Automatic Verification of Java Design Patterns

Alex Blewitt, Alan Bundy, Ian Stark
Division of Informatics, University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN, UK
Alex.Blewitt@ioshq.com
A.Bundy@ed.ac.uk
Ian.Stark@ed.ac.uk

Abstract

Design patterns are widely used by object oriented 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.

We attempt to show that many patterns (implemented in Java) can be verified automatically. Patterns are defined in terms of variants, mini-patterns, and constraints 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.*

1. Introduction

A *design pattern*, or simply a *pattern* is a specification of a set of classes and methods that work together to achieve a specific effect. Patterns have been studied for a number of years and are well known by the object oriented design community.

It is desirable to be able to verify that a pattern has been correctly implemented. Although designers may understand patterns well, developers may not have as much experience with them. This leads to the possibility that the pattern will be implemented incorrectly, or that coding errors which break a pattern may be introduced at a later stage.

In this paper, we present a way of representing patterns in a language called SPINE such that they may be verified by an automated proof tool, HEDGEHOG.

2. Design patterns

There are a number of books which catalogue and describe design patterns [4, 1, 6] including an informal description of the key features and examples of their use. However, at the moment there are no books which attempt to formalise these descriptions, possibly for the following reasons:

1. The implementation (and description) of the pattern is language-specific.
2. There are often several ways to implement a pattern within the same language.
3. Formal language descriptions are not common within the object oriented development community.

Instead, each pattern is presented as a brief description, and an example of its implementation and use. Designers and developers are then expected to learn the 'feel' of a pattern, without referring to any formal specification.

2.1. Types of pattern

In [4], patterns are broadly categorised into three different types:

Creational Patterns which are used to create new instances (such as `Factory`)

Structural Patterns that constrain classes' structural relationships (such as `Bridge`)

Behavioural Patterns whose implementation performs a common solution (such as `Command`)

The implementation of the patterns for each of these groups is not significantly different; the key difference is the pattern's *intent*. Most patterns (in [4], for instance) fall into the *Structural* category; few fall into the *Creational* category, and only a small number exist in the *Behavioural* category.

2.2. Implementation of patterns

The implementation (and specification) of a pattern is dependent on the target language. Although most object oriented languages support common features (inheritance, dynamically dispatched methods, instance variables), there are subtle differences (automatic or manual memory management, strong or weak typing, compile-time or run-time linking) which can have a profound affect on the way that a pattern may be implemented.

For example, when implementing the `Proxy` pattern in a untyped language (such as Smalltalk), only a single method (`doesNotUnderstand`) has to be implemented. (This method is automatically called whenever a Smalltalk instance receives a message that it does not understand.) In contrast, in a typed language (such as Java), a separate method has to be implemented for each message that it may receive.

Given such differences between target languages, it is not possible to create a concrete specification of patterns for all languages. The two solutions that present themselves are;

- **Create an abstract specification for patterns.**

  Eden has proposed a formalised graphical representation language for patterns, LePUS [3], which relates patterns at a higher level. In his language, a `Proxy` could be formulated graphically, with a rough translation of 'All messages received by the proxy are forwarded to the delegate'. However, such abstract statements may not be easy to verify.

- **Create a concrete specification for patterns for a specific target language.**

  Unlike an abstract approach, creating a concrete set of definitions for a target language can use some language-specific knowledge of how the pattern may be implemented. Instead of verifying an abstract specification, it can refer to ways in which the pattern may be implemented, and perform static analysis on the code itself.

HEDGEHOG takes the latter approach, using Java as the target language. In order to do this, the pattern must first be specified.

2.3. Constraints

A pattern specification may be thought of as a set of constraints which defines associations between classes, or between methods within a class. We categorise these constraints into:

Structural constraints How classes are related to each other through inheritance or polymorphic interfaces

Static semantic constraints How classes are related to each other by one-to-one or one-to-many relationships, and whether these are required or optional

Dynamic semantic constraints How implementations of particular methods operate; for example, a method may instantiate other classes, or may affect the runtime relationship between classes

A pattern specification can then be defined as a set of constraints on one or more classes. For example, in the `Proxy` pattern may be defined as:

- Class *C* implements interface *I* (structural constraint)
- Class *C* has instance variable *V* of type *I* (structural constraint)
- *V* cannot be `null` (semantic constraint)
- For each method *M* in interface *I*, *C.M* invokes *V.M* (semantic constraint)

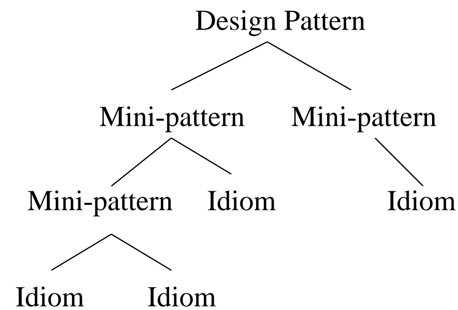

Figure 1. Relationship between patterns, mini-patterns and idioms

A number of these constraints may be combined to create a *mini-pattern* [2] or an *idiom* (see Figure 1). These mini-patterns may then in turn be used in the specification of other patterns. For example, the constraint "*V* cannot be `null`" is a common requirement across several patterns. This can be specified as a mini-pattern:

- *V* is assigned a non-`null` value during construction; and
- *V* is not updated by any other method

However, there are other possible implementations; for example, for any method, the variable is never assigned a (non-`null`) value.

2.4. Variations

Since there is generally more than one way to implement a pattern, we must formalise the specifications to allow different *variations* of a pattern. In order to show that a class *C*

is a Proxy, it is sufficient to show that the class implements InterfaceProxy (shown above) or SubclassProxy (not shown).

Another example is the Singleton pattern, which can be defined by several variations; the LazySingleton (which lazily instantiates the singleton instance), the FinalFieldSingleton (which instantiates a final field during construction) or the StaticSingleton (which is an implicit singleton since all the methods being static).

3 Formal specifications of patterns

In order to specify a pattern, we must first have a language for describing it. We have created SPINE, a typed first-order logic for describing patterns.

The format of the language is similar to Prolog; atomic terms start with a lower case letter, variables start with an upper case letter, and compound terms are denoted using parentheses to indicate sub-terms. However, unlike Prolog, each term has an associated type. The typing system is rich, and includes both primitive types such as boolean as well as Java meta-types such as Class, Method and Field.

Additionally, terms may be evaluated to result in a value. For example, the SPINE term "methodsOf('java.lang.String')" can be evaluated to return a list of Methods. This can be used with the forAll term, such as "forAll(M in methodsOf('java.lang.String'), hasModifier(M , public))"

It is then possible to encode pattern specifications using this language, as shown in Figure 2.

```
forAll(F in fieldsOf(ClassName),
  or(
    hasModifier(F,static),
    hasModifier(F,final),
    and(
      hasModifier(F,private),
      forAll(M in methodsOf(ClassName),
        not(modifies(M,F))
      )
    )
  )
)
```

Figure 2. Specification of the Immutable pattern

Each constraint in the specifications falls into one of the categories mentioned in Section 2.3; for example, hasModifier() is a **structural** constraint, whilst modifies() is a **semantic** constraint.

Note that it is possible to discharge some **semantic** constraints statically. For example, if a method does not contain any assignments, nor does the method call any other methods which contain assignments, then it cannot modify any variable.

We distinguish between the two types of **semantic** constraints as **static semantic** and **dynamic semantic**. Whilst both are associated with run-time semantics, it is possible to verify the former statically. This is an important consideration, because static semantic analysis is significantly easier than the dynamic semantic analysis.

Furthermore, most patterns do not require dynamic semantics to be specified. Although this is not a complete approach, it still has sufficient power to be able to prove many examples of real-world pattern uses.

4. HEDGEHOG

We have created a proof system HEDGEHOG, which is capable of reasoning about a simple subset of the Java language and associated patterns. It considers structural constraints (packages, inheritance, instance methods and fields, class methods and fields), and some static semantic constraints (such as modifies, forwards, nonNull and inert – a method is inert if it does not change or cause the change of any instance variable). It does not consider all aspects of the Java language; notably absent is support for inner classes and synchronization.

The proof system attempts to prove that a pattern instance is correctly implemented by a given class or set of classes. This may be annotated in the source code as a comment (such as @pattern: Singleton) which would allow the tool to scan the source for suspected patterns and to perform automatic verification.

