Java Design Pattern Obfuscation

Praneeth Kumar Gone  
Department of Computer Science  
San Jose State University  
San Jose, California 95192

Mark Stamp  
Department of Computer Science  
San Jose State University  
San Jose, California 95192  
Email: stamp@cs.sjsu.edu

Abstract—Software reverse engineering (SRE) consists of analyzing the design and implementation of software, where, typically, we assume that the executable file is available, but not the source code. SRE has many legitimate uses, including analysis of software when no source code is available and probing code for security vulnerabilities. Attackers also use SRE to search for weaknesses in closed-source software and to hack software activation mechanisms, or otherwise change the intended function of software.

There are many tools available to aid the aspiring reverse engineer. For example, there are several tools that recover design patterns from Java byte code or source code. In this project, we develop and analyze a technique to obfuscate design patterns. We show that our technique can defeat design pattern detection tools, thereby making reverse engineering attacks more difficult.

I. INTRODUCTION

Software reverse engineering (SRE) can be used to analyze executable files [14]. Examples of such analysis includes redocumentation of programs [3], code smell detection [8], renewal of software modules [19], migration of legacy code [4], translation of program from one language to another [17] and architecture recovery [2]. SRE is also used in software piracy, and other illegal activities.

Businesses spend an immense amount of time and money developing software. To protect their investment, it may be desirable to minimize the amount of information that rivals and attackers can gain through SRE. In this research, we propose and analyze techniques that make the recovery of high-level program design from Java class files more difficult. Specifically, our goal is to obfuscate design patterns.

In general, SRE of Java is much simpler than for programs that are compiled into native code. Java source code is compiled into machine-independent bytecode that can be executed by a Java Virtual Machine (JVM). This bytecode contains a great deal of information about the source code, making SRE of a typical Java class file a relatively easy task.

Software obfuscation is an anti-SRE mechanism that changes the structure of code, without changing its functionality. Many open source Java obfuscation tools are available [11, 25], including ProGuard [10], yGuard [29], SandMark [5], jarg [15], BebboSoft [27], and JavaGuard [28], as well as commercial tools such as Allatori [26], Zelix KlassMaster [30], and JShrink [7]. These obfuscation tools support a variety of obfuscation techniques, such as renaming classes, methods, fields and local variables to random strings, removing debugging information, removing dead code and constant fields, optimizing local variable allocation, and exception obfuscation.

However, existing obfuscation tools are not intended to perform design-level obfuscation. Obscuring the design requires changing the relationship between software system class components. Since design patterns use inheritance features, if we want to obfuscate such architectural-level information, we need to obscure the inheritance-level relationships between classes. Such obfuscation can be accomplished, for example, by removing interfaces or adding abstractions.

In this paper, we analyze a design obfuscation technique. We focus on the so-called Gang of Four (GoF) design patterns [12] and apply most of the design obfuscation techniques discussed in [20]. The effectiveness of the obfuscation is analyzed using existing design pattern recovery tools. Note that in [20], similar techniques are applied, but no testing is done to determine the effectiveness of the obfuscation on design recovery. The focus of [20] is on comparing the runtime performance of the obfuscated and unobfuscated code.

The remainder of this paper is organized as follows. In Section II, we discuss design patterns and pattern detection tools. Section III covers our approach to obfuscating design patterns. In Section IV we provide experimental results to illustrate the effect of our obfuscation on design pattern recovery. Finally, Section V concludes the paper and includes suggestions for future work.

II. DESIGN PATTERNS

Software design problems are simplified by using design patterns. These design patterns are reusable and rely on object oriented (OO) techniques. In this paper, we focus on the 23 GoF design patterns [12].

The GoF design patterns are grouped into three categories, namely, creational, structural, and behavioral patterns. Next, we provide information on each of these categories, including a list of the specific GoF patterns belonging to each.

As the name implies, creational patterns deal with the creation of objects. Creational patterns serve to encapsulate the knowledge of a given object in order to create and hide the individual instances that represent how these objects are created. The five GoF creational patterns are AbstractFactory, Builder, FactoryMethod, Prototype, and Singleton.

Structural patterns are used to realize relationships between different entities. A structural pattern may be used, for example, when adapting an object or when creating a complex type from simpler types. The GoF structural patterns are Adapter, Bridge, Composite, Decorator, Façade, Flyweight, and Proxy.

Behavioral patterns solve design problems by creating a common communication and implementation between entities.
Communicating between entities includes mediating between classes, notifying the state of an object, and selecting different algorithms at run time. The GoF behavioral patterns are CoR (Chain of Responsibility), Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, TemplateMethod, and Visitor.

