingleton(Class concrete, String newAbstract) {
  partialAbstraction(concrete, newAbstract);
  addSingletonMethod(newAbstract, concrete);
  ForAll e:ObjCreationExprn, classCreated(e)=concrete,
   e ∉ newAbstract {
    replaceObjCreationWithMethInvocation(e,
      newAbstract.getInstance());
  }
  makeConstructorProtected(newAbstract);
}
```

As an example, consider a metaprogramming definition of a **Singleton** pattern, such as Figure 4.8 (taken from [OC00, page 99]). This can be used to add a **Singleton** pattern to an existing class. However, although the pattern is parameterised with the name of the class (and any new abstracted name), it assumes a default name for the accessor method and any fields that it may use.

As such, it makes it very difficult to compare an existing instance of the pattern with the specification of how to create a new one, without perhaps instantiating an empty pattern and performing a like-for-like check between the two. Although this is only a simple example, the same holds true for other patterns defined in a metaprogramming way.

It should be noted that it is the *use* of the pattern specification which drives the way in which it is specified, rather than any approach being a right or wrong way to specify a solution. For example, the previous definition of a **Singleton** (in Figure 4.6) may be suitable for formally proving whether it is implemented correctly; however, it would be of little use if being used to realise the pattern.

4.3.3 Declarative constraint definition

The final way of specifying patterns is to represent them as a set of constraints in the way the pattern may be represented. These may be constraints on the structure (for example, expressing the existence of a particular inheritance hierarchy), or constraints on method implementations (this method should be side effect free). I presented an initial approach at solving this problem in [Ble00] and [BBS01].

These constraints can then be checked against the implementation to determine whether or not a pattern is present. In some respect, this is a combination of both approaches; it mixes in some semantic behaviour of the class (this method should invoke another method) as well as structural relationships (this class should implement this interface). Furthermore, these can be used to provide specific information when a proof has failed, because it is expecting a pattern to have a certain appearance/behaviour. Marco Mijers proposed [Mei96] breaking down the patterns into fragments as a means of defining them, and was used to build a framework for representing patterns in [Gru98].

Fortunately, it is not necessary to develop a full Java semantics in order to verify individual design patterns. Most design patterns only need basic structural and semantic requirements to be met in order to realise a pattern correctly, such as the ability to instantiate a new class, or to ensure that a method is forwarded from one class to another. As a result, a much weaker semantics (than a full language semantics) is required; nevertheless, one which is both useful and tractable.

As an example, consider the **Singleton** pattern once more; we can express it informally as requiring:

1. Provide a `private` constructor
2. Provide a single instance via a `static final` variable

Although this will admit only a certain type of **Singleton**, it is a specification which is relatively easy to specify, and discharge against an implementation. A more formal definition might look like Figure 4.9:

Figure 4.9: Declarative example of a **Singleton** pattern

```
realises('Singleton',[C]) :-
  exists(constructorsOf(C),true),
  forAll(constructorsOf(C),Cn.hasModifier(Cn,private)),
  exists(fieldsOf(C),F.and([
    hasModifier(F,static),
    hasModifier(F,public),
    hasModifier(F,final),
    typeOf(F,C),
    nonNull(F)
])).
```

In this example, most of the pattern's constraints are structural. The only non-structural constraint is '`nonNull`', which ensures that the field '`F`' is initialised with a non-`null` value. This is validated by ensuring that '`F`' has an explicit initialiser, or alternatively is assigned a value in the constructor of the class.

Of course, there are two limitations to using the structural constraints as outlined above:

1. It is language specific. In this case, the pattern described would be suitable for a Java implementation of the design pattern, but may not be suitable for a C++ or Smalltalk implementation. For example, in Smalltalk, it may be more practical to throw an error during construction than have it hidden with a `private` modifier.

2. It only captures a single variant of the **Singleton** pattern. There may be other types of singleton (e.g. one using lazy instantiation, or a registry of singletons) that are realisations of the same pattern but do not fit this restrictive specification.

In fact, this makes the pattern detection easier than if a language-independent choice was made, since language features often play a part in the way in which a pattern is realised. As an example, Objective C provides a mechanism that allows dynamic 'faulting' – when a method is received that it does not understand, it calls a default 'fault' (or error) method. This makes the realisation of the **Proxy** pattern incredibly easy by use of this feature; however, other languages (such as Java) do not have this faulting feature, and must realise the pattern in a completely different way.

The approach chosen for this work should be applicable to other programming languages as well; a different parser could be used to process the source code but the techniques will

be similar. Chapter 11 discusses the possibility of expanding this work to other programming languages.

4.4 Elements of patterns

In order to represent a pattern as a set of constraints, we need to identify what the key aspects of each design pattern are. This will also affect the requirements of the pattern language (covered in Chapter 5) and the proof engine (covered in Chapter 6).

Although the pattern catalogues such as [GHJV95] present a list of design patterns, the descriptions of design patterns are very abstract. Given that the book's intent is to provide a template or overview of how a pattern can be implemented, this is not unreasonable. Also, since the examples are expected to be exemplary rather than prescriptive, they may be shown in different languages (such as Smalltalk, C++ or Java). Furthermore, although a template may be shown for a design pattern, it is quite likely that realisations of the pattern in real-life code may be (subtly) different. As a trivial example, the **Command** pattern (shown in Figure 4.1) shows an abstract class named `Command`. However, it is not the name of the class that is important (or even that it be an abstract class; it could be an abstract interface). So even with a simple pattern, there may be many variations that exist in real software.

4.4.1 Reviewing existing patterns

A review of the patterns in [GHJV95] highlights some areas of commonality and different types of pattern. Although the book is partitioned into 'Structural', 'Creational' and 'Behavioural', there are other ways of partitioning the patterns. One trivial partitioning is whether there is a single class (e.g. **Singleton**) or multiple classes (e.g. **Bridge**). Another is whether the pattern is an implementation trick (e.g. **Template Method**) or a way of controlling execution flow (e.g. **Interpreter** or **Strategy**).

Idiomatic Both the **Template Method** and **Factory Method** are idiomatic patterns, in that they utilise an abstract method in order to delegate an implementation to a subclass. This is a widely used technique in object-oriented programming, and is a key feature in other patterns (such as **Command** and **State**). One of the problems with idiomatic patterns is that it is very easy to treat all abstract methods as being realisations of the **Template Method** pattern. (In fact, both **Template Method** and **Factory Method** are very similar; the latter is used to create new instances, whereas the former can be used for any purpose.)

Other works [Ede98, OC00] refer to mini-patterns (defined above in Section 4.2.5). These are pattern fragments which occur repeatedly in code bases, but which are probably too small to be considered a pattern by the community. A well-known example of this is the way in which a C or Java program processes an array of data (shown in Figure 4.10):

Figure 4.10: Example of processing a Java array

```
Object data[];
for(int i=0;i<array.length;i++) {
  Object element = data[i];
}
```

Whilst it is unlikely that looping through an array of objects would ever be called an object-oriented design pattern (since it does not directly do any processing that requires objects), it is this kind of repeated idiom that is likely to occur frequently in code systems. On the other hand, the use of an abstract method to delegate implementation to a subclass (the **Template Method**) is also frequently used, but does get a mention in GoF.

Additionally, patterns such as **Iterator** have become so well known in languages like Java that it may seldom be regarded as a separate pattern itself. This may have more to do with standard interfaces (such as `Enumerator` and `Iterator`) being built into the core data types; in effect, a validation of how popular the pattern has become.

It is expected that patterns falling into the idiomatic category will occur very frequently throughout a project, and may not lend themselves to automatic verification. What is more, the design may not have specifically documented the use of a pattern such as the **Template Method** to achieve its goal, and may not even be required for automatic verification.

Single-class Patterns such as **Singleton** and **Decorator** are normally implemented in a single class. Some patterns, such as **Decorator** and **Proxy**, need to be able to communicate with other classes at run-time. Despite this, the classes that they are talking to need not know of their existence as they are wrappers around functionality; and thus can be viewed as single-class patterns.

Invariably, patterns such as these have little structural component that can be used as a specification. Instead, they often rely on one (or more) methods being implemented in a particular way.

Multi-class Some patterns, particularly those that are implemented with abstract super-classes, must be implemented in terms of a family of related classes. Examples include **Bridge**, **Command** and **Abstract Factory**. These patterns are often realised with a single central class, and a number of other ancillary classes that either use subclasses or are directly navigable.

Multi-class patterns will have some level of structure that can be used to identify them. In most cases, this will be part of an object hierarchy (such as the abstract super-types of **Command** and **Visitor**), but may also have relationships between components (e.g. **Bridge**).

However, it will not be enough to rely on structure alone to locate or verify a design pattern. Apart from anything else, the pattern is also expected to have some form of behaviour; for example, whether a method instantiates a new object or forwards a message on from one to another. It is fairly obvious that a multi-class pattern will have a richer set of these requirements in order to be correctly validated.

4.4.2 Intent

A key reason for using a pattern is that it helps describe the system, as well as implement it. Thus, when a pattern is used (and documented) in a codebase, it aids other developers looking to extend the system. Many patterns have a high level of intent in the way in which they are applied; patterns such as **Command**, for example, have a very light structure but the intent of the pattern is clearly visible.

Similarly, patterns such as **Visitor** have a great deal of intent; and it is this intent that sets it aside from a class hierarchy with a number of methods. From both an implementation and structural point of view, the **Command** pattern and the **Strategy** patterns can look very similar. However, their intent is very different; the former is used to provide a very large number of potentially simple commands in an extensible application, whereas the latter is used to provide a small number of potentially complex algorithms in a system. There is no clear cut distinction between a simple command and an algorithm; for example, what is the difference between a '`sort`' command and a '`sort`' algorithm? It depends on where (and how) they are used.

It would therefore be difficult to create a pattern definition that captured a **Strategy** that did not capture a **Command**, and vice versa. It would also be difficult to construct the pattern definition in a way that would not also admit many false positives, which would devalue the benefit of such a pattern definition.

Fortunately, with patterns such as **Command** and **Strategy**, it is difficult to get the pattern wrong because of the limited structure that the pattern mandates.[5]

Verifying (or specifying) the intent of a pattern is something that is outside of the control of this thesis. It may be the case that with the addition of languages such as iContract the intent of the individual commands could be defined and verified, but this is tangential to the correct use of the pattern.

4.5 Summary

This chapter gave a history of design patterns along with terminology that will be used in the remainder of the thesis. Three approaches for defining patterns were proposed and compared:

1. Run-time semantic definition, in which pre- and post-conditions are defined for a pattern
2. Metaprogramming definition, in which executable code generates templates or extends existing code to realise a pattern
3. Declarative constraint definition, in which constraints on both the structure and semantic behaviour of methods define the pattern

As discussed, type 1 and 2 are not suitable for defining patterns for the purpose of pattern verification. The approach taken by the HEDGEHOG proof engine is to use type 3 of pattern definition, since this can be broken down by a proof engine and then discharged against existing source code. Furthermore, it gives extra flexibility when dealing with names of types, methods or fields that may not have been created by the same tool.

Finally, patterns themselves were investigated to determine which types of constraints are likely to be required, in order to sufficiently define a pattern that matches a number of pattern variants.

The next chapter deals with the SPINE language and shows how pattern variants can be defined; and Chapter 6 shows how these definitions can be used to verify pattern realisations.

[5] Of course, it's much easier to get the actual algorithm wrong – but whether the command is bug-free is a different problem than whether the **Command** pattern is implemented properly.

Chapter 5

The SPINE language

Patterns are defined in HEDGEHOG in a language called SPINE. This chapter describes the SPINE language and gives examples of its use.

SPINE is a declarative language that is used to define statements about Java classes. The propositions may be basic logical connectives (such as `and()` and `or()`) or evaluable propositions (such as `isAbstract()`) that can be evaluated with reference to Java source code. Lastly, the language can also have evaluable sets (such as `methodsOf()`) that can be expanded in the context of Java source code, for use by quantifiers such as `exists()` and `forAll()`. Design patterns are represented as a set of rules which allow a pattern to be defined in terms of its implementation constraints. Proving SPINE statements is discussed in more detail in Chapter 6.

In this chapter, $\mathcal{A}_1$ and $\mathcal{A}_n$ are variables ranging over formulæ. **MemberName** and **ClassName** represent names (which are either constant string literals, or variables bound to constant string literals), and `propositions()` and `evaluableSets()` are represented in fixed-width font. $\mathcal{S}$ is used to denote an evaluable set, and s is used to denote a set of constants that is the result of evaluating $\mathcal{S}$. x is used to denote a variable, and c to denote a constant. FV is a set of free variables.

5.1 Overview

The syntax of SPINE is based loosely on Prolog. However, unlike pure Prolog, evaluable sets and propositions can be used in rule definitions which are evaluated during the proof process. These include a number of built-in predicates that are provided specifically for interrogating Java code, such as '`constructorsOf`' and '`subclassesOf`' and are listed in Appendix C.

SPINE was based on Prolog because:

- Prolog rules are a natural way of defining rule-based proof systems
- The syntax of Prolog is very simple (and therefore easy to parse, understand and extend)
- The language allows other patterns to be easily defined

SPINE was not developed directly in Prolog, but a separate language was created because:

- Java has a number of APIs that are well suited to parsing and processing Java source files
- Prolog does not have an easy mechanism for evaluating expressions in the same way that functional programming languages (like ML) do
- Java execution environments are more widely available than Prolog or ML engines

Initial experiments to integrate an internal Prolog engine with a Java-based front end did not prove successful. Primarily this was because the entire Java source tree needed to be encoded in a set of Prolog statements and uploaded into the Prolog engine before any processing could be done; and as the size of the Java source base being considered grew, the conversion into the Prolog engine became prohibitively expensive. Since the front end also spent a lot of time starting up the Prolog engine and returning back into it again, and since the front-end was used to make a lot of interactive calls to find out the state of the proof tree, it was necessary to represent the source tree in Java using off-the-shelf Java parsers. After the proof engine runs, the proof tree can then be interpreted and shown graphically, which was another limitation of the initial Prolog approach. Instead, an interpreting engine was implemented in Java to provide the ease-of-specification whilst allowing the power of Java's APIs to interact with Java code.

The same observation has been made about LOOP [JvdBH+98], since this requires Java method implementations to be translated into PVS [ORS92] statements for proving. The fact that all methods must be encoded into proof statements in order to pass them into the proof engine was highlighted as an issue which may degrade the LOOP tool's performance (to the extent where it could crash):

> "The LOOP tool produces a single, big, proof obligation in PVS for every method, and then relies on PVS to reduce this proof obligation into even smaller ones which we can ultimately prove. ... A drawback of the LOOP approach is that the capabilities of the theorem prover become a bottleneck sooner than in other approaches. After all, there is a limit to the size of proofs that PVS– or any other theorem prover for that matter – can handle before becoming painfully slow or simply crashing." [JP03, page 15]

The alternative approach taken by HEDGEHOG is that the patterns are encoded in SPINE and loaded at startup; when a proof is required, the source code for the Java is loaded and parsed dynamically when needed, instead of having to be translated into predicates and bulk loaded into the proof engine first of all. The parsed data is cached for quick access should the same class be required later in the same proof.

5.2 Syntax

The syntax of SPINE can be defined recursively. By constructing the syntax over a set of free variables, it can be guaranteed that a well-formed `realises()` rule will have no free variables (other than those captured by the `realises()` rule itself) by construction, which ensures that all variables are instantiated before evaluation. The pattern specification library is a sequence of SPINE rules. Only variable-free formulæ may be used as proof goals.

- `true` and `false` are formulas over $\{\}$
- If $\mathcal{A}$ is an evaluable proposition (listed in Section 5.3.2) over FV
  then $\mathcal{A}$ is a formula over FV.
- If **Name** is a constant string literal
  then **Name** is a name over $\{\}$
- If **Name** is a variable
  then **Name** is a name over $\{$**Name**$\}$
- If **MemberName** is a name over FV
  then `argsOf(`**`MemberName`**`)` is an evaluable set over FV
- If **ClassName** is a name over FV
  then `constructorsOf(`**`ClassName`**`)` is an evaluable set over FV
- If **ClassName** is a name over FV
  then `fieldsOf(`**`ClassName`**`)` is an evaluable set over FV
- If **ClassName** is a name over FV
  then `methodsOf(`**`ClassName`**`)` is an evaluable set over FV
- If **ClassName** is a name over FV
  then `subclassesOf(`**`ClassName`**`)` is an evaluable set over FV

- If $\mathcal{A}$ is a formula over FV
  then `not(`$\mathcal{A}$`)` is a formula over FV
- If $\mathcal{A}_i$ is a formula over FV_{A_i} for $1 \leq i \leq n$
  then `and(`$[\mathcal{A}_1, \ldots, \mathcal{A}_n]$`)` is a formula over $FV_{A_1} \cup \ldots \cup FV_{A_n}$
- If $\mathcal{A}_i$ is a formula over FV_{A_i} for $1 \leq i \leq n$
  then `or(`$[\mathcal{A}_1, \ldots, \mathcal{A}_n]$`)` is a formula over $FV_{A_1} \cup \ldots \cup FV_{A_n}$
- If $\mathcal{S}$ is an evaluable set over FV_s and $\mathcal{A}$ is a formula over FV_a
  then `forAll(`$\mathcal{S}, x.\mathcal{A}$`)` is a formula over $FV_s \cup FV_a \backslash \{x\}$
- If $\mathcal{S}$ is an evaluable set over FV_s and $\mathcal{A}$ is a formula over FV_a
  then `exists(`$\mathcal{S}, x.\mathcal{A}$`)` is a formula over $FV_s \cup FV_a \backslash \{x\}$
- If $P, C_1, \ldots, C_m$ are names over FV
  then `realises(`$P, [C_1, \ldots, C_m]$`)` is a formula over FV
- If P is a constant name, $C_1 \ldots C_m$ are variable names
  and $\mathcal{A}_i$ is a formula over $\{C_1, \ldots, C_m\}$ for $1 \leq i \leq n$
  then `realises(`$P, [C_1, \ldots, C_m]$`)` :- $\mathcal{A}_1, \ldots, \mathcal{A}_n$ is a SPINE rule

5.3 Semantics

The semantics of SPINE consists of *basic propositions*, *evaluable propositions* and *evaluable sets*. The basic propositions define logical relationships such as `and()` and `or()`. The evaluable propositions and evaluable sets are evaluated in the context of a specific set of Java clauses. Thus, the semantics of SPINE is defined with respect to a given set of Java classes, and in addition, only defined for variable-free formulæ.

5.3.1 Basic propositions

- `true` is true
- `false` is false
- `and(`$[\mathcal{A}_1, \ldots, \mathcal{A}_n]$`)` is true iff every $\mathcal{A}_i$ is true for $1 \leq i \leq n$
- `or(`$[\mathcal{A}_1, \ldots, \mathcal{A}_n]$`)` is true iff at least one $\mathcal{A}_i$ is true for $1 \leq i \leq n$
- `not(`$\mathcal{A}$`)` is true iff $\mathcal{A}$ is false
- `forAll(`$\mathcal{S}, x.\mathcal{A}(x)$`)` is true iff $\mathcal{S}$ evaluates to s, and for every $c \in s.\mathcal{A}(c)$ is true

- `exists` $(\mathcal{S}, x.\mathcal{A}(x))$ is true iff $\mathcal{S}$ evaluates to s, and at least one $c \in s.\mathcal{A}(c)$ is true
- `realises` $(P, [C_1, \ldots, C_m])$ is true iff it is an instance of a SPINE rule `realises` $(P, [C_1, \ldots, C_m])$:- $\mathcal{A}_1, \ldots, \mathcal{A}_n$ and every $\mathcal{A}_i$ is true for $1 \leq i \leq n$

5.3.2 Evaluable propositions

The following are evaluable propositions which may be calculated directly from Java source; either *EP* or `not(`*EP*`)` holds (where `EP` is a variable-free evaluable proposition). Evaluation of evaluable propositions is only defined for those with no free variables. These are the axioms of the logical system and are described in Appendix C.

- `adds` (**MemberName**, **ClassName**, **FieldName**) is true iff the method **MemberName** adds an instance of **ClassName** to the collection referenced by **FieldName**
- `extends` (**ClassName**$_1$, **ClassName**$_2$) is true iff **ClassName**$_1$ is a subclass of **ClassName**$_2$
- `hasModifier` (**MemberName**, *modifier*) is true iff **MemberName** has a Java modifier *modifier* (e.g. `public`, `protected`)
- `implements` (**ClassName**$_1$, **ClassName**$_2$) is true iff **ClassName**$_1$ implements the interface **ClassName**$_2$
- `instantiates` (**MemberName**, **ClassName**) is true iff **MemberName** instantiates **ClassName**
- `invokes` (**MemberName**$_1$, **MemberName**$_2$) is true iff **MemberName**$_1$ invokes **MemberName**$_2$
- `invokes` (**MemberName**$_1$, **MemberName**$_2$, **FieldName**) is true iff **MemberName**$_1$ invokes **MemberName**$_2$ on **FieldName**
- `isAbstract` (**MemberName**) is true iff **MemberName** has the Java modifier *abstract*
- `isClass` (**ClassName**) is true iff **ClassName** is a class
- `isFinal` (**MemberName**) is true iff **MemberName** has the Java modifier *final*
- `isFriendly` (**MemberName**) is true iff **MemberName** does not have one of the Java modifiers *public*, *protected*, or *private*
- `isPrivate` (**MemberName**) is true iff **MemberName** has the Java modifier *private*
- `isProtected` (**MemberName**) is true iff **MemberName** has the Java modifier *protected*

- `isPublic(`**`MemberName`**`)` is true iff **MemberName** has the Java modifier *public*
- `isSideEffectFree(`**`MemberName`**`)` is true iff the execution of **MemberName** is side effect free
- `isStatic(`**`MemberName`**`)` is true iff **MemberName** has the Java modifier *static*
- `isInterface(`**`ClassName`**`)` is true iff **ClassName** is an interface
- `lazyInstantiates(`**`MemberName`**`,`**`FieldName`**`)` is true iff **MemberName** lazily instantiates **FieldName**
- `modifies(`**`MemberName`**`,`**`FieldName`**`)` is true iff **MemberName** modifies **FieldName**
- `navigable(`**`ClassName`**$_1$`,`**`ClassName`**$_2$`)` is true iff **ClassName**$_1$ and **ClassName**$_2$ are navigable
- `named(`**`MemberName`**`,`*name*`)` is true iff **MemberName** is called *name*
- `nonNull(`**`FieldName`**`)` is true iff **FieldName** is assigned a non-null value
- `prefix(`**`MemberName`**`,`*name*`)` is true iff **MemberName** has a prefix *name*
- `related(`**`ClassName`**$_1$`,`**`ClassName`**$_2$`)` is true iff **ClassName**$_1$ and **ClassName**$_2$ are related
- `removes(`**`MemberName`**`,`**`ClassName`**`,`**`FieldName`**`)` is true iff the method **MemberName** removes an instance of **ClassName** from the collection referenced by **FieldName**
- `returns(`**`MemberName`**`,`**`FieldName`**`)` is true iff the method **MemberName** returns **FieldName**
- `sameSignature(`**`MemberName`**$_1$`,`**`MemberName`**$_2$`)` is true iff the methods **MemberName**$_1$ and **MemberName**$_2$ have the same signature
- `sameSupertype(`**`ClassName`**$_1$`,`**`ClassName`**$_2$`)` is true iff **ClassName**$_1$ has the same supertype as **ClassName**$_2$
- `subtypeOf(`**`ClassName`**$_1$`,`**`ClassName`**$_2$`)` is true iff **ClassName**$_1$ is a subtype of **ClassName**$_2$
- `typeOf(`**`MemberName`**`,`**`ClassName`**`)` is true iff **MemberName** is of type **ClassName**

5.3.3 Evaluable sets

The following are evaluable finite sets (from Appendix C) which may be calculated directly from the Java source code. The sets may be empty, singletons, or have many values but they are

guaranteed to be finite because the source code is finite. They are only used within the direct context of a `forAll` or `exists` proposition. If the set is empty, `forAll` is deemed to be true and `exists` is deemed to be false. Evaluation of evaluable sets is only defined for evaluable sets with no free variables. Note that the sets only consist of constants.

- `argsOf(`**`MemberName`**`)` evaluates to the set of argument names for the given member (method or constructor) name
- `constructorsOf(`**`ClassName`**`)` evaluates to the set of constructor names for the given class name
- `fieldsOf(`**`ClassName`**`)` evaluates to the set of field names for the given class name
- `methodsOf(`**`ClassName`**`)` evaluates to the set of method names for the given class name
- `subclassesOf(`**`ClassName`**`)` evaluates to the set of class names that are subclasses of the given class name

5.4 Rules

SPINE rules are defined in terms of `realises()`. Each `realises()` rule has two arguments; a pattern name, and a non-empty set of class names which are provided as constants at the top level. Thus, all SPINE rules are of the form `realises(`$P,[C_1,\ldots,C_m]$`)` :- $\mathcal{A}_1,\ldots,\mathcal{A}_n$, where $\mathcal{A}_1,\ldots,\mathcal{A}_n$ may contain nested `realises()`, P is a constant and $C_1,\ldots,C_m$ are variables.

Given that SPINE rules can be defined in terms of others (for example, capturing the variants of a pattern or allowing mini-patterns to be defined), it would be possible to define a set of rules that are mutually recursive:

$$\texttt{realises}(P_1,[C_1,\ldots,C_m]) \ :- \ \texttt{realises}(P_2,[C_1,\ldots,C_m])$$
$$\texttt{realises}(P_2,[C_1,\ldots,C_m]) \ :- \ \texttt{realises}(P_1,[C_1,\ldots,C_m])$$

Such mutually recursive rules would result in `realises()` being ill-defined. To avoid this problem, a partial ordering on the `realises()` SPINE rules can be defined:

Definition 5.4.1. *A set of* SPINE *rules is* consistent *if it has a well-founded partial ordering such that if for every* SPINE *rule:*

$$\texttt{realises}(P_1,[C_1,\ldots,C_m]) \ :- \ \ldots\texttt{realises}(P_2,[C_1,\ldots,C_m])\ldots$$

then $P_1 \overset{def}{\succ} P_2$*, and that there exists exactly one rule for each* P_i*.*

If there is more than one rule for a given P, then it would not be possible to define an ordering based on the first argument alone:

$$\texttt{realises}(P,[C_1,\ldots,C_m]) \quad :- \quad \mathcal{A}.$$
$$\texttt{realises}(P,[C_1,\ldots,C_m]) \quad :- \quad \mathcal{B}.$$

Note however that this is identical to the following:

$$\texttt{realises}(P,[C_1,\ldots,C_m]) \quad :- \quad \texttt{or}([\mathcal{A},\mathcal{B}]).$$

Thus it is possible to merge multiple rules with the same first argument into a single rule.

5.5 Java constraints

In order to define patterns in SPINE, a number of Java-specific rules need to be defined. Patterns are defined in terms of structural constraints (such as relations with superclasses, interfaces and methods), as well as constraints on the implementation of the individual methods themselves.

The constraints on Java patterns can be split into:

Structural Constraints on how the Java classes are related to each other, such as inheritance relationships or association relationships

Semantic Constraints on how the Java methods are implemented or behave

Clearly, the former set of constraints are trivially provable (by direct appeal to the Java class hierarchy or interface definitions). The latter set are more difficult to prove.

5.5.1 Structural constraints

Structural constraints are those relating to inheritance hierarchies, existence of specific named methods, associations between classes and so on. They also include modifiers (such as `public` and `private`).

These definitions appeal directly to the implementation of the classes, either through the inheritance (superclasses) or by evaluating a set of methods (or fields) from a given class. The `fieldsOf` is evaluated to give a list of fields defined in the given class, and the `typeOf` evaluates to **true** if the type of the defined field is a type of the given class.

5.5.2 Semantic constraints

It is a lot more difficult to prove that semantically, a method does what it is required to. This difficulty stems from two quite different causes:

- It is a lot more difficult to accurately specify a semantic constraint
- It is a lot more difficult to prove a semantic constraint as it requires a full Java semantics

For example, one commonly used pattern is the **Command** pattern. This encourages an application to be built as a command-processing system, where each operation in the application is abstracted as a *command*. When an operation is requested, a command instance is obtained, and then executed by a command processor. Not only does this abstract the way that commands are processed (for example, it is easy to extend this system by adding security checking and multithreading), it also allows other features, such as an undo operation, to be added at a later stage. [**Command** is often implemented in conjunction with the **Flyweight** pattern; and for undo operations, the **Memento** pattern.]

However, specifying a **Command** would necessarily be complex; each different command would require a different specification in terms of what the command does, and how it changes the underlying data model. This is an example of the high-level semantic constraints which fall foul of both the difficulties mentioned above.

5.5.3 Weak semantic constraints

Instead of requiring a full Java semantics and a full semantic specification of a pattern, it would be possible to represent a pattern in terms of *weak semantics*, by defining patterns in terms of their structural definition and not directly depending on their behaviour. Some of these require specific analysis of the code (for example, whether a method changes or assigns a particular field) but this can be determined based on static analysis of the method's structure. This is a conservative approach but more tractable.

As an example, the **Immutable** pattern requires knowledge of whether a method might modify an instance variable or not. It is not necessary to know whether the execution of a method *always* modifies an instance variable; however, it is desirable to know whether a method *can* modify an instance variable.

Informally, this can be expressed as follows: for a given method '`M`' and instance variable name '`V`', '`M`' can only modify '`V`' directly if, in the body of M, there is an assignment statement of the form '`V = expression`'.

This can be statically determined by iteration over the structure of the method body. The set of assignments in the method can be calculated, and for each assignment, it can be compared with the instance variable name.

We can define other semantic structural (weak semantic) constraints:

isSideEffectFree if execution does not change the state at all

isConstant if execution results in the same value

nonNull the field F is assigned a non `null` value

These can be defined in terms of the structure of the statements and expressions, without needing to appeal to a full Java semantics to prove the general case. A full list of these are defined in Appendix C.

5.6 Patterns

Design patterns are encoded as rules that can be applied to Java classes. Each design pattern may only have a single rule, or if there are many variants (see Section 4.2.2) many rules. The proof engine (discussed in Chapter 6) then uses these rule definitions to prove the conjecture that '`realises(Pattern,[Class])`', where '`[Class]`' is a list of classes that realise '`Pattern`'.

5.6.1 Immutable

A class is said to be *immutable* if an instance, once created, cannot be modified. One advantage of an immutable class is that an instance of it can be safely passed as an argument to other functions, since they cannot modify the contents of the immutable instance.

In Java, immutable classes are developed by assigning all the instance variables during construction, and only providing read-only methods (ones that do not change the content of the instance fields). [Note that 'constant' fields (those defined with `final`) do not need to be considered, since these by definition cannot be changed.] The variables must be defined as `private` since it is not otherwise possible to prevent other classes changing the contents.

The **Immutable** pattern can be defined by the following SPINE rule:

```
realises('Immutable',[C]) :-
  forAll(fieldsOf(C),F.
    or([
      hasModifier(F,static),
      hasModifier(F,final),
      and([
        hasModifier(F,private),
        forAll(methodsOf(C),M.not(modifies(M,F)))
      ])
    ])
  ).
```

In general, most patterns use a similar set of logical constraints, which are built-in to the system, such as `hasModifier` and `fieldsOf`. However, each pattern may require a specific constraint to be defined, such as the `modifies` in the case of the **Immutable** design pattern.

As the pattern library grows, it is expected that fewer specific constraints will be needed in the system to define future patterns. This would follow since each pattern would likely be based on using existing SPINE rules, only infrequently needing a specific extension of a built-in SPINE rule.

5.6.2 Singleton

The **Singleton** pattern requires that at most one instance of a class be instantiated/accessed at one time. The usual implementation to achieve this is to store a single instance in a `static final` variable of the class itself; because it is `static`, there is only one copy of the variable, and because it is `final`, it cannot be changed once assigned (typically during class loading).

In order to prevent other classes instantiating the **Singleton** pattern, it is necessary to ensure that none of the constructors are publicly accessible. Because Java creates a default (`public`) zero argument constructor if no others are provided, we must disable this behaviour by creating at least one constructor in the class.

Because this requirement for non-instantiability is a fairly common requirement in other design patterns, it has been extracted to form a **Non Instantiable** mini-pattern. (Mini-patterns were discussed in Chapter 4.) The definition of the **Non Instantiable** mini-pattern is as follows:

```
realises('NonInstantiable',[C]) :-
  or([
    isAbstract(C),
    and([
      exists(constructorsOf(C),Cn.true),
      forAll(constructorsOf(C),Cn.hasModifier(Cn,private)).
    ])
  ]).
```

The definition of the (public) **Singleton** pattern therefore is as follows:

```
realises('PublicSingleton',[C]) :-
  realises('NonInstantiable',[C]),
  forAll(constructorsOf(C),Cn.hasModifier(Cn,private)),
  exists(fieldsOf(C),F.
    and([
      hasModifier(F,static),
      hasModifier(F,public),
      hasModifier(F,final),
      typeOf(F,C),
      nonNull(F)
    ])
  ).
```

As with the previous example, it is necessary to appeal to a new term, `nonNull`, in order to prove the **Singleton** pattern. This ensures that the value of the field `F` is assigned a value, and since Java's typing system only allows `F` to hold a value of type `C`, it will hold an instance of class `C`.

As discussed in Chapter 4, the **Singleton** pattern has several variants. These variants are accounted for with different rules that match the same term. A super-pattern rule can be created to tie these patterns into the same meaning:

```
realises('Singleton',[C]) :- or([
  realises('PublicSingleton',[C]).
  realises('PrivateSingleton',[C]).
  realises('LazySingleton',[C])
]).

