Reverse Engineering of Design Patterns for High Performance Computing

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

1 Introduction

Reverse engineering tools are typically used for program understanding. Popular code analysis and comprehension tools like CSCOPE [6] and RedHat Source-Navigator^{TM} [1] can recover program structure and relationships between program components. However, without proper documentation of legacy code, it would still take users a lot of work to understand the intent of the code. The need for a good reverse engineering tool becomes more vital to understand code designed for high performance computing.

A design pattern abstracts a reusable object-oriented design that solves a common recurring design problem in a particular context [14]. An object-oriented design describes the roles, responsibilities, and collaboration of participating classes and instances. Every design pattern has its own unique intent. By finding design patterns from source, we can bring program understanding to a higher level by revealing the architectural design of a system.

Past efforts have used structural relationships (such as the generalization and association relationships) to find design patterns. However, some patterns, e.g., the Strategy and the State patterns, are structurally identical but differ in behavior. To distinguish such patterns and to reduce the false positive rate, other approaches use dynamic analysis to verify pattern behavior. Dynamic analysis depends on the coverage of test data, which can increase the false negative rate if the test data is not complete.

Patterns for concurrent programs and distributed applications [17, 23] are defined similarly to sequential design patterns (i.e., the GoF design patterns [14]). The task to analyze code behavior when concurrency and synchronization are involved becomes more difficult. Unfortunately, dynamic analysis would not be helpful in analyzing concurrent behavior, considering it would need to deal with more complicated scenarios and larger amount of execution traces.

We propose to automate the pattern recognition process that combines structural analysis to look for pattern structures from source code and static program analysis to verify whether the semantics matches the pattern behavior. We have some preliminary results from our initial prototype in recovering design patterns from the Java AWT package. We believe that our approach will be a better fit for finding patterns designed for high performance computing. For example, our tool is also able to spot performance overhead by detecting a particular optimization pattern for concurrency [23].

2 Critique of Current Approaches

The GoF book [14] illustrates 23 common design patterns and categorizes them, based on their purposes, into three categories: creational, structural, and behavioral. Creational patterns focus on how objects get created. Structural patterns focus on class organization by roles using structural relationships, such as class inheritances, interface hierarchies, and attribute associations. Behavioral

patterns focus on separating object responsibilities based on polymorphism. While this categorization is useful for programmers, it is not helpful for pattern detection. Based on detection methods, current approaches can be categorized based on the kind of analysis they perform: pure static or a combination of static and dynamic.

2.1 Pure Static Approaches

Previous work [10, 8, 16, 19, 9, 24, 20, 27, 25, 22] uses structural analysis to find GoF design patterns [14] from source. Structural relationships of the code include class inheritance, interface hierarchies, attributes, method invocations, parameters and return types, object creations, and variable access within a method.

Some previous work [10, 16, 27] extracts structural relationships from C++ source and stores this information in a database. Designs patterns are recovered through queries to the database. Reference [9] uses the same schema defined for UML class diagrams for extracting abstract semantics graphs from C++ source. It defines an XML pattern language for users to define patterns. Patterns are recovered based on graph comparison. SPQR [25] first translates the abstract syntax graph obtained by gcc to a format recognized by a theorem prover. Then it runs the input on the theorem prover that recognizes some design patterns pre-defined using denotational semantics. Patterns are recovered based on formal analysis.

These approaches heavily rely on the accuracy of the information extracted in the first stage. Although extracting structural relationships seems straightforward, it is complicated by variations in the implementations of some relationships, such as aggregation [24, 19]. Thus, these approaches can result in either higher false positive or false negatives rates.

FUJABA [2, 19] extends the work from [24] and uses a bottom-up-top-down approach to speed up the search and to reduce the false positive rate. It uses a combination of structural relationships to indicate a pattern. Thus, when such information is obtained from the bottom-up search, even partially, FUJABA then assumes the existence of a possible pattern and tries to complete the rest of the search, that is the top-down search, to confirm that such a pattern actually exists. This iterative approach allows going back to the abstract syntax tree (AST) for further analysis on demand. Follow-on work [20] introduces fuzzy logic to make the search speed tolerable for larger-scale systems. Like [16], FUJABA is a semi-automatic detection tool. The pattern detection engine is bundled with the FUJABA Tool Suite RE (a software round-trip engineering tool for Java), which is in parallel with the work in Reference [7]. The pattern recognition process in FUJABA's recent work [18] is more user driven. They believe pattern detection requires human intervention to overcome scalability problems caused by implementation variations in different problem domains. Thus, this approach assumes users to have a fair amount of knowledge of the analyzed code. However, reverse engineering tools for design patterns are typically used in understanding legacy code, where users may not be able to provide any feedback during the reverse engineering process.

2.2 Static and Dynamic Approaches

The approaches in Section 2.1 are limited in finding patterns that are distinctive only in structure. However, some patterns aim at program behavior, which cannot be determined analyzing only structural relationships. Other approaches (e.g., those in References [15] and [28]) suggest using dynamic analysis to analyze behavior. They first obtain structural information from source code. Next for a particular pattern, they compute a list of candidate classes. Then, assuming what these candidates should behave, they verify the behavior during runtime. Reference [28] uses dynamic analysis as part of pattern identification. This approach complicates the search by expanding the set of candidate classes and results in analyzing more unrelated execution traces. We believe that structural analysis should be used to narrow down the search space.

Without any experimental results or proof, References [15] and [28] claim that traditional data-flow and control flow analysis should not be feasible when polymorphism and dynamic method binding are involved. However, the critical behavior in a design pattern is defined in the base class. Therefore, we rarely have to trace every possible path happening in the subclasses(s) (see the Chain

of Responsibility pattern and other behavioral patterns in [14]). And more importantly, 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 behavior of a particular pattern (e.g., a distribution might include a validation or benchmark suite).

KT [11] uses algorithms to search for patterns in programs written in SmallTalk. It excludes the search for patterns that are structurally identical, e.g., the Strategy, State and Command patterns. KT failed to find the Chain of Responsibility pattern. KT's search algorithm for the pattern is based on only dynamic analysis. It analyzes an object-message diagram interpreted from a call tree constructed during runtime. This process removes unnecessary message calls unrelated to the search. Then the pattern should be identified if the object-message diagram captures the right pattern behavior. However, this approach failed to find the Chain of Responsibility pattern due to improper message logging mechanism and insufficient test data.

2.3 Other Approaches

MAISA [4, 21, 26] measures software quality at the design level. From a system's architectural description (which includes UML class, activity, component, and sequence diagrams), MAISA is able to find design patterns and anti-patterns [12]. However, the number of patterns found were limited. Only the Abstract Factory pattern (which represents a "good" pattern) and the Blob anti-pattern[1] (which represents a "bad" pattern) were found in their analyzed system. Recovering design patterns from architectural descriptions is not likely to be effective in practice for two reasons. First, during software development, architectural requirements and descriptions are usually laid out at the beginning of the development cycle, but are rarely reiterated and detailed as the project evolves. Second, to use MAISA to find patterns, one needs to first extract a set of UML diagrams (including both structural and behavioral diagrams) from source; however, how to recover system behavior is still ongoing research.

3 Motivating Example

The Singleton pattern is probably the most commonly used pattern. It is generally perceived to be the simplest pattern to detect [27, 22], 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 [14]. However, to verify this intent is not an easy task.

The key features to implement the Singleton pattern in Java include: a private constructor (so that no other class — inside or outside of its package — can instantiate the Singleton class); a private static variable, instance, that holds the Singleton instance; and a public static getInstance() method that returns a Singleton class type. The getInstance() serves as a global access to instance. Some pattern detection tools, such as FUJABA and Reference [22], stop here and conclude that an implementation of the Singleton pattern is found. Other work described in Section 2 did not describe how their tools detect the Singleton pattern. However, the same structure requirement applies to some creational pattern that has no restriction on the number of instances being created. For example, one can define a class that structurally resembles the Singleton class but uses a private constructor and a global access point to control or to maintain registry for all created instances of this class. Thus, behavior must be analyzed to make sure that right pattern is identified.

Consider the following variations of implementing the Singleton pattern. If instance is initialized statically, then it is trivial to make sure that instance is not created again by any methods declared in the Singleton class. Now, if lazy initialization is used (instance created on first call to getInstance()), then it requires knowing when and how the Singleton instance gets created. Further, if more conditions are involved in getInstance() (e.g., getInstance() will not return instance unless it

[1]The Blob pattern describes the lack of OO design, which requires refactoring techniques to break the blob into object components. However, this work identifies the "Blob" when unsynchronized shared memory is found using the UML component and sequence diagrams.

is not currently in use), then it requires more intelligence to figure out under what circumstances will instance gets created and returned.

To recognize these variations, dynamic analysis is not helpful in verifying the intent. Dynamic analysis can spot a different address (object reference) being returned by getInstance(), so it can conclude the absence of the pattern. However, it cannot prove that a class implements the Singleton pattern simply because getInstance() seems to be returning the same address for a certain period of time. Thus, a different approach should be proposed.

4 Our Approach and Plans

As indicated in our motivating example (Section 3), dynamic analysis captures system behavior, but it is not practical in verifying the logic of a program. We propose to automate the pattern detection process using pure static analysis that combines structural and semantic analysis. Structural analysis helps narrow down the search space, while semantic analysis is used to verify behavior from method bodies and interactions with other classes. Here, behavior means in terms of statecharts and communication diagrams as defined in UML2 [13].

We modified Jikes [3] (a Java compiler written in C++) to analyze design patterns from Java source code. Our current focus is on analyzing the 23 GoF patterns. However, our present tool also checks for the Double-Checked Locking (DCL) pattern [23] when it recognizes a Multi-threaded Singleton pattern. The DCL pattern optimizes performance when implementing a thread-safe Singleton class. Our tool verifies whether the DCL pattern is correctly implemented in the source code.

The following sections discuss our search strategies and preliminary results in analyzing the Singleton pattern and the Chain of Responsibility (CoR) pattern in the Java AWT 1.3 package.

4.1 The Singleton Pattern

In analyzing the pattern, our first attempt assumes that instance is the only variable involved in getInstance(). Our strategy is to first make sure that only instance gets returned from getInstance(). Then, we simulate the execution paths to verify that instance is not modified in consecutive calls to getInstance(). This initial prototype is able to recognize correct implementations of the Singleton pattern in many common forms. It also rejects the incorrect implementations of the Singleton pattern that are falsely identified as Singleton patterns by FUJABA. We also consider Multi-threaded and Inherited Singleton patterns in our analysis.

4.2 The CoR Pattern

There are many ways to form a chain of request handlers. It is not required that all handlers should be subclasses of a base handler class, and neither should the handle methods be polymorphic [11]. For software maintainability, the GoF book defines a base Handler class. Handler points to itself through successor and defines the RequestHandle() methods.

Our current strategy is based on the form described in the GoF book. Our tool first finds an association relationship where RequestHandle() delegates its call to successor.RequestHandle(). Then, we examine Handler that contains RequestHandle(). Handler does not have to be abstract. However, Handler must be subclassed, and successor's type must have either a common super class or interface with Handler. Finally, we examine RequestHandle() by making sure that it produces at least two exit paths and that RequestHandle() only defers a request in one exit path.

4.3 Current Results

We have tested our tool on the Java AWT 1.3 package. The package contains 345 java files, 453 classes (including inner classes), 142800 lines of code. Our tool recognizes three instances of the Singleton pattern and one instance of the CoR pattern in AWT.

The Singleton classes are Toolkit, GraphicsEnvironment, and ColorModel. The first two are multi-threaded, but neither implemented the DCL pattern for performance optimization. The getInstance

methods are declared synchronized upon method declaration. Component and Container form the CoR pattern. Container extends Component and contains an array of Component. Component points to Container through a variable parent. This means Container can belong to another Container. Events are passed through the chain linked by parent, and the event handle methods are: getForeground(), getBackground(), getFont(), getLocale, and getInputContext(), etc. None of these instances of those two design patterns were discovered in References [19, 20, 24]. The Singleton instance Toolkit and the CoR instance are exactly those reported on the "Pattern Stories: JavaAWT" webpage [5]. We manually verified GraphicsEnvironment and ColorModel as correct Singleton instances.

The execution time performance of our tool is promising. On a Linux machine running on an Intel 1.4GHz processor with 512M of memory, it took less than a second to parse the source files and to find the instances of the two patterns in AWT described previously.

4.4 Plans

We plan to complete our tool by including the rest of the GoF patterns and patterns for parallel and distributed computing [23, 17]. We will also expand our tool to recognize most common variations of those patterns, which will require more complicated semantic analysis. We will consider using various program analysis techniques based on program slicing, interprocedural analysis, and control flow analysis.

Recognizing program behavior is, of course, an undecidable problem, hence a fully automated static analysis will not be able to achieve 100% accuracy. However, we believe that static analysis can significantly reduce the false positive and the false negative rates by recognizing common implementations for an intended behavior. We believe that our tool will be useful to detect performance overhead and more generally to enhance program understanding in high performance computing applications.

References

[1] CSCOPE. `http://cscope.sourceforge.net`.

[2] FUJABA. `http://www.fujaba.de`.

[3] Jikes. `http://www-124.ibm.com/developerworkds/ossjikes/`.

[4] MAISA. `http://www.cs.helsinki.fi/group/maisa/`.

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

[6] Source navigator. `http://sourcenav.sourceforge.net`.

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

[8] 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.

[9] 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.

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

[11] K. Brown. Design reverse engineering and automated design pattern detection in SmallTalk. Master's thesis, North Carolina State University, 1998.

[12] W. H. Brown, R. Malveau, H. W. M. III, and T. J. Mowbray. *Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, New York, New York, 1998.

[13] H.-E. Eriksson, M. Penker, B. Lyons, and D. Fado. *UML2 Toolkit*. Wiley Publishing, Indianapolis, Indiana, 2004.

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

[15] 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.

[16] 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.

[17] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading, Massachusetts, second edition, 2000.

[18] J. Niere, M. Meyer, and L. Wendehals. User-driven adaption in rule-based pattern recognition. Technical report, 2004.

[19] J. Niere, W. Shafer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of the 24 International Conference on Software Engineering*, pages 338–348. IEEE Computer Society Press, May 2002.

[20] 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.

[21] 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.

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

[23] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Chichester, England, 2000.

[24] 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.

[25] J. M. Smith and D. Stotts. SPQR: flexible automated design pattern extraction from source code. In *Proc. of the 18th IEEE International Conference on Automated Software Engineering*, pages 215–224. IEEE Computer Society Press, October 2003.

[26] A. I. Verkamo, J. Gustafsson, L. Nenonen, and J. Paakki. Design patterns in performance prediction. In *Proceedings of the Second International Workshop on Software and Performance*, pages 143–144. ACM Press, September 2000.

[27] M. Vokáč. An efficient tool or recovering design patterns from C++ code. *Journal of Object Technology*, July/August 2005. To appear.

[28] 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.