For the sake of brevity, throughout this paper, we focus on the following four design patterns in detail: Builder, FactoryMethod, Decorator, and Mediator. The report [9] includes a detailed analysis for 12 of the GoF design patterns.

A. Builder

A Builder pattern can be used to create complex objects from smaller objects according to an algorithm or procedure. Figure 1 shows a UML diagram for a Builder pattern, as discussed in the following example.

Example: Suppose a `Client` class calls the `main()` method that will initiate a `Builder` and `Director` class. A `Builder` class represents a complex object that needs to be built using other small objects and types. The `Director` receives this `Builder` class and is responsible for calling appropriate methods that create a complex object. A `Client` can call a respective `ConcreteBuilder` depending on the parameters defined to create different complex objects. An example would be a `TextConverter` that converts an RTF document to an ASCII document. An `RTFReader` class will be acting as a `Director`, where a `TextConverter` interface is a `Builder` interface and an `ASCIIConverter` is an implementation of `Builder`, i.e., `TextConverter`. An `ASCIIConverter` reads each character or string from an `RTFReader`, then converts and writes to an ASCII document by following the Builder pattern.

B. FactoryMethod

The FactoryMethod pattern solves the problem of creating objects without specifying an exact class initialization. Initiating different objects in the application could duplicate the use of code and might increase memory requirements. The FactoryMethod pattern defines a separate abstract method that can be overridden by all subclasses with the derived object used within the application [6].

Example: Consider a `Factory Product` with a `Factory Interface` that specifies generic behavior for products. The `Client` requests a product from the `ConcreteFactory` to initialize the `Product` variable which uses concrete products. `ConcreteProduct` is an implementation of the `Product` interface; there can be different implementations depending on the type of product. The UML diagram for this Factory Product example appears in Figure 2.

C. Decorator

A Decorator pattern is used to demonstrate the relationship, during runtime, between entities. In software development we can extend functionality of an object statically at compile time by using inheritance. However, in some situations we need to extend functionality dynamically during runtime.

Example: A graphical window used to create a `FrameWindow` class would decorate a `Window` class and a `FrameWindow` object would be created statically by the client program. This use of a `FrameWindow` needs to initiate different objects within the client program. A Decorator pattern can be used to create a `FrameWindow` dynamically, without creating objects in the client program. The UML diagram demonstrating this Graphical Window application, using a Decorator pattern, is shown in Figure 3.

D. Mediator

A Mediator pattern is a behavioral pattern that aids in the interaction of a large number of classes. A Mediator pattern can be used to remove tight coupling behavior. Figure 4, shows the UML diagram of a Mediator pattern, relevant to the following example.
Example: Consider the problem of developing a screen that contains different controls, that is, a case where various controls must interact with other controls. For example, if a button is pressed it must determine whether the data is valid in other controls. Therefore, in different applications these controls need to interact differently. To solve this problem we use a Mediator pattern that can be extended with different implementations.

E. Pattern Detection Tools

Design pattern detection is a reverse engineering technique that can aid in analyzing Java code. There are several pattern detection tools available, including Hedgehog [1], Reclipse [16], Pattern INference and recOvery Tool (PINOT) [23, 24], and Similarity Scoring [21, 22]. Next, we give a brief overview of each of these tools.

Hedgehog was developed using a pattern description language known as SPINE [1] (and hence the name). SPINE is a language similar to Prolog and contains typed first order logic for describing patterns; it is not currently available for download.

Reclipse is a reverse engineering tool for automatic pattern detection from Java source code. It uses UML 2.0 diagrams derived from source code to deduce the design. Reclipse provides graphical editors for structural and behavioral patterns. Detection of a specified pattern starts from detecting putative design pattern occurrences, or candidates. Dynamic analysis is used to confirm or reject candidates. Installation of this tool requires Eclipse IDE v3.6.1, and Eclipse Modeling Tools, version 3.6.1 [16], however, these versions of the software are no longer available.

In PINOT, Prototype and Iterator patterns are classified as language-provided patterns, since they are widely used and implemented in many languages. Classes that have inter-class relationships, such as Adapter and Façade, are identified as structure-driven patterns, while classes that differ in certain behavioral requirements, such as Singleton and Flyweight, are deemed behavior-driven patterns. Finally, GoF patterns such as Interpreter and Command are known as domain-specific patterns. PINOT focuses on detecting structure and behavior driven patterns [23].

Previous research has shown that PINOT has a significant false positive rate [13, 23]. Our results in Section III confirm these findings.