realises('PrivateSingleton',[C]) :-
  realises('NonInstantiable',[C]),
  exists(fieldsOf(C),F.
    and([
      hasModifier(F,static),
      hasModifier(F,private),
      typeOf(F,C),
      nonNull(F)
    ])
  ).
realises('LazySingleton',[C]) :- ...
realises('PublicSingleton',[C]) :- ...
```

5.7 Summary

This chapter presented the SPINE language as a way of defining logical statements about design patterns. The syntax is loosely based on Prolog, and rules are defined with SPINE terms to allow the proof engine HEDGEHOG (discussed in Chapter 6) to perform goal-oriented proofs.

The syntax was presented in Section 5.2; the semantics of terms in Section 5.3. Rules were presented and a consistent rule set defined in Section 5.4. The distinction between weak and strong semantics discussed in Section 5.5. Lastly, example design patterns were shown in Section 5.6 to show how the patterns are defined in terms of the SPINE language.

The **Singleton** pattern was used to demonstrate how SPINE patterns can be defined in terms of mini-patterns, and how different variants can be defined by specifying multiple rules for each pattern.

When a pattern is added to the SPINE library, it may require specific predicates to be added such as '`nonNull`'. As the pattern library grows, it is expected that the number of additional predicates required for each new pattern will tend to zero. It is possible to add extra patterns into the system at a later stage, providing that they do not need new predicates to be defined; alternatively, new predicates can be added in although this will require more work to do so.

Chapter 6

The HEDGEHOG proof engine

At the heart of HEDGEHOG lies the proof engine that verifies design patterns based on their SPINE definitions. It proves whether or not a Java class, or set of classes, realises a design pattern.

Since the HEDGEHOG proof engine is designed to be both fully automated and hidden from the end-user, it is necessary to use a proof system that can:

1. Be embedded within the verification tool
2. Provide mechanisms to automatically search for proofs without user input
3. Provide enough state about the proof tree afterwards so that it can be translated into a suitable error message in the case of proof failure

An initial investigation used the Oyster proof system [Hor88] (formerly known as Nuprl [CAB+86], part of the Oyster-Clam proof system [BvHSH90]). The approach taken was similar to that of [JvdBH+98], where the Java program was translated into a large set of constraints and then fed to the proof system remotely. However, this solution did not scale well with large programs, because the source code needed to be translated and sent to the proof system in its entirety before proof could start. The same observation is noted in [JP03, page 15] with its translation into PVS [ORS92]. A similar problem would exist if Isabelle/HOL [NPW02] was used for proving properties of program code.

Additionally, the strength needed for automated proof in the tightly constrained uses for pattern verification is not as high as with mathematical logic; no recursive behaviours are required nor higher order logic (except for the special cases of the `forAll()` and `exists()` quantifiers, which are discussed specifically). As a result, a proof engine was specifically created to support verification of patterns, to meet the above requirements:

Embedded Because the proof system is developed in Java, it can be embedded within the tool directly.

Automated The proof nodes are maintained as Java objects, and if a Java source file needs to be processed, it can be parsed and loaded into the same memory space as the proof engine. This obviates the need for all source files to be translated into logic statements prior to the execution of the proof; they can be dynamically calculated during the execution of the proof process.

Informative The proof node can be translated into an error message in the case of proof failure.

As a result a basic automated proof tool was created, based on experience with Oyster. Patterns are defined as rules in SPINE, which allows patterns to be defined externally and loaded into HEDGEHOG at a later stage. The rules are made up from *built-in* and *derived* predicates and functions that allow HEDGEHOG to reason about the structure and implementation of the Java classes. Java class definitions are loaded from the Java source files into memory as an abstract data structure.

The system is given a conjecture, such as:

```
realises('Singleton',['java.lang.System'])
```

HEDGEHOG then tries to automatically prove this statement using the rules and definitions encoded in the SPINE pattern library. At the end of this automated proof attempt, HEDGEHOG returns with one of three results:

Complete The class does realise the pattern

Failed The class does not realise the pattern

Unknown The proof system is unable to tell whether or not the class meets the pattern

As the proof system is intended to be used as an automated tool, it is not desirable to show proof trees or allow the user to interact with a failed (or partial) proof. However, the proof system does try to generate sensible messages in the event of a proof failure, and this is discussed in more detail in Chapter 7.

The representation of patterns in SPINE has been covered in Chapter 5; the processing of Java files is covered in Section 6.1 and the proof engine is covered in Section 6.2. More detailed information about the built-ins and the way they work is covered in Section 6.3.

6.1 Representing Java

This section discusses the way Java code is represented in memory to support the predicates used in the SPINE pattern definitions.

In order to process Java classes, HEDGEHOG represents Java class definitions as abstract data structures in memory. These class definitions are parsed from either:

Source files HEDGEHOG's built-in Java parsing engine reads the Java source file and creates a class definition. This is discussed further in Section 6.1.1.

Class files Java provides a way of interrogating Java classes at run-time using a process called *reflection*. This uses the compiled output — the `.class` file — to determine the signature of the class. However, reflection only provides information about the structure of the class (what methods exist, what their argument types are etc.) — and not implementation details of individual methods. This is discussed further in Section 6.1.2.

The definitions are represented in memory as an abstract data type structure. The elements of the AST are shown in Figure 3.3. Built-in SPINE functions and predicates, such as the `constructorsOf()` function, return information based on the contents of these data structures.

Every class has its own definition, which contains a set of member definitions (methods, constructors and fields). The method and constructor bodies consist of a set of statements, and since a statement block is considered a statement, statements can be nested. An expression is also considered a statement, and expressions are also built recursively.

6.1.1 Java source files

Java source files are parsed using a built-in Java parser, based on the freely available ANTLR [Par] parser for Java. This parser loads a Java definition into a tree-like data structure, which is then converted into a HEDGEHOG style data structure as shown in Figure 3.3.

Note that HEDGEHOG does not use Java language reflection. Although this provides a mechanism to detect the signature of a Java class (what methods exist, what modifiers methods have etc.), it does not provide any information about the implementation of those methods. It is therefore not suitable for parsing Java classes for HEDGEHOG's use, as most patterns require some kind of constraint on the implementation of methods and not just their signatures.

Amongst other things, the parser also cleans up references to fields and methods. In Java, it is possible to refer to a field just by using '*name*', instead of '`this.`*name*'. Similarly,

references to methods that do not have an argument are prefixed with '`this.`' to distinguish which methods are being referred to. This ensures that when variables are referred to by name in the methods, it is clear when a variable reference is for a local variable or an instance field.

Figure 6.1: Example Java code

```
public class Person {            // Class definition
  private String name;           // Field definition
  public Person(String name) {   // Constructor definition
    this.name = name;            // Body of constructor
  }
  public String toString() {     // Method definition
    return this.name;
  }
}
```

Figure 6.1 shows an example of Java code, and Figure 6.2 shows its abstract syntax tree. Figure 6.3 shows examples of built-in functions using the code in Figure 6.1. The data structure is accessed via built-in functions such as `constructorsOf()`. Note that the values displayed in '{' and '}' are results that cannot be entered directly into HEDGEHOG; they represent internal values that have been demonstrated here to show the evaluation of a function.

6.1.2 Java class files

HEDGEHOG does not currently support using class files as a source of Java code; this section explains what the problems are with this approach and why it does not matter that they are not supported.

Compiled Java class files consist of class, method, and field signatures (what the name is, what the type is, what the arguments are etc.). Method bodies consist of *byte-code*, which is an object-oriented stack-based processor language. An example of byte-code is shown in Figure 6.4.

Patterns that rely solely on the class and method signatures can use class files as the class definition, since this is all the information that is available via Java reflection. If implementation-specific details are required, then it would involve a different way of implementing the built-in predicates (which currently just work on source files).

Figure 6.2: Example Java code AST

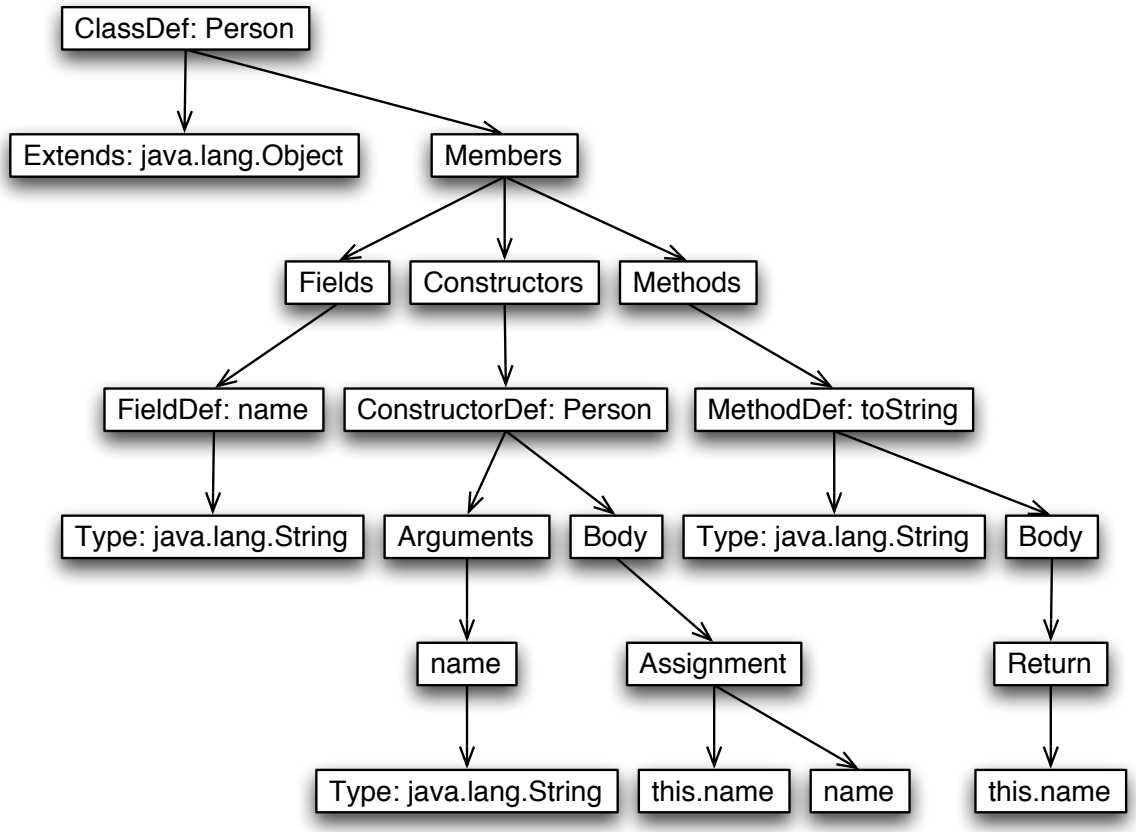


Figure 6.3: Example evaluation of functions

```
constructorsOf('Person') =
  {Person(java.lang.String)}.
body({Person(java.lang.String)}) =
  {assignment(fieldRef(this,name), name)}.
isPublic({Person(java.lang.String)}) =
  true.
isFinal({Person(java.lang.String)}) =
  false.
```

Figure 6.4: Example of Java byte-code

```
Method java.lang.String toString()
  0 new java.lang.StringBuffer
  3 dup
  4 invokespecial java.lang.StringBuffer()
  7 aload 0
  8 invokevirtual java.lang.Class getClass()
 11 invokevirtual java.lang.String getName()
 14 invokevirtual java.lang.StringBuffer
      append(java.lang.String)
 17 ldc String "@"
 19 invokevirtual java.lang.StringBuffer
      append(java.lang.String)
 22 aload 0
 23 invokevirtual int hashCode()
 26 invokestatic java.lang.String toHexString(int)
 29 invokevirtual java.lang.StringBuffer
      append(java.lang.String)
 32 invokevirtual java.lang.String toString()
 35 areturn
```

It is not possible to uniquely convert the set of byte-codes for a method's implementation into its original Java representation — a process which is known as *decompilation*. Although Java decompilers do exist, the generated output source will often not be the same as the original source. As such, patterns which are defined correctly in the source may be translated into a form which is unrecognisable by HEDGEHOG after decompilation. The original Java implementation of Figure 6.4 is shown in Figure 6.5, whilst the decompiled output might look like Figure 6.6.

Figure 6.5: Original Java implementation

```
public String toString() {
  return getClass().getName()
     + "@"
     + String.toHexString(hashCode());
}
```

Figure 6.6: Decompiled Java implementation

```
public String toString() {
  StringBuffer buffer = new StringBuffer();
  buffer.append(this.getClass().getName());
  buffer.append("@");
  int code = this.hashCode();
  buffer.append(String.toHexString(code));
  return buffer.toString();
}
```

Because several design patterns are dependent on the implementation of a method, it is not directly possible to just use the compiled `.class` files. Built-in functions which process Java statements or expressions (for example, searching through to verify whether or not a statement contains an assignment expression) will not be directly applicable to byte-code operations.

Since the patterns are defined in terms of these built-in functions, it may be possible to write separate implementations of built-in functions in order to deal with the class code instead of the abstract representations of statements and classes.

Implementing each built-in function for the two representations would be a significant amount of work. Additionally, it would provide little benefit; it is assumed the purpose of HEDGEHOG is to verify whether a class (or set of classes) realises a design pattern correctly at

the development stage, and thus if they do not realise it, they can then be changed. This implies the source code needs to be available for changing, so HEDGEHOG can be executed on that instead of any compiled output.

6.1.3 Inner classes

Since Java 1.1 it has been possible to define *inner classes*, which are classes defined within the scope of another class. There are several different types of inner classes, but essentially they are no different from classes that are defined normally.

The Java compiler translates the inner classes into separate classes at compile time; there is no concept of an inner class at the Virtual Machine level. If a class *HashMap* defines an inner class *Entry*, then the resulting classes are called `HashMap.class` and `HashMap$Entry.class`. (There is another style of inner class called an *anonymous inner class*. These do not have a name, but are simply numbered, so an anonymous inner class results in `HashMap$1.class`, and so on.) These are treated as two separate classes by the run-time system, so in principle, there is no difference between an inner class and a normal 'outer' class.

HEDGEHOG does not currently support inner classes. Although they are theoretically the same as outer classes, it complicates the parsing and processing engine that is used to parse source files and create class definitions. However, this does not affect the power of HEDGEHOG. If a design pattern was implemented using inner classes, then HEDGEHOG would not be able to verify it directly using the source. However, if an external translation tool converted the Java source files from using inner classes to using non-inner classes (using the same technique as the Java compiler uses) then it would be possible to run the verification on these expanded classes.

Note that if patterns could be verified using compiled output alone, it would be possible to run HEDGEHOG on the compiled output. Since there is no concept of an inner class at the VM level, HEDGEHOG would be able to treat these as separate outer classes.

6.1.4 Native methods

For similar reasons to those discussed in Section 6.1.3, HEDGEHOG does not support the use of native methods to implement patterns. There is no specific problem with using native methods in a class that realises a pattern, providing that it does not rely on the execution of the native method to realise the pattern correctly.

6.2 Proof engine

HEDGEHOG provides a simple user interface to the underlying proof engine. The user interface provides a read-prove-print loop that allows individual patterns to be tested against specific class files.

Because of the way in which the patterns are defined, the proof system will not be *complete*; that is, there will be implementations of patterns that HEDGEHOG is not able to prove. For example, a pattern may be implemented in a variant that is not known to HEDGEHOG, or for which no suitable definition in SPINE exists. Alternatively, the pattern may be implemented with a minor deviation of the SPINE definition (for example, using an interface instead of an abstract class). However, the system will be *correct*; that is, if the HEDGEHOG system proves that a given class (or set of classes) realises a pattern, then it will be the case.[1]

As HEDGEHOG is expected to be used as an automatic verification tool in conjunction with a developer, false negatives or unknown verdicts are not a disaster. When a negative verdict is raised, it simply means that further expert (human) verification is required, either because HEDGEHOG is unable to solve the problem, or because it does not meet HEDGEHOG's definition of a pattern.

6.2.1 Overview of proof process

The root proof node is used to represent the conjecture, which is usually of the form '`realises('Pattern',[Class])`'. This is rewritten using the SPINE rule that matches the given '`Pattern`' to produce a new set of goals. This process recurses until there are no more '`realises()`' left in the conjecture. (The proof tree is annotated with rewritings that occur in order to generate suitable error messages, as discussed in Chapter 7.)

Once the rewriting phase has finished, the proof process uses the built-in implicational rules (which are listed in Section 6.2.3.1) to advance the state of the proof.

A list of rules are then selected which are applicable to the current node. The first one is then used to generate a number of sub-goals for the current proof node. As the evolution of the proof is dynamic, the proof nodes are dynamic objects and may have children proof nodes generated and replaced during backtracking.

After these sub-goals have been generated, the state of the proof tree is updated, and the proof process recurses through the unproven proof goals, usually using a depth-first procedure.

[1] Obviously with respect to the SPINE definition; errors in the pattern definition excepted.

If a proof node is failed, then backtracking is used to archive the proof node's children, by applying the next applicable rule. This continues until either the proof node is complete or failed and there are no more applicable rules.

6.2.2 Proof tree

The proof tree is made up from *proof node*s. As a tree data structure, a single proof node may have zero or more children; a proof node with no children is a leaf. Each proof node has:

Goal The goal to be proven. This is a SPINE conjecture, such as a `realises()` predicate, to determine whether or not a class (or set of classes) realises a pattern. Once a proof node has been created with a given goal, the goal does not change.

State The proof state (described below), of the current node. As the proof progresses, the state of individual proof nodes change, and these changes propagate up the proof tree.

Children Each proof node can have zero or more children. If a proof node has a set of children, then the state of the parent proof node is a combination of the child proof states.

Rule The rule used to generate the children proof nodes. If no rule has yet been applied, this will be `null`. When the proof is finished, the proof nodes can be interrogated to determine what rule each node was satisfied by.

Alternate rules The rules that may be used in the case of backtracking being required. This may be an empty set.

The proof node is in one of four states:

None The proof node has not been attempted.

Partial Some child proof nodes have been (successfully) proven, but some still remain.

Complete The proof node (and its children) have been proven.

Failed The proof node has failed.

Of these, **complete** and **failed** are terminal states, whereas **none** and **partial** are non-terminal states. At the end of an automated proof attempt, if the root proof state is in a non-terminal state then the proof system outputs **unknown**.

Since each proof node has a single unchanging goal, it may be easy to confuse the two. However, the proof node is a *mutable* object, and refers to the current goal and the state of the proof at that part of the proof tree, whereas the goal is an *immutable* statement of what is required.

Each time a proof node's state changes (for example, from **none** to **partial**, or from **none** to **failed**) the parent node's state is recalculated. Thus changes in a proof tree are driven by changes in a child node's proof state, and the changes propagate upwards. By default, proof nodes merge their states such that when a child's state changes to **failed**, the parent node's state changes to **failed**. If the child's proof state changes to **partial**, the parent's node is upgraded from **none** to **partial**. Lastly, if the child's state changes to **complete** then the parent node is updated; if all children are marked as **complete** then the parent node is marked as **complete**; otherwise, it stays at **partial**.

Some built-in rules in HEDGEHOG combine the child proof states in a different way. For example, the **exists** quantifier combines the child's proof node state in such a way that only one child's success is required to cause the proof node to be marked as **complete**. The remaining child nodes are left as they are, and play no further part in the proof process.

6.2.3 Rules

HEDGEHOG has two types of rules that it can use. The first are the built-in rules that allow the implicational rules to evaluate the proof tree. These are listed in Section 6.2.3.1, and cannot be changed by the user. The second type of rules are the pattern definition rules that can be loaded in from external files when HEDGEHOG starts. The set of files to load are defined in HEDGEHOG's main configuration file, and the files make up HEDGEHOG's known library for design patterns. Additional patterns and other rules may be added into HEDGEHOG by editing these text files and restarting HEDGEHOG.

The proof advances by the application of *rule*s to proof nodes. The application of a rule either:

- Changes the state of the proof node directly, such as **complete** or **failed**.
- Creates a number of sub-goals that need to be proven.

Each sub-goal that is created is packaged in a proof node, and inserted into the proof tree as a child of the current proof node. When a new proof node is created, the state is set to **none** to indicate that the proof node has not yet been attempted.

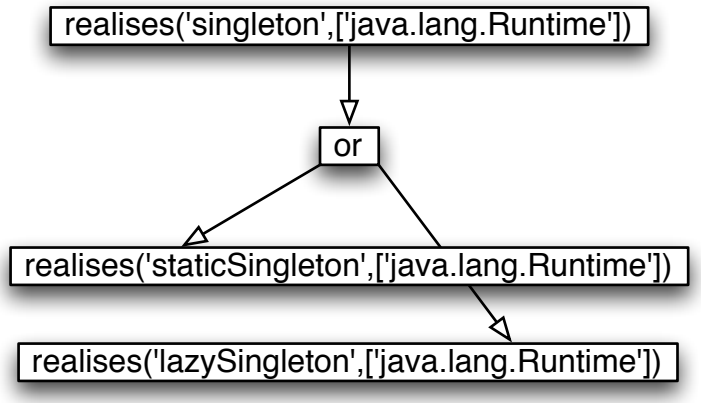
Not all rules are *applicable* to all proof nodes/goals. The proof engine uses one-way unification to determine whether or not the rule is applicable for a specific goal. This allows HEDGEHOG to have potentially many rules defined, but only be able to select a few rules for each proof node/goal.

The simplest type of rule matching operates on the term name; for example, there are a set of '`realises`' rules; one for each pattern. When proving a '`realises`' goal such as '`realises('Singleton',List)`', it is rewritten using the definition in the SPINE library to '`or([realises('LazySingleton',List),realises('PublicSingleton',List)])`'. (A subsequent step will then rewrite the other '`realises`' terms.)

Figure 6.7 shows the goal '`realises('Singleton',['java.lang.Runtime'])`' before the body is rewritten, and Figure 6.8 shows the expanded proof tree with sub-goals after it is rewritten.

Figure 6.7: Before the rewriting is applied

Figure 6.8: After the rewriting is applied



The choice of which rule to apply is decided by selecting the first applicable rule for the current node. If there is more than one applicable rule, the remaining applicable rules are

tacked onto the current proof node. During backtracking (see below), these rules are applied to generate fresh sub-goals.

6.2.3.1 Implicational rules

The axioms of the system are derived from the evaluable propositions (from Section 5.3.2):

$$\text{Either } \frac{}{EP} \text{ or } \frac{}{\texttt{not}\,(EP)} \text{ for every variable-free evaluable proposition } EP \tag{6.1}$$

The following are built-in implication rules in HEDGEHOG and cannot be changed by the user:

$$\frac{\mathcal{A}_1 \ \ldots \ \mathcal{A}_n}{\texttt{and}\,([\mathcal{A}_1,\ldots,\mathcal{A}_n])} \tag{6.2}$$

$$\frac{\mathcal{A}_i}{\texttt{or}\,([\mathcal{A}_1,\ldots,\mathcal{A}_n])} \text{for } 1 \leq i \leq n \tag{6.3}$$

$$\frac{\mathcal{A}(c_1) \ \ldots \ \mathcal{A}(c_n)}{\texttt{forAll}\,(\mathcal{S},x.\mathcal{A}(x))} \text{ where } \mathcal{S} \text{ is an evaluable set that evaluates to } \{c_1,\ldots,c_n\} \tag{6.4}$$

$$\frac{\mathcal{A}(c_i)}{\texttt{exists}\,(\mathcal{S},x.\mathcal{A}(x))} \text{ for } 1 \leq i \leq n \text{ where } \mathcal{S} \text{ is an evaulable set that evaluates to } \{c_1,\ldots,c_n\} \tag{6.5}$$

$$\frac{\mathcal{A}}{\texttt{not}\,(\texttt{not}\,(\mathcal{A}))} \tag{6.6}$$

$$\frac{\texttt{or}\,([\texttt{not}\,(\mathcal{A}_1),\ldots,\texttt{not}\,(\mathcal{A}_n)])}{\texttt{not}\,(\texttt{and}\,([\mathcal{A}_1,\ldots,\mathcal{A}_n]))} \tag{6.7}$$

$$\frac{\texttt{and}\,([\texttt{not}\,(\mathcal{A}_1),\ldots,\texttt{not}\,(\mathcal{A}_n)])}{\texttt{not}\,(\texttt{or}\,([\mathcal{A}_1,\ldots,\mathcal{A}_n]))} \tag{6.8}$$

$$\frac{\texttt{forAll}\,(\mathcal{S},x.\texttt{not}\,(\mathcal{A}(x)))}{\texttt{not}\,(\texttt{exists}\,(\mathcal{S},x.\mathcal{A}(x)))} \tag{6.9}$$

$$\frac{\texttt{exists}\,(\mathcal{S},x.\texttt{not}\,(\mathcal{A}(x)))}{\texttt{not}\,(\texttt{forAll}\,(\mathcal{S},x.\mathcal{A}(x)))} \tag{6.10}$$

6.2.4 Backtracking

Sometimes, the proof engine is unable to prove a goal one way or another; for example, the pattern is realised differently from the SPINE specifications. In these instances, the goal is left as unproven and the proof state remains **none** or **partial**.

If the current proof node cannot be further processed with the current rule, then proof may continue by backtracking. This is achieved by archiving the current proof tree, and applying an alternate rule to generate a different set of sub-goals. The process repeats until no more rules are applicable. When no more rules are applicable to the current proof node, and the proof cannot be moved forward any more, then proof stops. The failed proof nodes are archived (and so do not play any further part in the proof) but are kept in order to generate the error messages as discussed in Chapter 7.

6.2.5 Proof strategy

Once the '`realises`' predicates have been re-written, the proof strategy follows a depth-first proof, driven by rules (from Section 6.2.7). The strategy involves the following steps:

1. Start with a proof node `N` that has goal `G`.
2. If `N` is **complete** or **failed**, terminate.
3. Find all rules `Rs` that can apply to `G`.
4. Annotate `N` with this set of applicable rules `Rs`.
5. For each rule `R` in the set `Rs`, do until `N` is **complete**:
   (a) Archive any existing sub-nodes `SNs` for the current node `N`.
   (b) Apply rule `R` to `G` to generate a set of sub-goals `SGs`. Generate sub-nodes `SNs` for each goal in `SGs`
   (c) For each sub-node `SN` in `SNs`, recurse.
   (d) Update the state of node `N` for each sub-node that is proven or disproven.
6. If the set `Rs` is empty or has been completely traversed, and the goal has not been proven, return an 'unknown' verdict.

This process ensures that if the proof system cannot prove a goal with a particular rule, then it is possible to continue processing using another rule.

The state of sub-nodes are combined to update the parent node. The parent node is updated as each sub-node changes. If any child is **failed**, then the parent node is marked as **failed**; if all children are **complete**, then the parent node is marked as **complete**; if all children are **none** then the parent is **none**, otherwise the parent is marked as **partial**.

6.2.6 Soundness

The proof process is sound; if the proof completes from some initial goal, then that goal is true by the semantics in Chapter 5. This follows from the fact that each of the rules (Equation 6.11 – Equation 6.10) are sound with respect to the semantics already given, and by induction on the structure of the proof that the proof is also sound.

Each rule can be individually demonstrated to be sound; e.g., Equation 6.1 follows from the definition of the evaluable proposition itself. The only one that requires further analysis is Equation 6.11, which uses substitution of expression (and thus complicates the argument). In order to argue the soundness of this rule, we must consider the substitution of arbitrary formulæ into an arbitrary environment, and show that if the formulæ themselves are sound then so are the replacements into that environment. To do this, we need to define the following:

Lemma 6.2.1. *Assume given formulæ $\mathcal{A}$ and $\mathcal{B}$, and an enviroment E[_] (with potentially free variables in each, and binding operators binding free variables in the environment's hole _). If for all substitutions σ, binding all free variables in $\mathcal{A}$ and $\mathcal{B}$ to constants, we have that $\sigma(A)$ is true* iff *$\sigma(\mathcal{B})$ is true, then for all substitutions τ binding all free variables in E[$\mathcal{A}$] and E[$\mathcal{B}$] to constants, we have that $\tau(E[\mathcal{A}])$ is true* iff *$\tau(E[\mathcal{B}])$ is true.*

Proof. By induction on the structure of E. □

6.2.7 Termination

In order to show that the proof engine terminates on a consistent set of SPINE rules, the proof is processed in two distinct phases:

1. Recursive rewriting of all occurrences of `realises()` with their respective rule bodies.

2. Application of the implicational rules that allow the logic to be evaluated.

6.2.7.1 Rewriting of SPINE rules in the proposition

The first phase successively increases the depth of the proposition, since a `realises()` SPINE rule body is often a compound proposition. This increase continues until there are no `realises()` SPINE rules left to replace. All occurrences of a `realises()` formula are recursively rewritten using the following equality rule:

$$E[\texttt{and}([\mathcal{A}_1,\ldots,\mathcal{A}_n])] \equiv E[\texttt{realises}(P,[C_1,\ldots,C_m])] \quad \text{where } \texttt{realises}(P,[C_1,\ldots,C_m])\texttt{ :- }\mathcal{A}_1,\ldots,\mathcal{A}_n \text{ is a variable-free instance of a SPINE rule and } E[\mathcal{A}] \text{ is any expression with a subexpression } \mathcal{A} \tag{6.11}$$

It is necessary to allow re-writing at any depth of the expression, since it is unlikely that the `realises()` goal will only exist at the top level. It is also necessary to perform the re-writing steps first so that the second phase strictly reduces the depth of the proposition.

The E[$\mathcal{A}$] notation is used to denote rewriting at any level of the expression. For example, `or([realises('NonInstantiable', 'TheClass'), isAbstract('TheClass')])` matches the rewrite rule above, and can be rewritten into `or([...,isAbstract('TheClass')])` (where ... represents the body of the corresponding SPINE rule). Prior to this re-writing step, the rule's variables are renamed on both sides of the rule definition in order to avoid capturing occurrences of the same variable name in the containing expression.

This phase will terminate since there is a partial ordering on the `realises()` SPINE rules. If the set of rules is consistent, then there are no mutually recursively defined rules. Thus, each replacement of the `realises()` rules will replace it with either a proposition that contains no further `realises()` rules, or will contain a `realises()` rule that is strictly less than the one it is replacing. As the set of rules is finite, eventually this phase of the rule processing will finish. Since the rule set is consistent and the size of the rule bodies themselves are finite, then this must result in a finite sized proposition.

Lemma 6.2.2. *The first phase of the proof process (rewriting rule bodies of* `realises()` *rules) will terminate with a finite proposition.*

Proof.

- If a proposition has no `realises()`, then the first phase terminates.
- If a proposition contains `realises()`, then the rewriting will replace it with a finite body that contains zero or more `realises()`. Each of the `realises()` will be smaller under $\succ$ (from Definition 5.4.1). The `realises()` will keep getting smaller under $\succ$ until they are rewritten with a finite body with no further `realises()`, and then the first phase will terminate. □

6.2.7.2 Application of the remaining implicational rules

The second phase of the proof applies the implicational rules. Given that there are no more `realises()` rules to be applied, the proof will terminate if a measure of the proposition can be shown to be strictly decreasing.

The size of the proof space is dependent on the number of branches that the proof has. If the proof had infinite branches, it may require an infinite amount of time to finish. On the other hand, if the proof has finite branches and finite depth, then there is a finite amount of work to do and thus the proof will terminate. If we can show that the branching is finite, we can use a measure based on the depth of the proposition to prove that the proof process will terminate.

Lemma 6.2.3. *The branching of the proof tree is finite.*

Proof. The proof branching occurs as a result of:

- `and()` and `or()` propositions in the SPINE rules themselves
- `forAll()` and `exists()` introducing branching
- Backtracking with substitution of other rules

Each SPINE rule is finite, and therefore each of the coded `and()` and `or()` propositions in SPINE rules must also themselves be finite.

The `forAll()` and `exists()` propositions expand with one branch per set member of the (evaluated) evaluable set. However, this evaluable set must be finite in size, since the size of the Java code base is finite, and as a result of which the `forAll()` and `exists()` propositions must therefore generate a finite set of work.

The substitution of rules during backtracking must be finite, since there are a finite number of SPINE rules.

Therefore, the branching of the proof tree is finite. □

The rules Equation 6.1 — Equation 6.6 have a smaller depth of each of the premises than the conclusion. Thus, if these rules are applied, the proposition becomes strictly smaller.
The remaining rules Equation 6.7 — Equation 6.10 have the same depth of proposition for the premise as the conclusion. In each case, the depth of the proposition captured by the `not()` rule decreases. The premise of each of the following rules are not of the form `not(`$\mathcal{A}$`)`; therefore, these rules cannot be applied more than once to the formula as a whole. The only rules that match the premise are those that decrease the depth of the proposition further.

Combining these sets of rules will give (for example):

$$\frac{\frac{\texttt{not}(\mathcal{A}_1)\ \ldots\ \texttt{not}(\mathcal{A}_n)}{\texttt{and}([\texttt{not}(\mathcal{A}_1),\ldots,\texttt{not}(\mathcal{A}_n)])}}{\texttt{not}(\texttt{or}([\mathcal{A}_1,\ldots,\mathcal{A}_n]))}$$

To show that these rules terminate, a measure must be defined that is strictly decreasing for each of these rules. Multisets are used to show that either the depth of the proposition as a whole decreases, or that the depth of `not()` propositions strictly decreases.

A *multiset* [DM79] is a set that allows duplicate elements (also known as a *bag*), denoted by $\{\}$. Multisets are used to show that a measure of the rule's premise and conclusion decreases, in order to prove that the rules terminate. $s_1 \cup s_2$ is the multiset union of s_1 and s_2.

It is possible to define a well-founded ordering (denoted by $\gg$) of multisets such that they can be compared:

$$X \overset{def}{\gg=} Y \text{ iff } \forall y \in Y.\exists x \in X.x >= y.$$

$$\text{then } X \overset{def}{\gg} Y \text{ iff } X \gg= Y \text{ and } \exists y \in Y, x \in X.x > y$$

In the depth measures, specific SPINE formulæ are represented as α. (The reason for avoiding $\mathcal{A}$ is that the $\mathcal{A}$ variable can apply to any formula in the rules, and it is not possible to calculate the depth of any formula. However, by choosing an arbitrary specific formula α, we can calculate the depth of that specific term and use that to show that the measure holds.)

Definition 6.2.4. *The* depth *of a proposition* α *is denoted by* $|\alpha|_d$ *under the normal definition of depth.*

Definition 6.2.5. *The* negation depth *of a proposition* α *is denoted by* $|\alpha|_n$. *For* $f \neq$ not, $|f(t_1,\ldots,t_n)|_n \overset{def}{=} |t_1|_n \cup \ldots \cup |t_n|_n$. $|\texttt{not}(\alpha)|_n \overset{def}{=} \{|\texttt{not}(\alpha)|_d\}$, *where* $\{\}$ *is a multiset of the depths* $||_d$. *For constant or variable* α, $|\alpha|_n \overset{def}{=} \{|\alpha|_d\}$.

Definition 6.2.6. *The overall measure* $|\alpha|$ *of proposition* α *is a lexicographic combination of the normal depth as well as the negation depth:* $|\alpha| \rhd |\beta|$ *iff* $|\alpha|_d > |\beta|_d \vee (|\alpha|_d = |\beta|_d \wedge |\alpha|_n \gg |\beta|_n)$

To prove that the measure of the conclusion is greater than the measure of the premise, each rule needs to be shown to do so in turn. The proofs use the rules with each variable instantiated with a specific SPINE formula, and are represented as α_1 and α_n.

Using the conclusion and premise of the `and()` rule (Equation 6.2), it can be seen that the conclusion has a greater measure than the premise, by working backwards from the goal as follows:

Lemma 6.2.7. *The measure of the conclusion is greater than the measure of each of the premises for the* `and()` *rule (Equation 6.2).*

Proof. To show that the measure of the `and()` rule's conclusion is greater than its premises:

$$|\texttt{and}([\alpha_1,\ldots,\alpha_n])| \quad \rhd \quad |\alpha_i| \quad \text{for } 1 \leq i \leq n$$

it is sufficient to show that the depth is larger:

$$|\texttt{and}([\alpha_1,\ldots,\alpha_n])|_d \quad > \quad |\alpha_i|_d \quad \text{for } 1 \leq i \leq n$$

which is true by definition of $||_d$. □

Lemma 6.2.8. *The measure of the conclusion is greater than the measure of the premise for each of the* `or()` *rules (Equation 6.3).*

Proof. To show that the measure of each of the `or()` rule's conclusion is greater than its premises:

$$|\texttt{or}([\alpha_1,\ldots,\alpha_n])| \quad \rhd \quad |\alpha_i| \quad \text{for } 1 \leq i \leq n$$

it is sufficient to show that the depth is larger:

$$|\texttt{or}([\alpha_1,\ldots,\alpha_n])|_d \quad > \quad |\alpha_i|_d \quad \text{for } 1 \leq i \leq n$$

which is true by definition of $||_d$. □

Lemma 6.2.9. *The measure of the conclusion is greater than the measure of each of the premises for the* `forAll()` *rule (Equation 6.4).*

Proof. To show that the measure of the `forAll()` rule's conclusion is greater than its premises:

$$|\texttt{forAll}(\mathcal{S},x.\alpha(x))| \quad \rhd \quad |\alpha(c_i)| \quad \text{for } 1 \leq i \leq n \text{ and } \mathcal{S} \rightsquigarrow \{c_1,\ldots,c_n\}^2$$

it is sufficient to show that the depth is larger:

$$|\texttt{forAll}(\mathcal{S},x.\alpha(x))|_d \quad > \quad |\alpha(c_i)|_d \quad \text{for } 1 \leq i \leq n \text{ and } \mathcal{S} \rightsquigarrow \{c_1,\ldots,c_n\}$$

which is true by definition of $||_d$, given that $|x| = |x_i|$ for $1 \leq i \leq n$ (as a variable and a constant have the same depth). □

Lemma 6.2.10. *The measure of the conclusion is greater than the measure of each of the premises for the* `exists()` *rule (Equation 6.5).*

Proof. To show that the measure of the `exists()` rule's conclusion is greater than its premises:

$$|\texttt{exists}(\mathcal{S},x.\alpha(x))| \quad \rhd \quad |\alpha(c_i)| \quad \text{for } 1 \leq i \leq n \text{ and } \mathcal{S} \rightsquigarrow \{c_1,\ldots,c_n\}$$

it is sufficient to show that the depth is larger:

$$|\texttt{exists}(\mathcal{S},x.\alpha(x))|_d \quad > \quad |\alpha(c_i)|_d \quad \text{for } 1 \leq i \leq n \text{ and } \mathcal{S} \rightsquigarrow \{c_1,\ldots,c_n\}$$

which is true by definition of $||_d$, given that $|x| = |x_i|$ for $1 \leq i \leq n$ (as a variable and a constant have the same depth). □

[2] The sets only ever evaluate to ($\rightsquigarrow$) constants, from Section 5.3.3

Lemma 6.2.11. *The measure of the conclusion is greater than the measure of each of the premises for the double negation rule (Equation 6.6).*

Proof. To show that the measure of the `not(not())` rule's conclusion is greater than its premise:

$$|\texttt{not(not(}\alpha\texttt{))}| \quad \rhd \quad |\alpha|$$

it is sufficient to show that the depth is larger:

$$|\texttt{not(not(}\alpha\texttt{))}|_d \quad > \quad |\alpha|_d$$

which is true by definition of $||_d$. □

The rules involving pushing negation in (Equation 6.7 — Equation 6.10) have the same depth as defined by $||_d$, but have a reducing negation depth as defined by $||_n$:

Lemma 6.2.12. *The measure of the conclusion is greater than the measure of each of the premises for the negated* `and()` *rule (Equation 6.7).*

Proof. To show that the measure of the negated `and()` rule's conclusion is greater than its premise:

$$|\texttt{not(and(}[\alpha_1,\ldots,\alpha_n]\texttt{))}| \quad \rhd \quad |\texttt{or(}[\texttt{not(}\alpha_1\texttt{)},\ldots,\texttt{not(}\alpha_n\texttt{)}]\texttt{)}|$$

Since the depths of each are equal:

$$|\texttt{not(and(}[\alpha_1,\ldots,\alpha_n]\texttt{))}|_d \quad = \quad |\texttt{or(}[\texttt{not(}\alpha_1\texttt{)},\ldots,\texttt{not(}\alpha_n\texttt{)}]\texttt{)}|_d$$

it must be shown that the negation depth is greater:

$$|\texttt{not(and(}[\alpha_1,\ldots,\alpha_n]\texttt{))}|_n \quad \gg \quad |\texttt{or(}[\texttt{not(}\alpha_1\texttt{)},\ldots,\texttt{not(}\alpha_n\texttt{)}]\texttt{)}|_n$$

The negation depth of `or()` is defined as

a multiset of the negation depths of its subterms:

$$|\texttt{not(and(}[\alpha_1,\ldots,\alpha_n]\texttt{))}|_n \quad \gg \quad |\texttt{not(}\alpha_1\texttt{)}|_n \cup \ldots \cup |\texttt{not(}\alpha_n\texttt{)}|_n$$

The negation depth of a negated term is defined as

the depth of the negated term:

$$\{|\texttt{not(and(}[\alpha_1,\ldots,\alpha_n]\texttt{))}|_d\} \quad \gg \quad \{|\texttt{not(}\alpha_1\texttt{)}|_d \ldots |\texttt{not(}\alpha_n\texttt{)}|_d\}$$

which is true by definition of $||_d$. □

Lemma 6.2.13. *The measure of the conclusion is greater than the measure of each of the premises for the negated* `or()` *rule (Equation 6.8).*

Proof. To show that the measure of the negated `or()` rule's conclusion is greater than its premise:

$$|\texttt{not}(\texttt{or}([\alpha_1,\ldots,\alpha_n]))| \quad \rhd \quad |\texttt{and}([\texttt{not}(\alpha_1),\ldots,\texttt{not}(\alpha_n)])|$$

Since the depths of each are equal:

$$|\texttt{not}(\texttt{or}([\alpha_1,\ldots,\alpha_n]))|_d \quad = \quad |\texttt{and}([\texttt{not}(\alpha_1),\ldots,\texttt{not}(\alpha_n)])|_d$$

it must be shown that the negation depth is greater:

$$|\texttt{not}(\texttt{or}([\alpha_1,\ldots,\alpha_n]))|_n \quad \gg \quad |\texttt{and}([\texttt{not}(\alpha_1),\ldots,\texttt{not}(\alpha_n)])|_n$$

The negation depth of `and()` is defined as

a multiset of the negation depths of its subterms:

$$|\texttt{not}(\texttt{or}([\alpha_1,\ldots,\alpha_n]))|_n \quad \gg \quad |\texttt{not}(\alpha_1)|_n \cup \ldots \cup |\texttt{not}(\alpha_n)|_n$$

The negation depth of a negated term is defined as

the depth of the negated term:

$$\{|\texttt{not}(\texttt{or}([\alpha_1,\ldots,\alpha_n]))|_d\} \quad \gg \quad \{|\texttt{not}(\alpha_1)|_d \ldots |\texttt{not}(\alpha_n)|_d\}$$

which is true by definition of $||_d$. □

Lemma 6.2.14. *The measure of the conclusion is greater than the measure of each of the premises for the negated* `forAll()` *rule (Equation 6.9).*

Proof. To show that the measure of the negated `forAll()` rule's conclusion is greater than its premise:

$$|\texttt{not}(\texttt{forAll}(\mathcal{S},x.\alpha(x)))| \quad \rhd \quad |\texttt{exists}(\mathcal{S},x.\texttt{not}(\alpha(x)))|$$

Since the depths of each are equal:

$$|\texttt{not}(\texttt{forAll}(\mathcal{S},x.\alpha(x)))|_d \quad = \quad |\texttt{exists}(\mathcal{S},x.\texttt{not}(\alpha(x)))|_d$$

it must be shown that the negation depth is greater:

$$|\texttt{not}(\texttt{forAll}(\mathcal{S},x.\alpha(x)))|_n \quad \gg \quad |\texttt{exists}(\mathcal{S},x.\texttt{not}(\alpha(x)))|_n$$

The negation depth of `exists()` is defined as

a multiset of the negation depths of its subterms:

$$|\texttt{not}(\texttt{forAll}(\mathcal{S},x.\alpha(x)))|_n \quad \gg \quad |\texttt{not}(\alpha(c_1))|_n \cup \ldots \cup |\texttt{not}(\alpha(c_n))|_n$$

The negation depth of a negated term is defined as

the depth of the negated term:

$$\{|\texttt{not}(\texttt{forAll}(\mathcal{S},x.\alpha(x)))|_d\} \quad \gg \quad \{|\texttt{not}(\alpha(c_1))|_d \ldots |\texttt{not}(\alpha(c_n))|_d\}$$

which is true by definition of $||_d$, given that $|x| = |x_i|$ for $1 \leq i \leq n$ (as a variable and a constant have the same depth) and $\mathcal{S} \rightsquigarrow \{c_1,\ldots,c_n\}$. □

Lemma 6.2.15. *The measure of the conclusion is greater than the measure of each of the premises for the negated* `exists()` *rule (Equation 6.10).*

Proof. To show that the measure of the negated `exists()` rule's conclusion is greater than its premise:

$$|\texttt{not}\,(\texttt{exists}\,(\mathcal{S},x.\alpha(x)))| \quad \rhd \quad |\texttt{forAll}\,(\mathcal{S},x.\texttt{not}\,(\alpha(x)))|$$

Since the depths of each are equal:

$$|\texttt{not}\,(\texttt{exists}\,(\mathcal{S},x.\alpha(x)))|_d \quad = \quad |\texttt{forAll}\,(\mathcal{S},x.\texttt{not}\,(\alpha(x)))|_d$$

it must be shown that the negation depth is greater:

$$|\texttt{not}\,(\texttt{exists}\,(\mathcal{S},x.\alpha(x)))|_n \quad \gg \quad |\texttt{forAll}\,(\mathcal{S},x.\texttt{not}\,(\alpha(x)))|_n$$

The negation depth of `forAll()` is defined as

a multiset of the negation depths of its subterms:

$$|\texttt{not}\,(\texttt{exists}\,(\mathcal{S},x.\alpha(x)))|_n \quad \gg \quad |\texttt{not}\,(\alpha(c_1))|_n \cup \ldots \cup |\texttt{not}\,(\alpha(c_n))|_d$$

The negation depth of a negated term is defined as

the depth of the negated term:

$$\{|\texttt{not}\,(\texttt{exists}\,(\mathcal{S},x.\alpha(x)))|_d\} \quad \gg \quad \{|\texttt{not}\,(\alpha(c_1))|_d \ldots |\texttt{not}\,(\alpha(c_n))|_d\}$$

which is true by definition of $||_d$, given that $|x| = |x_i|$ for $1 \leq i \leq n$ (as a variable and a constant have the same depth) and $\mathcal{S} \rightsquigarrow \{c_1, \ldots, c_n\}$. □

These lemmas allow us to state:

Theorem 6.2.16. *The proof process will terminate.*

Proof.

- From Lemma 6.2.2, the first phase will terminate.
- From Lemma 6.2.3, the branching of the proof tree will be finite.
- From Lemma 6.2.7 — Lemma 6.2.15, the second phase will terminate.

Therefore, the proof process has a finite amount of work to do and will terminate. □

6.2.8 Complexity

The proof engine works by performing a depth-first search through the proof goal space. As such, the order of rules defined in the patterns (and the order in which the patterns are loaded in through '`init.sp`') is important. Furthermore, if the pattern is verified, then the proof process will terminate without scanning all the possible rules through backtracking; however, if the pattern cannot be verified then all rules will be exhausted prior to reporting failure.

At any one time, the proof process is only looking to determine if a set of classes meets a specified pattern. Thus the number of different patterns in the library has no direct effect on the size of the proof tree for verification purposes. However, the number of variants that a pattern

has will affect the proof process; the more variants a pattern has, the more possibilities must be tried by the proof engine to demonstrate a pattern.

These pattern variants are often combined into a super-pattern (e.g. the pattern variants **LazySingleton** and **PublicSingleton** are combined into **Singleton**). The variants are normally independent of each other in terms of their proofs; and in any case, there is a partial ordering on the patterns and their variants to prevent cyclic definition of patterns. As a result, the proof space is proportional to the number of variants that a pattern has.

The proof space is related to the number of branches that are encountered during the proof process. The branches may be a result of explicit `and()` and `or()` nodes from the SPINE rules, from implicit branches caused by backtracking, and finally the branches created as a result of the `forAll()` and `exists()` expansions. The explicit `and()` and `or()` nodes are relatively easy to determine from the SPINE rules themselves; the number of nested `and()` and `or()` nodes will be a measure of how complex the proof is from a static view.

Note that SPINE rules are defined as an implicit `and()` of propositions, and that backtracking will result in an implicit `or()` of the rules.

The number of branches that may occur due to explicit/implicit `and()` and `or()` can be statically determined from the SPINE rules. As noted above, the proof process only attempts to prove a set of classes meets a single pattern, and thus only a subset of the rules (those involving that pattern, its variants, and any nested `realises()` rules) will need to be considered.

As noted in Lemma 6.2.2, the rewriting of the proof goal by expansion of the `realises()` rules will result in a finite proposition. Moreover, at this point, the branching due to explicit `and()` and `or()` will have already been taken into account. Thus the complexity of any given proof goal can be calculated since it is proportional to its depth and branching content.

What is more interesting is the branching caused by the `forAll()` and `exists()` operators. These generate one branch per set member from the evaluation of evaluable sets such as such as `constructorsOf()` and `subclassesOf()`. Although these are finite (because the Java code base must be finite), there may be many tens or hundereds of members returned (particularly in an expression such as '`subclassesOf('java.lang.Object')`'). Given that several of these may be nested, there may be a potentially worse branching explosion because of these than explicit or implicit branches introduced with `and()` and `or()`, which causes the proof space to be polynomial in size of the Java source code being processed.

The worst two cases for quantifier set nesting occurs with the definition of the **Abstract-Factory** and **Bridge** patterns, which have the following respective set nestings:

```
subclassesOf() x subclassesOf() x methodsOf() x methodsOf()
```

```
subclassesOf() x subclassesOf() x methodsOf() x methodsOf() x fieldsOf()
```

Thus, the proof space in relation to the size of the Java code analysed (and therefore the time taken to search it) will be related to the following polynomial:

$$v \times c^2 \times m^2 \times f \tag{6.12}$$

where v is the number of variants, c is the number of classes, m is the number of methods and f is the number of fields.

Fortunately, the search space often works out much better than this. For example, there is only one variant for each of the **Bridge** and **AbstractFactory** patterns, and the number of subclasses are restricted to a specific parent class (in other words, not *`java.lang.Object`*). Additionally, classes in Java tend not to have many methods or fields; 100 fields and methods would certainly be a complex class indeed, and an **AbstractFactory** that produced product classes with that many fields and methods would be equally unmanageable for a human to process as a proof engine. Most of the patterns have 3 or less nested set operations, and some of these are ranged over by `exists()` rather than `forAll()`, so if a suitable element is found the proof does not need to continue.

The proof's memory requirements are that it should be sufficient enough to hold the state of the proof in as much detail as necessary to complete the proof. Given that the proof process is a depth-first search, the minimum amount of memory required is proportional to the maximum state of the proof tree during evaluation. However, the proof system also needs to retain information about successful and failed sub-proofs in order to generate error messages, so since each of the nodes visited is stored in memory until the proof is complete, the memory complexity is the same as the time complexity.

6.3 Built-in functions and predicates

A number of SPINE built-in functions and predicates exist which deal with the data structures representing the Java classes. These are evaluated by direct appeal to the data structure. For example, the function '`methodsOf('java.lang.Object')`' results in a list of methods defined in the '`java.lang.Object`' class. Each of these definitions contains the list of statements that are defined in the body of the method itself, which in turn contains a number of expressions.

One such built-in allows HEDGEHOG to determine if a specified field of a class can be modified, which is used to ensure that a class is immutable. This works by appeal to the method

implementations in a class; provided that the field is `private` (so that it can't be accessed from outside that class), the only way it is modified is if methods can access the variable. The built-in then walks through the method implementations searching for an assignment operator that refers to the given field on its left hand side, and if no such assignments exist, concludes that the field may not be changed. Constructors are not checked; invariably, the constructor is used to initialise the instance variables of an immutable object, and once constructed, the values cannot be changed.

A full list of the built-ins are described in Appendix C.

6.4 Summary

The HEDGEHOG proof engine provides a mechanism for manipulating proof trees. Each proof node in the tree has an associated goal, and the state of the proof node is one of **none**, **partial**, **complete** and **failed**. The first two are the nonterminal states of the proof node, and the last two are the terminal states of the proof node.

The root proof node is used to prove the top-level goal. A set of applicable rules, from a library of rulesets, are attached to the node. The application of a rule to a proof node results in either a change of proof state, or generation of a set of sub-goals (or both). For each sub-goal, a new sub-node is created as a child of the original proof node.

This process then iterates over the remaining proof structure, using a depth-first procedure to prove the remaining nodes. Only sub-nodes of nonterminal proof nodes are considered, since these are the the ones that make a difference. However, if a node is marked as failed, and there are other applicable rules, then these may be tried instead to generate a different set of sub-goals.

The success (or failure) of the entire proof tree is then the success of the top-level node after this iterative process is complete. Furthermore, each complete (successfully proved) node will have the associated rule which can be displayed as part of the proof output.

The result of the proof tree is then announced to the user. If HEDGEHOG is unable to prove the conjecture, or falsifies the conjecture, then error messages can be generated to indicate the problem and give directed help towards the solution. The generation of messages is covered in Chapter 7.

Chapter 7

Generating error messages

The main purpose of any automated proof system is to generate a result: whether the proof succeeded or not. However, although the message 'success' is sufficiently detailed for most purposes, a result of 'failure' does not give enough information to the user about what the cause of the problem is.

Users familiar with proof systems may be used to seeing proof trees and interpreting their failures. However, since HEDGEHOG is designed for Java developers, it is unlikely that they will be able to understand a failed proof tree. Instead, the failure must be converted into a succinct textual message that the developer can understand.

The failure of a proof is not necessarily the end of usefulness for an automated theorem prover. If the proof has failed because it is not able to connect two proof graphs, or only a small problem stops the proof mechanism from continuing, it may be possible to use that information to fix the problem, or to make the requirements stricter such that automated proof can continue. These techniques are used in current automated proof systems to try and improve the quality of the proof. For example, rippling [BSvH$^+$93] is a technique for controlling the key parts of proof by induction, used to suggest possible approaches in failed proofs [IB96].

However, not every problem can be automatically fixed. In most cases, the end-user will have to be involved to guide the automated proof system to allow it to successfully finish the proof. This is an active research area [LCSV96, LD97, BMZ00], based on increasing the quality of information given to the end-user [Moo93]. This is also used in other automated program analysis using proof systems [EI04] to increase the ways in which proof systems may be further automated in the case of proof failure.

HEDGEHOG's proof tree consists of simple connectives (such as '`and`' and '`or`'), existential quantifiers ('`forAll`' and '`exists`') and Java-related predicates ('`extends`' and '`implements`').

HEDGEHOG unpacks the logical connectives and discharges the Java-specific predicates using the abstract representation of the source code.

Generation of the error message consists of the following steps:

1. Filtering the tree to leave only failed nodes
2. Converting the contents of the tree into a suitable error message

7.1 Tree filtering

At the end of an automated proof, the tree will consist of a combination of success and failure nodes. In order to generate a message about the causes of the error(s), information about the errors needs to be considered. However, given that the proof tree contains both success and failure information, it can be condensed to just provide information about the problems that caused an error.

A node is *successful* if the proof node succeeded. A branch of the proof tree is said to be successful if the node is successful. Note that a successful node may contain unsuccessful children; for example the node `and([true,false])` will have one successful child node and one unsuccessful child node.

If the proof has failed, then the root node of the proof tree will be marked as failed. Clearly, this node will have at least one failed child (or cause) and possibly some success nodes as well.

The successful nodes are filtered from the top level so that only the unsuccessful nodes remain. Note that the nodes are not deleted from the proof tree; they are merely marked as hidden and play no further part in the generation of error messages.

This process then recurses down the proof tree. At the end of this process, the proof tree is reduced to a subtree of failed nodes.

7.2 Converting trees to error messages

Once the tree has been reduced to only those nodes representing failures, it is then possible to convert the resultant proof fragment into an error message. However, the trick lies in generating an error message that it is understandable, and does not have too much *noise*; that is, the message displays information that is directly relevant to the failure and not too much information from successful nodes.

7.2.1 Displaying a complete reason

A failed proof tree can be converted into an error message simply by recursing over the proof tree, and translating the logical connectives into suitable English. For example, consider the proof statement:

```
and([true,or([false,false]))
```

This could be translated literally as " 'and()' failed because 'or()' failed because both 'false' and 'false' failed."

Though accurate, this would not necessarily provide the end user with a clear idea of why the proof failed. It would be more useful if the direct connections were eliminated; in this case, removing the 'and()' and 'or()' nodes from the resultant message. A clearer message would then be "proof failed because both 'false' and 'false' failed."

7.2.2 Compressing the message

Each node in the resulting proof tree is evaluated to show how *interesting* that node is. The more interesting a node, the greater its chance of being displayed in the error message. Nodes that are not interesting (i.e. they have an interest value lower than a defined *interest threshold*) may not be displayed in the body of the error message. The threshold is defined so that the majority of successful proof nodes are filtered out, whilst those with clear failures are displayed.

For example, consider a failure proof tree that has five nodes; `A` — `E`. In this example, `A` has as its only (failed) child `B`, which has as its only failed child `C` and so on.

Now consider an example situation where a post-process of this tree decides that nodes '`A`' and '`D`' are interesting, but that '`B`', '`C`' and '`E`' are not. Instead of generating the message " '`A`' failed because '`B`' failed because '`C`' failed because '`D`' failed because '`E`' failed", it can generate the shorter " '`A`' failed because '`D`' failed."

7.3 Interesting errors

In order to compress the verbose error message, it is necessary to decide on those nodes which are interesting, and those which are not.

Nodes involving simple logical connectives (such as 'and' and 'or') can be pruned from the search tree. Clearly, if a requirement was required for both '`and([A,B])`' and '`B`' failed, then there is little point in introducing the '`and`' clause into the error message. In general, for any

logical connective, a node is only interesting if two (or more) sub-nodes failed. Thus if both '`A`' and '`B`' failed, it would be worth pointing out both of these failures. Similarly, in the case of an '`or`' failing (because none of the child nodes succeed) then it would be worth displaying that message as well.

At each stage of the message, it is necessary to determine whether the goal that was trying to be proved is `true` or `false`. In the case of negation, if the statement '`not(A)`' has failed, it is because '`A`' has been proven when it should not have done. This will be necessary information to convey to the user when the message is displayed.

In the case of the quantifiers '`forAll`' and '`exists`', a failure may be because one of the child nodes failed (or did not succeed). In general, these nodes are marked as interesting because it is usually the case that only one (or a small number) of child nodes failed, and these failures can be important to note.

The last case of errors are associated with the Java-specific primitives, such as the existence (or lack of) of a particular signature, or method implementation. These will always be reported as interesting errors, since these are usually the causes that developers want to know.

7.3.1 Chains with uninteresting beginnings or ends

When condensing a set of proof nodes, there may be chains which have uninteresting nodes at the beginning of the chain, or uninteresting nodes at the end of the chain. In these cases, is it desirable to prune the start or ending nodes, or merely the uninteresting nodes in the middle of a chain?

In some cases, it may be desirable to display the start of the chain so that the user can place the resulting failure in context. There is an argument, therefore, that the start of the chain should be displayed whether or not it is marked as 'interesting' by the decision procedure (see below).

One way of enforcing this is to modify the condensing procedure so that it does not prune 'uninteresting' nodes at the beginning or end of chains. However, this may not be desirable; for example, a proof failure may have a significantly large chain of 'uninteresting' nodes initially, followed by a single 'interesting' node towards the end of the chain.

A better way of modifying the condensation routine is to modify how the nodes are calculated as 'interesting' in the first place. A (two-pass) run of the chain can find the nodes of greatest importance, and then (for example) mark all nodes from the start of the chain to the first 'interesting' node as pseudo-'interesting'. This would then allow the condensation routine to be simpler, but with the same level of flexibility.

Hence, uninteresting nodes at the start and end of the chain can be treated in exactly the same way (and pruned) as uninteresting nodes in the middle of the chain.

7.3.2 Tree nodes with interesting children

Another problem occurs when branching nodes (or tree nodes) are marked as uninteresting, but directly or indirectly have interesting children.

There are two possibilities for processing these nodes:

- Leave the branch node in the resulting process, by marking it as 'interesting'
- Create a new node that contains the 'interesting' children

Of the two approaches, the latter is likely to result in smaller trees than the former. The reason for this is that an uninteresting branch node may have uninteresting branches beneath it, and that these nodes may be merged without affecting the cause-and-effect display of the output.

7.4 Pattern annotation

As well as providing error information from the proof tree, other explanations of why a feature is disallowed (or required) can be provided.

For example, in the case of the **Singleton** pattern, it is necessary that all constructors are `private`, and that there is at least one constructor. The first requirement ensures that the class cannot be instantiated outside of the class definition, whilst the second requirement ensures that there is a provided constructor. The second is necessary, because if there are no provided constructors (in the source file) the compiler will automatically generate a default (no argument) constructor which is public.

If an error message in the compiler suggested that there was no provided constructor, it may not be immediately obvious why a constructor is necessary. Furthermore, if the developer were to create an additional constructor, it would probably be created `public` (as most constructors are). Then, when the test was rerun, a second error indicating that the constructor should have been `private` would be generated.

It is therefore desirable to give the user some kind of indication of why the error has been raised in the first place, and what the solution to the problem is.

To do this, suitable descriptions are encoded within the pattern specification using ordinary comments. This allows the proof system to present a message to the developer, which can then be used to help the developer fix the problem specifically for the pattern. For example:

```
realises('PrivateSingleton',[C]) :-
  (* Must have at least one constructor *)
  exists(constructorsOf(C),Cn.true),
  (* All constructors must be private *)
  forAll(constructorsOf(C),Cn.hasModifier(Cn,private)),
  (* Must be a public static final variable *)
...
```

These comments can then not only be indications to the developer who is reading the pattern definition, but can also be applied to the proof tree. If a problem occurs whilst trying to prove the last '`exists`' statement, then the comment above can be displayed to guide the user in terms of correcting the statement.

Note that these messages are in addition to, rather than replacement of, the messages that are generated from the proof tree. For example, if there was a field which had the correct type, was instantiated and had both '`static`' and '`public`' modifiers, then the generated error message would indicate that the the '`final`' method was missing.

7.5 From nodes to explanations

Once the proof tree has been sufficiently filtered, it is then desirable to convert it into some kind of textual explanation as to why the proof failed.

There are two ways in which nodes are converted into textual answers: by translating the proof node type into text, and to associate specific error messages with each node. Both are useful in translating error messages to end users.

Furthermore, for errors associated with particular methods, fields, or code in the class, it may be possible to provide more specific file and line numbering information.

Some node types – such as '`and`', '`or`', '`forAll`', and '`exists`' – are translated directly into English equivalents. For example, if part of the proof tree has a constraint that all constructors are `private`, and one part of the proof tree fails, then it is possible to generate an error message like "All methods should be private; however, constructor X is not". Others, to do with the syntax of the Java language (for example, expecting a relationship between two classes to exist) are also relatively easy to translate into a textual output.

There are some other node types which may prove more difficult to translate directly. For example, if there is a complex requirement that uses a set of constraints, then it may be counter-productive to list individual failures instead of a single reason.

Whilst printing out a simple text string for a single reason works quite well, when multiple problems occur the combination of text messages and branches becomes more of a problem. For example, when attempting to prove that a class is a **Singleton**, it is required that it realises one of three variants. If one of these variants fails because of a minor difference, and the other variants fail because of major differences, then clearly the desired message is to highlight the 'close' implementation and ignore the others. However, it is not easy to merge the result of failed branches.

It is also difficult to represent a tree in a single text explanation, without using bracketing or nested lists. As the complexity and depth of the tree increases, such bracketing and nested list schemes can obscure the key points of the proof.

7.6 Example

To see how this process works in practice, let us consider the **Singleton** pattern (specifically, the **PrivateSingleton** variant), shown in Figure 7.1. In this pattern, only a single instance of an object is allowed at one time. This can be achieved by observing the following constraints:

- All of the constructors must be `private`
- There must be at least one constructor
- There must be a field `F`
  - Which is `static final` and of type *C*
  - Which is assigned a new `C` during construction
- There must be a `public` method that returns type *C*
  - Which returns the value of *F*

In the SPINE definition, comments may be inserted to describe the purpose of the specification, including using expressions to display the names of bound variables at that point in the proof tree. For example, if the constraint `nonNull(F)` fails, then the proof system can output the result "Field *F* must be instantiated during class construction".

Figure 7.1: **PrivateSingleton** SPINE definition

```
realises('PrivateSingleton',[C]) :-
  and([
     (* All constructors must be private *)
    forAll(constructorsOf(C),Cn.
      hasModifier(Cn,private)),
    (* There must be at least one constructor *)
    exists(constructorsOf(C),Cn.true),
    exists(fieldsOf(C),F.
      and([
        typeOf(F,C),
        hasModifier(F,private),
        hasModifier(F,static),
        (* Field $F must be instantiated *)
        nonNull(F),
        exists(methodsOf(C),M.
          and([
            hasModifier(M,public),
            hasModifier(M,static),
            typeOf(M,C),
            returns(M,F)
          ])
        )
      ])
    )
  ])
```

In the specification, *hasModifier* and *typeOf* are examples of static constraints, whilst *instantiates* and *returns* are examples of semantic constraints.

Consider the following class:

```
public class Example {
  private Example field;
  public Example getInstance() {
    return null;
  }
}
```

This does not meet our singleton specification in a number of different ways. Consider what happens when we try to verify the class, using the **PrivateSingleton** variant:

```
realises('Singleton',['Example']) :-
realises('PrivateSingleton',['Example']) :-
  and:
forAll(constructorsOf('Example'),Cn.true)
  -> forAll([],Cn.true)
  -> true
  exists(constructorsOf('Example'),Cn.true)
  -> exists([],Cn.true)
  -> false
```

The proof tree for this failure is very simple, and can be condensed into:

realises(‘Singleton’,[‘Example’])
→ realises(‘PrivateSingleton’,[‘Example’])
→ and
→ exists(Cn,constructorsOf(‘Example’),true).

In this particular case, most of the nodes are uninteresting. (The first and last nodes are the only ones of real interest in the sequence.) The proof tree can be filtered into a shorter failure:

realises(‘Singleton’,[‘Example’])
→ exists(Cn,constructorsOf(‘Example’),true).

This can be used to generate an error message “The class ‘Example’ does not realise the pattern ‘Singleton’ because: there must be at least one constructor”. These messages can be generated directly from the rules that are used to specify the pattern.

7.7 Summary

In the case of proof failure, HEDGEHOG's ability to explain the reason for the failure is an important part of HEDGEHOG's use. Without a mechanism for displaying the result to the end user, a message of proof failure would not give the end users an indication of why it has failed without an analysis of the code.

Additionally, the proof tree is processed to filter out both successful and uninteresting nodes. This means that the result displayed provides information that a Java developer could work with in order to locate and fix the problems.

Chapter 8

Worked examples

This chapter shows how HEDGEHOG attempts to prove that a set of classes realises a particular design pattern.

8.1 Startup

When HEDGEHOG starts up, it loads the built-in SPINE definitions from the pattern library. The rules are parsed and cached in memory, from which they can be referenced in the proofs.

The SPINE initialisation file, `init.sp` (shown in Figure 8.1), defines which other SPINE files to load. Each of these files contains rule definitions, which in turn, can load further rules.

Figure 8.1: SPINE initialisation file

```
(* init.sp *)
load('Singleton.sp').
load('Bridge.sp').
load('Factory.sp').
...  (* other pattern definitions loaded *)
```

The initialisation file above uses the special evaluable proposition '`load`' which causes a further SPINE file to be loaded. In this case, definitions for the '`singleton`', '`bridge`' and '`factory`' patterns are loaded in turn. The order of rules is significant: when a rule is matched against a term, the rules are tried in order until the rule is matched. During backtracking, subsequent rules are tried until there are no more possibilities, in which case the proof fails. Each of the individual SPINE files contains one (or more) rule definitions. Patterns are defined

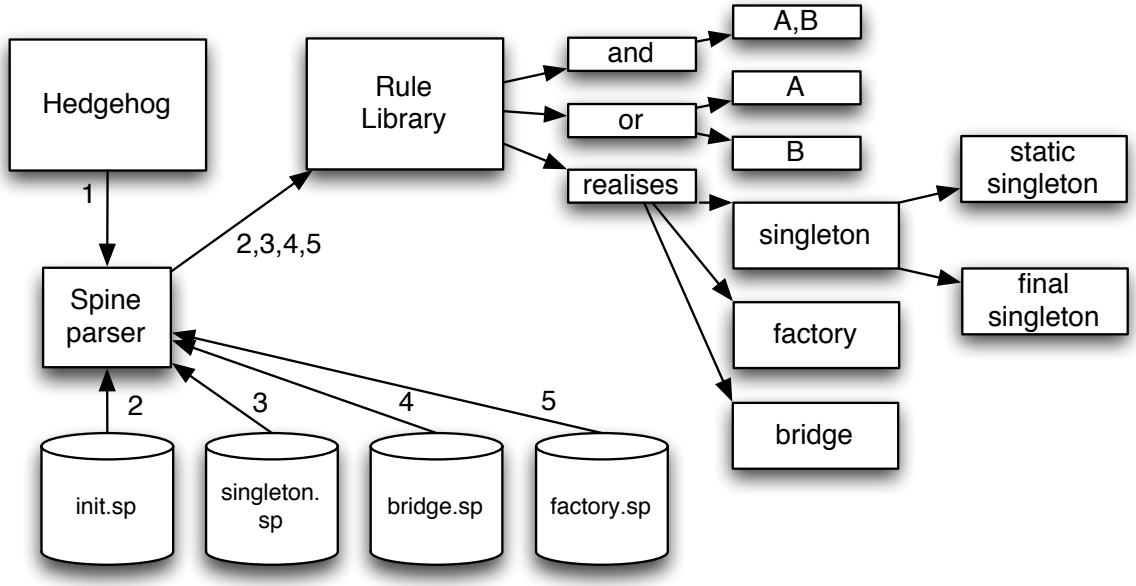
using the ‘realises’ term; for patterns with multiple variants, there may be multiple rules that realise the same pattern. (If so, the rules are combined at parse time to form one rule with a disjunction of all of the rule bodies.) The **Singleton** pattern, for example, can be realised in several ways; either using a `static final` variable, or using a lazy instantiation accessor. The definition of the **Singleton** pattern is given in Figure 8.2.

Figure 8.2: **PublicSingleton** SPINE definition

```
(* Singleton.sp *)
realises('Singleton',[C]) :- or([
  realises('PublicSingleton',[C]),
  realises('PrivateSingleton',[C]),
  realises('LazySingleton',[C])
]).

(* Public variant Singleton *)
realises('PublicSingleton',[C]) :-
  (* There must be at least one (private) constructor *)
  exists(constructorsOf(C),Cn.true),
  (* All constructors must be private *)
  forAll(constructorsOf(C),Cn.hasModifier(Cn,'private')),
  exists(fieldsOf(C),F.
    typeOf(F,C),
    hasModifier(F,'static'),
    hasModifier(F,'public'),
    hasModifier(F,'final'),
    nonNull(F)
  ).

(* Private variant Singleton *)
realises('PrivateSingleon',[C])
...

(* Lazy variant Singleton *)
realises('LazySingleton',[C])
...
```

The rules are kept in memory, and are called upon during inference. Figure 8.3 shows HEDGEHOG during the startup procedure and how the rules are inserted into the rule library.

Figure 8.3: HEDGEHOG during initialisation



8.2 Simple proofs

Once HEDGEHOG has finished loading, it is ready to accept requests for proofs. It is envisaged that the proof process will be integrated within an IDE or automated editor/testing environment; however, at present an interactive text interface allows a user to work with the HEDGEHOG engine.

The user enters a request into the system, whereupon it tries to automatically prove the result and output the conclusion. The SPINE parser is used to read the term in and construct a goal for that term.

The proof system then does the following:

- Generates a set of applicable rules for the term
- Applies the first rule to generate sub-goals
- Recurses through the proof tree until all sub-goals are proven

Figure 8.4: Setting up the initial proof goal

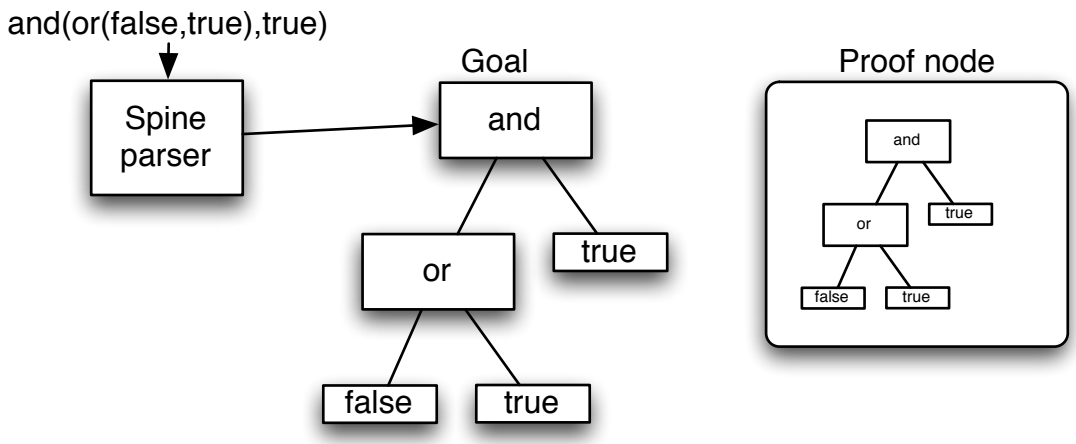


Figure 8.5: Applying the '`and`' rule

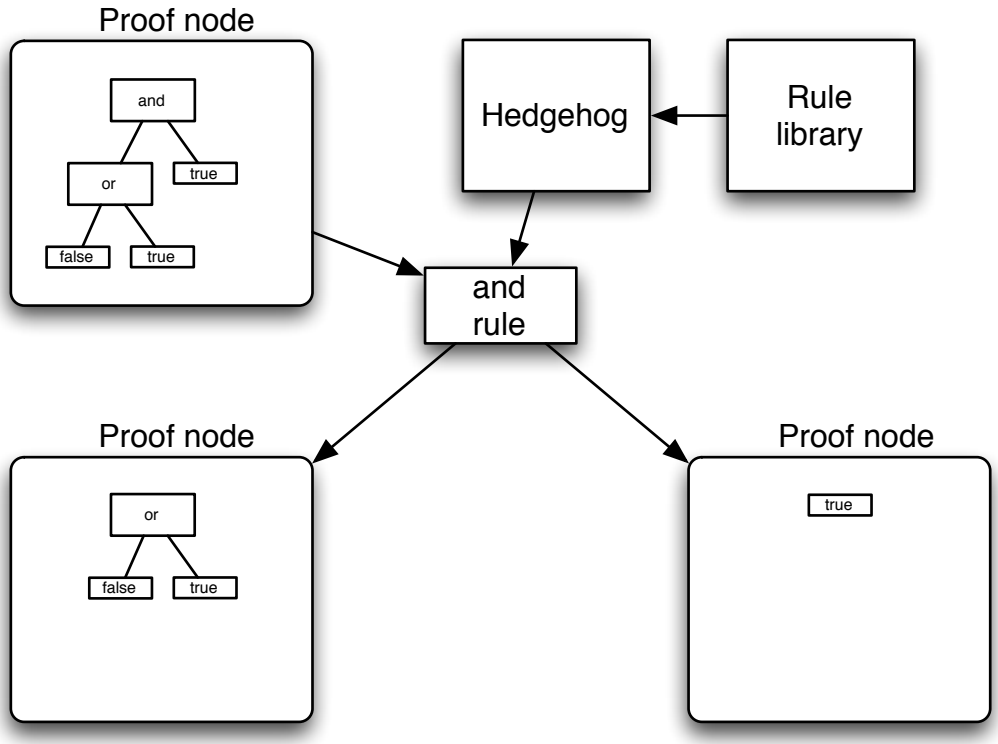
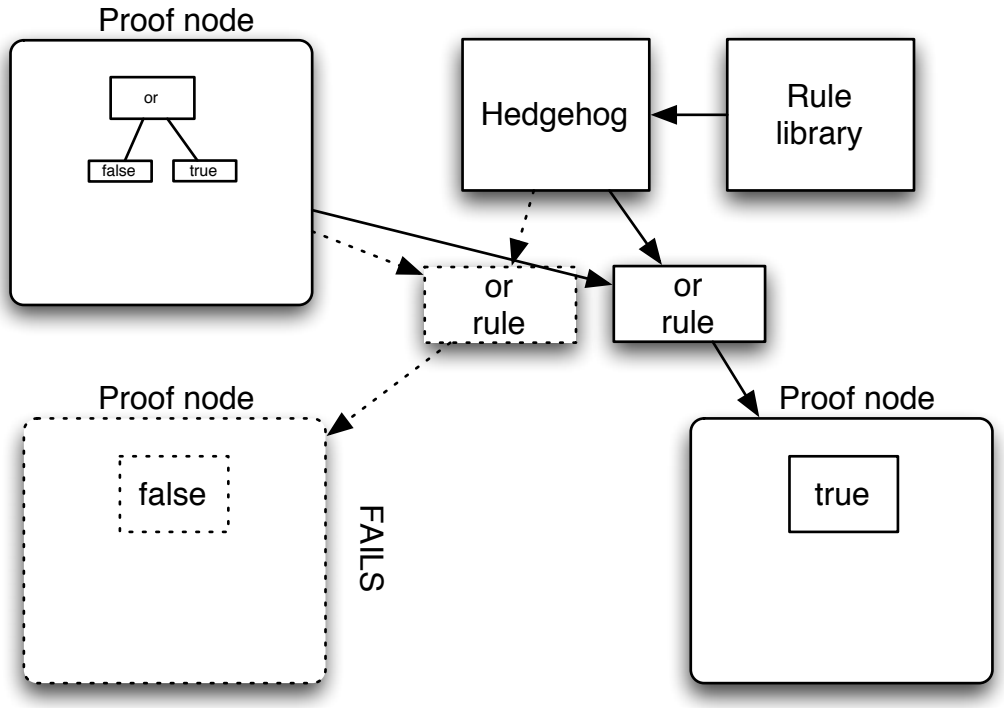


Figure 8.6: Applying the '`or`' rule, with backtracking



For example, if the user asks HEDGEHOG to prove '`and([or([false,true]),true])`' then the system would first generate a goal and wrap it in a proof node (Figure 8.4). It would then generate a list of the rules available for that proof node (which would only match the one '`and`' rule) and generate a pair of sub-goals (Figure 8.5).

The proof process then recurses down the tree; in the case of the '`true`' sub-goal, the (sub-) proof succeeds immediately. In the case of the '`or`' sub-goal, two rules are applicable. It first tries the '`false`' sub-goal, which fails; this causes a backtrack, and the second '`or`' rule is applied (Figure 8.6). Note that if the order of the '`or`' term had been reversed, then the proof would have succeeded without backtracking.

8.3 Proving a class realises a pattern

In the previous section, a simple proof was shown for a logical goal. This section presents an example of a simple pattern, **Singleton**, to demonstrate how HEDGEHOG works in conjunction with design patterns.

Each pattern has one or more '`realises`' rule associated; patterns with many variants can have a '`realises`' rule for each.

The '`realises`' rule takes a pattern name and a list of Java classes that may realise that pattern. In this first example, the pattern is associated with only one class; however, a later example will show multi-class patterns.

The goal '`realises('Singleton',['Test'])`' can be given to HEDGEHOG, which asks it to show that the class '`Test`' (shown in Figure 8.7) realises the **Singleton** pattern.

Figure 8.7: Implementation of the `Test` class

```
public class Test {
  public Test() {
  }
  private Test(int x) {
  }
  private static final instance = new Test();
}
```

When the HEDGEHOG system is given the goal '`realises('Singleton',['Test'])`' it rewrites the goal using the definition of '`realises`' from Figure 8.2 to produce the sub-goals:

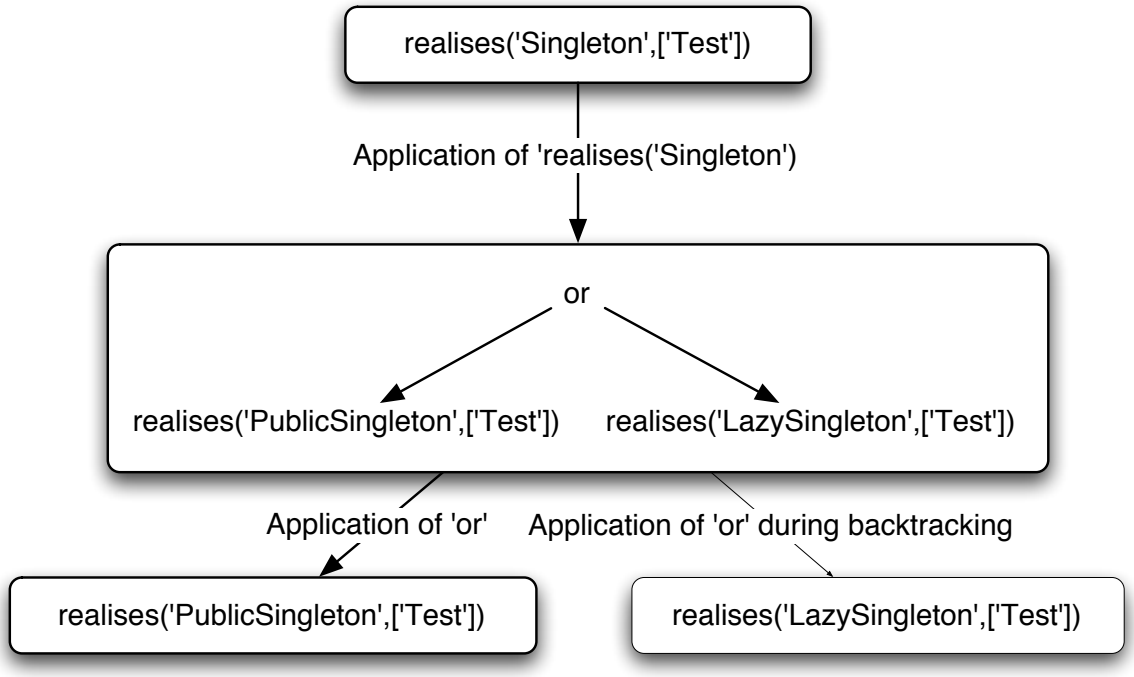
```
or([
  realises('PublicSingleton',['Test']),
  realises('LazySingleton',['Test']),
])
```

The automated proof procedure tries to prove these goals in order, so it starts by trying to prove:

```
realises('PublicSingleton',['Test'])
```

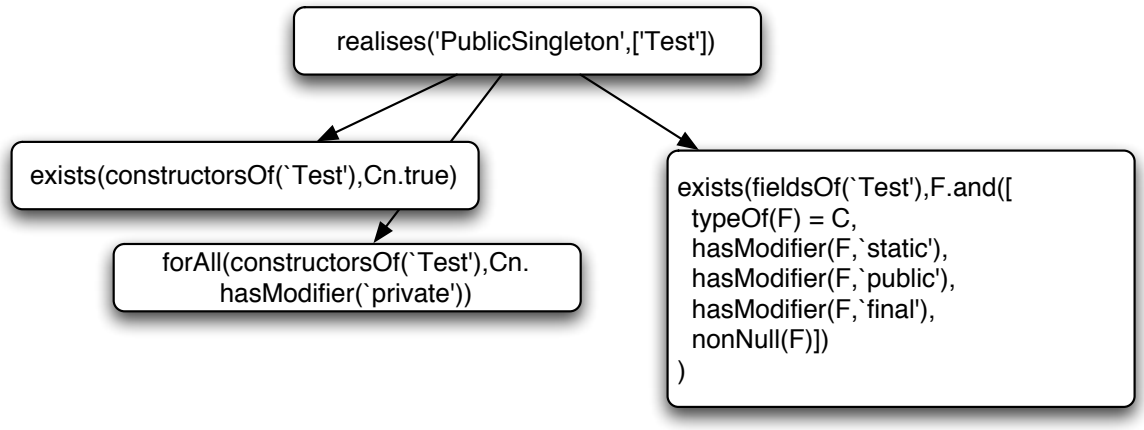
The proof tree now looks like the one in Figure 8.8. The '`or`' node is then broken down into two child proof nodes that contain each of the '`or`' goals).

Figure 8.8: Example of proof tree after initial pattern mapping



This, in turn, causes the following goals to be generated (as shown below and in Figure 8.9):

Figure 8.9: Example of proof tree with pattern variant proof nodes




```
exists(constructorsOf('Test'),Cn.true),
(* All constructors must be private *)
forAll(constructorsOf('Test'),Cn.
 hasModifier(Cn,'private')),
 exists(fieldsOf('Test'),F.and([
  typeOf(F,C),
  hasModifier(F,'static'),
  hasModifier(F,'public'),
  hasModifier(F,'final'),
  nonNull(F)
])).
```

8.3.1 Parsing the Java source

Java source files are kept in an in-memory cache and are dynamically loaded when requested. As such, the first expression to access the class (in this case, the '`constructorsOf('Test')`') causes the source file to be parsed and kept in memory as an AST. All references between classes are kept as textual references so they are not loaded until they are required.

Evaluation of the expression '`constructorsOf('Test')`' within the '`exists`' (a built-in HEDGEHOG operator) results in an access to the class-cache to find '`Test`'. If it is not already loaded, it is automatically loaded and parsed.

References to other classes inside '`Test`' are kept by fully qualified class name (for example, '`java.lang.String`'). That way, if a class is referenced but plays no part in the proof process, then the source is not loaded and processed.

The ‘constructorsOf(...)’ expression generates a list of Java constructors. All Java member types (constructors, methods, fields) are represented using the fully qualified class name, followed by the ‘#’ symbol, and then the method name, and lastly the argument types. For example, the method ‘valueOf(int)’ in the ‘java.lang.String’ class has a unique name in the cache of ‘java.lang.String#valueOf(int)’. Constructors are represented using the class name for the method, and fields are represented as field name alone. The resultant goal then looks like:

```
exists(['Test#Test()','Test#Test(int)'],Cn.true)
```

8.3.2 Applying quantifiers

Since the ‘exists’ quantifier is used to find whether there is a constructor that meets a particular condition, the goal that is generated is an ‘or’ block that instantiates ‘Cn’ to each of the members of the resultant list. In other words, the statement that is generated looks like:

```
or([true,true])
```

The sub-expression ‘true’ has been modified such that all occurrences of ‘Cn’ have been replaced with ‘Test#Test()’ or ‘Test#Test(int)’ as appropriate. Since ‘true’ does not have any occurrences of ‘Cn’, it is unchanged by this substitution. This is graphically shown in Figure 8.10:

The ‘forAll’ quantifier is the next goal on the list to be processed. The built-in function ‘constructorsOf(...)’ is evaluated to produce a list:

```
forAll(['Test#Test()','Test#Test(int)'],Cn.
  hasModifier(Cn,'private'))
```

The generated goal for the ‘forAll’ now uses an ‘and’ instead of an ‘or’ to generate the results:

```
and([hasModifier('Test#Test()','private'),
hasModifier('Test#Test(int)','private'])
```

In this case, the sub-expression ‘hasModifier('Test#Test()','private')’ fails (as shown in Figure 8.11) because the constructor ‘Test’ is marked as ‘public’ in the source code.

Figure 8.10: Showing expansion of ‘`exists`’

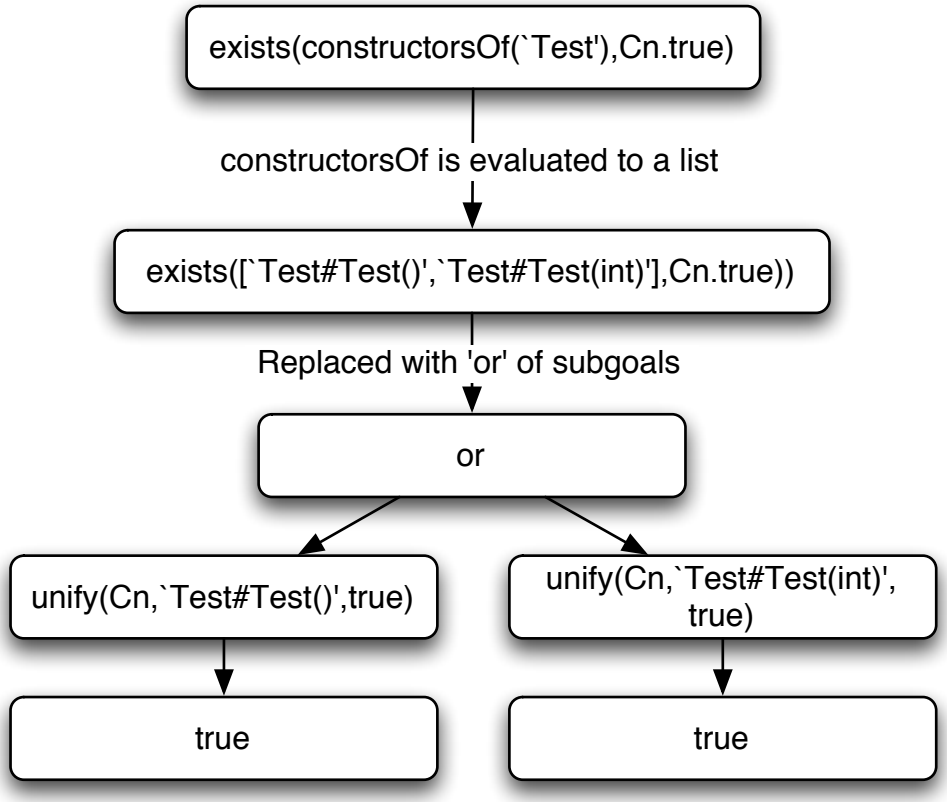
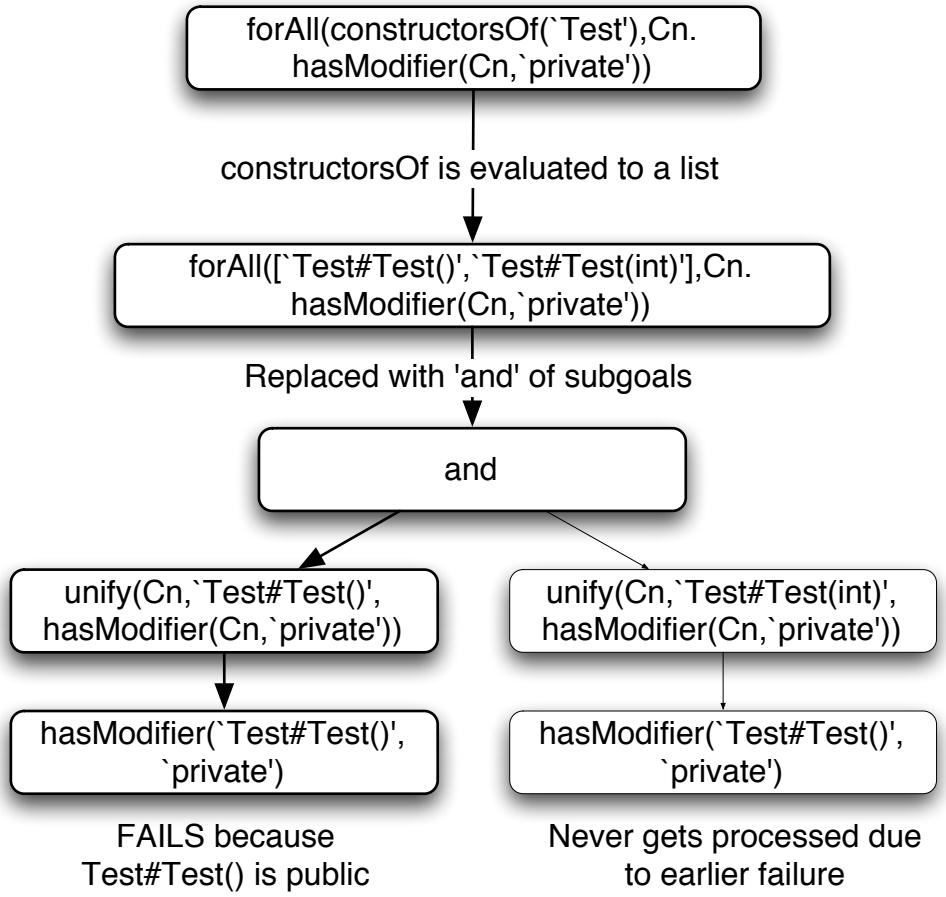


Figure 8.11: Failure in expansion of '`forAll`'



8.4 Multi-class patterns

Although some patterns are realised by only one class, some are realised by a set of classes together. For example, the **Abstract Factory** pattern consists of a 'factory' class and many 'product' classes. It is the relationship between these classes that defines the **Abstract Factory** pattern as well as the implementation of certain key features.

HEDGEHOG is capable of proving that a set of classes together realise the **Abstract Factory** pattern. In this case, instead of a single class being included in the '`realises`' goal, several classes are passed as a list; the definition of the factory pattern is shown in Figure B.1.

```
realises('AbstractFactory',['Factory','Product']).
```

Not all the classes in a pattern need to be provided individually in the '`realises`' modifier; for example, subclasses of the '`Product`' are not required (since HEDGEHOG can find a list of all the subclasses automatically).

The order of the classes and their roles depends on the pattern definition. In this case, the first class listed is the abstract *factory* class, and the second is the *product* class. In the case where a factory has multiple products that all share a common superclass, it is only necessary to provide that superclass.

The **Abstract Factory** pattern is defined in terms of relationships between classes. This works using several built-in SPINE functions and predicates (the full list of which is given in Appendix C) to determine whether or not these classes are part of the **Abstract Factory** pattern.

methodsOf is a list of methods defined in the given type

subclassesOf is a list of all subclasses of the given type

instantiates is true if the given method instantiates and returns the given type

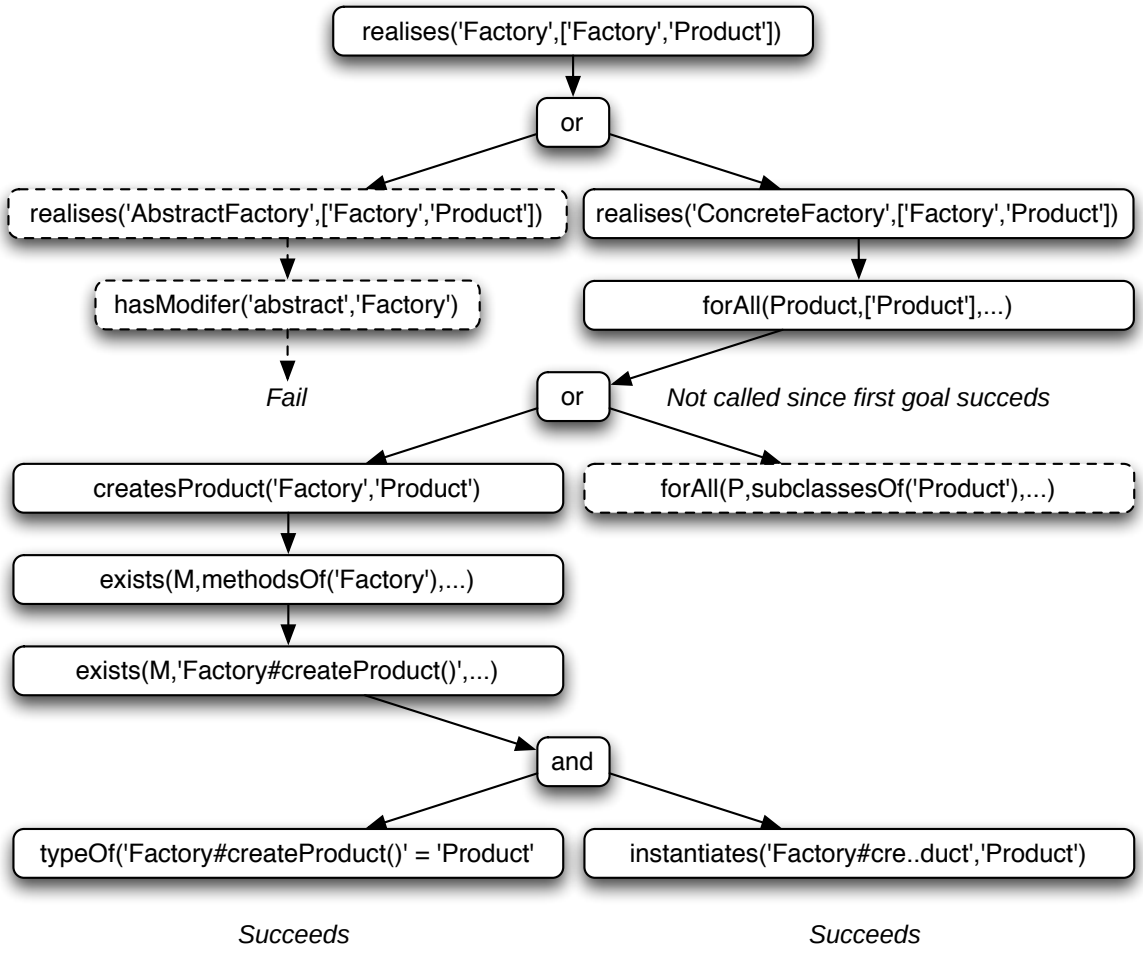
Figure 8.13 shows the final search tree of the factory example after the application of various rules in the process. In this case, the '`AbstractFactory`' is attempted first, but this fails due to the fact that `Factory` is not an `abstract` type. The proof process then tries to use the pattern definition for the '`ConcreteFactory`', which succeeds.

This demonstrates how HEDGEHOG is able to handle different variants of a pattern. By having multiple rules for defining a pattern (in this case, a concrete factory pattern as well as an abstract factory pattern) the backtracking will check out other variants of the same pattern.

Figure 8.12: Example factory and product

```
public class Factory {
  public Product createProduct() {
    return new Product();
  }
}
public class Product {
  // Implementation
}
```

Figure 8.13: **Factory** example



The success is due to the built-in SPINE predicate ‘`instantiates`’, which determines whether the method is implemented in such a way as to both instantiate and return an instance of the given type. This process is not shown in Figure 8.13, but works by running through the implementation of the method in question (in this case, the `createProduct()` method). It expands all of the possible call-paths (i.e. for each `if` statement, it creates two possible paths through the method) and then ensures that for each path, the type `Product` is instantiated, and then that value is returned.

This checking process does not work in the case of looping statements (`for`, `while` etc.) since it is not trivial to ensure that the loop executes a certain number of times. In these cases, the looping constructs are removed from the loop and are assumed to execute zero times, which is a conservative estimate of what may actually happen. In most cases, factory methods are not implemented using such loops (they do not occur within the Java libraries, for example) and so this restriction is not significant.

8.5 Dealing with failure

If a proof fails, the proof engine backtracks to the last available point. In Figure 8.13, the first attempt to show the factory was an ‘`AbstractFactory`’ failed and backtracking occurred by trying to show that it was a ‘`ConcreteFactory`’ instead. Similarly, during the proof process of the **Singleton**, the ‘`realises('PublicSingleton',['Test'])`’ fails, and so backtracking ensures that ‘`realises('LazySingleton',['Test'])`’ is attempted.

However, this will fail in a similar way due to the fact that the **LazySingleton** pattern also requires `private` constructors (as do other variants of the **Singleton** pattern). As the proof cannot be fulfilled, and the back-tracking process has run out of other ideas, the failure message is displayed.

The error message that is generated is condensed from the failure nodes of the proof tree. In this case, the error message compares the first level of the proof tree and concludes that the ‘`Test`’ class does not meet any variant of the ‘`Singleton`’ pattern. It would then detail the reason for this, which is that the constructor ‘`Test#Test()`’ should be ‘`private`’. It would also print out the comment associated with the failed reason; in this case “All constructors must be private,” which is extracted from the definition of the pattern itself. Error messages are discussed in more detail in Chapter 7.

8.6 Summary

This chapter presented a series of examples demonstrating how the HEDGEHOG system is used to verify that a class (or classes) realises a given design pattern. Although at present the user types in these requests manually, the further work section suggests that this tool be integrated with an automated testing environment or integrated developer environment to facilitate this procedure.

Chapter 9

Results

This chapter discusses the testing methodology used in determining the success of HEDGEHOG, and presents the results and analysis of the patterns and situations in which HEDGEHOG can be used.

The hypothesis, from Chapter 1, states:

> This thesis aims to prove the hypothesis that it is possible to represent patterns as as set of constraints on the implementation of one or more Java classes, such that it is possible to verify whether they realise a pattern correctly.

9.1 Critique of patterns as constraints

The first question that needs to be answered is: is it possible to represent a design pattern as a set of constraints on its implementation? In other words, do the pattern definitions match what would be expected in a design pattern, and do they avoid false matches?

A full list of the SPINE pattern definitions is given in Appendix B, mirroring those in [GHJV95]. One thing that is immediately noticeable is that not all patterns have a definition. Thus, it is the case that not every pattern can be represented as a series of constraints on their implementation. Whilst the hypothesis did not aim to show that every pattern can be formally defined, it is interesting to consider which patterns did not, and decide accordingly whether this is an important issue or not.

Firstly, unrepresentable patterns are discussed, explaining why they could not be represented in SPINE. Afterwards, the patterns that are defined in SPINE are covered individually, with a commentary on how close these pattern definitions are to the original pattern examples.

9.1.1 Unrepresentable patterns

There are two main types of pattern that are unrepresentable in SPINE:

- Those that are defined mostly by their intent
- Those which have a generic solution that admits too many false positives

Patterns that are mostly defined in terms of their intent are difficult to formally define, as the key feature is not how the pattern is implemented, but rather how it is used. This makes the pattern much more difficult to verify based on implementation[1] alone.

The **Interpreter** pattern is an example of such an intent-based pattern. In the **Interpreter** pattern, an abstract class hierarchy maps onto the grammar of a language. The **Interpreter** typically also uses aspects of the **Command** and **Visitor** patterns in order to provide a dual behaviour between language representation and language execution.

However, the key feature of this pattern is in its intent. The pattern is usually realised with one class per terminal in the language grammar; and choice operations are represented using an abstract class to group the choices together. The point at which the **Interpreter** pattern evolves from a grammatical structure of the the language is not well defined.

In essence, this is a similar problem to that of the **Command** pattern, which has an abstract super-type with an abstract method for execution (sometimes with a context). Given that it is difficult to express what 'execution' means, it is difficult to define whether an abstract class with an abstract method is really a **Command**, or whether it is just a **Template Method** for some other operation. As noted above, an **Interpreter** is an extension of the **Command** in which the execution of each terminal language literal is its own command.

It may be possible to define a specification that allows a proof engine to suspect an instance of the **Interpreter** pattern, by recognizing the combination of abstract and non-abstract types. However, such a loose definition could also match the **Command** pattern, along with abstract hierarchies (e.g. the `java.awt.Component`). But the key features of the **Interpreter** are its intent, and how it is used.

Fortunately, it is difficult to break the implementation of the **Interpreter** pattern; and, as a result, it tends not to decay over time. Partially, this is because an interpreter tends to be a fairly core component of the program that it is being used in, and as such functional tests often pick

[1] Although this thesis has concentrated mostly on whether patterns can be defined as constraints on their implementation, it may be possible to observe the existence of a pattern externally by identifying how other classes interact with it. This is commented on further in Chapter 11.

up any faults; but also, an interpreter is mostly defined by its intent, rather than any specific implementation. Perhaps this suggests that it is not necessary for a tool such as HEDGEHOG to be used to verify the correct implementation of a generic **Interpreter** pattern.

However, it is still possible to use the pattern specification language SPINE to good effect. Although there is no generic specification of an **Interpreter** pattern, it may be possible to translate a BNF specification of a language into a set of SPINE constraints. For example, if a simple mathematical language defined integer literals and various binary operators (e.g. plus, minus, multiply, divide), it would be possible to define a sequence of SPINE constraints that require the **Interpreter** has specific subclasses for each of the operators and for representing literals. If this process was automated, and a new operator was added to the language (e.g. modulus), then the new SPINE constraints would not match the implementation and an error condition could be raised.

9.1.2 Abstract Factory

The **Abstract Factory** pattern (defined in Figure B.1) describes `abstract` super-types for both the product and factory, with subclasses providing the implementation to select which product to instantiate.

The SPINE definition captures the requirement that each abstract factory and product pair has a pair of subclasses, such that the concrete factory class has a method which instantiates the concrete product. This matches the definitions given in [GHJV95].

Figure 9.1: Non **Abstract Factory** pattern that matches the SPINE definition