Structural analysis allows HEDGEHOG to prove simple Java-related constraints, such as inheritance relationships and interface implementations. These requirements can be matched by direct appeal to the structure of the Java class.

Full-blown Java semantics are not necessary to prove all patterns; instead, weaker specifications can be proven. Such constraints can be proven based on the static structure of the code; for example, the expression "new Object()" can never return a null value.

5. Results so far

A manual process of searching the Java Language source code [5] resulted in a set of 15 different types of pattern that can be found in many of the core classes. Some patterns occur infrequently; for example, the Singleton pattern is implemented by System, Runtime and few others; whereas others occur frequently; for example, the Factory pattern is implemented by URL, Date, String, Color and many others.

These examples were then processed with HEDGEHOG tool to determine whether or not the pattern instances were correctly implemented. All of the `Singleton` and `Immutable` instances were correctly verified, as were most of the `Factory` instances.

An unexpected and interesting result occurred verifiying the instance of the `Bridge` pattern in the AWT (GUI) `Component` classes. The specification of the `Bridge` pattern states that for every subclass of an abstract data structure *ADT*, there exists a corresponding subclass of *ADTPeer*. Whilst the original Java AWT used the peerage system, the newer Swing user interface does not. Since the `JComponent` Swing user interface component is a subclass of `Component`, the newer Swing classes break this pattern. Therefore, the Java AWT classes do not fully implement the `Bridge` pattern.

Of the different types of patterns described in Section 2.1, the most successful results have been obtained with the structural category of patterns. In part, this is because the logic for proving classes implement a structural pattern is much simpler than for those that implement specific operations.

Success has been made with creational patterns such as `Factory` and `Pool`. However, it is not as simple to prove that these patterns are correct, which explains why some of the patterns can be verified and some cannot within the Java language source code.

No success has been made with the behavioural patterns. In most cases, this is due to difficulty in specifying the behaviour for the classes. As an example, the `Command` pattern is almost impossible to verify, since there are very limited constraints which can be identified in such a pattern. Instead, most of the pattern is encoded in the actual intent of the command implementation. Since the commands can implement radically different operations (for example, opening a file or printing a document) it is very hard to encode a specification that could be used to verify it sensibly.

6. Conclusion

We have presented a methodology for verifying patterns in Java code, using a set of declarative specifications of design patterns and variations.

This methodology has been applied successfully to both structural and creational patterns such as `Bridge` and `Factory`, and been used to verify the existence of pattern implementations in the Java Language source. It has also been used to discover an incomplete pattern implementation.

Restricting the verification to a particular target language (Java) means that patterns can be more strongly specified than if a generic approaches such as LePUS are used. These stronger specifications then allow more instances of patterns to be verified. It may be possible to use the same methodology with other target languages (such as Smalltalk or C++) by redefining the patterns' specification.

Specifying the patterns as structural and static semantic constraints allow the pattern to be compared directly with the source code. Even with this incomplete specification, many different patterns can be verified. The main problem with this approach is that it suffers when a pattern's implementation does not match that of a known specification.

Since the pattern specifications (and variants) are defined in external text files, the system is extensible and allows the user to add the definitions of new patterns at a later stage.

7. Acknowledgements

This paper was funded under EPSRC grant number 98315335 in conjunction with International Object Solutions Limited. We are also grateful for the help of Richard Boulton who helped with earlier work.

References

[1] F. Buschmann. *Pattern-oriented Software Architecture: A System of Patterns*. Wiley, April 1996.

[2] M. Cinnéide and P. Nixon. A methodoloy for the automated introduction of design patterns. In *Proceedings of the International Conference on Software Maintenance*, Oxford, September 1999.

[3] A. Eden. *LePUS - A Declarative Pattern Specification Language*. PhD thesis, Department of Computer Science, Tel Aviv University, 1998. http://www.cs.concordia.ca/~faculty/eden/lepus/.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Professional Computing Series. Addison Wesley, 1994. ISBN 0-201-63361-2.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996. http://java.sun.com/docs/books/jls/html/index.html.

[6] J. Vlissides. *Pattern Hatching: Design Patterns Applied*. Software Pattern Series. Addison Wesley, July 1998.