Similarity Scoring is a design pattern extraction tool available from [22]. The process of detection relies on building matrices from Java class files and comparing them to known matrices [21]. The name Similarity Scoring derives from the graph matching algorithms used in the tool. This pattern detection tool does not depend on behavioral characteristics. By considering only structural characteristics it is difficult to detect certain patterns, such as State and Strategy, which only differ in behavior.

PINOT and Similarity Scoring are currently available and employ very different approaches to pattern detection. Consequently, in this paper, we use these two algorithms to measure the success of our proposed design pattern obfuscation technique.

III. DESIGN PATTERN OBFUSCATION

This section describes the obfuscation techniques that we apply to the GoF design patterns. But first, we test the pattern detection tools discussed in the previous section (i.e., PINOT and Similarity Scoring) on unobfuscated code, and on code that has been obfuscated using the well-known Java obfuscation tools Proguard [10] and SandMark [5]. These results will serve as a point of reference when we apply analyze our obfuscation technique in Section IV.

A. Unobfuscated Pattern Detection

For this test, we used a package that contains all 23 GoF patterns, with many patterns appearing more than once. The number of patterns detected using both PINOT and Similarity Scoring are shown in Figure 5.

The results for PINOT show poor detection of creational patterns. For example, only one out of four Singleton patterns
was detected. Also, PINOT had many false positives, as summarized in Table I.

**TABLE I: PINOT False Positives for Unobfuscated Files**

<table>
<thead>
<tr>
<th>detected</th>
<th>actual</th>
<th>occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facade Builder</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Flyweight Command</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Strategy AbstractFactory</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Observer Visitor</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Mediator Builder</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Mediator State</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Bridge Mediator</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

There were far fewer false positives using Similarity Scoring. With the exception of the false positives in PINOT, the results for Similarity Scoring and PINOT are comparable.

**B. Obfuscation using Proguard and Sandmark**

In this section we obfuscate the files considered in the previous section using two available obfuscators, namely, Proguard, and Sandmark. For Proguard, we test the obfuscated files using both PINOT and Similarity Scoring. However, for Sandmark we can only test the results using Similarity Scoring, since we cannot decompile the Sandmark obfuscated files, and PINOT requires the class files.

1) **Proguard:** Proguard is an open-source Java class file shrinker, optimizer, obfuscator, and preverifier [10]. The obfuscator option renames classes, fields, and methods, using meaningless names. For the test considered here, we only use the obfuscation option.

The results in Figure 6 show that PINOT and Similarity Scoring yield similar results on the Proguard obfuscated files as on the unobfuscated files. As with the unobfuscated files, PINOT produces many false positives; see Table II.

**TABLE II: PINOT False Positives for Proguard Obfuscated Files**

<table>
<thead>
<tr>
<th>detected</th>
<th>actual</th>
<th>occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bridge Mediator</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Flyweight Command</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Factory Memento</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Observer Visitor</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Composite Visitor</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

2) **Sandmark:** Sandmark is a tool developed for watermarking, tamper proofing, and obfuscation of Java bytecode [5]. The tool integrates a number of static and dynamic watermarking algorithms, a large collection of obfuscation algorithms, various code optimizers, and a tool to view and analyze Java bytecode. Here, we use the Sandmark obfuscation feature. In Sandmark, there are 39 different algorithms available to obfuscate Java bytecode. In our extensive experiments, we found that nearly all of the Sandmark obfuscation algorithms fail to obfuscate any design patterns. In fact, only three techniques—SplitClasses, Objectify, and OverloadName—were able to obfuscate any design patterns. The results for these three obfuscation techniques are summarized in Figure 7.

**Fig. 6: Detected Patterns after Proguard Obfuscation**

**Fig. 7: Detected Patterns using Similarity Scoring after Selected Sandmark Obfuscations**
C. Our Approach to Design Pattern Obfuscation

From the results in the previous section, it is clear that Proguard and Sandmark do not effectively obfuscate design patterns. Our goal is to develop a technique that can effectively obfuscate design patterns.

In [20], the authors consider design obfuscation based on class-coalescing, class-splitting, and type-hiding. Class-coalescing consists of combining two or more classes into a single class. At the extreme, we could replace all classes with a single class, effectively converting an OO program into a procedural program. Class-splitting consists of splitting one class into two or more classes. As with class-coalescing, class-splitting can have a major impact on program structure. Type-hiding introduces a number of Java interfaces that are implemented by existing classes, which can make the program structure more difficult to reverse engineer.

For our design pattern obfuscator, we employ class-coalescing and class-splitting. We are able to effectively hide design pattern information without employing type-hiding.

Next, we discuss the obfuscation we perform on the GoF design patterns. For the sake of brevity, we only provide details for the Mediator pattern. The discussion of the Mediator pattern below refers to the corresponding example presented in Section II. Many additional patterns are considered in the full report [9].