```
public abstract class Stack {
  public List abstract getList();
}
public class FIFO extends Stack {
  private List data;
  public List getList() {
    return new ArrayList(data);
  }
}
```

It is worth noting that this definition may match other classes which are not expected to be realisations of the **Abstract Factory** pattern. For example, a single abstract/concrete factory with a single abstract/concrete product could be matched with this pattern definition, but which would not be considered realisations of the **Abstract Factory** pattern. A contrived example is shown in Figure 9.1 that uses the Java collections data classes (in this case, the abstract product being *List* and the concrete product being *ArrayList*), where owing to a particular implementation, the concrete method in the subclass happens to create a new instance. This would match the SPINE definition of the pattern.

Although this can admit false positives, it is a type of false positive that is unlikely to arise in HEDGEHOG's use of verifying patterns in existing code. In order to determine whether a pattern is present, the system must be asked whether a pattern exists; and additionally, to provide the classes that are expected to be collaborators in the pattern. Thus, the system would only be asked to prove suspected cases of the pattern, or at least ones in which the user decided that a pattern should be present: in this case, they would probably not choose to apply HEDGEHOG to this particular non-example. If they did, they may be expecting that it did implement the **Abstract Factory** pattern, albeit in a fairly obtuse way.

Normally, the **Abstract Factory** pattern is implemented in such a way that the product names and factories share some kind of name, such that the concrete factory and its concrete products can be grouped together. For example, [GHJV95] uses '`ConcreteFactory1`' and '`ConcreteFactory2`' as the factories, and similarly '`ProductA1`', '`ProductA2`', '`ProductB1`' and '`ProductB2`' as the products. A good implementation of this pattern will use names that make it clear which products belong to which factory, and normally obey the fact that a concrete product is not shared between two or more concrete factories (although this is not an absolute requirement). Unfortunately, it is not possible to identify any naming conventions using SPINE and so this aspect of the pattern cannot be verified.

There are other implementations of this pattern that are not captured by the SPINE definition. For example, [GHJV95, page 91] discusses the possibility of an 'extensible' factory that uses parameterised methods to decide which product to instantiate. In this variant, it is not necessary to add a new method to the abstract factory for each product type; rather, one method can be used to create all product classes. Had this approach been taken with the *java.awt.Toolkit* class, some of the early problems with Java's AWT lack of extensibility may have been avoided.

9.1.3 Factory Method

The **Factory Method** pattern provides a way of instantiating classes using a method instead of a constructor. Since this is a very common idiom/mini-pattern in the object-oriented world, it may well not be explicitly declared in a design. The SPINE definition (shown in Figure B.2) uses the built-in operator '`instantiates`', which determines whether or not a method instantiates a given class. If a method instantiates the class, then it meets the pattern definition.

[GHJV95] shows as its example the **Factory Method** combined with an **Abstract Factory** pattern, and as such, it is quite difficult to determine which aspects are part of the **Factory Method** pattern and which are part of the **Abstract Factory** pattern. Given that the latter deals with the relationships between abstract and concrete factories (as well as abstract and concrete products), it is reasonable to assume that the **Factory Method** is solely responsible for the instantiation of the new object, as a method-based constructor. This allows a **Factory Method** to be used outside of an **Abstract Factory**, which might be useful for some realisations of this pattern.

This also introduces the possibility of a number of other false positives matching this pattern, given that object construction in methods is commonplace. As has been noted elsewhere, such small patterns tend to crop up frequently and are often not labelled as such owing to the extra noise that can occur in documentation and/or verification.

Although this pattern can be represented in SPINE, its usefulness due to the small pattern size and simplicity is questionable. Furthermore, some of the mini-patterns highlighted already are actually larger than this pattern.

9.1.4 Singleton

The **Singleton** pattern defined in [GHJV95] has a number of variants, including subclassing, lazy instantiation, and a registry of singletons. Several variants of the **Singleton** pattern are given in Figure B.4, but not including the subclassed or registry approaches. The rationale for not including these is that subclassing explicitly gives up control of creating the single instance (which is supposed to be a benefit of using the **Singleton** pattern in the first place); and the registry of **Singleton**s is equivalent to using a Java data structure such as a `Map`. In the latter case, it would be very difficult to correctly capture which uses of a `Map` were intended to be a **Singleton** pattern, and which were just being used for cacheing or searching.

For the pattern's variants that are defined in SPINE, they can be compared with the pattern definitions in [GHJV95, SM01] to see whether the pattern definition accurately represents it.

The SPINE specification requires that the **Singleton** be non-instantiable (Figure B.18). This requires the class to be either `abstract` or have one (or more) `private` constructors. Although this may seem to indicate that this allows `abstract` realisations of the **Singleton** pattern, the requirement for creating the instance of the **Singleton** class (either through a static initialiser or through a lazy instantiation) requires that the method instantiate a static field with a new instance of the same class. Given that it is not legal (in Java) to create a direct instance of an `abstract` class, it would not be possible to have a syntax error free class that allowed an `abstract` **Singleton**.

Although [GHJV95] provides examples in Smalltalk and C++ for the **Singleton** pattern, the approaches follow the SPINE definition. It is difficult to compare languages like-for-like, since the different languages approach construction slightly differently. As an example, since Smalltalk has no notion of `private`, the Smalltalk **Singleton** enforces non-instantiability by throwing an error during construction.

As noted earlier, the SPINE implementation does not attempt to formally define the registry of singletons by allowing the singleton to be subclassed; and therefore realisations using this variant will not be correctly verified.

9.1.5 Adapter

The **Adapter** pattern allows two systems to talk to each other despite having different external interfaces. This is often implemented as a wrapper class with one interface type delegating to an instance of a different type. Java examples include *`java.io.InputStreamReader`* and *`java.io.OutputStreamWriter`* types that convert between data streams of 8-bit bytes and 16-bit characters.

Given the level of indirection between the input and output types, it is not possible to capture all variations of the **Adapter** pattern. This is due to the fact that the **Adapter** pattern does not explicitly state what the relationship between the adapter and the adaptee is. [GHJV95, page 139] states “Adapter lets classes work together that couldn’t otherwise.”

The SPINE definition of **Adapter** (Figure B.5) simply defines a relationship between the adapater and the adaptee classes; specifically, that the adapter has a (single) reference to the adaptee, and that at least one method in the adapter class calls at least one method in the adaptee. Of course, it may be the case that a particular realisation of the **Adapter** pattern should delegate all methods to the adaptee, or it may be the case that the adapter does not hold an instance to the adaptee but instead looks it up dynamically.

Given the ambiguity of "work together", a generic implementation is defined in HEDGEHOG that covers most standard variants of the **Adapter** pattern. This is likely to result in both some false positives (for example, the `LayoutManager` may be mis-interpreted as an adapter, since it forwards messages to a `Container`) and some false negatives (for example, where the adapter does not hold an explicit reference to its adaptee).

9.1.6 Bridge

The **Bridge** pattern (Figure B.6) defines a family of adapter classes such that a source type has one (or more) target types that they adapt to. The SPINE definition is shown in Figure B.6.

As with the **Adapter** pattern, the nature of the bridging classes and how they are related are not well defined. In this case, the SPINE definition looks for a pair of types such that for each subclass of the source class, there is a related subclass in the target subclass such that the source forwards at least one method to the target class. Thus the kinds of problems associated with the **Adapter** pattern are the same as for the **Bridge** pattern.

9.1.7 Composite

The **Composite** pattern provides a way of grouping items together in a recursive manner. It may be implemented either with a generic component super-type and a composite sub-type, or with a composite super-type that allows all types to contain other components.

The SPINE definition (shown in Figure B.7) of the **Composite** pattern defines a relationship between the composite (grouping) class and the generic type of data to be added. It also adds the requirement that it must be possible to navigate from the composite to its child components; and that there must be a way of adding (and removing) components from the composite.

In line with standard Java practice, the methods for adding and removing items from the component should be prefixed '`add`' and '`remove`' respectively, to fit in with the JavaBeans naming conventions. Of course, this introduces a possible source for false negatives, since other implementations may choose to avoid this standard naming convention.

It also may be the case that the JavaBeans listener mechanism appears to be a minor type of container, since each JavaBean is a container for the component listeners that are interested in observing events. This may not be something that many developers would classify as a realisation of the **Composite** pattern, although in principle it is a specialised type of **Composite**.

9.1.8 Decorator

The **Decorator** pattern is similar to the **Adapter** pattern, in that the definition of the decoration is ambiguous. [GHJV95, page 175] says "Decorators attach additional responsibilities to an object dynamically" and thus has the same set of problems as the **Adapter** pattern.

The main difference between the **Decorator** and the **Adapter** patterns is that the former is supposed to keep the same interface, but the latter is normally expected to deal with two different interfaces. In this case, the SPINE definition (Figure B.8) requires the **Decorator** to have some parent class that it both sub-types and delegates one (or more) methods to.

As with the **Adapter** pattern, this introduces a possible source of false negatives, since it's not necessarily the case that the **Decorator** always forwards messages on to its enclosing instance, or that it may not store the reference in an instance field but look it up dynamically. It also opens up the possibility for many false positives, where non-realisations could be verified as a correct realisation of the **Decorator** pattern.

It is also potentially the case that a **Proxy** could be mistaken for a **Decorator**, or vice-versa. Of course, the fact that the purpose of a **Proxy** is to forward its messages to another target means that it could be described as a decorator; the only significant difference is in their intent. In the case of a **Proxy**, the intent is to be able to forward messages (potentially remotely, or across an encrypted bridge) to another object. The **Decorator** usually changes the behaviour of the enclosed object, whilst still (generally) forwarding messages to the enclosed object.

9.1.9 Proxy

The **Proxy** pattern is very similar to the **Decorator** pattern, in that the proxy class is expected to forward requests to an enclosing instance. In the SPINE definition in Figure B.10, the proxy and subject share a common super-type, and there is a field in the proxy that is used to invoke a method of the proxy class. This is very similar to the **Decorator** pattern, and it's possible that a class that meets the **Decorator** pattern will also meet the **Proxy** pattern as well.

As highlighted above, the difference between the **Proxy** and **Decorator** is that the intent of the two are different, rather than necessarily the implementation. It may be difficult to tell the two apart from a static analysis alone.

Also, there may be other classes that would not be considered a **Decorator** or **Proxy** but which match the implementation. For example, the *Stack* class is implemented using an internal *Vector*, and forwards messages such as '`size()`' to the enclosed *Vector*. This meets the definition of the **Proxy** pattern (since both share a super-type) but is unlikely to be thought of as a **Proxy** in terms of a pattern realisation.

This is explicitly raised in [GHJV95, pages 219—220] which discusses the comparison of similar patterns. It compares **Adapter** and **Bridge**, and highlights the fact "the key difference between these patterns lies in their intents. **Adapter** focuses on resolving incompatibilities between two existing interfaces. **Bridge** bridges an abstraction and its (potentially numerous) implementations."

The discussion also focuses on **Composite**, **Decorator** and **Proxy**. Although it considers the possibility that **Decorator** may be a degenerate case of **Composite**, it says "the similarity ends at recursive composition, again because of their differing intents." It also compares **Decorator** with **Proxy**, saying "both patterns describe how to provide a level of indirection to an object, and the implementations of both keep a reference to another object to which they forward requests. Once again, however, they are intended for different purposes."

It finishes the discussion with:

> "These differences are significant because they capture solutions to specific recurring problems in object-oriented design. But that doesn't mean these patterns can't be combined. You might envision a **Proxy**-**Decorator** that adds functionality to a proxy, or embellishes a remote object. Although such hybrids might be useful, they are divisible into patterns that are useful."

9.1.10 Iterator

The **Iterator** pattern allows an external client to iteratively step through the contents of a data structure, without knowing how that data structure is organised.

Given that it is difficult to know how a data structure is implemented, or can be traversed, it is difficult to know whether the **Iterator** pattern is correctly implemented by analysis of the code. However, given that Java has standard interfaces for defining the **Iterator** pattern, we can identify when this pattern is used by reference to its interface hierarchy.

Of course, this will not capture all realisations of the **Iterator** pattern, since a hand-rolled implementation of **Iterator** may use different method names and interface names. Furthermore, this definition is only particularly useful for Java implementations, since other languages will not necessarily have a specific interface for representing this pattern. Some languages may be able to use a similar technique; for example, a Python class can define that it supports the '`__iter__`' method that returns an **Iterator**.

It is unlikely that this will capture false negatives, simply because the interfaces in Java are so well-known that they explicitly imply this pattern.

9.1.11 Observer

The **Observer** pattern, defined in Figure B.13, captures the JavaBeans [Jav97] style of listener, in which the observable keeps a track of all registered observers. It uses the naming/coding convention that expects there to be methods to add and remove listeners to the observable, and that the observable invokes at least one method of the listener interface.

This definition is slightly different from the one used in [GHJV95], if only because that uses the terms 'attach' and 'detach' for registering and de-registering the observers. It also enforces the observer to get state manually after a notification; however, in the JavaBeans specification, the event data is propagated at the same time as the event notification. So whilst it may be the case that the observer invokes methods on the observable subject, it should not be taken as a prescriptive part of the pattern. Indeed, there is a discussion [GHJV95, page 298] that compares differences between "the push model, in which the subject sends observers detailed information about the change; the pull model, in which the subject sends nothing but a notification" in the notes regarding implementation specifics.

It may be the case that this pattern produces false positives; for example, determining whether an item in a `List` will invoke the `equals()` method on its contents; and since the `List` class also has `add()` and `remove()` methods, it might be mistaken for the structure of an **Observer**.

9.1.12 Template Method

The **Template Method** pattern defines an abstract method that is overridden by subclasses to provide behaviour specific to that subclass. The SPINE definition in Figure B.15 looks for a concrete method and an abstract method in an abstract class, such that the concrete method calls the abstract method.

Given the **Template Method**'s simplicity, it is likely that there will be no false negatives. However, it's such a small pattern that it is likely to crop up in many places in an existing codebase. Because it is so common, it's quite likely that HEDGEHOG could deduce its existence in classes where it was not explicitly expected; but having found it, it's quite likely that it would be a **Template Method** that has evolved from code reorganisations.

9.1.13 Visitor

The **Visitor** pattern allows behaviour to be externally applied to an existing object hierarchy without having to modify the existing objects. The SPINE definition in Figure B.16 ensures that for each object in the external hierarchy, a method exists on the visitor's interface that can accept that kind of data object.

This allows the basic aspect of the **Visitor** pattern to be elegantly captured; and furthermore, if additional subclasses of the data object are created, will ensure that appropriate methods are applied to the visitor class.

There are other variations of the **Visitor** pattern; for example, some Java implementations only have a single `public` method that takes the data super-type, and use dynamic introspection to delegate to an appropriate internal method. This has both advantages and disadvantages; it means that the caller of the **Visitor** pattern need not do introspection of the different types in order to decide which method to call. However, an abstract visitor super-class can be defined which uses **Template Method**s to process each of the different data type classes.

It's unlikely that this pattern will match false positives; since the object hierarchy of the data object and the methods of the visitor are tightly related. It may be the case that for some data types with single sub-types it is matched accidentally; or ones where the data object is `final` and thus has no sub-type.

9.2 Testing procedure

It is necessary to use real-world examples of design patterns in order to test HEDGEHOG's ability to represent and validate patterns. To this end, it is necessary to obtain:

- Java classes that correctly realise patterns from existing external source code
- Java classes that do not correctly realise patterns from existing external source code
- Java classes that do not realise any pattern

The most obvious source of design patterns would be to choose some of the 'standard texts' of patterns, such as [GHJV95, Bus96]. However, most of these books were written using Smalltalk and C++ as their examples, and do not show any Java examples as they were written prior to the development of Java. However, subsequent books have been written specifically for the Java platform, and [SM01] was developed by Sun Microsystems specifically to provide Java examples of design patterns. As a result, this provided one of the key sets of examples for testing HEDGEHOG.

It is also important that HEDGEHOG be capable of using patterns from real-world examples, and be extensible to allow new patterns to be defined in the future. There are many open-source or source-available projects that are written in Java and could be used, but perhaps the most obvious choice is to run tests against the Java language itself. This has evolved since 1995 and has had a new version released every two years (approximately), often with changes to the underlying code – although backward compatibility has always been paramount.[2]

Java 1.0 was seen by most as a toy language; it was not until Java 1.1 was released that it was accepted as a real language for development work. Significant changes were made to Java 1.2 (including the introduction of a new trademarked term 'Java 2'), and since then, there have been relatively few changes to the core API; instead, subsequent releases have provided additional capabilities such as sound sampling and 3D rendering.

As a result, changes between Java 1.1 and Java 1.2 are likely to be the most interesting, and thus examples of patterns from these versions were used. In particular, code that realised a pattern correctly in Java 1.1 were found not to realise a pattern in 1.2.

Testing was performed on the following sources of patterns:

AJP Applied Java Patterns [SM01], a book containing Java design pattern examples

PatternBox Eclipse Plugin The PatternBox Eclipse Plugin [Ehm] creates realisations of patterns based on internal hard-coded pattern definitions

Java Source The source code for Java 1.1 and 1.2 provided many examples of patterns that were (mostly) correctly realised

9.3 Selecting the examples

In order to verify that patterns were realised correctly, it was necessary to search the sources for pattern realisations that were either explicitly or implicitly documented, or were considered by other developers to be a clear example of a pattern.

The AJP book [SM01] provided a list of examples of Java patterns along with a description of the pattern itself, which therefore immediately provided a source of positive examples that could be used to test HEDGEHOG.

[2] Some APIs are marked as *deprecated*, meaning that they should not be used in the future. However, to date, the APIs marked deprecated in the original release of Java have not yet been removed from the current production versions of Java.

In order to find a set of patterns from the Java language libraries, a manual search of the source code/JavaDoc was required. Some patterns (such as **Singleton** and **Abstract Factory**) were obviously realised in classes such as `java.lang.Runtime` and `java.awt.Toolkit`[3]; the JavaDoc comments and class names gave sufficient clues to be able to deduce this. However, for source files without such obviously identifying marks, it was necessary to make a judgement about whether individual classes realised a pattern or not. This resulted in eliminating clearly negative examples of patterns from the available examples.

The examples generated by the PatternBox Eclipse Tool were generated from a menu which allowed an example to be generated of a specific pattern, and thus was a source of positive examples. However, it was possible to generate some negative examples of the patterns by removing or changing signatures in the generated code.

9.4 Selecting non-examples

In order to ensure that HEDGEHOG was not reporting patterns where none existed, it was also run against a selection of other classes. For example, as well as showing that the `String` class realises **Immutable**, it was possible to show that it did not realise **Iterator**. (`String` also realises other patterns; for example **Prototype** is realised by `String` in addition to **Immutable**.)

It was also possible to get selections of code that did not implement any pattern, from other classes from [SM01]. These were run against a set of patterns to determine if any of them matched the definitions of patterns, with incorrect results.

Lastly, non-examples were created by 'breaking' implementations of patterns in [SM01], by removing methods or fields that played a part in the pattern.

9.5 Results

When attempting to verify that a class (or set of classes) realises a pattern, there are four possible outcomes:

True positive **($\surd$+)** A pattern that is correctly realised, and recognised by HEDGEHOG as such.

False positive **($\times$+)** A pattern that is recognised by HEDGEHOG, but where no pattern exists (or is incorrectly realised).

[3] **Abstract Factory** is also known as **Toolkit**

True negative **($\surd$-)** HEDGEHOG recognises that the pattern is not correctly realised.

False negative **($\times$-)** HEDGEHOG claims that no pattern exists, but where a pattern should have been recognised.

However, there are also two other possibilities that prevent a pattern being tested (and hence unable to produce a result):

Unfound **($\emptyset$)** No pattern instance could be found in a specific test source (by the tester). This does not preclude the possibility of pattern instances being found in other test sources.

Unrepresentable **(—)** It is not possible to create a pattern definition in SPINE of the pattern, and as such no instances can be verified from any test source.

Of these six outcomes, true positive ($\surd$+) and true negative ($\surd$-) results are the successes for HEDGEHOG to report. These are generated when HEDGEHOG has given the correct answer.

A false negative ($\times$-) result is not a success, but may not be harmful; the intended purpose of HEDGEHOG is to draw attention to where it suspects there may be a problem, and if the human user finds no flaw then they can ignore the result (or improve HEDGEHOG by adding a new SPINE definition of that pattern). However, some uses of HEDGEHOG (such as automated discovery of patterns in existing source code, as discussed in Chapter 11) may be more sensitive to false negatives.

A false positive ($\times$+), recognising an incorrect realisation as a correct pattern, is a failure for HEDGEHOG. This would cause a human user to think that a pattern was correctly realised whereas it would in fact contain one or more errors. Furthermore, since HEDGEHOG works by erring on the side of caution, a positive result is less likely to be investigated by a human user than a negative result, and thus this class of errors may be more hidden from the end user.

It is possible (even likely) that a test source will not contain many instances of each pattern, or even that a given test source will contain a single instance of every pattern defined. These are termed *unfound* ($\emptyset$), and are shown in the results table to indicate that there was no example to test against the individual source. However, other sources may have instances of the pattern that can be tested.

An unrepresentable (—) pattern is a limitation of HEDGEHOG, rather than a failure of the system at run-time. As discussed in Section 5.5, there are some statements that SPINE is unable to represent without detailed knowledge of the pattern's intended use, which is a hard problem to solve.

The results shown in Table 9.1 are based on the categories defined in [GHJV95], broken down into the pattern name and testing component. Analysis of the results follows in Section 9.6.

Table 9.1: Results

Pattern	AJP [SM01]	PB Eclipse	Java 1.1	Java 1.2	Java Implementation
Creational					
Abstract Factory	√+	√+	√+	√+	`java.awt.Toolkit`
Builder		—			
Factory Method	√+	√+	√+	√+	`java.awt.Toolkit`, `java.net.URL`
Prototype	×-	×-	∅	∅	`java.lang.Cloneable`
Singleton	√+	√+	√+	√+	`java.awt.Toolkit`, `java.lang.Runtime`
Structural					
Adapter	√+	√+	√+	√+	`java.io.InputStreamReader`, `java.io.OutputStream-Writer`
Bridge	√+	∅	√+	√-	`java.awt.Component`, `java.awt.ComponentPeer`
Composite	√+	√+	√+	√+	`java.awt.Component`
Decorator	√+	∅	×-	×-	`java.io.Buffered-InputStream`, `javax.swing.JScrollPane`
Façade		—			
Flyweight	√+	√+	√+	√+	`java.awt.Color`
Proxy	√+	∅	∅	×-	`java.util.Collection$ UnmodifiableCollection`
Behavioural					
Chain of Responsibility		—			

continued on next page

continued from previous page

Pattern	AJP [SM01]	PB Eclipse	Java 1.1	Java 1.2	Java Implementation
Command		—			
Immutable	√+	√+	√+	√+	`java.lang.String`
Interpreter		—			
Iterator	√+	×-	√+	√+	`java.util.Enumeration`, `java.util.Iterator`
Mediator		—			
Memento		—			
Observer	√+	√+	√+	√+	`java.util.Observer`
State	×-	√+	∅	∅	
Strategy	√+	×-	∅	∅	
Template Method	√+	√+	√+	√+	`java.net.InetAddress`
Visitor	√+	∅	∅	∅	

The results are summarised in Table 9.2: of the 24 pattern types, 7 could not be defined in SPINE. That leaves 17 patterns, of which there were 4 examples each (less 13 which could not be found) which gives a total of 55 pattern instances to be verified. Of these, 47 were correctly recognised (46 true positives and 1 true negative), whilst 8 were incorrectly classified as not realising the design pattern (false negative). There were no false positives.

Table 9.2: Summary of results

	True (√)	**False (×)**	Other	
Positive (+)	46	0	**Unfound (∅)**	13
Negative (-)	1	8	**Unrepresentable (—)**	7
Total	47	8		20

9.6 Analysis

The results show that some patterns are easily verified by HEDGEHOG, whereas others are difficult or impossible to verify. There is also a lack of some realisations of the [GHJV95] patterns in some systems; indeed, the reason why [SM01] covers them all is due to the fact that it is a superset of those defined by [GHJV95]. Thus not all patterns in [GHJV95] turn up in certain large systems; rather, some (like **Immutable**) occur frequently, whereas others (like **Visitor**) occur infrequently.

To some extent this discrepancy can be related to the size of the pattern. Smaller single-class patterns (such as **Template Method**, **Immutable** and **Singleton**) are relatively common, as opposed to larger multi-class patterns (such as **Visitor** and **Bridge**) which tend to be used for specific cases where it is necessary to link together many classes, and occur less frequently. Also, multi-class patterns are usually more specialised (the **Visitor** pattern is used for traversing ASTs inside compilers, for example) and so are likely to be used in fewer situations.

Patterns from [Ehm] were found to realise most of the patterns, but since it is an ongoing development it does not have all the patterns listed in [GHJV95] at present. However, the tool is geared towards producing the patterns from [GHJV95] and therefore in the future it may be possible to use more examples generated from this tool.

More examples were found in [SM01] since it was a book specifically aimed at educating the user towards using design patterns in Java. Its pattern realisations were successfully verified in HEDGEHOG, bar two; **Prototype** (because HEDGEHOG has a fairly limited notion of prototypes; see Section 9.6.5.1) and **State** (because the example found was defined with inner classes; a known limitation of HEDGEHOG is that it cannot process source code with inner classes, as described earlier in this thesis).

9.6.1 No pattern definition

Several patterns from [GHJV95] could not be represented in a way amenable to HEDGEHOG and SPINE. The common feature of all of these patterns is related to semantic understanding of the problem and how the pattern is used, as opposed to anything specific in their realisation. Thus the problem moves out of SPINE's weak semantic constraints (as discussed in Section 5.5) into the stronger semantic constraints or intent, and out of HEDGEHOG's current grasp.

Perhaps unsurprisingly, most of the patterns that could not be defined fall into [GHJV95]'s 'behavioural' category. However, over half of the patterns defined in that category can be proven, so the category itself is difficult rather than impossible. (5 out of the 12 patterns in the 'behavioural' category were not able to be represented in SPINE.)

9.6.1.1 Façade

A **Façade** is designed to encapsulate a set of functionality (such as a subsystem) to hide the internal details from the users of the subsystem. However, it is very difficult to encapsulate this requirement in a form that is compatible with HEDGEHOG and SPINE; for example, simple approaches (such as "There is one class which interacts with many other classes but does not allow them to be accessed directly") are vague and may preclude some realisations (for example, accessing data through a **Façade** may require that a type from within the subsystem be returned).

Part of the reason for this problem is that it is difficult to define what the contents of a subsystem is. Java uses *packages* to group together related types, and in some cases this packaging structure may be used to define subsystems (such as the `java.awt.peer` package). But not all packages are subsystems, and so it would not be sensible to assume this mapping.

It is often the case that a **Façade** is realised in terms of a type which encapsulates access to the subsystem. However, merely recognising an abstract type would be incredibly difficult for HEDGEHOG without the potential for many false positives.

There may be a solution for certain types of **Façade** pattern, but not that would solve the example in [SM01, page 468].

9.6.1.2 Command, Interpreter and Chain of Responsibility

The **Command**, **Interpreter** and **Chain of Responsibility** patterns do not have a recognisable structure or form which could be used to represent the pattern. These are semantic patterns, in which the implementation of the methods is the key part of the pattern as opposed to the object-oriented structure.

As such, the approach for recognising (and verifying) either of these patterns requires the ability to reason in detail with the full Java execution semantics, and thus whether or not the methods are implemented correctly. Because these patterns cannot be defined in SPINE, HEDGEHOG cannot recognise these kinds of patterns.

Even though these patterns cannot be completely specified, they may be able to be partially defined. For example, the **Command** pattern could be defined as in Figure 9.2. However, the fact that the prefix of the execution method is defined as '`execute`' means that it will not recognise a number of different but correct realisations of the **Command** pattern.

Removing the constraint that the execution method need be called '`execute`' does not resolve the problem. If we do this, the pattern specification becomes very vague, and will match many implementations that are not realisations of the **Command** pattern. It will match

Figure 9.2: An attempt at specifying the **Command** pattern

```
realises('Command',[C]) :-
  isAbstract(C),
  exists(methodsOf(C),M.
    and([
      prefix('execute',M)
      isAbstract(M),
      forAll(subclassesOf(C),CC.
        exists(methodsOf(CC),M2.
          and([
            not(isAbstract(CC)),
            not(isAbstract(M2)),
            sameSignature(M1,M2)
          ])
        )
      )
    ])
  ).
```

any abstract class that has one or more abstract methods; and there are many of those (e.g. *java.awt.Toolkit*) that could not possibly be considered realisations of the **Command** pattern.

It also does not capture patterns that may be implemented using interfaces; for example, the `java.awt.event.ActionListener` is often used as a **Command** pattern for connecting GUI elements such as buttons to a specific command (or a `javax.swing.Action`). Although this requirement could be captured by replacing the specification for the parent to be an abstract type (rather than an abstract class), this raises even more possibilities of false pattern identification. For example, the `java.lang.Runnable` interface could match the definition for the command-as-interface pattern (and in some cases, the `java.lang.Runnable` is used to implement commands, particularly when posting to the Swing event queue). However, not all `java.lang.Runnable` implementors are **Command**s.

This reinforces the observation that whilst the **Command** and **Template Method** patterns are very similar in terms of implementation, their intent is very different. As noted in [Bro96], "it is naïve to assume that a program can comprehend the intent of a pattern ... what may be detectable, however, are the artefacts of implementation the solution of the design pattern." And since other idioms use the same implementation technique, telling the difference between

a **Command** and set of `java.lang.Runnable` implementors is the key problem. This cannot be differentiated by either structure or on constraints on the implementation of the **Command** pattern. As such, the **Command** cannot be represented with enough accuracy to make it either useful or not capture many false positives.

9.6.1.3 Builder, Mediator and Memento

The remaining patterns that cannot be defined are due to a lack of knowledge in how they will be used. For example, a **Memento** is merely a data object that can be used to store data between invocations/page displays; in essence, it is a data object only. However, it is how it is used and applied to the system that turns it into a **Memento**, which will therefore provide greater problems in proving that it is a design pattern. Equally, you could argue that any definition that admitted **Memento** would capture a whole class of 'Plain Old Java Objects' (or POJOs) incorrectly as **Memento**. A pattern definition that generated far too many false positives would be of little use.

A similar argument also holds for **Builder** and **Mediator**, except that these patterns are more dependent on how they are used than how the class itself is implemented. As a result, it is very difficult to create a pattern representation for these items that is likely to match more than one specific implementation. Of course, if a small set of 'standard' implementations could be found, then it may be possible to extract the common features, but there is far more choice in implementing one of these three patterns than in the other patterns. This makes it impossible to create any definitions for these patterns that HEDGEHOG can use.

9.6.2 True positives

The true positives are the success of HEDGEHOG working and giving the correct answer (that is, the pattern is correctly realised). The majority of the patterns (that could be defined) could be correctly verified by HEDGEHOG. Out of the 55 pattern examples that were representable in HEDGEHOG (covering 16 different patterns), 46 of them were true positives, giving HEDGEHOG a success rate of 83%. (Including the true negative shown in Section 9.6.3 below, the success rate goes up to 85%).

9.6.3 True negatives

The true negative found by HEDGEHOG is an example of showing how it can be used to find where patterns become broken over time. The **Bridge** pattern was realised correctly in Java 1.1's AWT, but during the transition to Java 1.2 and replacing the AWT with Swing broke the pattern.

The fact that HEDGEHOG could pick this up shows that it is useful in determining issues that change over time, and that a pattern that may have been correctly realised in a prior version may not hold in a later version.

9.6.3.1 Bridge

Although the **Bridge** pattern was verified in [SM01] and the Java 1.1 release, the pattern was not verified in Java 1.2. In fact, this is a success since the example of **Bridge** used is the `java.awt.Component` hierarchy which changed during the transition between 1.1 and 1.2 releases.

In Java 1.1, the `java.awt.Component` hierarchy was mapped onto the `java.awt.peer.ComponentPeer` hierarchy. Each subclass of `java.awt.Component` had a corresponding `java.awt.peer.ComponentPeer` class, so that (for example) `java.awt.Button` had a corresponding `java.awt.peer.ButtonPeer`. In essence, this is the purpose of the **Bridge** pattern.

However, in Java 1.2, the old 'heavyweight' AWT classes were superseded by the newer 'lightweight' Swing classes. The key difference between the two was that a `java.awt.peer.Component` no longer needed a `java.awt.peer.ComponentPeer`; instead, the rendering was done by Java's graphics libraries directly. A whole host of Java components were created such as `javax.swing.JButton`, `javax.swing.JComponent` etc. (the "Swing" components) which are subclasses of `java.awt.Container`. Although `java.awt.Container` has a peer (`java.awt.peer.ContainerPeer`), the corresponding subclasses do not have peers, which breaks the **Bridge** pattern.

In this instance, the designers actively chose to depart from the **Bridge** pattern for both performance and functional reasons, so HEDGEHOG is correct to declare that the 1.2 release of the `java.awt.Component` hierarchy no longer realises the **Bridge** pattern.

9.6.4 False positives

No false positives were reported by HEDGEHOG against the samples given. However, searching for false positives is a non-trivial task since it is easy to verify the existence of a false positive, but not to prove that they do not exist.

In order to test for false positives, correct realisations of patterns were modified until they no longer passed the verification. By the time the pattern was broken such that HEDGEHOG no longer recognised it as a pattern, it could not be called a correct realisation of a design pattern in any case (and hence was reported as a true negative).

In part, the reason for the lack of false positives generated can be attributed to not having pattern definitions for vaguely specified patterns. For example, a pattern definition of **Memento** would likely be very vague, and hence generate a selection of both true and false positives. By avoiding such vague pattern definitions, no false positives were seen.

This is desirable since a number of false positives would have given users of HEDGEHOG difficulty in determining which were true or false positives, and possibly therefore also losing faith in what HEDGEHOG is capable of doing. By avoiding vague patterns, the chances for false positives are decreased and so the end user's feeling of confidence in a true positive is increased.

9.6.5 False negatives

False negatives were reported by HEDGEHOG when patterns could not be correctly verified, even though they were valid realisations of a design pattern. Out of the 55 design pattern examples given to HEDGEHOG, 8 of these were reported as false negatives, or 14%.

As noted before, the fact that a pattern is raised as a false negative is not a complete failure. The goal of HEDGEHOG is to be automatically run against a set of classes and given design patterns, such that any irregularities (such as a design pattern becoming broken, as with the **Bridge** discussed in Section 9.6.3.1) are reported and further human action can be taken. In this case, some patterns cannot be correctly realised for one reason or another and the human user can investigate further to find out why the pattern is reported as not being correctly realised. Messages generated by HEDGEHOG (see Chapter 7) would help track down the problem and determine whether or not the problem is a serious one.

One of the reasons for HEDGEHOG's false negatives is that it does not support inner classes, and several of the patterns were defined in terms of inner classes. If the examples were rewritten to avoid the use of inner classes, it may have been possible to further reduce the number of false negatives reported by the system.

The other main reason for a false negative was a realisation of a design pattern which was subtly different from the pattern definition in HEDGEHOG itself. As such, patterns like **Iterator** and **Prototype** were defined in terms of standard Java interfaces, and those patterns that did not use those interfaces were not recognised. A discussion for each of these false negatives are presented below.

9.6.5.1 Prototype

The first failure is the realisation of **Prototype** in [SM01]. Part of the reason for this failure is the difficulty in creating a **Prototype** definition; indeed, in [GHJV95, page 121] the authors discuss **Prototype**'s advantages in a static language like C++ but note its lack of advantages when using a more dynamic language like Smalltalk (and also Java):

> Prototype is particularly useful with static languages like C++, where classes are not objects and little or no type information is available at run-time. It's less important in languages like Smalltalk or Objective C that provide what amounts to a prototype (i.e., a class object) for creating instances of each class.

There is a standard way of providing copies in Java using the *java.lang.Cloneable* interface. A class that implements *java.lang.Cloneable* has a method `clone()` which can be used to return a copy[4] of the current instance.

If one were to make use of the **Prototype** pattern in Java, it would therefore be sensible to base it on the *java.lang.Cloneable* interface. As a result, the SPINE definition for the **Prototype** pattern uses this fact, and [SM01, page 357] does not make use of this interface in its implementation; hence the proof fails for this pattern instance.

It may be possible to modify the pattern definition (or create a variant) to deal with this specific case, but it would not be of great use unless other systems used this variant as well.

Indeed, the failure of this example (linked with the documentation of what the **Prototype** pattern hopes to achieve and how it works) may have resulted in the developers of the [SM01, page 357] example using a more standardised[5] way of realising the pattern.

However, that is not to say that other pattern realisations could not be achieved in Java; for example, dynamic access of the *java.lang.Class* instance and `newInstance()` invocation could be another way of cloning a prototype.

[4] The Java semantics do not say whether a shallow copy or deep copy is required, just that a clone is returned. By default, a shallow copy is returned but the developer can program a deep copy if that is desired.

[5] Of course, the authors may have been showing a specific standalone example of all the interfaces working together rather than relying specifically on a Java class library, but this is of marginal value to a student learning the pattern.

Java's run-time nature, including its ability to load classes dynamically, has resulted in the **Prototype** pattern not being widely used. As an example, database connections require per-database specific drivers, which would (in other languages) be an ideal candidate for the **Prototype** pattern. In Java, drivers are configured by using their class name (such as *com.ibm.db2.jdbc.net.DB2Driver*) and using the *java.lang.Class* `newInstance()` factory method. Dynamic loading of classes complicates the ability to reason about the pattern's realisation and thus such esoteric variants of **Prototype** have not been modelled in HEDGEHOG.

9.6.5.2 Proxy

The **Proxy** pattern in Java is realised by few classes directly. Most of the proxies used in Java are in automatically generated classes (such as distributed technologies such as those used in RMI). The standard class libraries have few examples; the only pattern instance that could be found is that of the *java.util.Collections* class that uses internal wrappers for providing read-only or synchronized access to an underlying data structure. Unfortunately, the implementation of the *java.util.Collections* class uses named inner classes in order to define the *UnmodifiableCollection* class, and one of HEDGEHOG's limitations is that it cannot process inner classes directly. If the classes are extracted into their own separate compilation units, however, then it is possible to verify that the *SynchronizedCollection* do realise the **Proxy** design pattern; however, the read-only equivalents do not – since (understandably) they only forward the method calls that result in no change to the underlying structure. As a result, because they do not delegate methods such as `remove()`, they do not fully realise the **Proxy** pattern.

9.6.5.3 Decorator

The **Decorator** pattern bears a similarity to the **Adapter** pattern, both in terms of structure and intent. The key difference between the two is that the **Decorator** is not expected to change the component's interface, whereas the **Adapter** pattern is.

It is therefore expected that the **Decorator** supports the same interface as the component that it is decorating. As far as SPINE is concerned, it is not possible to determine whether the component supports the *intent* of the interface; only whether (structurally) it matches the specification.

Since the purpose of a **Decorator** is to modify the capabilities of the item it is decorating, but not fundamentally change it, the SPINE definition (in Figure B.8) requires that methods

in the **Decorator** are delegated to the component that it is decorating. This would allow an instance of the decorator to be used in place of the item it is decorating without affecting clients that use it.

This behaviour is seen in the *java.io.BufferedInputStream* class, which buffers data from an underlying *java.io.InputStream*. Whilst this is a good example of the **Decorator** pattern in use, it does not match the SPINE definition, since not all methods are directly invoked on the decorated component. For example, the '`read()`' method is not called on the underlying *java.io.InputStream* at all; rather, all methods for reading data are channeled through the more efficient '`read(byte[])`' method. So the requirement that all parent methods are called by the decorator class is falsified, and thus HEDGEHOG reports this as a failure.

A similar behaviour is seen in the *javax.swing.JScrollPane* class. Logically, this puts scroll bars on an existing component, and it is used as an example in [GHJV95] for when a **Decorator** may be applicable. However, the Java implementation puts an extra indirection through a *javax.swing.JViewport*, between the scroller and the original component. So whilst the *JScrollPane* acts as a decorator on top of its *JViewport*, the *JViewport* itself is decorating the underlying *Component*. Furthermore, both the *JScrollPane* and *JViewport* do not forward all the messages known by the parent class (*JComponent*) to the decorated container; they only deal with the graphical painting methods.

As a result, the examples found in the Java class libraries show that the **Decorator** SPINE specification does not match some more involved uses of the **Decorator** pattern. It would be possible to reduce the constraint that all methods are forwarded to the decorated item (for example, there must be at least one method that is forwarded) but this may not be desirable. Alternatively, variants of the **Decorator** pattern could be created (e.g. **FullDecorator** or **PartialDecorator**), and the user could decide which of these applied on a case-by-case basis.

9.7 Summary

HEDGEHOG is capable of recognising a number of design patterns realised not only in small code examples, but also real Java systems. There are some features of design pattern that HEDGEHOG can easily recognise:

- Smaller design patterns consisting of one or a small number of classes (as the problem becomes more difficult as the number of classes increases)

- Patterns that are defined by their structure or relationships

It cannot recognise patterns for which it has no SPINE definition, such as those which:

- Are defined in terms of how they are used by others (e.g. **Façade**)
- Have specific semantic requirements for how methods should behave (e.g. **Command**)
- Are defined by the intent of how it is used rather than implementation (e.g. **Memento**)

Of the categories defined in [GHJV95], the 'creational' and 'structural' patterns (12 in total) are relatively easy to verify with HEDGEHOG, with only a few cases (3) where the pattern cannot be defined. For the 'behavioural' category, of the 12 patterns there are 5 that cannot be defined. Interestingly, though the 'behavioural' category specifically groups patterns with a specific intent, it is still possible to capture the behaviour of half of the patterns in that category.

The remainder of the patterns show that the method of defining patterns in terms of their structure and weak semantic constraints works well; the majority can be recognised correctly. Limitations in HEDGEHOG's ability to correctly process inner classes accounts for several of the remaining failures; some implementations of patterns in the test data set used inner classes, and thus could not prove a pattern was present.

Lastly, the fact that HEDGEHOG is capable of showing when a pattern realisation is broken (the realisation of **Bridge** in Java 1.1 to 1.2) validates the fact that HEDGEHOG can be used to verify patterns and notify developers when such pattern realisations are broken over time.

These results show that the hypothesis from Chapter 1 (which was restated at the beginning of this chapter) has been met; it is possible to represent patterns based on implementation (listed in Appendix B), and that this specification is sufficient to be able to verify their realisations in existing code. Although not all patterns from [GHJV95] could be represented, those that could were very amenable to representation in this form.

Chapter 10

Related work

This chapter discusses related work, covered as an overview in Chapter 2, and presents a more detailed comparison of these with HEDGEHOG and SPINE.

10.1 ESC/Java

The Extended Static Checker for Java (ESC/Java) uses a similar process to HEDGEHOG's proof process; it uses an internal proof engine (called *Simplify* [Nel80]) and parses the source files to create an internal representation of the source code. It then uses static analysis of the files to detect potential errors which may not have been explicitly defended against by the user (such as whether a reference is assigned a non-`null` value).

ESC/Java also allows programs to be annotated with *invariants* [LS97], which declare valid ranges of values and other properties which are constant throughout an instance's lifetime. Such invariants may say that a 'direction' may only be one of 'up', 'down', 'left' or 'right', and take no other values. It then uses these annotations to verify that calls to these annotated methods do so with correct values.

HEDGEHOG does a similar task in trying to prove static semantic analysis of the code. This is very similar in purpose to that of ESC/Java; rather than trying to reason with a fully dynamic representation of the Java language (such as those dealing with $Java_s$ and $Java_{light}$), it deals with a much stricter set of constructs (such as determining whether a non-`null` value is assigned to a field) which is easier to work with than a fully dynamic analysis of the code.

Additionally, the authors of [FLN+02] make some very good points about dealing with this type of extended analysis of source code, which are also applicable to HEDGEHOG:

- Although general proof of whether an expression evaluates to `null` or not is undecidable in general, "the kinds of programs that occur in undecidability proofs rarely occur in practice."

- Modular analysis of Java source code allows the process to be scalable. "Both ESC/Java and ESC/Modula-3 perform modular checking; that is, they operate on one piece of a program at a time." Although patterns may consist of several Java classes (rather than just one), it is only a part of a much larger system. Furthermore, each occurrence of a pattern can be processed independently of others in a large system. "Consequently [the authors of [FLN+02]] consider modular checking to be an essential requirement [of a scalable automatic checker]."

- Although it is desirable for an ideal automated system to be both sound and complete, the authors of [FLN+02] do not take both of these to be an absolute requirement. "The competing technologies (manual code reviews and testing) are neither sound nor complete ... if the checker finds enough errors to repay the cost of running and studying its output, then the checker will be cost-effective, and a success." In other words, the cost of reviewing an incorrectly detected error is outweighed by other errors being successfully isolated, and the cost of an error slipping through this net is comparable with not having the tool in the first place.

What is interesting about their approach is that they have chosen a more formal representation of Java, but still are only concerned with a subset of possible Java programs. It is clear that the authors believe that although formally[1] it is a subset, in practice this covers the majority of 'normal' Java programs.

HEDGEHOG uses a similar level of analysis in its implementation. As with ESC/Java, it is clear that a less powerful formalisation is still capable of dealing with a number of common situations that may occur in real programs. Importantly, patterns are defined in terms of their implementation, and not by their executable semantics (similar to ESC/Java) so that it does not concern itself with traditional 'hard' problems such as recursion or termination.

However, whilst HEDGEHOG is concerned with the verification of patterns, ESC/Java is concerned with the correct implementation of individual methods. As a result, ESC/Java does not consider the object typing hierarchy as part of its method-based constraints, which is often a key part in how design patterns are represented. Furthermore, the fact that they are all method-based constraints means that it is very difficult to ensure that a pattern is present, because it is

[1] They note that it is neither sound nor complete

not possible to represent a pattern based on its execution semantics alone as previously noted in Section 4.3. Even if instance invariants are considered (constraints which are valid across any method), it is still not possible to enforce typing or inheritance constraints.

Lastly, patterns often span multiple classes and require the relationships between those classes to be maintained, rather than just implementations of specific methods to be constrained. It may be difficult to express the typing relationship in an ESC/Java constraint that relates all the classes of a **Visitor** pattern, for example.

Chapter 11 notes that a hybrid of HEDGEHOG and ESC/Java may prove a useful symbiosis, since patterns can be represented in SPINE, and the implementation of some of the built-in predicates that use the weak semantics could take advantage of ESC/Java's proof engine to show that the built-in holds. The two can complement each other since they focus on different levels of verification.

10.2 The fragment model

Marco Meijers [Mei96] proposed a mechanism for representing patterns as a set of fragments in what he termed "The Fragment Model." This breaks down the pattern in terms of what it must provide: relationships with other classes (including inheritance), methods that must be present, and how they are connected with one another. Each one of these requirements is a fragment, and a collection of fragments together under one 'root' fragment defines a pattern. As a comparison, the **Observer** pattern from [GHJV95] is shown in Figure 10.1, and the fragment representation in Figure 10.2 (taken from [Mei96, page 58]). The corresponding SPINE definition is presented in Figure B.13 for comparison.

The aim of the fragment model is to provide a definition of design patterns that can be used in a practical tool to allow the developer to instantiate patterns from scratch or from existing code. To that extent, the fragment model is useful for instantiating new patterns, but not necessarily for verification of existing patterns.

Each fragment represents a key feature of the pattern that must be present. In some cases, the purpose of a fragment is to link other fragments together; for example, in Figure 10.2 the '`ObserverHierarchy`' fragment exists solely to tie the '`Observer`' and '`ObserverA`' fragments together (and similarly '`Observer`' and '`ObserverB`'). Each fragment has associated conditions and information which affect how the pattern is realised.

Unlike HEDGEHOG, the fragment model exists to allow an interactive tool to add fragments to an existing code-base. This allows patterns to be instantiated by attaching the fragment to the

Figure 10.1: The **Observer** pattern from GoF

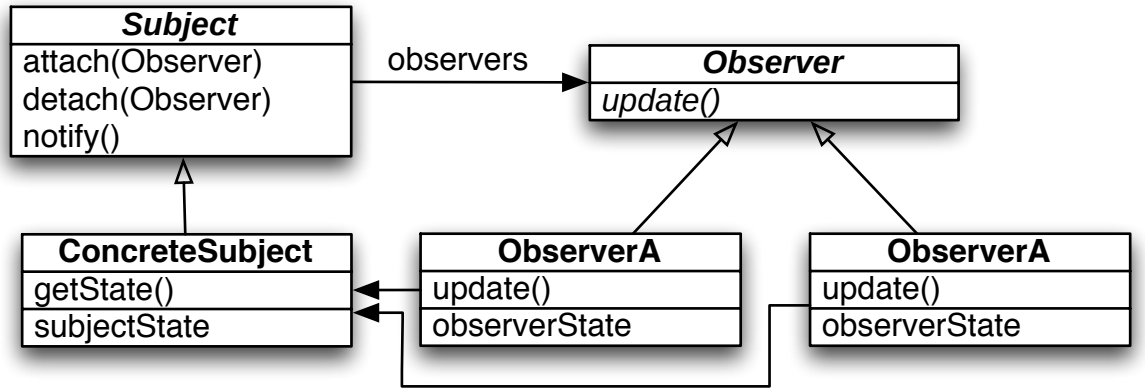
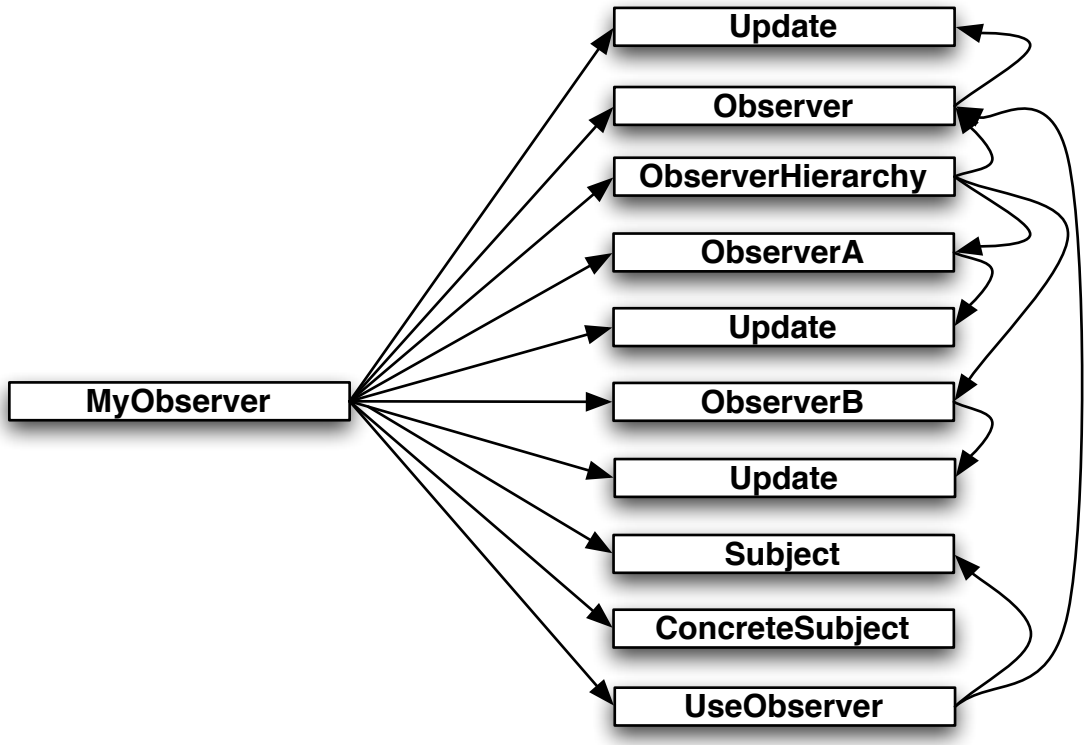


Figure 10.2: The **Observer** pattern using the fragment model



code representation within the tool. Additionally, triggers can be associated with fragments so that (for example) when a subclass is added to an existing class, its attached fragments are run to determine if any further changes need to be applied. This is cited as useful in the **Visitor** pattern when a new subclass may require the creation of a new method in the visitor implementation. It is not clear whether triggers were implemented in the fragment browser, however.

What is different between the fragment model and HEDGEHOG's use of constraints is that the former implicitly defines the pattern by the association of fragments. The fragment model defines the relationships (and what type of relationships there are) by interactively building up and associating fragments with a particular class in the run-time tool. This allows the tool to modify the fragments afterwards, or even introduce new ones by applying a new pattern to it.

In order to verify the fragments are still valid, a post-processing validation mechanism can run that is encoded into each individual fragment. This can ensure that (for example) a fragment has the correct number of actors; in the case of the Hierarchy fragment, there must be one parent and at least one child role fulfilled.

HEDGEHOG's approach to verification of patterns is to define them all as constraints that are applicable to any pattern, rather than having to associate fragments prior to use. The fragment tool includes a bottom-up mechanism that must be used on existing code first, before fragment operations can begin; and it requires the user's help to identify which fragments should be added.

The fragment tool does discuss the possibility of including a more formal constraint against the code that can be run after validation of the fragments themselves. In essence, HEDGEHOG's approach to representing the whole pattern as a constraint is similar to the combined approach of the fragment model and invariant construct. However, in multi-class patterns, each class will have its own fragments associated with it, whereas with SPINE, there is only the one pattern definition. This is more flexible when using external code systems that do not have fragments built up in the interactive tool; the pattern definition uses the appropriate predicates (such as using '`forAll`' and '`subclassesOf`') to dynamically generate a set of constraints spanning a number of classes.

It's not clear why the fragment tool uses two different mechanisms for representing the pattern's structure. On the one hand, the "invariant" constraint exists to ensure that the invariant for the pattern is valid before and after changes; but on the other, it also needs the fragments on an instance-by-instance basis in order to validate the pattern. HEDGEHOG's approach is to use the same unified mechanism for dealing with roles and behaviour by expressing the constraints for the entire pattern as a SPINE constraint. Additionally, since this is boolean logic,

it is possible to trivially combine the requirements for two separate patterns using a SPINE conjunction, and no further changes need to be made. This is not the same for the fragment model, which actually requires the fragment instances to be merged with the class instances if two fragments are combined.

The implementation of constraint checking is also not presented in [Mei96], which instead concentrates on the individual fragments and their relations. It appears that the constraints are based on the fragment implementations, and that these fragment implementations answer the predicates such as '`aClass` **`provides:`** `aMethod`' themselves. Thus the fragments are a necessary part of the tool's operation, since they also provide the invariant processing.

This also highlights another difference between HEDGEHOG's approach and the fragment model; the former needs nothing other than a goal and access to the source code, whilst the latter needs to have fragments adorned on to existing code before it can be checked. This has a couple of important ramifications:

- The variant of the pattern is explicitly encoded in the fragments attached to the source code. It isn't possible to change the source code outside of the tool to another variant and still have it recognised by the fragment tool.

- The checks that the fragment tool performs are actually checking whether the fragments have been initialised correctly, rather than whether the code exhibits the pattern. This is especially true in the case of the invariant checking, which requires the fragments to be created before they can be used.

Dennis Gruijs [Gru98] provided another example of the **Observer** pattern in the fragment language, shown in Figure 10.3. This describes the **Observer** pattern, which needs a number of collaborators in order to be instantiated. Some of these collaborators are other classes that can be interacted with (e.g. '`Subject`' and '`Observer`') whilst others are methods and data (e.g. '`notify`' and '`subState`').

In this representation, each pattern is defined as a Smalltalk instance that contains the fragments. To associate a pattern with a particular object instance, the pattern is cloned (an example of the **Prototype** pattern in use), and populated with appropriate values. In this case, the cloning of the '`Hierarchy`' fragment defines an inheritance relationship between the '`Observer`' and '`ConcreteObserver`' types. (In SPINE, this would be represented as '`extends(ConcreteObserver,Observer)`'.)

Figure 10.3: Example of another fragment **Observer** definition

```
Observer
{ | Subject notify observers attach detach ConcreteSubject
 subState StateObserver update ConcreteObserver obState |
  Hierarchy clone
  { superclass = Subject;
    subclass = ConcreteSubject;
  }
  Hierarchy clone
  { superclass = Observer;
    subclass = ConcreteObserver;
  }
  pluralType clone
  { plural = observers;
    singular = Observer;
  }
  Subject.methods += attach;
  Subject.methods += detach;
  Subject.methods += notify;
  Subject.notificationMechanism += notify;
  Subject.attributes += observers;
  ConcreteSubject.attributes += subState;
  ConcreteSubject.stateChangingMethods = ;
  Observer.methods += update;
  Observer.attributes += obState;
}
```

When this fragment is associated with an existing class, it sets up the relationships between the classes and the methods that they have together. One advantage of this approach is that any modifications to the classes can validate the changes against the fragments automatically. For example, it is possible to define '`handlers`' that are kicked off when a change occurs, such as adding a new method on one of the classes. The handler '`on new subclass@Subject`' will be invoked when a new subclass of '`Subject`' is created, to decide whether or not the new subclass should participate in the event notification part of the **Observer** pattern.

The approach of building up meta-information that is stored externally to the class is useful if development is only done in this tool. However, the meta-information is not portable outside of the fragment tool, and existing code-bases need to be migrated to instantiate the fragment models for each class that plays a part in the design pattern itself. Whilst one hopes that this meta-information could be persisted between invocations of the tool, it doesn't seem likely that this is the case. However, using HEDGEHOG as a verification tool does not require the use of any existing meta-information; the only external input that is required is the challenge that a class (or set of classes) realises a pattern, which HEDGEHOG then attempts to prove. This requirement could be added as a simple JavaDoc comment (or, in Java 1.5, an *annotation*).

10.3 Refactoring of design patterns

Unlike other approaches described in this chapter, [OC00] describes design patterns in the Java language. The work is tightly focussed on Java (as HEDGEHOG is) and also notes that there may be some advantages/disadvantages with the language-specific approach.

It also describes ([LK98]) a number of ways in which design patterns can be represented:

role model The most abstract representation of a pattern, defined in terms of the actors involved and their essential collaborations. These definitions are abstract and imply constraints that any refinement must respect.

type model A refinement of the role model where roles are replaced by domain-specific types that define the concrete syntax for operations and add to the abstract semantics of the role model.

class model A refinement of the type model that is the actual deployment of the pattern in terms of concrete classes.

The role model is akin to UML diagrams in which actors show parts of a use case. Thus, a pattern may be defined by the set of interactions that it has with its other collaborators, often represented as a use case diagram.

The type model may be drawn in the form of a UML interaction diagram. This shows some of the types of the actors, but abstracts away the implementation. LePUS can be considered to be operating between the role model and type model definitions.

The class model can be represented in the form of UML class diagrams, showing not only collaboration with other actors but also introducing methods and fields in order to achieve these goals.

HEDGEHOG operates at the level of the class model, taking into account the relationship of the classes' implementation. [OC00] uses a similar model for representing patterns, although possibly at a slightly higher level than HEDGEHOG.

It is interesting to note from [OC00, page 27] that:

> Pattern structure is insufficient in exact design pattern recognition as the pattern structure may be present, but not dynamic relationships or intent. Also, several patterns have the same pattern structure, and it is only the non-structural characteristics that differentiate between them.

This is largely true, and as such, HEDGEHOG cannot deal with highly abstract patterns such as the **Command** pattern (see earlier discussions in Section 9.6.1.2). In this case, most of the pattern implementation is tightly integrated with the intent of the application; as such, it is very difficult to separate the generic pattern from the application's use of the pattern.

This problem occurs more often with the "Behavioural[2] patterns" in [GHJV95], since these are implicitly associated with intent as opposed to structure. Some analysis of the "Behavioural" patterns using the semantic SPINE operators can be achieved, which are discussed more in Chapter 9.

[OC00]'s work provides transformations into such patterns, but hard-codes the names of the methods involved (in **Command**'s case, `execute()`). Whilst this approach is sufficient for transformations, it cannot be used for pattern detection since it is fairly likely that other method names could be used instead (e.g. `run()`, `process()` etc.), or extra arguments could be provided. This approach is very similar to [Tok99], which refers to them as refactorings.

Whereas HEDGEHOG uses a lower-level implementation of a pattern, [OC00] uses a slightly higher level to represent interactions and relationships between classes. HEDGEHOG's main focus is of the verification of patterns, whereas [OC00] presents transformations of ways in

[2]The others are "Structural" and "Creational" patterns

which patterns can be transformed and instantiated. This focus affects how the patterns can be specified; transforming into a pattern can be specified by a tight pattern definition, whereas verification of existing patterns may be looser in their implementation and thus harder to verify.

HEDGEHOG and [OC00] refer to the term mini-pattern, but use it in slightly different ways. Both identify repetition within the patterns and the way that they are used, such that the patterns are more modular. However, since the intent of HEDGEHOG and [OC00] are different, the two treat repetition slightly differently.

HEDGEHOG breaks down mini-patterns into identifiable elements that can be recognised and verified separately from others. This allows patterns to be defined based on a shared mini-pattern library. However, since [OC00] focuses on the transformation of code via mini-patterns, it uses parts of the design pattern that are amenable for transformations, and terms them *mini-transformation*s. For example, the **Factory Method** [OC00, page 71] is broken down into the following four mini-patterns:

1. Abstraction: the Product class must have an interface that reflects how the Creator class uses the instances of Product that it creates.

2. EncapsulateConstruction: In the Creator class, the construction of Product objects must be encapsulated inside dedicated, overrideable methods, which we term construction methods.

3. AbstractAccess: Apart from within the construction methods described in (2) the Creator class must have no knowledge of the Product class except via the interface described in (1)

4. PartialAbstraction: The Creator class must inherit from an abstract class where the construction methods are declared abstractly.

Each of these is implemented as its own mini-transformation which are then combined to make the **Factory Method**.

This level of detail is lower than the mini-patterns are defined in SPINE; possibly to the extent that the definitions would actually be one or two SPINE predicates. However, there is no reason why the SPINE library could not be extended with a reusable set of predicates if desired. It is most likely that the transformation of existing code into a design pattern needs to perform more work than verifying one that already exists; and as such, mini-transformations need to be more detailed.

As an example, a simple requirement like '`Abstraction`' requires the mini-transformation to perform the creation of an interface, and the extension of an existing class to support that interface, along with any methods that the class may provide. It should be noted that this is a fairly common refactoring also known as '`Extract Interface`' [Fow00, pages 341–343], and since it is a good coding practice to code to an interface rather than an implementation [Blo01, page 84–90] it is perhaps unsurprising that this mini-transformation is used frequently. In any case, this example would be represented as a single SPINE predicate '`isAbstract`' or '`isInterface`' and may not be eligible as a component for reuse.

10.4 LePUS

[Ede00] presents two ways of representing design patterns, based on evolution from earlier works [EGY97, Ede98].

The first approach was based on a metaprogramming representation of a design pattern, in which the pattern was encoded in the sequence of actions required to add a pattern to an existing system. The Patterns Wizard used *trick*s[3] to introduce artefacts into an object-oriented implementation.

HEDGEHOG initially took a similar approach to representing patterns, by encoding them in Java classes representing logical requirements. However, this suffered from exactly the same problem as the Patterns Wizard in that the pattern definitions could not be easily changed or investigated by the end user. Unlike the Patterns Wizard, the HEDGEHOG pattern definitions were always constraints on the implementation; externalising it as SPINE did not change the fundamental way that the pattern was defined.

The metaprogramming approach works well for the application of patterns, and is used by other approaches that modify existing patterns, whether by creating/editing [Mei96] or through refactoring [OC00, TB01]. However, this approach does not lend itself to working with a proof engine.

The second method uses a declarative approach [Ede00, page 47] for specification of design patterns. It started out as a graphical language called "The LanguagE for Patterns Uniform Specification" or LePUS that showed how the classes are related to one another in a similar vein to UML. It was subsequently retrofitted to a formalisation in higher order monadic logic or *HOML* [Bar76]. In this approach, artefacts are represented as members of a set, and these sets

[3]The name was chosen, somewhat unwisely in retrospect, because wizards are famous for performing tricks and magic

can be part of higher-order sets that represent classes, which in turn are part of higher-order sets that describe patterns between classes.

The declarative approach shares much more ground with SPINE (since both are constraint-based ways of defining design patterns) and so is worthy of closer comparison.

10.4.1 Graphical representation

The LePUS graphical notation of the **Observer** pattern is shown in Figure 10.4. One of the main benefits that LePUS claims to give is a precise but concise definition of a design pattern, and as the adage has it, a picture is worth a thousand words. As with any graphical notation such as LePUS and UML, it is only useful if the graphical symbols have defined/understandable meanings that enable the viewer to interpret what the diagram means; the definition for LePUS is covered in [Ede98, Ede00].

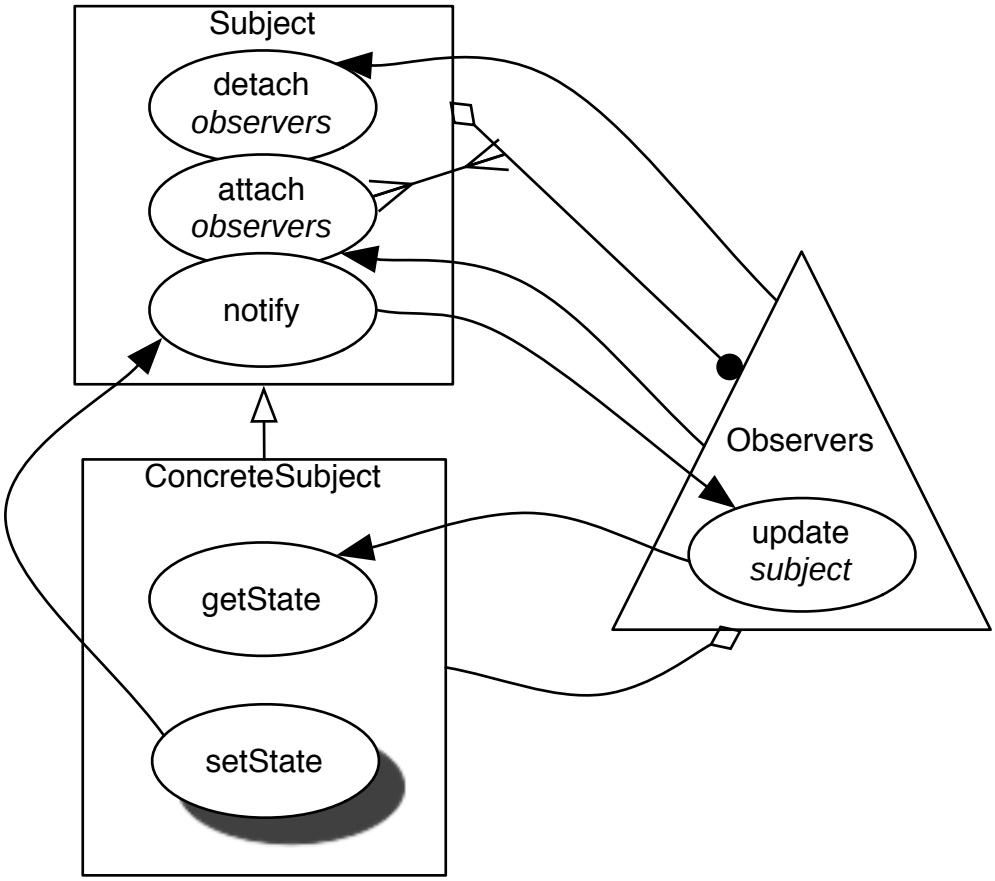
The square boxes with '`Subject`' and '`ConcreteSubject`' represent two classes in the system; the arrow between the two indicates that the former is the superclass of the latter. The ovals represent methods within the classes, whilst the shadowed oval represents a set of methods. (In the **Observer** pattern, there may be many methods that update the state; for example, an `Account` class may have methods '`withdraw`' and '`deposit`' that both affect the '`balance`' state of the `Account`.) The triangle with '`Observers`' represents a set of classes; in this case, there can be multiple classes that are observers of the subject.

The black-headed arrows show the flow of methods between the classes. For example, when a '`setState`' method is called on the '`ConcreteSubject`', it must invoke '`notify`' on the '`Subject`'. This in turn must invoke '`update`' on the '`Observers`'; because it is a set, it must invoke it on all members of that set. In turn, each '`Observer`' then calls '`getState`'.

The connector between the '`Observers`' and the '`Subject`' is a one-to-many relationship, and the connector between this relationship and the '`attach`' method indicates assignment; in other words, the '`attach`' method assigns the observer to the relationship.

This graphical representation therefore shows a combination of static relationships (one-to-many relationship between subject and observers, inheritance) as well as dynamic invocations (this method calls that method). It can be seen as a combination of UML class diagrams and UML collaboration diagrams, and may be useful for showing not only design patterns but also compact representations of an object-oriented system at run-time.

Figure 10.4: The **Observer** pattern in LePUS



Unlike UML, this notation permits sets to be presented. For example, the observer pattern allows many different observer classes to be used in its implementation. However, in an object-oriented environment, this could be equally well captured through the use of an `abstract` type such as an interface, and enforce a particular implementation[4] for implementors. It is also worth noting that UML has a notation for showing method broadcasts to multiple instances through the use of comments like '`[for all observers]`'.

10.4.2 Textual representation

As well as the graphical representation, LePUS has a more formalised textual representation in higher order monadic logic (HOML). Essentially, each of the icons in a LePUS diagram has an associated *formula*. The connecting arrows correspond to *relations* between the participants. Patterns can then be defined in terms of a set of formulæ. The textual representation of the **Observer** pattern shown in Figure 10.5 is the equivalent of the one in Figure 10.4.

In this example, the **Observer** pattern is defined as a conjunction of predicates as a formula. This is similar to a '`realises`' predicate defined in SPINE; however, the format of the formulæ in LePUS always consists of a conjunction of predicates. SPINE, on the other hand, is extensible and predicates can use combinations of conjunctions and disjunctions, as well as definitions of other predicates. This extensible definition is a necessary part of defining different variants of patterns, as well as reusable mini-patterns.

The relations ($\rightarrow$ and $\rightarrow H$) are ones that operate on a method-to-method, method-to-class, class-to-class, or sets of the same. As an example, '`Invocation` $\rightarrow$`(setState,notify)`' is a constraint that the '`setState`' method must call the '`notify`' method. This is a one-to-one mapping; every '`setState`' method must call one '`notify`' method. The alternative relation $\rightarrow H$ is a constraint against a hierarchy of classes; so '`Invocation` $\rightarrow^{H}$`(notify,update)`' means that for all subclasses that have a '`notify`' method, they must call an '`update`' method on all subclasses. A similar approach can be encoded in SPINE:

```
exists(methodsOf(ConcreteSubject),M.and([
  named(M,'setState'),
  exists(methodsOf(Subject),N.and([
    named(N,'notify'),
    invokes(M,N)
  ])
])
```

[4]This would be a good use of design-by-contract constraints; but can equally be done with JavaDoc comments

Figure 10.5: The **Observer** pattern in LePUS formulæ

$$
\begin{array}{l}
\exists\ \texttt{attach,detach,notify,getState} \in \texttt{F};\\
\quad \texttt{setState, update} \in 2^{F};\\
\quad \texttt{Subject,ConcreteSubject} \in \texttt{C};\\
\quad \texttt{Observers} \in \texttt{H}:\\
\qquad \texttt{clan(update,Observers)} \wedge\\
\qquad \texttt{clan(attach,Subject)} \wedge\\
\qquad \texttt{clan(detach,Subject)} \wedge\\
\qquad \texttt{clan(notify,Subject)} \wedge\\
\qquad \texttt{clan(getState,ConcreteSubject)} \wedge\\
\qquad \texttt{tribe(setState,ConcreteSubject)} \wedge\\
\qquad \texttt{Invocation}^{\rightarrow H}\texttt{(Observers,attach)} \wedge\\
\qquad \texttt{Invocation}^{\rightarrow H}\texttt{(Observers,detach)} \wedge\\
\qquad \texttt{Invocation}^{\rightarrow}\texttt{(setState,notify)} \wedge\\
\qquad \texttt{Invocation}^{\rightarrow H}\texttt{(notify,update)} \wedge\\
\qquad \texttt{Invocation}^{\rightarrow}\texttt{(update,getState)} \wedge\\
\qquad \texttt{ReferenceToSingle}^{\rightarrow H}\texttt{(Observers,ConcreteSubject)} \wedge\\
\qquad \texttt{ReferenceToMany}^{\rightarrow H}\texttt{(Subject,Observers)} \wedge\\
\qquad \texttt{Aargument-1}^{\rightarrow H}\texttt{(attach,Observers)} \wedge\\
\qquad \texttt{Aargument-1}^{\rightarrow H}\texttt{(detach,Observers)} \wedge\\
\qquad \texttt{Aargument-1}^{\rightarrow H}\texttt{(update,Subject)} \wedge\\
\qquad \texttt{Inheritance(ConcreteSubject,Subject)} \wedge\\
\qquad \texttt{Assignment(attach,Subject,Observers)}
\end{array}
$$

The advantage of the SPINE representation is that other constraints can be placed on the methods ‘`setState`’ and ‘`notify`’. For example, we could constrain the ‘`notify`’ method to be implemented as a `protected` member, which would prevent external objects performing direct notification. Additionally, we have finer control over the set of classes that can be processed; whereas $\rightarrow^H$ deals with all subclasses, we can use SPINE’s ‘`forAll`’ and ‘`subclassesOf`’ to deal with subsets of classes rather than an all-or-nothing hierarchy.

The clans and tribes defined in LePUS are higher-order sets (sets of sets). A clan is a group of related items: in this case, the ‘`setState`’ methods, as there can be many methods that change the state. (An example might be a bank account class, which has a state variable ‘`balance`’ but multiple methods for changing it such as ‘`withdraw`’ and ‘`deposit`’.) A clan of clans is called a tribe; so because the ‘`ConcreteSubject`’ is itself a clan, the ‘`setState`’ methods are termed a tribe.

Clans and tribes are not relevant in SPINE, because they can be handled by nested ‘`forAll`’ and ‘`exists`’ predicates. In fact, although it is not commonly seen, there is no reason for the ‘`forAll`’ to be arbitrarily nested to any level, whilst LePUS stops at tribes.

Although LePUS and SPINE allow a pattern to be described as a set of constraints, SPINE allows more flexibility by splitting down a requirement into arbitrary nesting of conjunctions, disjunctions, and use of existential quantifiers over subclasses, methods and so forth. Whilst LePUS defines the terms *clan* and *tribe* to describe higher order sets-of-sets, these can be more clearly presented as nested ‘`forAll`’ quantifiers in a SPINE predicate.

The set of LePUS relations[5] and SPINE predicates is very similar:

SPINE predicate	LePUS relation
invokes	invocation, forwarding
instantiates, lazyInstantiates, nonNull	creation
returns	production
implements, extends, subType	inheritance
adds, removes	assignment
signature	argument, return-type

[5]Assignment is only used in LePUS in defining relationships for the **Observer** pattern; similarly, ‘`adds`’ is only used in the **Composite** pattern for maintaining a one-to-many relationship

One key aspect of [Ede00] is that the design pattern specification in LePUS is programming-language agnostic (other than assuming a common set of object-oriented features). Thus a pattern may be described precisely as its interrelationships between classes, but not have enough information to process a pattern in a specific programming language (such as Java). The author of [Ede00] notes:

> Naturally, different [programming languages] incorporate different constructs, and the specification of design patterns at the appropriate level of abstraction required more than the lowest common denominator. Furthermore, a relation can have more than one syntactical form. As a more elaborate example take the term *Inheritance* in Java, where it is being assigned with two separate linguistic constructs that bear distinct notions. Thus, if we decided to incorporate in LePUS relations that are not necessarily primitives of all OOPLs there is the question of limits: What properties may be designated as ground relation? Why do we choose this set of ground relations and not any other?
>
> To answer this question we relate back to the purpose of our specification language: LePUS was conceived for the specification of lattices of design patterns. As such, the building blocks employed in its expressions, most notably the ground relations, should be descriptions made by the authors of these patterns.

In other words, the specification of the design pattern has necessarily been chosen in LePUS as a step above the actual programming language details.

Conversely, HEDGEHOG works at the programming language level – Java – so that it can verify that the language-specific implementation is correct, and not just the signature of the pattern. It therefore bridges some of the work that LePUS has achieved and the work that ESC/Java has done in analysis of the Java code directly.

10.5 Detection of patterns

Although detection and verification are not the same, they share a number of common features and points. Detection tends to look for pointers or hints as to where patterns may be present; and ideally, be fast to compute. Once a suspected location is found, more complex processing can be performed to determine whether or not a pattern really is present or not.

It is perhaps unsurprising that a verification tool will use much the same identifiers as a detection tool in specifying patterns. However, it is not necessarily the case that detection will always be a subset of verification; there may be trade-offs and a certain amount of false positives or noise that is desirable with a detection tool; as opposed to a verification tool in which false positives would potentially break the purpose of the tool.

What is clear is detection of a pattern from source can only be based on implementation, which also holds true of verification. The main problem is, as noted in Section 9.6.1, that in a number of cases (**Command**, **Interpreter** and **Memento**), the key part of the pattern lies not in its implementation, but in its intent. [Bro96] chooses to use a subset of patterns (**Composite**, **Decorator**, **Strategy** and **Chain of Responsibility**) since those can be identified based on their implementation, whilst acknowledging that patterns such as **Command** and **Interpreter** hold generic implementation but specific intent. [Bro96] suggests that although some patterns may be detectable, they may be ambiguous – in other words, there may be identifying hints that could indicate the presence of a pattern, but not be able to determine which one. Alternatively, they may be so common (such as **Template Method**) that they are identified almost everywhere throughout an application.

The other key difference between a verification tool and a detection tool is that the former is expected to be called into action against known (or suspected) instances already. A detection tool would have to do a pattern-mining approach across all instances before it could suspect whether a pattern is present or not. For some patterns, such as **Chain of Responsibility**, this is just a case of following navigable links; but for others, such as **Composite**, it may involve complicated walks over the object instance tree to discover cycles that may indicate recursive composition.

[PC00]'s approach represents objects and methods using a finite alphabet, then looking for sequences of method calls. For example, their definition of an **Iterator** is represented by the sequence $CcHh(NnHh)*$. Since an **Iterator** is a Java object which has two methods (`hasMoreElements()` and `nextElement()`) that are alternately invoked, an object that repeatedly calls M1 followed by M2 may be seen to be an **Iterator**. The syntax used in the sequence string is a call *H* and a return *h* of the `hasMoreElements()` method, and then a call to *N* and a return *n* of the `nextElement()` method. The call *C* and return *c* refers to the creation of an **Iterator**.

This approach aims to recognise patterns by their use, rather than by a specific pattern definition. It follows that if a pattern is not being used in a system (e.g. there is a class *`java.lang.Runtime`* that is a **Singleton**, but is not directly referenced by sample code) then it cannot be detected by this approach.

It is also not clear that a pattern can be represented cleanly using a regular expression to determine its use. For example, if an **Iterator** pattern called another method between the calls to `nextElement()` and `hasMoreElements()`, this would not still be recognised by the regular expression above.

Lastly, the approach does not verify that a class adheres to a pattern, but declares that the use of the class is similar to another pattern. For example, displaying elements in an array and calling a method to print out two properties of each element (for example, `a[i].getName()`, `a[i].getTitle()`) may be inappropriately recognised as an **Iterator** because of interleaving method calls.

10.6 Summary

This chapter discussed in more detail some of the work introduced in Chapter 2, and compared it to HEDGEHOG's approach of defining and processing patterns.

[LNS00] was presented as a Java-language aware constraint checker against existing code, and noted that although it is primarily aimed at finding (by preventing) bugs, the constraint mechanism may be suitable for implementing future SPINE predicates. However, since the tool does not express inheritance, typing, and relationships via the constraints on code, and that these are key features in many design patterns, it may not be capable of defining design patterns in ESC/Java directly. The point that ESC/Java is neither sound nor complete, but can still be a useful tool (and better than the alternative manual checking approach) is just as applicable to HEDGEHOG.

Different mechanisms for representing design patterns were also presented. The fragment tool [Mei96, Gru98] used a way of representing patterns by instantiating and relating fragments. Verification is performed against these fragments after they have been created; fragments are a necessary part of this tool and it is not possible to use it for verification purposes without setting them up first. Although there is a notion of invariants associated with this tool, it uses invariants to specify run-time behaviour and fragments to specify typing and relationships. This differs from HEDGEHOG's single use of SPINE to represent both in the same constraint.

The refactoring works [OC00, Tok99] tended to specify patterns as transformations which would refactor/instantiate a pattern from existing source code. A refactoring tool has a different set of requirements from a validation tool; in particular, it has more free choice in names of key methods (such as '`execute`' in the **Command** pattern) if they don't already exist; whereas they would be expected to exist with arbitrary names in a verification tool (if they were implemented correctly).

LePUS was covered since it provided two separate ways of representing design patterns: a metaprogramming method that allowed patterns to be instantiated by executing code called tricks, similar to the refactoring works; and a declarative method (similar to SPINE) that represented patterns as a graphical higher-order model. Unlike SPINE, the graphical language (and its textual translation) used a set-based logic to represent patterns. This set-based logic may be less customisable, since the translation can only be represented as a conjunction (there being no ordering in the graphical diagram). SPINE allows predicate logic, along with existential quantifiers such as '`forAll`' and '`exists`' which can be used to arbitrarily nest logical constraints.

Finally, a note about detection of design patterns and their relationships with a verification tool were presented; something that is discussed in further detail in Chapter 11.

Chapter 11

Further work and conclusions

This chapter presents some of the possibilities that may follow on from this work, and concludes with a summary of the work presented in this thesis.

11.1 Further analysis of design patterns

This work has presented a way of representing design patterns in terms of their implementation in Java classes (referred to as the class model in [LK98]). The results show that it is possible to represent some (but not all) design patterns using this method.

Part of the result of representing patterns in terms of their implementation shows that many patterns are actually implemented using common mini-patterns. For example, the **Factory Method** described in [GHJV95] could be described as a mini-pattern, from which other (larger) patterns such as **Abstract Factory** are implemented.

Specific investigation into patterns and their implementation in Java has led to some obvious candidates for mini-patterns:

Abstract definition: a method is to be defined abstract in a superclass, such that (concrete) subclasses are forced to provide an implementation

Accessor: a pair of methods (usually named getXxx and setXxx) that allow a field's value to be retrieved or a dynamic value calculated

Enum: an extension of singletons where multiple (shared) instances can be created instead of a single instance.

NonInstantiable: a class is non-instantiable if it has one or more constructors, and all are private; or if it is abstract

Further investigation of these mini-patterns may provide insights into how to discover new design patterns, or simplify the specification of existing design patterns. This may be even more useful for work that involves instantiating design patterns; it may be the case that such mini-patterns are used more frequently than their larger pattern counterparts.

Additionally, by considering the implementation of design patterns, several variants have been shown for some common design patterns (see Section 4.2.2). It may be that many design pattern implementations have further variants that have not been considered, and abstracting the common features of a variant may result in other super-patterns being discovered.

11.2 Applicability to other languages

It would be interesting to find out whether or not the approaches of using the class model are applicable to other languages. Since this work has focussed entirely on Java, it may be the case that there are aspects of HEDGEHOG that have been made possible because of using the Java language. There may also be additional ways of representing patterns that have not been considered in HEDGEHOG because of limitations or difficulties in defining the patterns in the Java language.

It is highly likely that those languages will be object-oriented (the [GHJV95] patterns are described only in terms of objects, for example) but it may be possible that mini-patterns exist in other non-object-oriented languages and that therefore the approach chosen by HEDGEHOG could be equally applicable to those languages. (It is not possible to claim that non-object-oriented languages can have patterns, because the design pattern community is by evolution a highly object-orientation affiliated community; however, structured programming techniques can approximate object-orientation and thus it may be possible to approximate design patterns in such a structured language.)

Clearly this would involve a major rework of HEDGEHOG's implementation, since all of the SPINE predicates are tightly focussed on the Java implementation. However, the concepts and directions chosen by HEDGEHOG could be used to start afresh in another language.

11.3 Reformulation of patterns using ESC/Java

Although the proof engine within HEDGEHOG allows patterns to be verified based on their SPINE definitions, it may be desirable to specify patterns in a more formal manner. One such possibility would be to use ESC/Java, which provides static checking against Java code.

Such checking would be ripe for use within HEDGEHOG, since it would be possible to prove or define a number of the SPINE predicates that are used in the representation of Java patterns. Indeed, it would have been a good starting point for HEDGEHOG had ESC/Java been publicly available at the start of this project; however, it was not released until nearly the end of the project. The two could complement each other; patterns could still be defined in SPINE, whilst ESC/Java could be used to implement some of the built-in predicates such as '`notNull`'.

It would be interesting to know whether or not a more powerful proof engine could affect the pattern verification rate for some of the difficult patterns such as **Command** and **Iterator** that HEDGEHOG is unable (or finds it difficult) to prove. Alternatively, it may be the case that these patterns are, in general, difficult to represent for the purposes of verification.

Should this approach work, it may be possible to build the same process with ESC/Modula to use a different language for implementing patterns.

11.4 Integration with IDEs

Clearly, the goal of HEDGEHOG is to be used by developers, preferably without any knowledge or experience of proof systems. Ideally, therefore, HEDGEHOG should be embedded in one of the common and popular IDE systems, so that it can be launched using the IDE's graphical user interface. Although an early release of HEDGEHOG supported launching from within the Visual Age for Java IDE, the product has been dropped by IBM in preference to its new WebSphere Studio toolset (which is based upon the open Eclipse IDE).

A more automated front end could be built, for example by scanning Java source files for specified identifiers. A developer could use a JavaDoc tag `@pattern singleton` to document that the given class realises the **Singleton** pattern, and this could then be fed into a back end process that validated that pattern.

Whilst integration with such an IDE would be of obvious benefit to users of HEDGEHOG, it is not within the scope of this project to make such an integration happen and so is left as a future exercise.

11.5 Integration with automated building tools

Current software best practices place high importance on running tests frequently, as advocated in Extreme Programming (aka XP) [Bec99]. The idea of automated testing is not new, but the frequency of these automated tests has been brought forwards to many times a day, or at least, once per day.

The reasoning behind this is that the later an error is detected in a system, the more costly it is to fix. Thus, if the error can be caught as soon as it is made (or as soon after as makes sense) then the cost in fixing this error will be dramatically lower.

Software projects using XP often have an automated set of tests that are run by the build machine on a nightly basis, with failures being mailed to the responsible people on the team. It would be most advantageous to integrate HEDGEHOG with this process, for its ability to detect both patterns, and also (and potentially more usefully) mini-patterns in the code. Thus, any errors introduced by mistake could be picked up at the end of each day, and an opportunity given to fix the mistake the next morning.

One of the most commonly used frameworks for automated testing is JUnit [GB]; but as with the integration with IDEs described above, it would be of great use to a developer working with HEDGEHOG, but is not an important part of this project as a whole.

11.6 Automated searching

One possible use of an automated verification system is that of searching for patterns in existing code. Several systems [Ban98, PC00] have tried to automate searching by looking for tell-tale markers in the classes to determine if a particular pattern is present or not.

HEDGEHOG could be used to perform a similar task, simply by attempting to prove pairs of class/pattern combinations. However, with N classes and M patterns, this becomes a total of $N \times M$ goals, which will take polynomial time to search. [Although most patterns will probably be discharged very quickly, it will still be polynomial in terms of the search space to cover, both from the number of classes to search and the number of patterns defined within HEDGEHOG.]

It is debatable how much use automated detection of design patterns actually is; in most cases, a system should have a design document[1] which will explain key design decisions and patterns used in the system. However, older (legacy) systems may not have design documents available that document the use of patterns.

[1] How well this is kept up to date is a different matter.

It is also possible that a design pattern will manifest itself in the most popular parts of an application, so it may be possible to intelligently sort the classes based on popularity (some kind of measure of the number of times a class is referred to from another class; in network theory this could be a highly connected node). It may then be possible to set a popularity threshold below which classes are not searched for pattern implementations.

11.7 Automated introduction of design patterns

One of the potential benefits of this verification approach is when a pattern is mostly (but not fully) implemented, the proof tree failure will have a record of what is still required (as discussed in Chapter 7). This may be used, not only to inform the end user of such a failure, but also to introduce the possibility of fixing the design pattern. [Eclipse already has this kind of quick code-assist feature for compilation errors; this could be extended to apply to design patterns as well.]

It is then a simple step from a pattern that is mostly implemented by a class, to a pattern that is not implemented by a class, but can have that pattern added using the error messages and introduction of missing parts.

Other works have focussed on introducing design patterns into existing applications [OC00, AACGJ01, Tok99] and there may well be crossover between HEDGEHOG's approach and some of the other existing approaches for introduction of design patterns.

11.8 Conclusion

The hypothesis, from Chapter 1, stated:

> This thesis aims to prove the hypothesis that it is possible to represent patterns as a set of constraints on the implementation of one or more Java classes, such that it is possible to verify whether they realise a pattern correctly.

In order to investigate this hypothesis, it was necessary to investigate design patterns to try and ascertain what makes a design pattern have the Quality without a Name [Ale79], or to identify the key features of design patterns in order to verify their correct implementation.

Although catalogues such as [GHJV95, Bus96, Vli98] present a set of design patterns, there is relatively little in the way of design pattern analysis. For example, [GHJV95] splits patterns into three separate sub-groups; behavioural, structural and creational. However, there is some

overlap between the three groups, and there are other ways of categorising patterns (e.g. whether there is single or multiple participants, or whether the pattern is defined by its intent or its collaborators). Additionally, it is clear from an analysis of design patterns (e.g. **Singleton** and **Utility**) that there are common features of both that can be thought of as a mini-pattern in its own right (see Section 4.2.4 and Section 4.2.5). Indeed, the approach outlined here would work for other common idioms (e.g. lazy initialisation) and could be used to help verify smaller building blocks than design patterns have traditionally been associated with. This backs up the findings of other works that have investigated the decomposability of patterns [OC00, Gru98].

Using a declarative language for specifying design patterns allows a proof system to be able to reason about design patterns. Other ways of representing design patterns (graphical representation or programmatic introduction/transformation) have been investigated by other works [Ede98, OCN99, OC00, Mei96, Gru98], and a comparison was given in Section 4.3. Each type of representation has its advantages (and disadvantages) and each are suited to the type of work that it is being used for. However, only a declarative language potentially allows a number of different uses, from verification (investigated in this thesis) to generation and transformation (highlighted in this chapter). It may even be possible to use a declarative language to generate transformational scripts to introduce patterns into new code.

The declarative specification language SPINE was presented in Chapter 5, along with the evaluable propositions that can be used to interrogate the Java implementation. Most of these propositions are generic, and deal with object-oriented concepts such as inheritance, method delegation and relationships; but some patterns (such as **Observer** and **LazySingleton**) require particular evaluable propositions that work at the method implementation level rather than the class signature level. Only a limited number of these propositions are needed in order to be able to define a number of different patterns.

Given that the pattern is specified as a set of constraints on its implementation, it's necessary to realise that patterns can be implemented in different ways. Thus, even for common patterns such as **Singleton**, there may be several ways of solving the problem. The ability to group several variations of a pattern together is an important part of any work that deals with patterns, since there is no one standard way of defining a specific pattern. One of the key strengths of this approach is that the SPINE definition is configured in a set of external rules that can be amended or added to at a later stage, so that if new variants are found, they can be added in by the user without having to change the internals of the HEDGEHOG proof system. On a related note, patterns are often implemented very differently in different languages (largely to take advantage of specific language features) so the definition of a specific pattern variant will obviously be specific to the language that it is implemented in.

The HEDGEHOG proof engine (discussed in Chapter 6) looked at the way in which the proof goals were discharged using the rules and predicates of the SPINE semantics. By implementing a proof engine in Java, it is possible to embed the proof engine into the Java-based IDE that HEDGEHOG was integrated with. Not only did that provide a better interface between the IDE and the proof system, it also allowed for certain optimisations, such as the ability for the IDE to query the state of the proof tree efficiently at the end of a proof, or for the proof process to interrogate the Java source on-demand. Unlike other systems where provers have been used, HEDGEHOG does not need to have the entire Java source tree translated into proof obligations that are uploaded before any work can be done; instead, the propositions are evaluated when they are needed and are discharged against the Java source as it stands. This lazy evaluation of evaluable propositions (as opposed to a pre-computed set of propositions) allows the proof system to scale and deal with thousands of classes and yet still be responsive to the end user. The same approach could be used with any other proof system that needs to access a potentially small subset of a large amount of data in order to only evaluate what is needed; however, this requires a proof system that is capable of such interaction hooks.

Because the specification is declarative, if one of the requirements is missing, it is possible to translate that failed requirement into a suitable error message that can be displayed to a user who is unaware as to how the proof system is being used. Generating error messages from proof trees was presented in Chapter 7. Since the proof system runs in the same Java process, it is easier for the result of the proof tree (including failed nodes) to be dynamically queried. This avoids the need to translate the proof tree to some intermediate format (text, xml etc.) which, for large proof trees, may take up a large amount of memory.

Given the nature of the proof constraints, it is perhaps unsurprising that the largest group of patterns recognised belonged to the ‘Structural’ and ‘Creational’ categories of [GHJV95]. These are (by definition) patterns that are defined by the structure of the code, or by the way in which methods are used to instantiate new objects (which is a fairly recognisable operation in Java). These type of patterns, and others in these categories, will generally be well suited to pattern processing of any kind, because they have a well defined set of requirements from an implementational standpoint, and so will be easy to define either for verification purposes or for transformational purposes.

The ‘Behavioural’ category was much less suited to pattern processing. This is due to the fact that patterns are defined in terms of their *intent*, rather than their implementation – in other words, what makes it a design pattern is not intrinsic to the pattern itself, but in terms of how that code is used by others. As an example, the **Command** pattern doesn’t have any

strict requirements in terms of what a command is; but the intent of the pattern when it is used is clear. Additionally, the number of variations that can be used with the pattern (having an abstract super-class, or an interface, or whether the method takes arguments or a state reference, or whether undo functionality is included etc.) means that an implementation of the **Command** could look almost like any code whatsoever. The only thing that would identify the pattern as a command to another developer would be the names (e.g. `Action` or `Command`) or the name of the method (e.g. `execute()` or `run()`). That is not to say that an analysis of these patterns is not possible; for example, an analysis of how a set of classes interacts with a pattern may provide a signature that could be used for identification purposes: however, the approach of specifying a pattern as constraints on its implementation is unlikely to work without resulting in many false positives.

Not all patterns can be represented in a way which is sufficiently unique to allow distinctions to be made between related patterns, or to avoid generating a number of false positives. For example, the **Adapter** and **Decorator** patterns are very similar in terms of their implementation; it is only the *intent* that is different between the two. As a result, a number of the patterns could not be defined in a way that is amenable to verification, and this is discussed in the analysis, Section 9.6.

It is now possible to answer the original hypothesis; it *is* possible to represent patterns as a set of constraints on the implementation of one or more Java classes, such that it is possible to verify whether they realise a pattern correctly. It is not possible to do this for every pattern; the more intent-based and less structure-based it is, the less likely that this approach will work. However, the analysis of the different types of patterns suggests that smaller idiomatic fragments of code or mini-patterns would be equally well suited for this kind of representation, and that it is possible to build design patterns from these smaller pattern types. This has been shown by building a pattern language (SPINE), and a proof tool (HEDGEHOG) that can be integrated within an IDE, and a set of results from known design patterns in order to test the hypothesis. Although this work focussed on Java (both as the implementation language and also as the pattern language) there is nothing Java-specific about the processing that is done, so the same techniques should be applicable to other pattern-hosting languages.

11.9 Summary

This chapter discussed some of the ways in which further work could be performed with HEDGEHOG.

- **Further analysis of design patterns:** more could be investigated into the different types of pattern (and mini-pattern) and whether variants are an interesting way of grouping patterns
- **Applicability other languages:** to see if the approaches used in HEDGEHOG are Java-specific, or whether they may be used in other languages (even if the implementation cannot be)
- **Integration with ESC/Java:** to provide strong statements about implementation of Java
- **Integration with IDEs and build tools:** to allow developers to work with HEDGEHOG more easily
- **Automated searching or introduction of design patterns:** to see if the approaches used to model patterns in HEDGEHOG are applicable to searching detection or instantiation of patterns in existing code bases

The conclusions that can be drawn from this work can be summarised as follows:

- **Declarative specification:** is well suited for verification, but could also be used for other uses e.g. pattern creation or introduction
- **Error messages:** can be generated from failed proof trees by compacting the proof tree to leave just the 'interesting' nodes, and messages that have been attached to the proof tree can be displayed to the user
- **Pattern categorisation:** is often based on the categories from [GHJV95], but there are more ways of categorising patterns than this
- **Pattern variants:** are neccesary, since there is more than one way to implement most patterns. Any pattern specification or processing definition needs to take this into account, and if possible, allow the end user to define their own variants or style of pattern

- **Patterns as constraints:** works for a number of different types of pattern, as long as there is some structure and not too much intent that defines the pattern; this split is similar for other types of pattern processing tools
- **Lazy evaluation:** is essential for proof systems that have a large environment where not all of that environment is necessary for the correct operation of the proof itself, to cut down on the amount of traffic between the proof engine and the system that is using it
- **Mini-patterns:** are smaller building blocks that can be used to assemble larger patterns, or simply stand on their own (also called idioms or mini-transformations)
- **Intent-based patterns:** are difficult to define because they are often defined in terms of how they are used rather than how they are implemented – and as such, do not work well with the constraints on implementation

Appendix A

Glossary

artefact Part of a design pattern (method call, class signature, relationship) that is specified in SPINE.

AST Abbreviation for Abstract Syntax Tree.

behavioural pattern A category in [GHJV95] that describes patterns that have a strong basis in how methods execute, or how patterns are used to achieve a particular run-time effect.

compile-time error Problem raised during the compilation of a program.

conjunction A combination of boolean results 'or'ed together. The result will be true iff there is at least one true boolean result. See also disjunction.

connective A term which combines (normally two) boolean values into a single boolean value.

creational pattern A category in [GHJV95] that describes patterns that have a strong basis in how classes are built or instantiated.

disjunction A combination of boolean results 'and'ed together. The result will be true iff each of the boolean results is true. See also conjunction.

false negative A pattern is recognised by HEDGEHOG, but where no pattern exists.

false positive A pattern is not recognised by HEDGEHOG, but where a pattern does exist.

first order logic A logic in which terms may be represented as variables, but where names of terms themselves cannot be represented by a variable.

FOL Abbreviation for first order logic.

function A term which can be evaluated to produce a result, such as a number, term or list.

goal A desired statement to be proven.

GoF Abbreviation for Gang of Four, the authors of [GHJV95].

HEDGEHOG The proof system that allows Java classes to be automatically verified against patterns defined in SPINE.

higher order logic A logic in which both terms and functions of terms may be represented as variables.

HOL Abbreviation for higher order logic.

iff Abbreviation for 'if and only if.'

instance An individual entity of a specific class.

instantiate A class is instantiated to create a new instance, which in Java uses the `new` keyword

JVM Java Virtual Machine.

mini-pattern A small idiom that is not listed in [GHJV95] as a pattern, nor would be big enough to be considered a full pattern. May be just the implementation of one or a couple of methods.

pattern A common solution to a common object-oriented program.

predicate A logical term which is either true or false.

realises A class (or set of classes) realises a design pattern if the implementation of the class(es) is a recognisable instance of the design pattern. This may also be known as *implements*, but since this would conflict with Java's interface implementation, a different term was chosen.

rule A statement which defines a goal in terms of other (sub-)goals.

run-time error Problem raised during the execution of a program.

SPINE A logical language used to define patterns for use in HEDGEHOG.

structural pattern A category in [GHJV95] that describes patterns that have a strong basis in relationships between classes.

sub-goal One of potentially many (sub-)goals, which when combined with a disjunction or conjunction proves the parent goal.

term A statement in the SPINE language.

true negative HEDGEHOG correctly claims that no pattern exists.

true positive HEDGEHOG correctly claims that the pattern exists.

type-safe A program (or language) is type-safe if all uses of a particular expression are consistent with the type of variable or argument. Typed languages, such as Java and C, are type-safe languages.

unfound A specific pattern could not be found in a specific test set for testing.

unrepresentable A pattern could not be defined in SPINE, so could not be tested against any pattern examples.

Appendix B

List of design patterns

This appendix contains a list of design patterns recognised by HEDGEHOG and the SPINE definition associated with it. Where pattern realisations are found in the Java language, the names of the classes are provided as reference.

The reader is assumed to be familiar with [GHJV95] and the description/purpose of each pattern, so these are not shown. The section titles are based on the design pattern categories.

Some of the design patterns could not be represented in SPINE; the reasons for the individual patterns are discussed below.

B.1 Creational patterns

Creational patterns are defined in [GHJV95] as patterns that result in the creation of new instances, or manage the process for obtaining instances of classes.

B.1.1 Abstract Factory

The **Abstract Factory** (also known as **Toolkit** or simply **Abstract Factory**) allows a different creational unit (factory) to be replaced with one that generates a family of related classes. In Java, it is realised in the `java.awt.Toolkit` class.

There are four classes of participants in the **Abstract Factory**:

Abstract Factory The abstract super-type of the factory itself

Concrete Factory One (or more) concrete implementations of the factory

Abstract Product The generic product(s) that are created by the factory

Concrete Product The specific product(s) that are created by the factory

Note that there may be variations; for example, the factory may just be the concrete factory class, or the factory super-type does not need to be abstract. The SPINE definition is shown in Figure B.1.

Figure B.1: Definition of the **Abstract Factory** pattern

```
realises('AbstractFactory',[AF,AP]) :-
  forAll(subclassesOf(AF),
    CF.exists(subclassesOf(AP),
      CP.realises('AbstractFactory2',[AF,CF,AP,CP])
    )
  )
(* CF is a sub-type of AF and CP is a sub-type of AP,
   such that CF generates CP *)
realises('AbstractFactory2',[AF,CF,AP,CP]) :-
  subtypeOf(CF,AF),
  subtypeOf(CP,AP),
  exists(methodsOf(AF),
    M1.and([
      typeOf(M1,AP),
      isAbstract(M1)
      exists(methodsOf(CF),
        M2.and([
          sameSignature(M1,M2),
          typeOf(M2,AP),
          instantiates(M2,CP),
        ]))
    ])).
```

B.1.2 Builder

The **Builder** cannot be represented in SPINE. The design pattern allows objects to be created by using an external class and then configuring its attributes based on a defined set of methods/data. This means that the pattern must do two things:

1. Create an instance of a class (possibly dynamically, or a specific sub-type based on configured data)

2. Configure the instance by setting up its instance variables appropriately

An example of a **Builder** in use may be creating an `Appointment` based on an external `vCal` text file. The purpose of the **Builder** would be to parse the entries in the `vCal` file and construct new `Appointment` instances, such that there needs to be no dependency between the `vCal` format and the `Appointment` class.

Although SPINE predicates could be used to determine when an instance of `Appointment` is created, it could not determine when the attributes were configured based on external data, because the scope of such a builder would remain very high. Furthermore, a simplistic definition of the pattern just detecting instantiation would result in a large number of false positives for the pattern; in effect, all it would be searching for would be instantiations of a named class. This is unlikely to provide useful for a developer, and combined with SPINE's inability to detect the 'correct' configuration of the instance, the net effect is that the **Builder** pattern cannot be represented in SPINE.

B.1.3 Factory Method

The **Factory Method** is one that is used to instantiate an instance by calling a method. It is implicitly used by the **Abstract Factory**, and represented in SPINE as a predicate `instantiates`. For completeness with other patterns, a `realises` predicate allows classes to implement a factory method.

When the factory method is static, and creates an instance of itself, it is called a **Static Constructor**, and whilst it is not a pattern defined in [GHJV95] it is provided here as a shorthand to be able to verify it. Similarly, the mini-pattern **Non Instantiable** is not listed in [GHJV95] but crops up so often it is worth noting independently.

Both implementations are shown in Figure B.2.

Figure B.2: Definition of the **Factory Method** and **Static Constructor** patterns

```
realises('FactoryMethod',[F,P]) :-
  instantiates(F,P).
```

B.1.4 Prototype

The **Prototype** allows objects to be copied (or cloned) from a default value. Section 9.6.5.1 discusses in more detail the problems in representing a **Prototype** as defined in Figure B.3.

Figure B.3: Definition of the **Prototype** pattern

```
realises('Prototype',[C]) :-
  implements(C,'java.lang.Cloneable').
```

B.1.5 Singleton

The **Singleton** pattern allows a single instance of a class to exist at one time. There are however, different variants that allow a singleton to be defined as shown in Figure B.4.

B.2 Structural patterns

Structural patterns define how classes are related to one-another in terms of inheritance, associations, or in the way that they are managed.

B.2.1 Adapter

The **Adapter** pattern allows classes of two different types to communicate with one another. In essence, the **Adapter** is a set of one (or more) methods delegated from one type to another, often with some conversion of the methods built-in. In this case, the '`Adapter`' needs a class to communicate with (called an '`Adaptee`') and then delegates some method calls from '`Adapter`' through. However, not all methods will be delegated; the specific calls that are or are not delegated are up to the individual developer coding the adapter class. As such, the SPINE statement has a relatively weak requirement that at least one method must be delegated, as shown in Figure B.5.

B.2.2 Bridge

The **Bridge** can be defined as a set of adapter classes on an object hierarchy, such that there are mappings between the two inheritance trees. The bridge class and peer class must be `related`

Figure B.4: Definition of the **Singleton** pattern

```
realises('Singleton',[C]) :- or([
  realises('PublicSingleton',[C]),
  realises('PrivateSingleton',[C]),
  realises('LazySingleton',[C])
]).
realises('PublicSingleton',[C]) :-
  realises('NonInstantiable',[C]),
  forAll(constructorsOf(C),Cn.isPrivate(Cn)),
  exists(fieldsOf(C),
    F.and([isStatic(F),isPublic(F),isFinal(F),
         typeOf(F,C),nonNull(F)])
  ).
realises('PrivateSingleton',[C]) :-
  realises('NonInstantiable',[C]),
  exists(fieldsOf(C),
    F.and([isStatic(F),isPrivate(F),
         typeOf(F,C),nonNull(F)])
  ).
realises('LazySingleton',[C]) :-
  realises('NonInstantiable',[C]),
  exists(fieldsOf(C),
    F.and([isStatic(F),isPrivate(F),
         typeOf(F,C),exists(methodsOf(C),
           M.lazyInstantiates(M,F))
    ])
  ).
```

Figure B.5: Definition of the **Adapter** pattern

```
realises('Adapter',[A,AE]) :-
  exists(fieldsOf(A),F.typeOf(F,AE),
    exists(methodsOf(A),
      M.exists(methodsOf(AE),
        D.invokes(M,D,F)))).
```

in some way that may be defined on a per-class basis, or may be due to casting to a particular type. For example the `Button` is related to `ButtonPeer`, but not to `LabelPeer`. The SPINE definition is shown in Figure B.6.

Figure B.6: Definition of the **Bridge** pattern

```
realises('Bridge',[C,Peer]) :-
  forAll(subclassesOf(C),
    X.exists(subclassesOf(Peer),
      P.and([related(X,P),
        realises('Adapter',[X,P])]))
).
```

B.2.3 Composite

The **Composite** pattern suggests there is a component class, and a container class, and that the latter has the ability to store many instances of the former. Often, the container is also a sub-type of a component which allows nesting, but this is not strictly necessary for the pattern, as shown in Figure B.7. In Java, the most obvious types are the `Container` and `Component` classes from the '`java.awt`' package, but additionally any of the data types (such as `ArrayList`) are also representations of the **Composite** pattern; though in this case, the component is `Object`.

B.2.4 Decorator

Like the **Adapter** pattern (Section B.2.1), the **Decorator** pattern can be defined in terms of its relationship to a parent super-type and methods delegated. In this case, the 'Decorator' is a sub-type of 'Parent' (which it is expected to decorate). For each method defined in the parent type, the decorator is expected to delegate to the original implementation at some point.

In most cases, this will give a reasonable match for the **Decorator** pattern. For example, the *`java.util.Collections`* class provides an implementation for synchronised access to a *`java.util.List`* that decorates the list implementation with a single-threaded synchronisation point. However, the *`java.util.Collections`* class also provide a method to wrap a list with an unmodifiable decorator, that explicitly does not delegate the `add` or `remove` methods. This is still a valid use of the **Decorator** pattern but one which this definition would not match.

Figure B.7: Definition of the **Composite** pattern

```
realises('Composite',[Component,Container]) :-
  or(
    typeOf(Component,Container),
    typeOf(Container,Component)
  ),
  navigable(Container,Component),
  exists(methodsOf(Container),M.and([
    prefix(M,'add'),
    exists(argsOf(M),A.and([
    adds(M,A,Container),
    typeOf(A,Component)]))])),
  exists(M,methodsOf(Container),M.and([
    prefix(M,'remove'),
    exists(argsOf(M),A.and([
    removes(M,A,Container),
    typeOf(A,Component)]))])),
  exists(M,methodsOf(Container),M.and([
    prefix(M,'get'),
    typeOf(M,Component)]).
```

Figure B.8: Definition of the **Decorator** pattern

```
realises('Decorator',[Parent,Decorator]) :-
  subtypeOf(Decorator,Parent),
  exists(fieldsOf(Decorator),P.and([
    typeOf(P,Parent),
    forAll(methodsOf(Parent),
      PM.exists(methodsOf(Decorator),
        DM.invokes(PM,DM,P)))
    ])
  ).
```

B.2.5 Façade

As discussed in Section 9.6.1.3, the **Façade** is a design pattern that does not have an easy-to-capture representation amenable to SPINE. This is due to the fact that the **Façade** is largely defined by the way that the classes are used, as opposed to defined; this means that SPINE's weak semantic analysis is not powerful enough to define a **Façade**.

B.2.6 Flyweight

The **Flyweight** pattern provides a set of predefined instances that can be used (shared) across multiple classes, to save creating new instances. In Java, these are seen by the `String` and `Color` classes. The pattern definition is shown in Figure B.9.

Figure B.9: Definition of the **Flyweight** pattern

```
realises('Flyweight',[Element]) :-
  realises('Flyweight2',[Element,Element]).
realises('Flyweight2',[Pool,Element]) :-
  or([
    realises('StaticFinalFlyweight',[Pool,Element]),
    realises('ImmutableFlyweight',[Pool,Element]),
  ]).
realises('StaticFinalFlyweight',[Pool,Element]) :-
  exists(fieldsOf(Pool),
    F.and([
      isStatic(F),
      isFinal(F),
      nonNull(F),
      typeOf(F,Element)
    ]))
realises('ImmutableFlyweight',[Pool,Element]) :-
  realises('Immutable',Element),
  realises('Container',Pool).
```

B.2.7 Proxy

The **Proxy** pattern allows one object to proxy messages to another object, possibly involving doing something in the process (logging, security etc.) In Java, the `UnmodifiableCollection`

provides a mechanism to proxy methods from one type to another, but does not proxy the writeable methods; hence it allows the underlying collection to remain read-only.

Unlike **Adapter**, the **Proxy** often shares a common super-type between the subject and the proxy itself. The representation is shown in Figure B.10

Figure B.10: Definition of the **Proxy** pattern

```
realises('Proxy',[Subject,Proxy]) :-
  sameSupertype(Subject,Proxy),
  exists(fieldsOf(Proxy),F.and([
    typeOf(F,Subject),
    exists(methodsOf(Proxy),
      M.exists(methodsOf(Subject),
        D.invokes(M,D,F)))])).
```

B.3 Behavioural patterns

Behavioural design patterns put constraints on how an instance may behave at run-time. Although HEDGEHOG can only work with a limited set of weak semantic constraints, it is still possible to verify most of the key patterns.

B.3.1 Unrepresentable patterns

There are several behavioural patterns that cannot be represented in SPINE because the meaning of the pattern is very dependent on how (and where) it is used, or has dependencies on the semantic interpretation of methods. These patterns are **Command**, **Chain of Responsibility**, **Interpreter**, **Mediator** and **Memento**. Since SPINE only works at a weak semantic level, it cannot reason about patterns that rely on specific behaviours outside of its built-in predicate set.

A more complete discussion is presented in Section 9.6.1.

B.3.2 Immutable

The **Immutable** pattern allows a developer to force a read-only view of an object by preventing the instance fields from being modified after they have been initialised. An example of an **Immutable** object in Java is the `String` class. The definition is shown in Figure B.11.

Figure B.11: Definition of the **Immutable** pattern

```
realises('Immutable',[C]) :-
  forAll(fieldsOf(C),
    F.or([isStatic(F),isFinal(F),
       and([hasModifier(F,private),
         forAll(methodsOf(C),M.not(modifies(M,F)))
       ])
    ])
  ).
```

B.3.3 Iterator

The **Iterator** in Java is provided by the Java types `Iterator` and `Enumeration`. SPINE is capable of defining types in terms of these elements, but not of verifying the semantic correctness of what iteration means. As with **Prototype**, it is possible to determine at a very loose level whether something is an `Iterator` by determining if it implements the standard Java interfaces. However, this will therefore have the possibility of admitting false negatives; for example, it is possible to realise the **Iterator** design pattern without using the standard built-in types, but HEDGEHOG will not be able to recognise it (as discussed in Section 9.6.5).

Figure B.12: Definition of the **Iterator** pattern

```
realises('Iterator',[C]) :- or([
  implements(C,'java.util.Enumeration'),
  implements(C,'java.util.Iterator')
]).
```

B.3.4 Observer

The **Observer** is used to allow a number of instances to synchronise with each other. This is achieved by having a set of observers maintained by the event source, such that when something occurs, the observers are subsequently notified.

In Java, this is seen in the `Observer` and `Listener` interfaces. The SPINE definition is shown in Figure B.13.

Figure B.13: Definition of the **Observer** design pattern

```
realises('Observer',[Observable,Listener]) :-
  navigable(Observable,Listener),
  exists(methodsOf(Observable),M.and([
    prefix(M,'add'),
    exists(argsOf(M),A.and([
    adds(M,A,Observable),
    typeOf(A,Listener)]))])),
  exists(methodsOf(Observable),M.and([
    prefix(M,'remove'),
    exists(argsOf(M),A.and([
    removes(M,A,Observable),
    typeOf(A,Listener)]))])),
  exists(methodsOf(Observable),
    M.exists(methodsOf(Listener),
      D.invokes(M,D))).
```

B.3.5 State

The **State** pattern allows an object to change its behaviour based on an internal state object. Methods are delegated to the state object, and when the state needs to change, the instance field is replaced with a new value. This does not directly occur in Java, but can be defined in Figure B.14.

B.3.6 Template Method

The **Template Method** pattern allows a placeholder method to be provided in a super-type that is then defined in the sub-type. The super-type invokes the template method which is thus defined in the subclass. The definition in Figure B.15 shows how this can be recognised.

B.3.7 Visitor

The **Visitor** pattern is used to add behaviour to a class hierarchy without having access to the hierarchy's source code. Each element in the hierarchy corresponds to a unique visit method in the visitor interface, and many different visitors can then process the same hierarchy.

Figure B.14: Definition of the **State** pattern

```
realises('State',[C,State]) :-
  isAbstract(State),
  exists(fieldsOf(C),
    F.and([
      typeOf(F,State),
      nonNull(F),
      forAll(methodsOf(State),
        Ms.exists(methodsOf(C),
          Mc.invokes(Mc,Ms)
        )
      )
    ])
  ).
```

Figure B.15: Definition of the **Template Method** pattern

```
realises('TemplateMethod',[C]) :-
  isAbstract(C),
  exists(methodsOf(C),
    AM.exists(methodsOf(C),CM.and([
      isAbstract(AM),
      not(isAbstract(CM),
      invokes(CM,AM)
  )])))).
```

Figure B.16: Definition of the **Visitor** pattern

```
realises('Visitor',[Visitor,AST]) :-
  isAbstract(Visitor),
  forAll(subclassesOf(AST),
    C.exists(methodsOf(Visitor),
      M.exists(argsOf(M),A.
        typeOf(A,C)))).
```

B.4 Mini-patterns

These mini-patterns are not defined in [GHJV95] but are commonly repeated by many of the pattern definitions. In some cases, they are used by developers as patterns in their own right (they are repeated to solve a common problem); however, some may see these patterns as 'too small' to be called patterns in their own right. Hence the term 'mini-pattern' is used to describe these patterns.

It could be argued that **Factory Method** is an example of a mini-pattern, rather than a fully fledged pattern. However, it is listed in [GHJV95] as a pattern, and because of this has become accepted as a first-class citizen in the pattern community.

B.4.1 Static Constructor

The **Static Constructor** mini-pattern uses a static method to obtain an instance, as opposed to using a constructor. It is an adaptation of the basic **Factory Method** since it also prevents the constructor being used. The definition is shown in Figure B.17.

Figure B.17: Definition of the **Static Constructor** mini-pattern

```
realises('StaticConstructor',[C]) :-
  realises('NonInstantiable',[C]),
  exists(methodsOf(C),M.and(
    isStatic(M),
    realises('FactoryMethod',[M,C])
  ).
```

B.4.2 Non Instantiable

The **Non Instantiable** mini-pattern is used to prevent a type being instantiated. It's a common requirement in patterns such as **Singleton** and **Factory Method**, and as such is shown in Figure B.18.

B.4.3 Lazy instantiation

Lazy instantiation is a technique common to Java programmers, and is a low-level programming idiom that can be used to instantiate an instance field on the first invocation, and thereafter return

Figure B.18: Definition of the **Non Instantiable** mini-pattern

```
realises('NonInstantiable',[C]) :- or([
  and([exists(constructorsOf(C),true),
       forAll(constructorsOf(C),Cn.isPrivate(Cn))]),
  isAbstract(C)]).
```

it.

Because of its high dependency of Java implementation, this is presented as a built-in predicate in SPINE. However, it may be desirable for a developer to ask HEDGEHOG to verify that the lazy instantiation works as expected, so a mini-pattern is provided to allow a developer to access the SPINE built-in predicate, and is shown in Figure B.19.

Figure B.19: Definition of the **Lazy Instantiation** mini-pattern

```
realises('LazyInstantiation',[M,F]) :-
  lazyInstantiation(M,F).
```

Appendix C

List of SPINE functions and predicates

This chapter presents a list of the functions and predicates used in the definitions of patterns throughout this thesis and in Appendix B.

C.1 Functions

SPINE functions are used to directly interact with the Java source stored in HEDGEHOG's Java AST cache. The functions are built-in directly into HEDGEHOG, and cannot be extended through SPINE; however, they may be developed in Java and installed as part of HEDGEHOG's core abilities. It is expected that for most pattern definitions, the built-in functions and predicates will be enough to extend HEDGEHOG by simply writing derived predicates.

argsOf(*M*) is a set of arguments of method *M*

constructorsOf(*T*) is a set of constructors that are defined in *T*

fieldsOf(*T*) is a set of fields that are defined in *T*

load(*File*) loads the SPINE definition file into memory – note that this causes further rules to be added to the rule library and does not have a specific value

methodsOf(*T*) is a set of methods that are defined in *T*

subclassesOf(*C*) is a set of subclasses of C

C.2 Predicates

SPINE predicates are used to determine if a certain property or feature holds. Unlike functions, the predicates may be *built-in* or *derived*. A built-in predicate is implemented, like a built-in function, using Java classes and internally defining it within HEDGEHOG. However, unlike functions, predicates may be added externally to the core of HEDGEHOG by writing SPINE rules, which then define the derived predicates.

All of the pattern definitions in HEDGEHOG are derived predicates called '`realises`', although in the derivation of those predicates, other derived and built-in predicates, and built-in functions are used.

C.2.1 Built-in

adds(*M*,*Type*,*Collection***)** the method *M* adds an instance of *Type* to *Collection*

exists(*Set*,*X*.*P*(*X*)**)** at least one *X* in *Set*, then $P(X)$ holds

extends(*C*,*P***)** class *C* extends (is a subclass of) *P*

forAll(*Set*,*X*.*P*(*X*)**)** for every *X* in *Set*, then $P(X)$ holds

hasModifier(*X*,*M***)** *X* has modifier *M*

implements(*C*,*I***)** class *C* `implements` interface *I*

instantiates(*M*,*T***)** method *M* creates an instance of *T* and returns it, although the method may have a declared return type of *T* or one of its super-types

invokes(*Method*,*Delegate***)** code in *Method* invokes the method called *Delegate*

invokes(*Method*,*Delegate*,*Field***)** code in *Method* invokes *Delegate* on *Field*

isClass(*T***)** *T* is a class

isConstant(*E***)** *E* does not change value

isSideEffectFree(*E***)** *E* does not change the object's state

isInterface(*T***)** *T* is an interface

isLiteral(*E***)** *E* is a literal Java expression

lazyInstantiates(*M*,*F*) *M* lazily instantiates *F* and returns it

modifies(*M*,*F*) method *M* modifies the value of field *F*

navigable(C_1,C_2) it is possible to navigate between C_1 and C_2

named(*M*,*name*) method/field *M* is called *name*

nonNull(*F*) the field *F* is non-null (i.e. it has been assigned an instance, either in the default field initialisation or in the constructor)

prefix(*M*,*name*) method/field *M* begins with the prefix *name*

related(C_1,C_2) the classes are related to each other, for example, by referencing the type in the source file

removes(*M*,*Type*,*Collection*) method *M* removes an instance of *Type* from *Collection*

returns(*M*,*F*) method *M* returns the value of field *F*

sameSignature(M_1,M_2) method M_1 has the same signature as method M_2

sameSupertype(C_1,C_2) class C_1 has the same supertype as class C_2

subtypeOf(*X*,*T*) *X* is a subtype of *T* (or equal to *T*)

typeOf(*F*,*T*) *F* is a field of type *T*

typeOf(*M*,*T*) *M* is a method that has a declared return type of *T*

C.2.2 Derived

and([*A*,*B*,*C*,**...**)] the conjunction of *A*,*B*,*C*...

implies(*A*,*B*) the logical equivalent of or(not(*A*),*B*)

isAbstract(*X*) *X* is a field, method, or type that is `abstract` – uses **hasModifier**

isFinal(*X*) *X* is `final` – uses **hasModifier**

isFriendly(*X*) *X* is `friendly` – uses **hasModifier**

isPrivate(*X*) *X* is `private` – uses **hasModifier**

isProtected(*X*) *X* is `protected` – uses **hasModifier**

isPublic(*X*) *X* is `public` – uses **hasModifier**

isStatic(*X*) *X* is `static` – uses **hasModifier**

not(*A*) the negation of *A*

or([*A***,***B***,***C***,...**)] the disjunction of *A*,*B*,*C*...

realises(*P*,*Set*) the participants in *Set* realise the design pattern *P* predicate

Bibliography

[AACGJ01] Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In *16th IEEE conference on Automated Software Engineering (ASE'01)*, pages 166–173. IEEE Computer Society Press, 11 2001.

[AISJ77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, and Max Jacobson. *A pattern language*. Oxford University Press, 1977.

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[ASF] Apache Software Foundation ASF. Apache Byte-Code Engineering Library. `http://jakarta.apache.org/bcel/`.

[Ban98] Jagdish Bansiya. Automating design-pattern identification. *Dr Dobb's Journal*, June 1998. `http://www.ddj.com/articles/1998/9806/9806a/9806a.htm?topic=patterns`.

[Bar76] J. Barwise. *Handbook of Mathematical Logic*. Elsevier Science Pub Co, 1976. ISBN 0444863885.

[BBS01] Alex Blewitt, Alan Bundy, and Ian Stark. Automatic verification of Java design patterns. In *ASE 2001: Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pages 324–327. IEEE Computer Society Press, November 2001. `http://www.ed.ac.uk/~stark/autvjd.html`.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change (XP Series)*. Addison-Wesley, 1999.

[Bec02] Kent Beck. *Test Driven Development*. Addison-Wesley Professional, 2002.

[Ble00] Alex Blewitt. A formal catalogue of design patterns. Technical Report Blue book note 1373, Division of Informatics, Edinburgh University, 7 2000.

[Blo01] Joshua Bloch. *Effective Java*. Java Series. Addison Wesley, 2001.

[BMZ00] Alan Bundy, Johanna D. Moore, and Claus Zinn. An intelligent tutoring system for induction proofs. In E. Melis, D. Scott, et al., editors, *CADE-17 Workshop on Automated Deduction in Education*, pages 4–13, Pittsburgh, USA, June 2000.

[Bro96] K. Brown. Design reverse-engineering and automated design pattern detection in Smalltalk. Technical Report TR-96-07, Unviersity of Illinois at Urbana-Champaign, 1996.

[BSvH+93] Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: a heuristic for guiding inductive proofs. In *Journal of Artificial Intelligence*, volume 62, pages 185–253, 1993.

[Bus96] Frank Buschmann. *Pattern-oriented Software Architecture: A System of Patterns*. Wiley, April 1996.

[BvHSH90] Alan Bundy, Frank van Harmelen, Alan Smaill, and Christian Horn. The oyster-clam system. In M.E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Design*, pages 647–648. Springer-Verlag, 1990.

[CAB+86] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[CHWC05] Andy Clement, George Harley, Matthew Webster, and Adrian Colyer. *Eclipse AspectJ: Aspect-Oriented Programming*. Addison Wesley, 2005.

[Coh97] R. Cohen. The Defensive Java Virtual Machine, 1997. `http://www.cli.com/software/djvm/`.

[Cop95] Jim Coplien. *Pattern Languages of Program Design 1*. Pattern Languages of Program Design. Addison Wesley, June 1995.

[DE97] Sophia Drossopoulou and Susan Eisenbach. Java is type safe — probably. *Lecture Notes in Computer Science*, 1241:389, 1997. `http://citeseer.nj.nec.com/drossopoulou96java.html`.

[DLNS98] David Detlefs, K. Rustan Leino, Greg Nelson, and James Saxe. Extended static checking. Technical Report 159, Systems Research Center, Digital Equipment Corporation, December 1998.

[DM79] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. In *Communications of the ACM*, volume 22, pages 465–476, August 1979.

[DV00] Sophia Drossopoulou and Tanya Valkevych. Java exceptions throw no surprises. Technical report, Department of Computing, Imperial College of Science, Technology and Medicine, 3 2000. `http://www-dse.doc.ic.ac.uk/projects/slurp/pubs.html`.

[DVE00] Sophia Drossopoulou, Tanya Valkevych, and Susan Eisenbach. Java type soundness revisited. Technical report, Department of Computing, Imperial College of Science, Technology and Medicine, 11 2000. `http://www-dse.doc.ic.ac.uk/projects/slurp/pubs.html`.

[Ede98] Amnon Eden. LePUS - a declarative pattern specification language. Technical report, Tel Aviv University, 1998. `http://www.cs.concordia.ca/~faculty/eden/lepus/`.

[Ede00] Amnon Eden. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD thesis, Department of Computer Science, Tel Aviv University, 2000. `http://www.eden-study.org`.

[EGY97] Amnon Eden, Joseph Gil, and Amiram Yehudai. Precise specification and automatic application of design patterns. In *12th Annual Conference of Automated Software Engineering*, 1997.

[Ehm] Dirk Ehms. Patternbox eclipse tool. `http://www.patternbox.com`.

[EI04] Bill J. Ellis and Andrew Ireland. An integration of program analysis and automated theorem proving. In E.A. Boiten, J. Derrick, and G. Smith, editors, *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM-04)*, number 2999 in Lecture Notes in Computer Science. Springer Verlag, 2004.

[FLN+02] Cormac Flanagan, K. Rustan M. Leino, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, June 2002. `http://www.hpl.hp.com/personal/Mark_Lillibridge/ESCOverview/revised-p25-leino.pdf`.

[FMvW97] Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool support for object-oriented patterns. In *Proceedings of ECOOP 97*, pages 472–498, 1997.

[Fow00] Martin Fowler. *Refactoring: Improving the design of existing code*. Object Technology Series. Addison-Wesley, 2000.

[FS03] Martin Fowler and Kendall Scott. *UML Distilled*. Addison-Wesley Professional, September 2003.

[GB] Erich Gamma and Kent Beck. JUnit cookbook. `http://www.junit.org`.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Professional Computing Series. Addison Wesley, 1995. ISBN 0-201-63361-2.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996. `http://java.sun.com/docs/books/jls/html/index.html`.

[GR83] Adele J. Goldberg and David Robson. *SmallTalk-80: The Language and its Implementation*. Addison Wesley, Reading, MA, 1983.

[Gru97] Dennis Gruijs. A framework of concepts for representing object-oriented design patterns. Technical Report INF-SCR-97-28, Utrecht University, November 1997.

[Gru98] Dennis Gruijs. *A Framework of Concepts for Representing Object-Oriented Design and Design Patterns*. PhD thesis, Department of Computer Science, Utrecht University, August 1998.

[HHF+97] Peter Heuchert, Frederik Hæsbrouck, Norio Furukawa, Ueli Wahli, Christian Michel, and Mike Cowlishaw. *Creating Java Applications using NetRexx*. Number SG24-2216-00 in IBM Redbooks. IBM, September 1997. `http://www.redbooks.ibm.com/redbooks/SG242216.html`.

[HHJT98] Ulrich Hensel, Marieke Huisman, Bart Jacobs, and Hendrik Tews. Reasoning about classes in object-oriented languages: Logical models and tools. *Lecture Notes in Computer Science*, 1381:105–121, 1998.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. In *Communications of the ACM*, volume 12(10), pages 576–580,583, October 1969.

[Hor88] Christian Horn. The Oyster proof development system, 1988.

[IB96] Andrew Ireland and Alan Bundy. Productive use of failure in inductive proof. In *Journal of Automated Reasoning*, volume 16, pages 79–111. Kluwer Academic, 1996.

[Jav97] Java Beans specification. Internet, July 1997. `http://www.javasoft.com/beans/docs/spec.html`.

[Joh99] Kim Howard Johnson. *The First 28 Years of Monty Python*. St. Martin's Press, 11 1999.

[JP00] Bart P. F. Jacobs and Erik Poll. A logic for the Java Modeling Language JML. Technical Report CSI-R0018, Computing Science Institute, University of Nijmegen, December 2000. `http://www.cs.kun.nl/csi/reports/info/CSI-R0018.html`.

[JP03] Bart Jacobs and Erik Poll. Java Program Verification at Nijmegen: Developments and Perspective. Technical Report NIII-R0318, University of Nijmegen, Netherlands, 2003. `http://www.cs.ru.nl/research/reports/info/NIII-R0318.html`.

[JvdBH+98] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Ulrich Hensel, and Hendrik Tews. Reasoning about Java Classes. In *Object-Oriented Programming Systems, Languages and Applications*, pages 329–340. ACM Press, 1998.

[KMM00] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moor, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. ISBN: 0-13-110362-8.

[Kra98] R. Kramer. iContract - The Java Design by Contract Tool. In *Proceedings of the Technology of Object Oriented Languages and Systems*. IEEE Computer Society, 1998.

[LCSV96] Helen Lowe, Andrew Cumming, Michæl Smyth, and Alison Varey. Lessons from experience: Making theorem provers more co-operative. In *Second International Workshop on User Interfaces for Theorem Provers*, pages 67–74, July 1996. `http://citeseer.ist.psu.edu/123678.html`.

[LD97] Helen Lowe and David Duncan. XBarnacle: Making theorem provers more accessible. In *Conference on Automated Deduction*, pages 404–407, 1997. `http://citeseer.ist.psu.edu/lowe97xbarnacle.html`.

[Lei97] K. Rustan Leino. Ecstatic: An object-oriented programming language with axiomatic semantics. In *4th International Workshop on Foundations of Object Oriented Languages*, January 1997.

[LK98] Anthony Lauder and Stuart Kent. Precise visual specification of design patterns. In *Proceedings of the European Conference on Object-Oriented Programming*. LNCS, July 1998.

[LNS00] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Report 2000-002, Compaq Systems Research Center, October 2000. `http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-2000-002.html`.

[LS97] K. Rustan M. Leino and Raymie Stata. Checking object invariants. Technical Report 1997-007, DEC Systems Research Center, January 1997. `http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-1997-007.html`.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996. `http://java.sun.com/docs/books/vmspec/html/index.html`.

[Mei96] Marco Meijers. Tool support for object-oriented design patterns. Master's thesis, Utrecht University, 1996. INF-SCR-96-28.

[Mey99] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1999.

[Mey02] Bertrand Meyer. *Design by Contract*. Prentice Hall, 2002.

[MLMN99] Bob Maatta, Leonardo Llames, Jennifer Maynard, and Mohammad Omar Nishtar. *Building AS/400 Applications with Java*. Number SG24-2163-02 in IBM Redbooks. IBM, 1999. `http://www.redbooks.ibm.com/redbooks/SG242163.html`.

[Moo93] Johanna D. Moore. What makes human explanations effective? In *Proceedings of the Fifteenth Annual Meeting of the Cognitive Science Society*, Hillsdale, NJ, 1993. Lawrence Erlbaum Associates.

[Nel80] Greg Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1980.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[NvO98] Tobias Nipkow and David von Oheimb. $Java_{light}$ is type-safe — definitely. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 161–170. ACM Press, New York, 1998. http://isabelle.in.tum.de/Bali/papers/POPL98.html.

[NvOP02] Tobias Nipkow, David von Oheimb, and Cornelia Puscha. microJava: Embedding a programming language in a theorem prover. In *Foundations of Secure Computation*. IOS Press, January 2002. `http://isabelle.in.tum.de/verificard/`.

[OC00] Mel Ó Cinnéide. *Automated Application of Design Patterns: A Refactoring Approach*. PhD thesis, University of Dublin, Trinity College, October 2000.

[OCN99] Mel Ó Cinnéide and Paddy Nixon. A methodoloy for the automated introduction of design patterns. In *Proceedings of the International Conference on Software Maintenance*, Oxford, September 1999.

[Ohe01] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. `http://www4.in.tum.de/~oheimb/diss/`.

[OJ90] William Opdyke and Ralph Johnson. Refactoring: an aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, September 1990.

[Opd92] W. F. Opdyke. *Refactoring Object-Oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992. `ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z`.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[Par] Terrence Parr. Antlr specification. Web. `http://www.antlr.org`.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994.

[PC00] Michæl P. Plezbert and Ron K. Cytron. Recognition and verification of design patterns. Technical Report 00-01, Department of Computer Science, Washington University in St. Louis, January 2000. `http://www.cs.wustl.edu/cs/techreports/2000/wucs-00-01.ps.Z`.

[Pus99] Cornelia Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. *Lecture Notes in Computer Science*, 1579:89–103, 1999. `http://citeseer.nj.nec.com/pusch99proving.html`.

[RBJ97] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. In *Theory and Practice of Object Systems*, volume 3, 1997.

[Rig03] Roger Riggs, editor. *Programming Wireless Devices with the Java2 Platform Micro Edition*. Addison-Wesley, June 2003. ISBN: 0321197984.

[SLU] SLURP. Sound languages underpin reliable programming. `http://www-dse.doc.ic.ac.uk/projects/slurp/index.html`.

[SM01] Stephen Stelting and Olav Maassen. *Applied Java patterns*. Java series. Prentice Hall, December 2001.

[Sym97] Donald Syme. Proving Java type soundness. Technical Report 427, University of Cambridge, Computer Laboratory, June 1997.

[Sym98] Donald Syme. *Declarative Theorem Proving for Operational Semantics*. PhD thesis, University of Cambridge, Computer Laboratory, 1998.

[TB01] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. In *Automated Software Engineering*, pages 89–120, 2001.

[Tok99] Lance Tokuda. *Evolving Object-Oriented Designs with Refactorings*. PhD thesis, The University of Texas at Austin, December 1999.

[Vli96] John Vlissides. *Pattern Languages of Program Design 2*. Pattern Languages of Program Design. Addison Wesley, June 1996.

[Vli98] John Vlissides. *Pattern Hatching: Design Patterns Applied*. Software Pattern Series. Addison Wesley, July 1998.

[Win93] Glynn Winskel. *The formal semantics of programming languages*. MIT Press, February 1993.

[WK03] Jos Warmer and Anneke Kleppe. *Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Professional, August 2003.

Index