Report Number: WUCS-00-01

2000-01-01

Recognition and Verification of Design Patterns

Michael P. Plezbert and Ron K. Cytron

In this paper we consider the automatic discovery of design (programming) patterns. While patterns have surfaced as an effective mechanism for authoring and understanding compelx software, popular languages lack facilities for direct specification of patterns or verification of pattern usage in program specifications. Static analysis for patterns is provably undecidable; we focus on discovery and verification of patterns by analyzing dynamic sequences of method calls on object. We show a proof-of-concept of our approach by presenting the results of analyzing a Java program for Iterator patterns.

Recognition and Verification of Design Patterns

Michael P. Plezbert and Ron K. Cytron

WUCS-00-01

January 2000

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130

Recognition and Verification of Design Patterns

Michael P. Plezbert* and Ron K. Cytron
Washington University in St. Louis
Dept. of Computer Science
St. Louis, MO 63130

January 23, 2000

Abstract

In this paper we consider the automatic discovery of design (programming) patterns. While patterns have surfaced as an effective mechanism for authoring and understanding complex software, popular languages lack facilities for direct specification of patterns or verification of pattern usage in program specifications. Static analysis for patterns is provably undecidable; we focus on discovery and verification of patterns by analyzing dynamic sequences of method calls on objects. We show a proof-of-concept of our approach by presenting the results of analyzing a Java program for Iterator patterns.

1 Introduction

Design patterns have quickly gained prominence as an effective approach to authoring and understanding complex software. [7] However, current popular programming languages offer little in the way of direct support for patterns; and while languages which incorporate patterns as first-class elements are likely to exist in the future, they are still a long way off.

While patterns elevate the level of programming-language discourse, there is currently no mechanism for discovering or verifying a given pattern in a program. In particular, we seek to answer the following questions:

- Does pattern P exist in a program?
- Do classes $C_1, C_2, \ldots, C_n$ implement pattern P?
- At what point does a program *fail* to manifest pattern P?
- What patterns are in a program?

To illustrate the possible benefits of such an analysis, we begin with an example. The experiments we describe in this paper included analysis of whether the *Iterator* pattern exists in a program. We applied this analysis to some packet-filtering software, and informed the author of his Iterators. The author responded that there was a class C that implements iteration, but we missed it. Our analysis was then able to show that class C was, in fact, not an Iterator. Moreover, we could show the programmer where the class failed to implement the Iterator pattern. In this case, the class accessed

*Contact author: plezbert@cs.wustl.edu. This work is supported by the National Science Foundation under grant NSF 9711531.

data without prior test that data was present—a violation of the Iterator pattern and a bug in the program. While it is likely that these errors eventually would have been caught without our analysis, the automatic detection of the errors helped make the difficult task of debugging easier.

In this case, patterns elevated the level of program debugging: it is unlikely that a bug of this nature could be discovered by conventional debugging tools. The program was valid; no exceptions were thrown. Instead, the errors were simple misuses of the objects in question; methods were being called in a manner that could lead to undefined behavior.

In this paper, we investigate the extent to which patterns can be automatically discovered and verified in programs. Ideally, we would offer static analysis that could show that some pattern P holds over all executions of some program. Unfortunately, this approach quickly subsumes undecidable problems. We focus instead on showing that in a *given execution*, a given pattern exists or does not exist. We can catch violations of a given pattern in a run, but we cannot guarantee that a pattern holds over all runs. Nonetheless, when a program is executed repeatedly, the body of evidence that a given patterns holds will increase. This will suggest optimization opportunities, as described in Section 2.

Our paper is organized as follows. Section 2 presents the background for this problem. Section 3 describes a mechanism for representing patterns that suffices for discovery of three simple patterns: Singleton, Iterator, and Adapter. Section 4 describes our implementation for these patterns and reports on our success in finding these patterns in software.

2 Overview

We begin by examining the potential benefits from having a mechanism that recognizes or verifies patterns in software.

Debugging: The automatic recognition and verification of design patterns affords many benefits for testing and debugging software. These include:

- Detection of incorrect use of objects or classes. A programmer may assert that a set of classes possesses a certain pattern. If it can be shown that the pattern is not faithfully represented, then this may reveal a bug in the program. Our example in Section 1 illustrates this point.
- Suggestion of explicit use of a pattern when appropriate. A set of objects may behave according to a given pattern. If this can be recognized, then perhaps language constructs could help enforce the continuation of the pattern. For example, consider a class that is supposed to be instantiated only once (*i.e.*, the *Singleton* pattern). If the pattern recognizer detects the Singleton pattern, then the program might be revised so it is impossible to instantiate the relevant class more than once. Such revision serves to annotate the code more appropriately as well as guard against future modifications that might violate the pattern.

Optimization: Pattern analysis can reveal the interaction of software components at a very high level. Program optimization can then be applied to improve the performance of these components once the pattern is recognized. Some possibilities are:

- Replace an inefficient implementation with an efficient one. A pattern such as Iterator can have many implementations. Once the pattern is known, this information could participate in automatic algorithm choice, similar to the automatic data structure choice problem [2].

```
public interface Enumeration {
  boolean hasMoreElements();
  Object nextElement();
}
```

Figure 1: The `Enumeration` Interface

```
Enumeration enum = vector.elements();
while (enum.hasMoreElements()) {
  System.out.println(enum.nextElement());
}
```

Figure 2: Typical `Enumeration` use.

- Customize a pattern for a particular use. Java offers an `Enumeration` type, which iterates only once. In some applications, a more efficient enumerator would be resettable, so that it could re-enumerate as many times as needed. Pattern analysis can help show that a `RecyclableEnumeration` can replace an `Enumeration` type. This point is considered in greater detail below.
- The Adapter pattern interposes a layer of software. For efficiency, this layer could be in-lined and optimized, once the pattern is known.
- A Singleton object can be instantiated only once. Thus, global variables or registers could be devoted to referencing such objects, once the pattern is known.

Consider the simple case of the Java `Enumeration` interface, shown in Figure 1. The `Enumeration` is a simple implementation of the Iterator pattern. Iterators are used to iterate over aggregate structures or collections of objects.

`Enumerations` are meant to be used in a very specific manner. The typical usage is illustrated in Figure 2. In particular, there is a prescribed ordering in which the methods are to be called: `hasMoreElements()` should be called before every call to `nextElement()`, and `nextElement()` should only be called if the preceding call to `hasMoreElements()` returned `true`. While it may be safe to violate this rule in specific instances, in general the behavior is undefined if `nextElement()` is called without a preceding call to `hasMoreElements()`. Unfortunately, Java, like most popular programming languages, has no mechanism for specifying or enforcing such usage constraints.

While in this particular example an `Enumeration` class could conceivably be implemented in such a way as to enforce proper usage by keeping track of the last method called in its internal state, such an approach adds complexity, incurs runtime cost, is prone to errors, and does not generalize well to more complex situations.

Consider the problem of repeatedly enumerating over the same space, such as the predecessors or successors of a node in an unchanging graph. If `Enumeration` is the abstraction for the Iterator, then each time enumeration is needed, a fresh `Enumeration` must be created.

The examination of the automatic verification of patterns naturally raises the question, is it possible to automatically find new patterns? That is, can we find heretofore unknown patterns using automatic techniques?

If we represent patterns as regular expressions, then the problem reduces to learning a regular set. Unfortunately, it has been shown that learning a regular set is inefficient under widely held assumptions [5]. There are several promising heuristic approaches, and it is currently an area of considerable active research [3].

In addition, our problem is made more difficult than the problem of learning a regular set, because we are interested in formulating regular sets from subsequences of a program trace. Thus, we do not have positive and negative examples for automatic learning, as would normally be the case.

3 Representing Patterns

The notion of a pattern, while meaningful in discourse, has not been set forth with a precision that allows the analysis we wish to undertake. Each pattern often has several variations and any number of possible implementations. Aspects such as intent and context often appear in informal descriptions of pattern applications. As such, patterns are currently described using prose, and will continue to be for the foreseeable future. [4]

While prose might serve when using patterns as design guides, such descriptions do not lend themselves well to automatic analysis. We need a more formal and precise representation. In this section, we describe our approach for recognizing and verifying patterns. For our purposes, a pattern's representation need not capture the entirety of a pattern's prose description. Aspects such as purpose and intent would not only be difficult to capture formally, but also are not necessary for our purposes. Instead, we need an approach that is more "what I do" not "what I say." We therefore focus on capturing the actual behavior of the objects involved in implementing a pattern; the intention behind the pattern is secondary.

In general, three components contribute to our characterization of patterns:

- Object type
- Type placement in a type hierarchy
- Object behavior

The first two properties are static: an object's type and its placement in a type hierarchy do not change as a program executes. The third component, object behavior, is specific to a given run. Our idea is to characterize potential object behavior in terms of method calls. For the purposes of this paper, we limit ourselves to *regular expressions* over method calls, where the alphabet of the regular expression is matched to actual method names.

3.1 Patterns Specified by Regular Expressions

How can we specify a set of object behaviors that represents a programming design pattern? For the patterns we discuss in this paper, it (mostly) suffices to examine actual object behavior. Given a pattern specification and a trace of the methods called during an execution, we seek to verify that the methods do conform to the specification.

In this paper, we investigate the use of (paramaterized) regular expressions to designate a desired pattern of method calls and returns. In our notation, a capital letter designates a method call and its lower case letter designates a return from the method. Continuing our Iterator example, we could attempt to represent the Iterator pattern used by the Java `Enumeration` with a regular expression such as

$$C\,c\,H\,h\,(N\,n\,H\,h)^*$$

where $C\ c$ represents calling and returning from the constructor, $H\ h$ represents a call/return for `hasMoreElements()`, and $N\ n$ represents a call/return for `nextElement()`.

For the Iterator, we need not distinguish method call from method return, since the pattern contains no nesting. However, other patterns (for example, the Adapter, which is described below) require separating the two and representing them individually.

The destruction of the object is not included in the regular expression, in part because Java does not have explicit destructors. The call to `finalize()` could have been included, but would not serve much purpose—we can simply assume an implicit call.

It might seem that including the constructor in the regular expression is unnecessary, also. If an implied destructor, why not an implied constructor? In this particular case, the inclusion of the constructor is not absolutely necessary. However, it helps clarify that the regular expression is meant to match each individual instance of an iterator class.

For example, consider the Singleton. The basic Singleton pattern might be represented as $C\ c\ (\overline{C \mid c})^*$, where C again corresponds to the constructor, and $\overline{C}$ means any method *except* the constructor. Unlike the Iterator, where the regular expression was meant to represent the behavior of *each* iterator object, the Singleton expression is meant to match over *all* instances of a given class (though with a singleton, there should be only one).

The adapter pattern might be represented as $(A\ B\ b\ a)^+$. In this case, the regular expression is meant to map to each method of a given class. In other words, class X adapts class Y if the methods of X and Y can be matched so that each pair satisfies the expression.

Determining if a method trace matches the regular expression for a pattern involves more than simply determining if a string fits a regular expression. The main difficulty lies in determining the correspondence of actual methods to the parametric alphabet characters in the regular expression. Also, the pattern we seek is a *subsequence* of the complete method trace.

A formal statement of our problem is as follows:

> Consider an alphabet $\Sigma = \{a, b, c, \ldots\}$, a string S over Σ, an alphabet $\Gamma = \{\alpha, \beta, \gamma, \ldots\}$ (where $|\Gamma| \leq |\Sigma|$), and a regular expression R over Γ. Is there a one-to-one mapping M from Γ to Σ such that the regular expression R is satisfied by S^M, where S^M is the *subsequence* of S consisting of all the characters from Σ used in the mapping M?

In general, this problem is NP-Complete (a proof is offered in the full paper). However, there are practical approaches to solving this problem for the patterns of interest, as follows:

- In practice, Γ is much smaller than Σ. Recalling the Iterator, Γ takes 3 calls and 3 returns to characterize the pattern, for a total of 6 symbols. A program may contain hundreds or thousands of actual methods. Matching Σ to Γ is an exercise of magnitude $C(|\Sigma|, |\Gamma|)$. This is a polynomial of degree Γ if Γ is constant. For small Γ, as with the Iterator, this can be reasonable.
- In practice, a matching tends to fail quickly, before much of the method call sequence is seen.

4 Implementation and Experiment

As proof of concept, we have developed software that will search a method trace for objects whose method calls match a regular expression. Our current implementation is very inefficient and must be tailored to each regular expression, but is meant primarily as a preliminary proof-of-concept.

A necessary prelude to performing this matching is the ability to generate the method traces. We have modified Sun Microsystems' `Java Virtual Machine` to generate traces of method calls. Any

```
Calling java.lang.Class@EE303A78.EE34E128 invokestatic_quick Tomita.TomitaParser.main([Ljava.lang.String;)V (1) EE3000A8
Calling java.lang.Class@EE303AA8.EE34E258 invokestatic_quick Engine.parser.<clinit>()V (0) EE3000A8
Leaving Engine.parser.<clinit>(parser.java:15) EE3000A8
Calling Engine.parser@EE303E20.EE34EB50 invokenonvirtual_quick Engine.parser.<init>(Ljava.lang.String;)V (2) EE3000A8
Calling java.io.FileInputStream@EE303E28.EE34EB88 invokenonvirtual_quick java.io.FileInputStream.<init>(Ljava.lang.String;)V (2) EE3000A8
Calling java.io.FileInputStream@EE303E28.EE34EB88 invokenonvirtual_quick java.io.InputStream.<init>()V (1) EE3000A8
Leaving java.io.InputStream.<init>(InputStream.java) EE3000A8
Calling java.lang.Class@EE300100.EE333F20 invokestatic_quick java.lang.System.getSecurityManager()Ljava.lang.SecurityManager; (0) EE3000A8
.
.
.
```

Figure 3: Java method trace

Java code that is run on our modified JVM will produce a trace of all methods called during execution of the program.

Figure 3 shows a few lines from a typical trace. As can be seen, the trace contains both entries into and exits from the methods that are called.

Our current implementation of the matching software uses JLex to perform the regular expression matching. JLex is a lexical analyzer generator for Java, in the tradition of lex and flex.

The current implementation of our software requires a separate JLex specification for each expression we wish to try to match. The specification used to search for Iterators in shown in Figure 4. As indicated in the figure, the methods are each mapped to a pair of characters (indicating method entry and exit). For example, the method hasMoreElements() is mapped to the character pair CD in Figure 4. The choice of characters is arbitrary, but it must be consistent with those assigned when processing the method trace, as descibed below.

JLex matches over strings, not subsequences. Therefore, in order to use JLex for our purposes, it is necessary to preprocess the method trace to extract only those objects against which a match is currently being attempted. For example, to test if object foo is an Iterator, it is necessary to extract all references to object foo from the method trace to create a subsequence of the trace consisting only of calls to the methods of foo. Figure 5 shows the extracted trace for a single instance of java.util.HashtableEnumerator from a run of the packet-filtering software.

This partial method trace is then mapped to a sequence of characters. The choice of characters is arbitrary, but must be consistent with those used in the JLex specification. The trace in Figure 5 would therefore be mapped to the string ABCDEFCD.

The lexical analyzer generated by JLex is then run with this string as input. It will return true if the input string satisfies the regular expression, indicating that the object for which the string was obtained adheres to the pattern represented by the regular expression. For example, the HashtableEnumerator which generated the trace in Figure 5 would be found to adhere to the Iterator pattern since the string ABCDEFCD would be accepted by the JLex specification given in Figure 4.

For patterns such as the Iterator, which are dependent on each individual instance of a class, this process of extracting the partial trace, converting it to a string, and feeding it to the lexical analyzer must be performed for *every* object of every class we wish to test. Then, if *all* instances of a given class adhere to a pattern, the class is assumed to implement that pattern.

The process for patterns such as Singleton, which encompass all instances of a class rather than individual instances, is similar except that the partial traces are extracted on a per-class basis rather than a per-instance basis.

In analyzing the packet filtering software mentioned in Section 1 and looking for Iterators, we found the following:

```
package Matcher;

import java.lang.*;

/**
 * The JLex specification for the Iterator pattern.
 */

%%

%integer
%yyeof

%{
public static final int MISMATCH = 0;
public static final int ITERATOR = 1;
%}

ObjectCreation = AB
HasMoreElements = CD
NextElement = EF

%%

{ObjectCreation}({HasMoreElements}{NextElement})*{HasMoreElements}\n { return ITERATOR; }

.\n { return MISMATCH; }
. { return MISMATCH; }
```

Figure 4: The JLex specification for the Iterator pattern.

```
Calling java.util.HashtableEnumerator@EE307140.EE3613D0 invokenonvirtual_quick java.util.HashtableEnumerator.<init>([Ljava.util.HashtableEntry;Z)V (3) EE3000A8
Leaving java.util.HashtableEnumerator.<init>(Hashtable.java) EE3000A8
Calling java.util.HashtableEnumerator@EE307140.EE3613D0 invokeinterface_quick java.util.HashtableEnumerator.hasMoreElements()Z (1) EE3000A8
Leaving java.util.HashtableEnumerator.hasMoreElements(Hashtable.java) EE3000A8
Calling java.util.HashtableEnumerator@EE307140.EE3613D0 invokeinterface_quick java.util.HashtableEnumerator.nextElement()Ljava.lang.Object; (1) EE3000A8
Leaving java.util.HashtableEnumerator.nextElement(Hashtable.java) EE3000A8
Calling java.util.HashtableEnumerator@EE307140.EE3613D0 invokeinterface_quick java.util.HashtableEnumerator.hasMoreElements()Z (1) EE3000A8
Leaving java.util.HashtableEnumerator.hasMoreElements(Hashtable.java) EE3000A8
```

Figure 5: Extracted single-object trace.

Total objects	2148
HashtableEnumerators	203
VectorEnumerators	55
False Iterators	122

Out of 2148 objects, 258 were Iterators and matched the expression $C\ c\ (H\ h\ N\ n)^*\ H\ h$. (And as mentioned in Section 1, two Iterator objects were being used incorrectly and did not match the expression.) But there were also 122 objects that matched the expression which were *not* Iterators.

Upon closer examination, 110 of these false Iterators were found to have "empty" iterations; that is, they matched the expression $C\ c\ H\ h$. *All* of these false positives could have been eliminated by examining the *type signatures* of the methods involved. For example, the method that matches with $H\ h$ (which corresponds to `hasMoreElements()` should take no arguments and return `Boolean`.

When searching for Singletons, 11 classes were found that fit the pattern; that is, they were instantiated only once. However, one of the those classes, namely `java.io.DataInputStream`, was not in actuality a Singleton—it simply happened to be instantiated only once in this particular program.

The current version of our software is extremely inefficient. It took over 30 minutes on an UltraSparc 1 to search for iterators with a trace file of about 42000 entries. However, most of this time is taken up by our ineffiicent method of extracting the subtraces and processing the trace files. We believe this time can be reduced sufficiently to allow interactive response times.

While it is an excellent tool, `JLex` is not well suited to the task at hand. In particular, there are three aspects of our problem that we need to address:

Dynamic creation of regular expressions: While we hope to create expressions for the "Gang-of-Four" 's 23 patterns, it is likely that the users of a tool such as we propose would wish to find other patterns. Our tool should allow the declaration of patterns at runtime without the need to pre-compile the patterns.

Matching over a subsequence of the method trace: Tools such as `JLex` typically match a regular expression to an input string. We require matches over *s*ubsequences of a string. Our current implementation (using `JLex`) requires that we create the possible subsequences beforehand, by preprocessing the method trace for every object or group of objects we wish to test. A solution which matched a regular expression to subsequences directly would be both more efficient and more flexible.

Mapping of re alphabet to methods: In the examples we have examined thus far, we have been able to use a "trick" to implicitly perform the matching of the regular expression alphabet to the method calls in the trace. This is an artifact of the regular expressions used thus far. Specifically, the particular regular expressions used allow us to put a fixed (partial) ordering to the alphabet. For example, all strings generated by the Iterator expression described in Section 3.1 will start with the prefix *CcHh* (or *CcHhNn* for strings over four characters long). This means that the first two (or three) method call/return pairs must match to these characters.

Our final implementation needs to be able to perform this matching for any arbitrary case. Unfortunately, as discussed in Section 3.1, the general matching problem is NP-Complete. However, we hope to be able to use heuristics based on the concrete problem to improve performance.

In our final implementation, we intend to create a solution that addresses all of these aspects.

Our regular expression pattern matching problem differs from classical regular expression matching in two ways: first, we are searching for subsequences rather than substrings, and second, we are looking for *any* (actually, *all*) mappings of the string alphabet to the regular expression alphabet that will cause the expression to be satisfied by the string.

One way of approaching this problem which addresses both of these aspects would be to simply create an NFA which encompasses all possibilities and then use standard methods for reducing the NFA to a DFA and scanning the string. However, with the alphabet matching problem being NP-Complete, this straightforward approach could be very inefficient.

Though our matching problem is NP-Complete in the general case, we hope to be able to use a couple of features of the concrete problem we are trying to solve to our advantage. First, the problem is actually only exponential in the size of Γ, the regular expression alphabet. In our design pattern matching problem, Γ is the (generic) methods of the objects that form the pattern for which we are searching; the size of Γ is therefore likely to be small. Second, it is likely that most of the possible matches can be eliminated quickly, in the first few steps in scanning the string.

The pattern matching can possibly be speeded up even more by preprocessing the regular expressions and/or the string. Though we wish to allow searching for arbitrary patterns in real-time, we can speed up the search for common patterns (or subsequent searches for arbitrary patterns) by preprocessing the expressions to facilitate the searching. Similarly, performing multiple searches on the same trace can be facilitated by processing the string.

5 Conclusion

We have introduced the problem of discovering design patterns automatically in programs. Our approach in this paper is quite simple, relying on regular expressions to find the patterns. In particular, we make no use of type information, by way of type signatures or type location within a type hierarchy (*i.e.*, "extends" information). For the three patterns described in this paper—Iterator, Adapter, and Singleton—the approach we propose in this paper suffices; experimentally, our results are positive for Iterator and Singleton. We have not tried as yet to find Adapter patterns. Future work will focus on the set of patterns defined in [4] and will incorporate type information to improve the precision.

Although our results are preliminary, they do indicate initial success at recognizing or verifying the existence of simple patterns in software. This step toward bringing patterns "down to earth" is necessary if programming languages are to include pattern assertion or specification as language features.

Acknowledgements

The authors thank John Vlissides and Josh Auerbach of IBM T. J. Watson Labs for discussions on this work.

References

[1] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2), 1996.

[2] J. Cai, P. Facon, F. Henglein, R. Paige, and E. Schonberg. Type analysis and data structure selection. In *Constructing Programs from Specifications*. North-Holland, 1991.

[3] Yoav Freund, Michael Kearns, Dana Ron, Ronitt Rubinfeld, Robert E. Schapire, and Linda Sellie. Efficient learning of typical finite automata from random walks. *ACM STOC*, 1993.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] Mike Kearns and Umesh Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[6] Lutz Prechelt and Ghristian Kramer. Functionality versus practicality: Employing existing tools for recovering structural design patterns. *Journal of Universal Computer Science*, 4(12):866–882, December 1998.

[7] John Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, 1998.