Obfuscating a Mediator pattern can be achieved using class-coalescing. Specifically, we remove the Mediator interface and replace Mediator references to respective mediator implementations. The example in Figure 8 illustrates the process, where the Mediator interface is removed and an ApplicationMediator is implemented as an ordinary class with all the necessary implementation. This ApplicationMediator is to be instantiated into a Colleague class All Mediators must be implemented and be used in respective Colleague classes.

D. Obfuscation of the GoF Patterns

The obfuscation techniques we apply to the 23 GoF design patterns are listed in the Table III. Note that “✓” denotes that the specified technique (i.e., class-coalescing or class-splitting) is applied, while “✗” implies the technique is not used.

<table>
<thead>
<tr>
<th>GoF Pattern</th>
<th>Obfuscation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbstractFactory</td>
<td>✓</td>
</tr>
<tr>
<td>Builder</td>
<td>✓</td>
</tr>
<tr>
<td>FactoryMethod</td>
<td>✗</td>
</tr>
<tr>
<td>Prototype</td>
<td>✓</td>
</tr>
<tr>
<td>Singleton</td>
<td>✗</td>
</tr>
<tr>
<td>Adapter</td>
<td>✓</td>
</tr>
<tr>
<td>Bridge</td>
<td>✓</td>
</tr>
<tr>
<td>Composite</td>
<td>✓</td>
</tr>
<tr>
<td>Decorator</td>
<td>✓</td>
</tr>
<tr>
<td>Façade</td>
<td>✓</td>
</tr>
<tr>
<td>Flyweight</td>
<td>✓</td>
</tr>
<tr>
<td>Proxy</td>
<td>✓</td>
</tr>
<tr>
<td>CoR</td>
<td>✓</td>
</tr>
<tr>
<td>Command</td>
<td>✓</td>
</tr>
<tr>
<td>Interpreter</td>
<td>✓</td>
</tr>
<tr>
<td>Iterator</td>
<td>✓</td>
</tr>
<tr>
<td>Mediator</td>
<td>✓</td>
</tr>
<tr>
<td>Memento</td>
<td>✓</td>
</tr>
<tr>
<td>Observer</td>
<td>✓</td>
</tr>
<tr>
<td>State</td>
<td>✓</td>
</tr>
<tr>
<td>Strategy</td>
<td>✓</td>
</tr>
<tr>
<td>TemplateMethod</td>
<td>✗</td>
</tr>
<tr>
<td>Visitor</td>
<td>✓</td>
</tr>
</tbody>
</table>
in Section III. We have reproduced the results here to illustrate the effectiveness of our obfuscation.

In Figure 9 (b), we see that Similarity Scoring was unable to detect any design patterns in the our obfuscated code. PINOT did detect a few patterns, but most of these are false positives; as we have seen previously, PINOT tends to produce a significant number of false positives. For example, the two Observer patterns detected by PINOT in Figure 9 (b) are both false positives that are actually Visitor patterns.

B. Runtime Analysis for Case 1

Runtime analysis for the individual patterns in Test Case 1 appears in Figure 10. Note that in some cases, obfuscation greatly improves runtime (e.g., AbstractFactory and Bridge). This improvement tends to occur when class-coalescing is used most heavily. Interestingly, class-splitting does not tend to cause an increase in runtime, as might be expected. Overall, the results in Figure 10 indicate that our obfuscation technique is likely to have no detrimental effect on the runtime performance of most code.

C. Test Case 2

For this test case we used GoF patterns from Grand’s book [18]. These patterns make heavy use of object oriented structures, such as inner classes and multilevel inheritance. Figure 11 gives detection results for PINOT and Similarity Scoring, for both the unobfuscated code and after our obfuscation is applied. Again, we see almost no patterns detected by Similarity Scoring while most of the PINOT-detected patterns are false positives.
V. CONCLUSION AND FUTURE WORK

In this paper, we showed that standard Java obfuscation tools have little effect on the ability to recover design patterns. We then discussed and tested an obfuscation strategy aimed at thwarting design pattern recovery. We provided test results showing that our technique is highly effective, and has no adverse effect on runtime performance. In this research, we focused on the 23 GoF design patterns, but our technique should be equally effective at preventing any high-level design recovery.

Our obfuscation technique employs class-coalescing and class-splitting. While this approach is highly effective, it could likely be further improved by including type-hiding, as suggested in [20]. Another area of future work is to improve the usability of the current tool, which is simply a proof-of-concept prototype.

REFERENCES

[12] E. Gamma, et al, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley