Reverse Engineering of Design Patterns from Java Source Code

Nija Shi and Ronald A. Olsson
Department of Computer Science
University of California, Davis
Davis, California 95616-8562 USA
{shini,olsson}@cs.ucdavis.edu

Abstract

Recovering design patterns can enhance existing source code analysis tools by bringing program understanding to the design level. This paper presents a new, fully automated pattern detection approach. The new approach is based on our reclassification of the GoF patterns by their pattern intent. We argue that the GoF pattern catalog classifies design patterns in the forward-engineering sense; our reclassification is better suited for reverse engineering. Our approach uses lightweight static program analysis techniques to capture program intent. This paper also describes our tool, PINOT, that implements this new approach. PINOT detects all the GoF patterns that have concrete definitions driven by code structure or system behavior. Our tool is faster, more accurate, and targets more patterns than existing pattern detection tools. PINOT has been used successfully in detecting patterns in Java AWT, JHotDraw, Swing, Apache Ant, and many other programs and packages.

1 Introduction

Program understanding tools today are able to extract various source information, such as class structures, inter-class relationships, call graphs, etc. Some may even produce a subset of UML diagrams. However, without proper documentation, it would still take a lot of effort for a developer to become proficient with the source code. Therefore, a powerful program understanding tool should be able to extract the intent and design of the source code. To fulfill this goal, we need some kind of code pattern that bears intent and design as source facts to analyze against. A design pattern abstracts a reusable object-oriented design that solves a common recurring design problem in a particular context [15]. A design pattern has its own unique intent and describes the roles, responsibilities, and collaboration of participating classes and instances. Thus, by extracting design patterns from source code, we are then able to reveal the intent and design of a software system.

Design patterns are typically used as guidelines during software development. Thus, the GoF book [15] presents a pattern catalog for forward engineering, but the same classification can be misleading for reverse engineering. Current approaches lack a proper pattern classification for reverse engineering. A pattern classification for reverse engineering should indicate whether or not each pattern is detectable and if there exist traceable concrete pattern definitions to categorize detectable patterns. Thus, we reclassified the GoF patterns into five categories in the reverse-engineering sense (see Section 4). Based on this reclassification, we automated the entire pattern recognition process using only static program analysis. This relatively simple approach has proven effective. We have some promising results — both accuracy and speed — from our initial prototype, PINOT (Pattern INference and recOvery Tool), in recovering design patterns from the Java AWT package, JHotDraw (a GUI framework), Swing, and Apache Ant.

The rest of this paper is organized as follows. Section 2 critiques current pattern detection tools. Section 3 presents examples that motivated our approach. Section 4 explains our reclassification of the GoF patterns. Section 5 illustrates how we identify structure- and behavior-driven patterns. Section 6 describes our initial prototype of PINOT Section 7 discusses results from using PINOT. Section 8 concludes the paper and covers our future work.

2 Critique of Current Approaches

Approaches to design pattern recognition fall into two main categories: those that identify the structural aspect of patterns and others that take a further step to distinguish the behavioral aspect of patterns.

2.1 Targeting Structural Aspects

These approaches analyze inter-class relationships to identify the structural aspect of patterns, regardless of their behavioral aspect. The targeted inter-class relationships include: class inheritance; interface hierarchies; modifiers of classes and methods; types and accessibility of attributes; method delegations, parameters and return types.

Some approaches first extract inter-class relationships from source code and then perform pattern recognition based on the extracted information. For example, DP++ [7], SPOOL [20], Osprey [5], and Reference [28] extract inter-class relationships from C++ source to a database; patterns are then recovered through queries to the database. Reference [13] combines the Columbus reverse-engineering framework with the MAISA architectural metrics analyzer (which analyzes software at the design level and had reported limited results on recovering anti-patterns [23]) to build a pattern recognizer. However, pattern recognition requires analyzing program behavior, which can be abstracted away at the design level. Reference [6] use the Columbus schema for the extracted abstract semantics graphs (ASG) and recover patterns based on graph comparison. Reference [4] extracts inter-class relationships and then uses software metrics to reduce search space. SOUL [11] is a logic inference system, which has been used to recognize patterns (in Java and SmallTalk) based on inter-class-based code idioms and naming conventions. SPQR [26] uses denotational semantics to find patterns on the ASG obtained by gcc. The accuracy of these approaches depends in part on the capability of the program facts extractors they use.

FUJABA [21] extends the work from [25] and uses a bottom-up-top-down approach to speed up the search and to reduce the false positive rate (due to more complicated inter-class relationship, such as aggregation [25, 21]). It uses a combination of inter-class relationships to indicate a pattern. Thus, when such information is obtained from the bottom-up search, even partially, FUJABA assumes the existence of a possible pattern and tries to complete the rest of the search — i.e., the top-down search — to confirm that such a pattern actually exists. This iterative approach allows going back to their annotated abstract syntax tree (AST) for further analysis on demand.

2.2 Targeting Behavioral Aspects

The approaches discussed in Section 2.1 are unable to identify patterns that are structurally identical but differ in behavior, such as State vs. Strategy and Chain of Responsibility (CoR) vs. Decorator. Approaches that target behavioral aspects seek to resolve this problem using machine learning, dynamic analysis, and static program analysis.

Machine Learning

These approaches attempt to reduce false positives by training a pattern recognition tool to identify the correct implementation variants of a pattern. Such approaches are semi-automatic: user intervention guides pattern recognition. Follow-on work of FUJABA [22] associates fuzzy values to pattern definitions. Pattern recognition is driven by a semi-automatic iterative process. PTIDEJ [3] recognizes distorted implementations of patterns, thus detected pattern instances are associated with a similarity rate. A related work of PTIDEJ [17] uses program metrics (such as size, cohesion, and coupling) and a machine learning algorithm to *fingerprint* roles of a pattern's participating classes. These fingerprints are learnable facts for the pattern constraint solver. Reference [12] (follow-on to [6, 13]) incorporates machine learning techniques to train its pattern recognition tool. Each pattern is defined with a set of predictors, whose values are used in the learning process. They tested their method on the Adapter and Strategy patterns.

Most GoF patterns (including the two above) have concrete definitions on their realization in code structure and system behavior. Such concrete definitions are traceable (see Section 4). Thus this category does not seem to solve the fundamental problem (see further Section 7).

Dynamic Analysis

These approaches use runtime data to help identify the behavioral aspects of patterns. KT [10] hard-coded its detection algorithms to search for patterns in programs written in SmallTalk. KT uses only dynamic analysis to identify the CoR pattern, but the result was unsuccessful, due to improper message logging mechanism and insufficient test data. Follow-on work to FUJABA [29] and Reference [19] suggest using dynamic analysis to analyze behavior. First, they obtain inter-class information from source code. Next, for a particular pattern, they compute a list of candidate classes. Then, assuming how these candidates should behave, they verify the behavior during runtime.

Dynamic analysis relies on a good coverage of test data to exercise every possible execution path; such test data is not often available. Even if test data is available in a distribution, the runtime results may be misleading since the data was not originally designed for recognizing the behavior of a particular pattern (e.g., a distribution might include a validation or benchmark suite). Moreover, dynamic analysis is not able to verify pattern intent that is not observable, such as verifying *lazy instantiation* and *single instance assurance* for the Singleton pattern,

Static Program Analysis

These approaches apply static program analysis techniques to the AST in method bodies. FUJABA, in its current implementation, identifies path-insensitive object creation statements for recognizing Abstract Factory and Factory Method patterns. Reference [9] is a design pattern verification tool for Java. The tool consists of the HEDGE-

HOG proof engine and the Prolog-like SPINE specification language. HEDGEHOG identifies inter-class relationships and then applies some inter-procedural but path-insensitive analysis techniques to verify some weak semantics (e.g., whether a method modifies the value of a field) defined in method bodies. SPINE is not able to capture program intent, thus patterns that are vaguely defined or lack clear realization are not representable in SPINE. HEDGEHOG has an accuracy rate of 85.5% for all SPINE-representable patterns. Since the analysis for verifying weak semantics are hard-coded in HEDGEHOG, the false negatives comes from HEDGEHOG's limitation of recognizing implementation variants (see further Section 7).

3 Motivating Examples

Current pattern recognition approaches fail to properly verify pattern intent, which is an important aspect of patterns.. For example, the Singleton and the Flyweight patterns are both object-creational patterns, but each has a unique intent that can be implemented in various ways.

Example: the Singleton Pattern

The Singleton pattern is probably the most commonly used pattern. Figure 1 shows a common implementation of the Singleton pattern. It is generally perceived to be the simplest pattern to detect [28, 24], since it does not require analyzing its interaction with other classes. The intent of the Singleton pattern is to ensure that a class has only one instance [15]. However, to verify this intent is not an easy task and is typically omitted or limited in current recognition tools.

```
public class SingleSpoon {
  private SingleSpoon();
  private static SingleSpoon theSpoon;
  public static SingleSpoon getTheSpoon() {
    if (theSpoon == null) theSpoon = new SingleSpoon();
    return theSpoon;
  } }
```

Code based on http://www.fluffycat.com/Java-Design-Patterns/Singleton

Figure 1. An Example of a Singleton Class

For example, FUJABA's recognition is solely based on inter-class relationships, which identifies a Singleton class with the following criteria: (1) has class constructors regardless of accessibility, (2) has a static reference, regardless of accessibility, to the Singleton class, and (3) has a public-static method that returns the Singleton class type. Thus, without further static behavioral analysis in the method bodies, FUJABA identifies a Singleton class as long as it matches these constraints. In fact, (1) and (2) are incorrect. The constructors have to be declared private (unless the class is abstract) to control the number of objects created, and the Singleton reference also has to be private to be prevent external modification. Even with these constraints modified, the Singleton class structure only prevents external instantiation and modification. The real pattern intent is embedded in the public-static method's body.

As another example, HEDGEHOG uses limited static semantic analysis to verify implementation of lazy instantiation (illustrated in getTheSpoon() of Figure 1). The lazy-instantiation analysis is hard-wired in HEDGEHOG. Based on the other semantic analysis techniques discussed in Reference [9], HEDGEHOG is not able to recognize other forms of lazy instantiation, such as using boolean (or other data types) flags to guard the lazy instantiation or using a different program structure (illustrated in Figure 2).

```
public static SingleSpoon getTheSpoon() {
   if (theSpoon != null) return theSpoon;
   theSpoon = new SingleSpoon();
   return theSpoon;
}
```

Figure 2. Lazy Instantiation Variant

Example: the Flyweight Pattern

The Flyweight pattern has various interpretations. Because it is categorized as a structural pattern by GoF, many believe the pattern is based on inter-class relationships. However, the pattern consists of a flyweight factory (that manages a pool of sharable-unique flyweight objects), which makes it more of a creational pattern and thus requires verifying pattern intent. The GoF book specifies that a flyweight object is created upon request. Each created flyweight object stored in a flyweight pool is associated with a unique key for later retrieval. Figure 3 shows an implementation of the GoF interpretation. The real pattern intent resides in the method body of getFlyweight(...).

```
public class FlyweightFactory {
  Hashtable hash = new Hashtable();
  public BallFlyweight getFlyweight(
    int r, Color col, Container c, AStrategy a) {
    BallFlyweight tempFlyweight =
           new BallFlyweight(r,col,c,a),
       hashFlyweight =
           ((BallFlyweight)hash.get(tempFlyweight));
    if(hashFlyweight != null) return hashFlyweight;
    else {
      hash.put(tempFlyweight,tempFlyweight);
      return tempFlyweight;
    } } }
```

Code based on
http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/FlyweightPattern.htm

Figure 3. Implementation of getFlyweight()

FUJABA interprets a flyweight class of having a container and a method that keeps and creates flyweight objects, respectively. This strategy fails to capture the pattern intent and significantly increases the false positive rate.

HEDGEHOG, on the other hand, interprets an immutable class as a flyweight class and any static-final variables as sharable flyweight objects. However, such interpretations tend to be overly restrictive and fail to recognize the GoF interpretation of the Flyweight pattern.

4 GoF Patterns Reclassified

The GoF book [15] illustrates 23 common design patterns and categorizes them based on their purposes (for creational, structural, or behavioral) and scopes (that are either class- or object-based). Based on the GoF categorization, some researchers [6, 29] believe structural patterns can be identified based on only inter-class relationships and require the least effort to analyze. Creational patterns come next, since statements of object creation can be easily detected. Behavioral patterns are considered the most difficult to detect, since analysis on the behavior in the method body is required. However, that view is not entirely accurate. As discussed in Section 3: the Singleton pattern (a creational pattern) requires not only detecting the existence of object creation, but it also requires verifying the behavior of the method body that creates and returns the Singleton instance; the Flyweight pattern (a structural pattern) requires behavioral analysis to verify whether all flyweight objects in the flyweight pool are singletons and are created on demand. The Template Method and Visitor pattern (both behavioral patterns) define their behavior in the class definitions, which can be identified based on static structural analysis (see Sections 4.2). While this categorization is useful for programmers, it is not helpful for pattern detection.

Instead of using purposes and scopes, patterns should be categorized, in the reverse-engineering sense, by their definitions from the structural and behavioral aspects. Some patterns are driven by code structure and are designed to structurally decouple classes and objects; but, other patterns are driven by system behavior and require specific actions implemented in the method bodies. Thus, we divide the GoF patterns based on their structural and behavioral resemblances into five categories: patterns that are already provided in the language (Section 4.1); patterns that are driven by structural design and can be detected using static structural analysis (Section 4.2); patterns that are driven by behavioral design and can be detected using static behavioral analysis (Section 4.3); patterns that are domain-specific (Section 4.4); patterns that are only generic concepts (Section 4.5). Figure 4 illustrates this reclassification and highlights our search strategies. The squares are the design patterns, and ovals are structural sub-patterns (which are the building blocks of the design patterns; some of the sub-patterns here are used in References [21, 26]). The unboxed texts indicate the searching criteria along the edge to another design pattern. A standalone square indicates that the pattern is detectable either by its inter-class properties or using additional domain-specific knowledge and heuristics.

4.1 Language-provided Patterns

Design patterns are so widely used today that many languages (e.g., Java, Python) and packages (e.g, JDK, STL) implement some common design patterns to facilitate programming. Java provides the Iterator (as in java.util.Enumeration, java.util.Iterator, and the foreach loop) and Prototype (as the clone() method in java.lang.Object) patterns. In practice, developers tend to use such built-in facilities[1] to efficiently and effectively build software systems. Such pattern instances can be recognized by matching specific names for methods or checking if a class implements a specific Java interface, which is used in HEDGEHOG [8].

4.2 Structure-driven Patterns

Patterns in this category can be identified by inter-class relationships. Such relationships establish the overall system architecture but do not specify the actual system behavior. Inter-class relationships are used to separate class responsibilities that contain declarations, generalization, association, and delegation relationships. Structure-driven patterns include the Bridge, Composite, Adapter, Facade, Proxy, Template Method, and Visitor patterns. The Bridge and Composite patterns separate class hierarchies based on generalization and association relationships; the Adapter, Facade, and Proxy patterns separate class roles based on class association and method delegation relationships; the Visitor and Template Method patterns defer class responsibilities through method declarations and delegation.

4.3 Behavior-driven Patterns

Some patterns are designed to realize certain behavioral requirements. Such a design pattern is embedded with a program intent that is carried in inter-class relationships and method bodies. Behavior-driven patterns include the Singleton, Abstract Factory, Factory Method, Flyweight, Chain of Responsibility (CoR), Decorator, Strategy, State, Observer, and Mediator patterns.

The GoF creational patterns are driven by some constraints on object creation, such as the number and type of

[1] java.util.Observable implements the Observer pattern; it implements a fixed subject-listeners communication mechanism, and the order in which notifications will be delivered is unspecified (see the Java2 API). In practice, the Observer pattern is applied to various contexts with different internal data structures and communication mechanisms. Thus, we include the Observer pattern in the Behavior-driven Patterns category (Section 4.3).

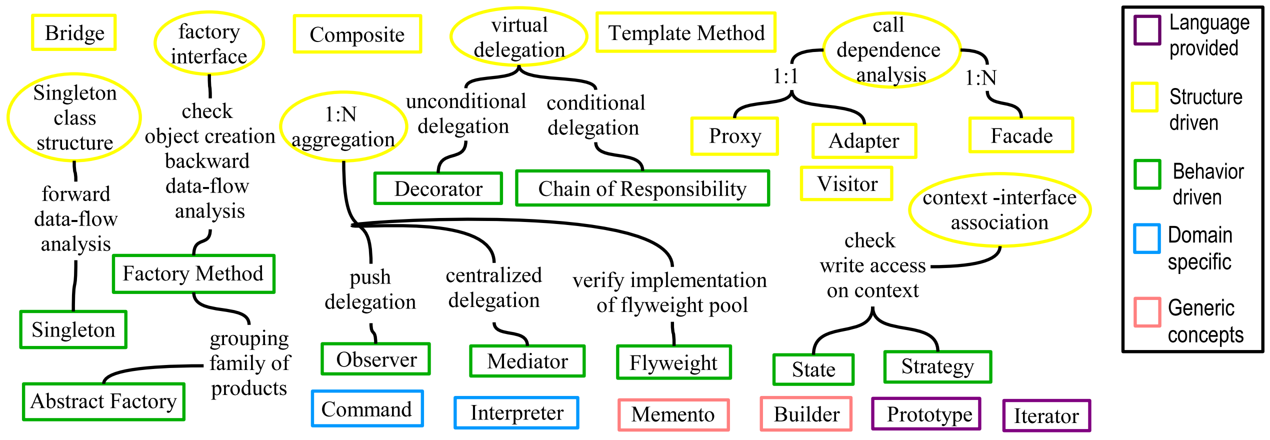


Figure 4. A Reclassification for Reverse Engineering of the 23 GoF Patterns

objects to be created. For example, the Singleton pattern ensures that a class has at most one instance during the entire program execution. The Flyweight pattern, although classified as a GoF structural pattern, is designed to effectively manage a pool of sharable objects.

The CoR and Decorator patterns define different behavior based on how a request is passed along a list of handlers. The Decorator pattern lets every handler process the same request, while the CoR pattern passes a request along the list until the right handler processes it.

The Strategy and State patterns share identical inter-class structures but differ in behavior. Each pattern involves a context class that has an attribute that takes a role of either a strategy or a state. The two patterns differ in how the attribute gets modified. In the State pattern, the attribute is passively modified by other state objects. In the Strategy pattern, the attribute is actively modified by other class entity through the context class.

The Observer (subject vs. observers) and Mediator (mediator vs. colleagues) patterns share the same 1:N aggregation relationship but differ in communication styles. The subject class of the Observer pattern broadcasts messages to its observers, while the mediator of the Mediator pattern serves as a communication hub for its colleagues.

4.4 Domain-specific Patterns

The Interpreter and Command patterns combine other GoF patterns and are specialized to suit a particular domain. The Interpreter pattern uses the structure of the Composite pattern and the behavior of the Visitor pattern. Based on this formation and a grammar of a language, the Interpreter pattern interprets the language. We consider this pattern as a special case of realizing the Composite and the Visitor patterns. The Command pattern is basically a realization of the Bridge pattern that separates the user interface from the actual implementation for command execution. The Command pattern also suggests incorporating the Composite pattern to support multi-commands and undoable operations and using the Memento pattern to store the history of executed commands. Such patterns are possible to detect, but their detection requires analysis that incorporates domain-specific knowledge.

4.5 Generic Concepts

While useful in practice, the Builder and Memento patterns are only generic concepts lacking traceable implementation patterns. The Builder pattern is a creational pattern that separates the building logic from the actual object creation, so that the building logic is reusable [15]. In practice, this pattern is often used for system bootstrapping, of which object creation may not be involved with initial configuration. The Builder pattern was detected in Reference [6] with a 86% false positive rate. The Memento pattern "captures and externalizes an object's internal state so that the object can be restored to this state later" [15]. However, the pattern neither defines the representation for a state nor the requirement of a data structure for the memo pool. This pattern has not been addressed in any pattern detection tools discussed in Section 2, because similar to the Builder pattern, these patterns are generic concepts that lack definite structural and behavioral aspects for pattern detection.

5 Approach to Pattern Detection

A design pattern is an abstraction of source code design and can be realized in many ways, which makes it nontrivial to detect. However, a pattern can be effectively detected using various program analysis techniques if it has a concrete definition of how it realizes its structural and behavioral aspects. Thus in our current scope, we exclude detection for domain-specific patterns and generic concepts. We also exclude language-provided patterns, since they are included in the language and require only trivial keyword analysis. In this paper, we focus on detecting the structure- and behavior-driven patterns.

5.1 Detecting Structure-driven Patterns

Section 4.2 discussed how such patterns can be detected by their inter-class relationships. Information on various inter-class relationships can be obtained through parsing. Then, specific analysis is applied to different patterns.

The Bridge, Composite, and Template Method patterns have been successfully identified in previous work (that target structural aspects) based on inter-class relationships. We use the same approach in this case.

The Visitor pattern provides a way to define a new operation to be performed on an already-built object structure without changing the classes of the elements on which it operates [15]. The inter-class relationships involved are: a method declaration accept (e.g., void Accept(Visitor v)), defined in the element class to invite a visitor; and a method invocation visit (e.g., v.visit(this)), where an element exposes itself to the visitor.

The Object Adapter (adapter vs. adaptee), Facade (facade vs. subparts), and Proxy (proxy vs. real) patterns share a common goal: to define a new class to hide other class(es) for system integration or simplification. We will refer to the Object Adapter pattern as the Adapter pattern. The Adapter and Proxy patterns each hides one class, whereas the Facade pattern hides multiple classes (to be distinguished from the Adapter pattern). By "hiding", we mean the hidden classes are not directly accessed (by reference or delegation) from others except for the one that is hiding.

Some other basic inter-class structures also need to be identified for detecting behavior-driven patterns (see further Section 5.2). The Singleton class structure is based on the structural features described in Section 3. The sub-patterns (as the ovals in Figure 4) are also identified for further behavioral analysis. These inter-class sub-patterns can be identified by analyzing class inheritance, class and method declarations, and method delegations. The next section further discusses these structures.

5.2 Detecting Behavior-driven Patterns

The inter-class analysis (defined in Section 5.1) identifies the structural aspect of a pattern, and most importantly narrows down our search space to particular methods for further static behavioral analysis. For example, identifying the Singleton class structure determines whether the singleton instance is created (1) once upon declaration or (2) by lazy instantiation. Then, static behavioral analysis is applied to each candidate method's body to verify whether for (1) it simply returns the instance or for (2) it correctly implements lazy instantiation.

There are several ways to understand program behavior. A common technique is template matching, which is often used in detecting malicious or buggy code (e.g., [18]). If applied to pattern detection, we can perhaps characterize certain pattern behavior into a sequence of states, then make it a template to match a target method. However, design patterns are not defined for detection or verification. Instead, they serve as guidance for various reification. Such sequence matching techniques can be limited in recognizing more common implementation variants.

Traditional data-flow analysis analyzes the entire AST of the method body. However, each behavior-driven pattern has a unique behavior that defines a target variable or statement for detection. To determine if an implementation is a correct pattern instance, we only need to verify whether the target does the right thing under the right condition. For example, if the getInstance() method of the Singleton pattern implements lazy instantiation, then it guarantees that the singleton instance gets created only once upon initialization. Here only the sub-AST that covers the lazy-instantiation mechanism requires full data-flow analysis.

Therefore, our approach uses data-flow analysis on ASTs in terms of basic blocks. As it processes each method body, it identifies the basic blocks, each of which contains statements that are executed under the same condition(s). Our approach links together the basic blocks based on execution flow to form a control-flow graph (CFG) for the method body. To illustrate, we present two examples: the Singleton and Flyweight patterns.

Example: the Singleton Pattern

Consider our static behavioral analysis to determine lazy instantiation in a method body of getTheSpoon() in Figure 1. First, we build the CFG shown in Figure 5. (Control

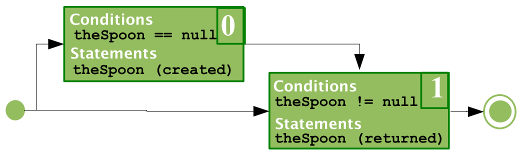


Figure 5. CFG of getTheSpoon() from Figure 1

flow is indicated through directed edges.) Then, the CFG is scanned to determine which basic block instantiates and which returns the singleton instance. The main actor here is the singleton variable that has the roles of being instantiated and returned. Based on the pre-determined actor and roles, our algorithm tells us BasicBlock0 creates the singleton instance and BasicBlock1 returns the singleton variable.

Then, we examine the conditions guarding BasicBlock0. Since only the last program state before return matters, here we use backward data-flow analysis on the flag variables involved in the conditions to verify if the contained sequence of statements guarantee single entrance to this basic block. Next, we check for the rest of the basic blocks if the flag variables can be modified, using backward data-flow analysis, elsewhere besides BasicBlock0.

Lazy instantiation can take many different forms. For example, one may use boolean types as flags, or use a different program structure, such as reversing the create-and-then-return order (shown in Figure 2). Such realistic variants of code map to (structurally) the same CFG as that in Figure 5, so we can use the same algorithm to track variable activities. Other behavior-driven patterns can also be detected using similar approaches.

Example: the Flyweight Pattern

Our approach to detecting the Flyweight pattern is based on a similar technique, which analyzes the method that potentially returns a flyweight object. For example, consider

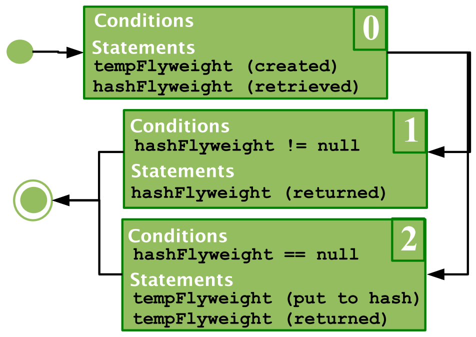


Figure 6. CFG of getFlyweight() **from Figure 3**

the code in Figure 3. Our inter-class analysis pinpoints that getFlyweight(...) is a candidate method that returns a flyweight object; then our static behavioral analysis is applied to this method. We build the CFG shown in Figure 6 (which shows roles, described later). Similar to detecting the Singleton pattern, the actors, which are the flyweight instance and flyweight pool, and their roles must then be determined. In this case, a flyweight instance is determined at a return statement. A flyweight pool is indicated by its data type, which is often a container class (such as a hashtable in this case). Based on the actors and their use and interaction in a statement, our algorithm assigns a role to each flyweight instance. Using backward analysis on this CFG, we can easily verify that the implementation either returns an existing or a new flyweight object.

Transforming to basic blocks not only flattens an AST of a method body, but also facilitates the detection of whether a target statement is executed in all paths. For example, the similarity between the CoR (with chained handlers) and the Decorator (with linked decorators) patterns is that each invokes the same polymorphic method of the adjacent node, but the difference is that this call is conditional for CoR and mandatory for Decorator. The same technique also applies to detection of loops to distinguish between the Observer and Mediator patterns.

6 PINOT

Based on our methodology (Section 5), we implemented a fully automated pattern detection tool, called PINOT (Pattern INference recOvery Tool). The current implementation of PINOT recognizes all the GoF patterns in the structure- and behavior-driven categories.

PINOT is built from Jikes (an opensource Java compiler written in C++) with an embedded pattern analysis engine. There are number of advantages of using a compiler as the basis of a pattern detection tool. A compiler constructs symbol tables and AST that facilitate the inter-class and static behavioral analyses. Compilers also perform some semantic checks that help pattern analysis. For example, Jikes prints out warnings when a local variable shadows (has the same name as) a global variable, which helps disambiguate delegation relationships. Most importantly, compilation errors reflect the incompleteness of symbol tables and AST, which result in incorrect pattern detection results. However, some tools, such as FUJABA and PTIDEJ, are able to partially (with a fuzzy number) detect patterns from incomplete source. Such tools can be desirable if pattern detection is used as part of software forward-engineering, such as building and incorporating patterns on the run. In our case, pattern detection is reserved for reverse-engineering, where accuracy is vital.

PINOT begins its detection process for a given pattern based on what is most likely to be most effective in identifying that pattern (i.e., declarations, associations, or delegations). This reduces the search space by pruning the least likely classes or methods. The completeness of a pattern detection tool is determined by the ability of recognizing pattern implementation variants. For practical reasons, PINOT focuses on detecting common implementation variants used in practice. Thus, some behavioral analysis techniques are not fully applied to each behavior-driven pattern. As an example, data-flow analysis is applied to analyzing the activities of the flag variable that guards the lazy instantiation in the Singleton pattern. The flag can have any data type, but java.lang.Object (when the reference for the Singleton instance also acts as the flag) and boolean are more common. Although a flag may be an integer, it is not as common in this case and would require much more computation. Thus, PINOT only analyzes lazy instantiation that uses boolean or java.lang.Object types. Inter-procedural data-flow and alias analyses are only used for detecting patterns that often involve method delegations in practice, such as Abstract Factory, Factory Method, Strategy and State patterns.

Some patterns, such as Decorator, CoR, Observer, and Mediator patterns, require only identifying the condition of which the target method delegation statement takes place. In particular, the Observer pattern involves a subject notifying a list of listeners. In Java, the listeners are usually stored in an array or a java.util.Collection class. If the latter, the iteration is often handled using java.util.Iterator. PINOT identifies arrays and array indexing, as well as classes that implements java.util.Collection and their use of java.util.Iterator. PINOT does not recognize any user-defined or user-extended data structures.

7 Results

We compared PINOT with two other similar tools: HEDGEHOG [9] and FUJABA 4.3.1 (with Inference Engine version 2.1).

HEDGEHOG (see Section 2.2) reads pattern specifications from SPINE, which allows users to specify inter-class relationships and other path-insensitive semantic analysis (e.g., for Factory Method pattern, the predicate "instantiates(M, T)" checks whether a method M creates and returns an instance of type T.), but other more complicated semantic analysis is hard-wired to its built-in predicates (e.g., "lazyInstantiates(...)"). Thus, SPINE is bounded by the capability of semantic analysis provided by HEDGEHOG. To use the tool, the user specifies a target class and a target pattern to verify against (i.e., attempt to recognize).

FUJABA has a rich GUI for software re-engineering. Its pattern inference engine provides a UML-like visual language for user-defined patterns. The language allows specifying inter-class relationships and a "creates" relationship (which is the same as the "instantiates" predicate defined in SPINE). FUJABA is easy to use: the user simply specifies the location of the source code and then runs the pattern inference engine. FUJABA displays the results graphically. FUJABA can run entirely automatically or incorporate interactive user guidance to reduce its search space.

PINOT is fully automated; it takes a source package and detects the pattern instances. All detection algorithms are currently hard-coded to prove the correctness of our techniques on the structure- and behavior-driven patterns.

Although these three tools were built for different uses, they all involve pattern recognition. Thus, we compare these tools in terms of accuracy. Table 1 shows the results of testing each tool against the demo source from "Applied Java Patterns"(AJP) [27]. Each AJP pattern example is similar to the one illustrated in the GoF book [15], except for

	Tools		
	PINOT	HEDGEHOG	FUJABA
Creational			
Abstract Factory†	✓	✓	×
Builder	–	–	–
Factory Method†	✓	✓	×
Prototype	–	×	–
Singleton†	✓	✓	✓
Structural			
Adapter*	✓	✓	×
Bridge*	✓	✓	✓
Composite*	✓	✓	×
Decorator†	✓	✓	×
Facade*	✓	–	✓
Flyweight†	✓	✓	×
Proxy*	✓	✓	–
Behavioral			
CoR†	✓	–	×
Command	–	–	–
Interpreter	–	–	–
Iterator	–	✓	×
Mediator†	✓	–	×
Memento	–	–	×
Observer†	✓	✓	×
State†	✓	×	–
Strategy†	✓	✓	✓
TemplateMethod*	✓	✓	✓
Visitor*	✓	✓	–

$\star$: a Structure-driven Pattern; $\dagger$: a Behavior-driven Pattern

✓ the tool claims to recognize this pattern and is able to correctly identify it in the AJP example.

× tool claims to recognize this pattern but fails to identify it in AJP.

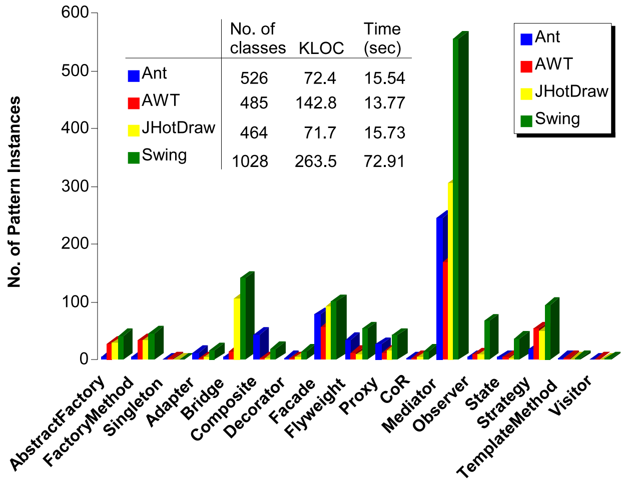
– the tool excludes recognition for this pattern.

Table 1. Pattern Recovery Results on AJP

the Flyweight pattern. The AJP Flyweight example does not define a flyweight pool; instead, the flyweight objects are statically instantiated and are static-final fields of the flyweight factory class. Table 1 shows that PINOT is able to recognize all the structure- and behavior-driven patterns in AJP. Because PINOT is a pattern detection tool, it assumes a class can participate in any pattern. Thus, PINOT tests a class against all pattern definitions. FUJABA was also tested in the same fashion. HEDGEHOG, however, is not an automated verification tool and users are responsible of picking the patterns to verify against the target class. Thus, HEDGEHOG's results shown in Table 1 were based on prior knowledge of the source and only likely patterns were verified against a class [9].

Patterns can have various reification, and it is impossible for a pattern recognition tool to be complete. Thus, a tool's pattern-recognition ability depends on its interpretation of pattern implementation. As an example, the Observer pattern defines how a Subject class notifies its Listener classes. FUJABA recognizes a variant of this pattern and calls it the "Broadcast Mediator" pattern. It specifies that Subject has a container class for the Listeners, and there exists a method delegations from Subject to Listener. HEDGEHOG, on the other hand, first checks for a container (as does FUJABA) and then checks if Subject defines the following methods: a method that starts with prefix name "add", another that starts with "remove", and finally one method delegation that invokes some method in Listener. HEDGEHOG checks if the first two methods actually *add* and *remove* an object of Listener type from the container [8]. However, FUJABA's and HEDGEHOG's approaches do not capture the real intent of the pattern, which is the "broadcasting of notifications" as in a push-model communication. PINOT recognizes this intent by first identifying a container in a Subject class (based on inter-class relationships) and then using static behavioral analysis (using techniques similar to those illustrated in Section 5.2) to identify a loop control (e.g, in a notify method) that iterates through the container and invokes the same method (e.g., in an update method) of each contained Listener object.

We also tested PINOT on several real Java applications. Figure 7 shows only the results for Java AWT 1.3, JHotDraw 6.0, Java Swing 1.4, and Apache Ant 1.6; see [2] for results on other applications, such as javac, java.io, and java.net packages. PINOT analyzes all classes, including anonymous and inner classes. A pattern instance is a collection of participating classes, and a class may participate in several other patterns. For example, in AWT, java.awt.Component and java.awt.ComponentPeer form one Bridge pattern instance; java.awt.Component and java.awt.Container together form one Composite and one CoR pattern instances.



	No. of classes	KLOC	Time (sec)
🟦Ant	526	72.4	15.54
🟥AWT	485	142.8	13.77
🟨JHotDraw	464	71.7	15.73
🟩Swing	1028	263.5	72.91

Figure 7. Pattern Instances Recovered

We ran PINOT on each of these packages on a Linux machine running on a 3GHz Intel processor with 1G of RAM. Compared to times for PINOT (Figure 7), FUJABA took 22 minutes to analyze the AWT and PTIDEJ took 2-3 hours to analyze JHotDraw. FUJABA was tested on a Pentium III 933MHz processor with 1G of memory. The reported time excludes parsing [21], but we are not certain if this time includes displaying the results graphically. PTIDEJ was tested on an AMD Athlon 2GHz 64b processor. PINOT is faster because the recognition algorithms are hard-coded and compute common sub-patterns patterns once.

The PINOT website [2] comprehensively discusses the recovered pattern instances. Our test results were verified against an authoritative discussion pattern discussion board [1], documentation written by original developers [14], and manual verification. We found some false positives in PINOT's results: 23.75% of Factory Method instances are considered Prototype instances, of which the classes implement java.util.Cloneable and override the clone method. Such Prototype instances are trivial to identify using keyword matching. However, user-defined variants that do not implement the Java built-in types may require heuristics to verify the "cloning" intent within method bodies.

Due to the impreciseness of some GoF definitions, PINOT recognizes other common implementation variants of the Flyweight and Mediator patterns. In particular, PINOT recognizes *immutable* classes as a common implementation variant of the Flyweight pattern [8]. We found 13.69% of Flyweight instances as *immutable* classes. Moreover, PINOT detects a Mediator variant (in AJP and GoF sample code) that allows colleagues to be individual instances in a Mediator class (i.e., a variant 1:N relation). In this case, the Mediator class serves as a facade that shields direct communication from one colleague to another. We found 24.93% of the Mediator classes as Facade classes.

Unfortunately, we are not able to compare our results with other pattern recognition tools. HEDGEHOG verified 5 correct pattern instances [8] (that have also been identified by PINOT, see [2]) within the AWT, but the tool is not publicly available (unlike PTIDEJ and FUJABA). PTIDEJ [16] analyzes patterns at the bytecode-level and was tested on AWT and JHotDraw, but the results were not comprehensive and only presented recall results for the Composite pattern. FUJABA [21, 22, 25] was tested on the entire AWT 1.3, but only 3 pattern instances were reported (also identified by PINOT) and it is not clear whether the published results of pattern instances were comprehensive. Our experimentation with PTIDEJ and FUJABA indicates that PTIDEJ is not stable and lacks user documentation, while FUJABA works on small programs but has limited pattern recognition capability on larger programs.

8 Conclusion and Future Work

This paper discussed the state-of-the-art pattern detection tools. Our contributions include: reclassifying the GoF patterns to facilitate pattern recognition; claiming that pattern definitions are either driven by code structure or system behavior; using our lightweight static program analysis techniques to efficiently recognize complicated program behavior; and implementing PINOT, a fully automated pattern detection tool that is faster, more accurate, and more comprehensive than existing tools. Our future work with PINOT will: expand its pattern recognition capability to recognize more complicated user-defined data structures; explore its use to detect design patterns in specific application domains, such as concurrent and real-time patterns; experiment with its use in tracking software evolution by design; and extend its overall usability by providing a visual specification language for defining patterns and exporting our analysis results as XMI for external viewing.

Acknowledgments

Todd Williamson helped greatly with testing PINOT.

References

[1] Pattern Stories: JavaAWT. `http://wiki.cs.uiuc.edu/PatternStories/JavaAWT`.

[2] The PINOT Website. `http://www.cs.ucdavis.edu/~shini/research/pinot`.

[3] H. Albin-Amiot, P. Cointe, Y.-G. Guehéneuc, and N. Jussien. Instantiating and detecting design patterns: putting bits and pieces together. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering*, pages 166–173. IEEE Computer Society Press, Nov. 2001.

[4] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *Proc. of the 6th International Workshop on Program Comprehension*, pages 153–160. IEEE Computer Society Press, June 1998.

[5] A. Asencio, S. Cardman, D. Harris, and E. Laderman. Relating expectations to automatically recovered design patterns. In *WCRE*, pages 87–96, 2002.

[6] Z. Balanyi and R. Ferenc. Mining design patterns from C++ source code. In *Proc. of the International Conference on Software Maintenance*, pages 305–314. IEEE Computer Society Press, September 2003.

[7] J. Bansiya. Automating design-pattern identification – DP++ is a tool for C++ programs. *Dr. Dobbs Journal*, 1998.

[8] A. Blewitt. *HEDGEHOG: Automatic Verification of Design Patterns in Java*. PhD thesis, University of Edinburgh, 2005.

[9] A. Blewitt, A. Bundy, and I. Stark. Automatic verification of design patterns in Java. In *ASE*, pages 224–232, 2005.

[10] K. Brown. Design Reverse Engineering and Automated Design Pattern Detection in SmallTalk. Master's thesis, North Carolina State University, 1998.

[11] J. Fabry and T. Mens. Language Independent Detection of Object-Oriented Design Patterns. *Computer Languages, Systems and Structures*, February 2004.

[12] R. Ferenc, Á. Beszédes, L. Fulop, and J. Lele. Design pattern mining enhanced by machine learning. In *ICSM*, pages 295–304, 2005.

[13] R. Ferenc, J. Gustafsson, L. Müller, and J. Paakki. Recognizing design patterns in C++ programs with the integration of Columbus and Maisa. *Acta Cybern.*, 15(4):669–682, 2002.

[14] E. Gamma. Becoming a Programming Picasso with JHotDraw. `http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-jhotdraw.html`.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[16] Y.-G. Guehéneuc and N. Jussien. Using explanations for design patterns identification. In *Proceedings of the 1st IJCAI Workshop on Modelling and Solving Problems with Constraints*, pages 57–64, August 2001.

[17] Y.-G. Guehéneuc, H. Shraoui, and F. Zaidi. Fingerprinting design patterns. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 172–181, Nov 2004.

[18] S. Hallem, B. Chelf, Y. Xie, and D. R. Engler. A system and language for building system-specific, static analyses. In *PLDI*, pages 69–82, 2002.

[19] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe. Automatic design pattern detection. In *Proc. of the 11th IEEE International Workshop on Program Comprehension*, pages 94–103. IEEE Computer Society Press, May 2003.

[20] R. Keller, R. Shauer, S. Robitaille, and P. Pagé. Pattern-based reverse-engineering of design components. In *Proc. of the 21st International Conference on Software Engineering*, pages 226–235. IEEE Computer Society Press, May 1999.

[21] J. Niere, W. Shafer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *ICSE*, pages 338–348. IEEE Computer Society Press, May 2002.

[22] J. Niere, J. P. Wadsack, and L. Wendehals. Handling large search space in pattern-based reverse engineering. In *Proc. of the 11th IEEE International Workshop on Program Comprehension*, pages 274–279. IEEE Computer Society Press, May 2003.

[23] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A. I. Verkamo. Software metrics by architectural pattern mining. In *Proceedings of the International Conference on Software: Theory and Practice*, pages 325–332. 16th IFIP World Computer Congress, August 2000.

[24] I. Philippow, D. Streitferdt, M. Riebisch, and S. Naumann. An approach for reverse engineering of design patterns. *Software Systems Modeling*, pages 55–70, 2005.

[25] J. Seemann and J. W. von Gudenberg. Pattern-based design recovery of Java software. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 10–16. ACM Press, 1998.

[26] J. M. Smith and D. Stotts. SPQR: flexible automated design pattern extraction from source code. In *ASE*, pages 215–224. IEEE Computer Society Press, October 2003.

[27] S. Stelting and O. Maassen. *Applied Java Patterns*. Prentice Hall, Palo Alto, California, 2002.

[28] M. Vokáč. An efficient tool for recovering design patterns from C++ code. *Journal of Object Technology*, 5(2), March-April 2006.

[29] L. Wendehals. Improving design pattern instance recognition by dynamic analysis. In *Proc. of the ICSE Workshop on Dynamic Analysis (WODA)*, pages 29–32. IEEE Computer Society Press, May 2003.