Spring 2012

JAVA DESIGN PATTERN OBFUSCATION

Praneeth Kumar Gone
San Jose State University

JAVA DESIGN PATTERN OBFUSCATION

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Praneeth Kumar Gone

May 2012

The Designated Project Committee Approves the Project Titled

JAVA DESIGN PATTERN OBFUSCATION

by

Praneeth Kumar Gone

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2012

Dr. Mark Stamp	Department of Computer Science
Dr. Chris Pollett	Department of Computer Science
Dr. Jon Pearce	Department of Mathematics

ABSTRACT

Java Design Pattern Obfuscation

by Praneeth Kumar Gone

Software Reverse Engineering (SRE) consists of analyzing the design and implementation of software. 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, porting old software to a modern programming language, and analyzing code for security vulnerabilities. Attackers also use SRE to probe for weaknesses in closed-source software, to hack software activation mechanisms (or otherwise change the intended function of software), to cheat at games, etc.

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.

ACKNOWLEDGMENTS

I would like to thank Dr. Mark Stamp, for guiding me through this research project and working with me to achieve this. I also thank him for his suggestions and contributions for handling some of the difficulties faced during the course of this project. Without him, this would not have been possible.

I would like to thank members of the committee, Dr. Chris Pollett and Dr. Jon Pearce, for their valuable feedback, encouragement and advice.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

CHAPTER 1

Introduction

Business organizations and companies spend an immense amount of time and money on computer software development in order to drive a product from conception to release. Important software development tasks include the design and integration of complex modules into an application. In the field of software system developers, a need to understand how existing software works is imperative. This process of analyzing source code and technological principles such as functions, structures from existing software binaries (exes, C/C++ object files or java class files) is called Software Reverse Engineering (SRE) [8]. SRE involves an analysis of a software systems components, its internal structure, or its design from software binaries. An SRE is used for purposes listed below [6, 9]:

1. Documentation of the software or to understand how existing software works to extend its functionality;
2. Monitoring access to resources like files, system registry [27];
3. Security audit of software applications [48];
4. Cryptanalysis of famous cryptographic systems;
5. Reverse engineering of Protocols, typically includes reversing of encryption and hashing functions [48];
6. Software benchmarking, verification of software includes analyses and match user expectations [48];

7. Analysis of digital rights of a system;

8. Cheating in games;

9. and Stealing and replicating someone's idea.

1.1 Reverse Engineering

Software reverse engineering (SRE) can be used for analysis of executable files [17]. Examples of such analysis includes redocumenation of programs [4], code smell detection [12], renewal of software modules [28], migration of legacy code [5], translation of program from one language to another [25] and architecture recovery like recovery of design patterns [2]. SRE is also used in piracy of software, and other illegal activities.

Software Reverse Engineering (SRE) is used for the analysis of executable files [17]. Examples of these analyses includes re-documentation of programs [4], code smell detection [12], renewal of software modules [28], migration of legacy code [5], translation of a program from one language to another [25] and architecture recovery such as recovery of design patterns [2]. SRE is also used in software piracy, and other illegal activities.

SRE of the software binaries is accomplished in three steps [6]:

1. Parsing and semantic analysis of code;

2. Extracting information from the code;

3. and Dividing the product into components.

Disassembling tools are used to parse software binaries and then a semantic analysis is performed on each parsed code. Information gathered is stored in an

informational base and is used to understand the softwares design and functionality. Details obtained can be used to develop functionally superior software or similar software with different abstractions.

Essential tools for SRE are both a disassembler and debugger [27]. Disassemblers are used for software parsing by converting an executable (Windows exe) to the assembly code. Issues faced in the disassembling process are separating the code and gathering data where there is no information regarding variable names and/or label names. Disassembling an executable file for analysis and reassembling it into a functioning executable file is not always possible. Debuggers are used for dynamic analysis tasks such as setting break points while the software executes and analyzes the program flow.

Software Reverse Engineering for Java software programs is much simpler than a native assembly code. The Java source code, when compiled, generates bytecode that will be executed by a Java Virtual Machine (JVM). This byte code will have more information regarding the source code than a native code will contain. This creation of a bytecode that runs on JVM makes the Java language platform independent as shown in Figure 1 below.

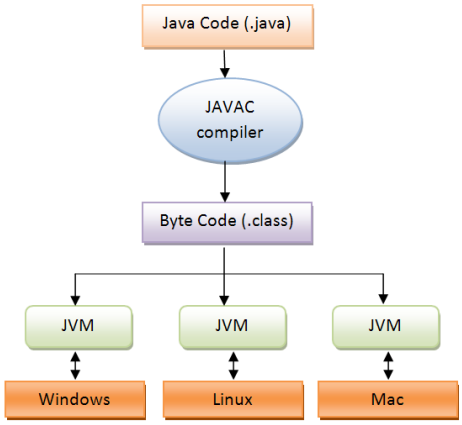

Figure 1: Platform independent Java language.

A JVM must be installed on a computer in order to use Java language. To execute a Java application, these intermediate symbols are read by a JVM and are converted to machine code in order to run the process as shown in Figure 2 below.

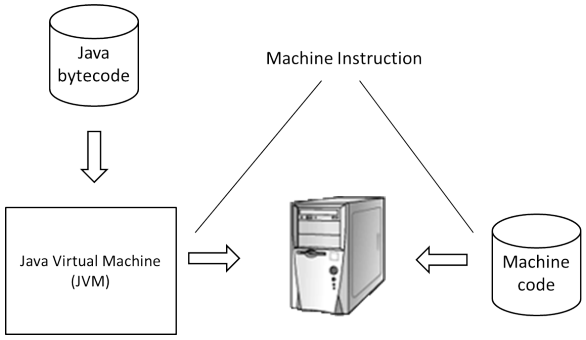

Figure 2: Execution of Java bytecode vs Machine code.

The SRE of a Java bytecode, as explained, uses a HelloWorld example that involves a mapping of the HelloWorld binary to a Java class file format; the extraction of information occurs by opening the Java class file using a text editor. The HelloWorld binary is shown in Figure 3 and the mapping of a HelloWorld binary to Java class file format is shown in Figure 4 and Figure 5.

The Java class file format specification document [23] gives details regarding the protocol used in forming class files by using symbols defined in the bytecode. There are 10 basic sections to the Java class file structure [47] as demonstrated below:

1. **Magic Number:** 0xCAFEBABE
2. **Version of Class File Format:** the minor and major versions of the class file
3. **Constant Pool:** this pool consists of constants of the class
4. **Access Flags:** specified whether the class is abstract, static etc.,
5. **This Class:** name of the current class

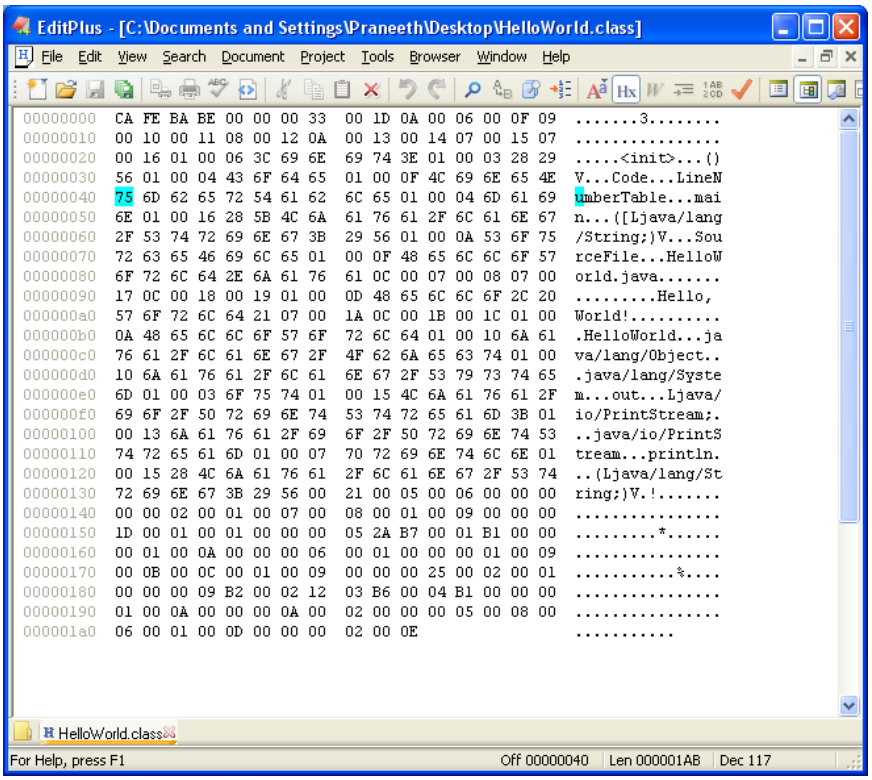

Figure 3: Hex view of HelloWorld.class file.

6. **Super Class:** name of the super class

7. **Interfaces:** interfaces used in the class

8. **Fields:** fields declared in the class

9. **Methods:** methods declared in the class

10. **Attributes:** such as name of the source file, etc.,

An analysis of a HelloWorld class file [38], according to the Java class file format,gives information such as method references within the class; an example would be `System.out.println` with the parameter `HelloWorld!` inside the `main` method. A native code in assembly language will not provide this information regarding the

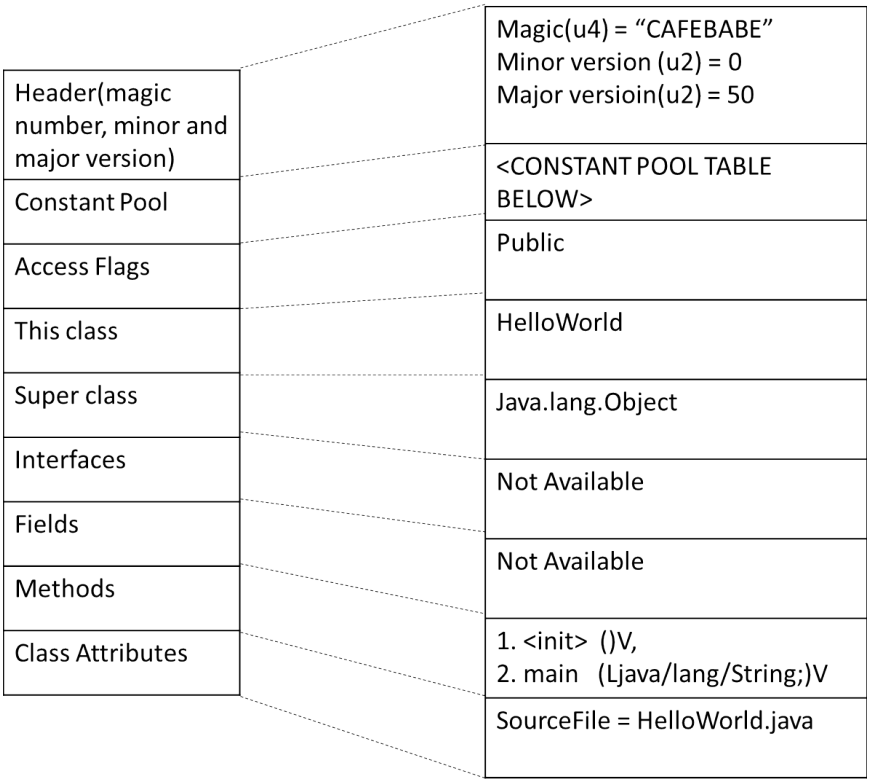

Figure 4: Class file view (HelloWorld.class).

software binary and its reverse engineering is more complex and tedious, even with the use of SRE tools.

This SRE example of using a simple HelloWorld program demonstrates a mechanism for using and extracting basic information through a text editor, without the help of an SRE tool. Extensive documentation detailing the Java bytecode and the JVM is readily available and helpful in the development of an automated SRE tool in order to decompile Java class files. Presently, tools, such as decompilers and pattern detection tools are readily available to reverse engineer Java source and class files. Decompilers are used to extract source code from Java binaries, and pattern detection tools are used where analyzing design patterns from an extracted Java source or its binaries is needed.

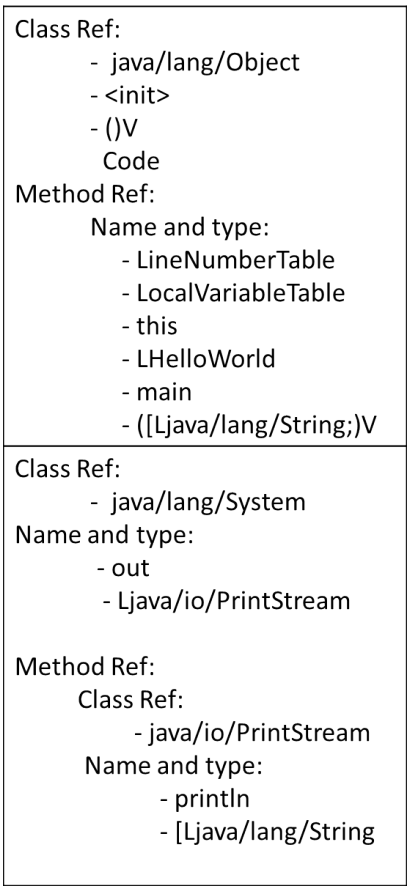

Figure 5: Constant Pool table.

1.2 Anti-reverse Engineering

Anti-reverse engineering is the process to protect software from reverse engineering. The software community is facing a tough challenge in order to protect software from attackers and to prevent its misuse. According to [9], The patent system is not quite as effective with software as it is with traditionally engineered tangible artifacts. While a patent mandates IP protection it is next to impossible to prove or even suspect any IP theft in a software product that might have been the result of a malicious reverse engineering attack on a patented competitor. The goal of any anti-reverse engineering technique is to increase the amount of work needed to reverse engineer software and increase the reversing time beyond the life-time of a software

application.

1.3 Reversing and Anti-reversing Tools

There are many tools available in order to reverse engineer software applications: reverse engineering tools such as disassemblers that extract system blocks information using an assembly code from an executable file; debuggers used for dynamic analysis; and decompilers to gain the source code from software binaries. Tools can be open source and/or commercial software [37]. Most use disassembling and debugging tools available include OllyDbg, IDA Pro, and WinDbg. Apart from disassembling and debugging, these tools can also produce additional information [9].

A Java byte code is not created in a human readable format; in fact, it has a close resemblance to the source code and will help in understanding basic details of a program. The tools available for reverse engineering Java are JaD [36], JODE [22], MOCHA [18], DJ Java decompiler [3], and PINOT [34].

In order to defend the need for reversing software binaries, necessary research is being conducted within the software community. Suggested mechanisms for making it more difficult to reverse engineer software include: software obfuscation; physically protecting a software application platform; encrypting an executable; and watermarking the software [9]. The main idea of our project is to use code obfuscation; we start with an analysis of the design pattern detection tools and implement our obfuscation tool by following the techniques described in [29].

1.4 Related Work and Project Goal

Software obfuscation is an anti-reverse engineering mechanism that changes the structure of a given code with no change to its functionality. There are three types

of obfuscation used: control-flow obfuscation in order to obscure the flow of control; dataobfuscation that makes understanding data fields difficult; and layout-obfuscation where we split logic into separate procedures. Many obfuscation tools are available; a list of open source obfuscation tools is given in [35, 14] and these are ProGuard [13], yGuard [49], SandMark [7], jarg [19], bb_mug (BebboSoft) [41], JavaGuard [43], as well as commercial tools such as Allatori [40], Zelix KlassMaster [50], and JShrink [11]. These obfuscation tools support functions such as:

1. Rename class, method, field and local names to some random meaningless strings;
2. Removing debugging information;
3. Removing dead code and constant fields (Shrinking the code);
4. Optimizing local variable allocation;
5. and Exception Obfuscation.

The tools listed above implement a control-flow obfuscation, data-obfuscation and layoutobfuscation; however, these tools do not perform a design level obfuscation. Functions supported by these tools do not bring a structural change in the class level (i.e., removing interfaces, adding abstractions) and will not bring a major change in class level interactions. Obscuring a design requires change in the relationship between software system class components. Design patterns mainly use an inheritance feature of an object oriented program; we need to obscure this inheritance level relationship between classes in order to hide and shield architecture from a pattern detection tool.

Present obfuscation tools do protect programs and make it difficult to reverse engineer. However, these obfuscation tools cannot hide the software system design

level mechanism. In Chapter 3, we describe obfuscation using current available tools that do not show hiding patterns from detection tools. These experiments clearly demonstrate that three obfuscations, control-flow; dataflow; and layout obfuscations, do not completely obfuscate the design level architecture.

Design obfuscation is necessary in order to obfuscate object oriented programs that protect applications from reverse engineering. Software obfuscation techniques described in [29], class-coalescing, class-splitting, and type-hiding, can be used for design obfuscation. In a class-coalescing technique two or more classes are combined into a single class; for class-splitting a single class is split into a number of classes with the functionality divided between them. Type-hiding is used to increase the number of interfaces that are implemented by classes.

These techniques are implemented into Design Obfuscator Java (DOJ) application [29] that uses Soot optimization framework to analyze the bytecodes within an application. A DOJ tool design obfuscation was tested on medium and large sized programs. Results demonstrate that class-splitting resulted in an overhead of 10

Tasks completed in this project were to implement a design obfuscation tool that will obfuscate 23 Gang of Four (GoF) design patterns that hide an internal architecture from reverse engineering. Design obfuscation techniques, described in [29], such as class-coalescing and class-splitting are applied to binaries. This obfuscated source is tested for design pattern detection by running pattern detection tools. Before and after obfuscation results are compared and analyzed in Chapter 5.

1.5 Organization of the Report

This project report is organized as follows:

- Chapter 2 will describe the decompilers used in Java reverse engineering, introduction to design patterns, and pattern detection tools.
- Chapter 3 explains an obfuscation approach used for important design patterns.
- Chapter 4 presents the implementation details used by the developed software tool.
- Chapter 5 shows results from two pattern detection tools, and comparing proposed obfuscator to present obfuscators.
- Chapter 6 concludes our report and provides future work.

CHAPTER 2

Reverse Engineering Java

The reverse engineering of a Java class file is simple, as class files contain most of the information about the original source code. Many decompilers, and design extraction tools for source and class files are available.

2.1 Java Decompilers

A Java decompiler is used to covert our java class files (*.class) into source code files (*.java). Many software applications do not provide their source code; however these applications can be reverse engineered by using decompilers in order to obtain source Java files for analysis. Many Java decompilers are available [37], and a few effective tools are as follows:

1. Jad Java decompiler [36]
2. DJ Java decompiler [3]
3. JODE decompiler and optimizer [22]
4. Mocha java decompiler [18]
5. JD (Java Decompiler) [21]

Our list includes a decompiler that can be run through either a command line or GUI. The inner working of these decompilers by decompiling HelloWorld class file and evaluation of various decompilers [20], will be explained in the next subsections.

2.1.1 Reverse Engineering HelloWorld Java program

By decompiling a simple HelloWorld class file we can demonstrate the working of all these decompilers. The following below in Figure 6 is the code for the HelloWorld.java program. We compiled this program using a Java compiler in order to create a HelloWorld.class file.

```
//HelloWorld.java
import java.lang.*;
public class HelloWorld {
      public static void main(String args[]) {
            System.out.println("Hello World!");
      }
}
```

Figure 6: HelloWorld Source

2.1.1.1 JAD

The Java Decompiler (JAD) is an old decompiler and currently not under maintenance [36]. JAD provides a command line interface that decompiles class files to Java files. Decompiled program of HelloWorld.class file is as shown in Figure 7.

```
// Decompiled by Jad v1.5.8f. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   HelloWorld.java

import java.io.PrintStream;
public class HelloWorld
{
    public HelloWorld()
    {
    }

    public static void main(String args[])
    {
        System.out.println("Hello World!");
    }
}
```

Figure 7: JAD decompiled HelloWorld

2.1.1.2 DJ Java Decompiler

A DJ Java decompiler is the GUI version of JAD and one of the most widely used decompilers. A full version of DJ Java decompiler is available for $19.99 [3], and a decompiled HelloWorld program, using a Trial version, is shown in the Figure 8.

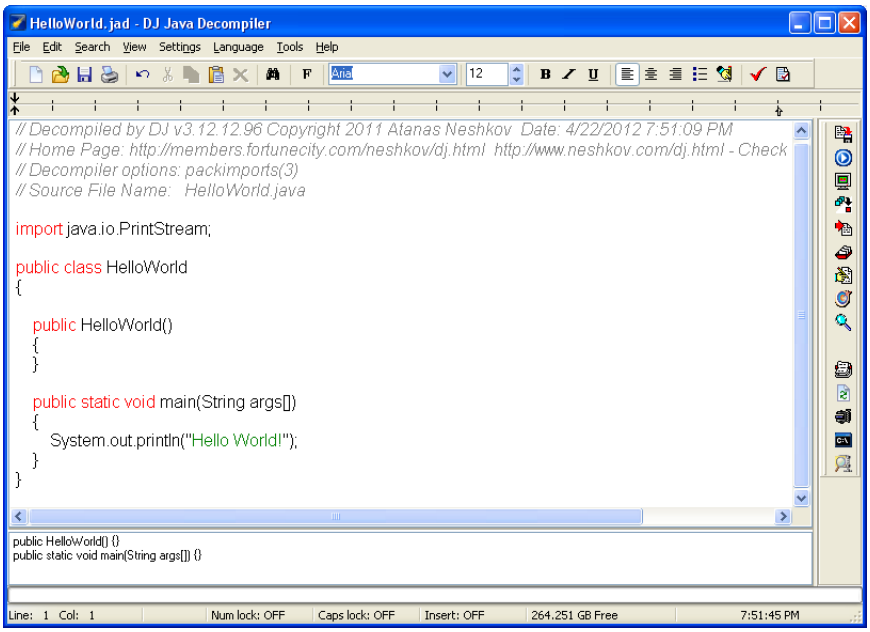

Figure 8: DJ Java decompiled HelloWorld.java

2.1.1.3 JODE

After installing the JODE decompiler, we start decompiling the process of our HelloWorld.class file using the following command as shown in Figure 9 below:

```
>java jode.decompiler.Main --classpath "C:\" HelloWorld
Jode (c) 1998-2001 Jochen Hoenicke <jochen@gnu.org>
HelloWorld
/* HelloWorld - Decompiled by JODE
 * Visit http://jode.sourceforge.net/
 */

public class HelloWorld
{
    public static void main(String[] strings) {
        System.out.println("Hello World!");
    }
}
```

Figure 9: JODE decompiled HelloWorld.java

JODE can decompile all class files that do not contain any dependencies; it displays an error trying to decompile class files with dependencies and also it cannot

understand complex expressions or statements within the program.

2.1.1.4 Mocha

This decompiler is also not under maintenance and it reportedly has problems decompiling class files created by recent a Java compiler version [18]. The decompiled HelloWorld.java program is shown in Figure 10 below:

```
/* Decompiled by Mocha from HelloWorld.class */
/* Originally compiled from HelloWorld.java */
import java.io.PrintStream;
public class HelloWorld
{
       public static void main(String astring[])
       {
              System.out.println("Hello, World");
       }
       public HelloWorld( )
       {

       }
}
```

Figure 10: Mocha decompiled HelloWorld.java

2.1.1.5 JD - Java Decompiler

A Java decompiler (JD) is available as a library JD-Core, in GUI JD-GUI and a plugin for eclipse JD-Eclipse [21] is also available. A decompiled HelloWorld class file, on JD-GUI, is shown in Figure 11:

2.1.2 Evaluation of Decompilers

Evaluating decompilers requires tests that contain different types of source code; an example would be a code with exceptions, try-catch blocks, and interface abstractions. In [20] the authors used a set of classes to evaluate commonly used and available

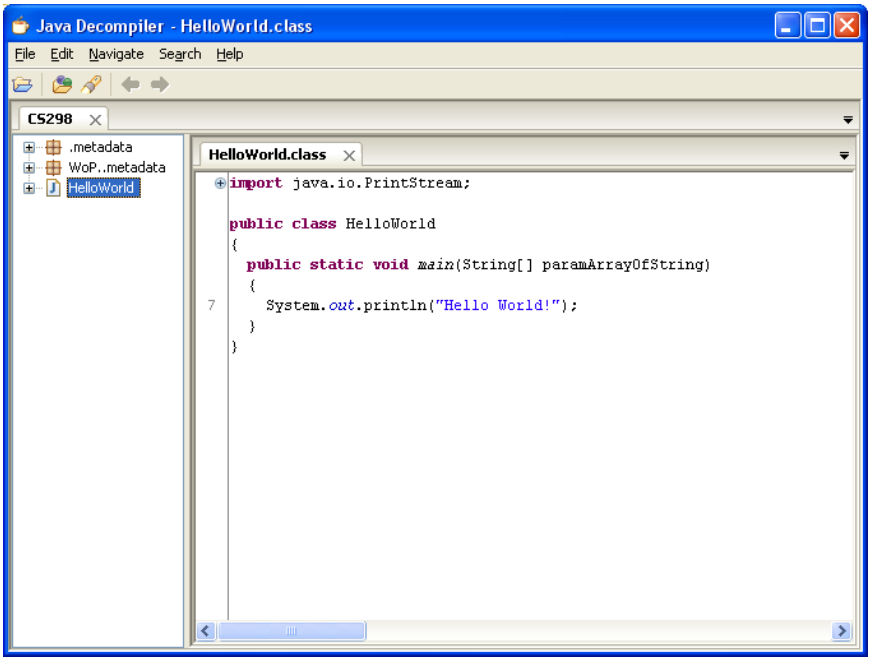

Figure 11: JD decompiled HelloWorld.java

decompilers. The decompilers that were evaluated are shown in Figure 12, below.

decompiler	type	status	2003 version	current version	last update
Mocha	free	obsolete	0.1b	0.1b	1996
SourceTec	commercial	obsolete	1.1	1.1	1997
SourceAgain	commercial	obsolete	1.10j	1.1	2004
Jad	free	unmaintained	1.5.8e	1.5.8e	2001
JODE	open-source	unmaintained	unknown	1.1.2-pre1	2004
ClassCracker3	commercial	obsolete	3.01	3.02	2005
jReversePro	open-source	unmaintained	1.4.1	1.4.2	2005
Dava	open-source	current	2.0.1	2.3.0	2008
jdec	open-source	current	N/A	2.0	2008
Java Decompiler	free	current	N/A	0.2.7	2008

Figure 12: Java Bytecode Decompilers [20]

The decompilers are scored according to output results shown in Figure 13, and decompiler test results are shown in Figure 14.

Decompiler results conclude that ClassCracker3, jdec, jReversePro, Mocha, SourceTec (Jasmine) decompilers preform poorly and are not able to decompile a single program from the given set. Dava, Jad, and JODE decompilers performed similarly when decompiling five programs from the set; but for the most part, JODE

Score	semantics	syntax	output result	examples
0	correct	correct	semantically and syntactically correct program with perfect/-good source code layout	perfect decompilation
1	correct	correct	semantically and syntactically correct program with 'ugly' source code layout and/or no type inference	unreconstructed control flow statements, unreconstructed string concatenation, unused labels, no type inference
2	incorrect	incorrect	easy-to-correct syntax errors which produce a semantically correct program	boolean typed as int, missing variable declaration
3	incorrect	incorrect	difficult (but possible) to correct syntax errors which produce a semantically correct program	code with goto statements
4	incorrect	incorrect	very difficult (or nearly impossible) to correct syntax errors required to produce a semantically correct program	invalid variable use, obviously incorrect code, massive source re-write required
5	incorrect	correct	easy to correct semantic errors which produce a semantically correct program	missing typecasts
6	incorrect	correct	difficult (but possible) to correct semantic errors which produce a semantically correct program	incorrect control flow
7	incorrect	correct	very difficult (or nearly impossible) to correct semantic errors required to produce a semantically correct program	incorrectly nested try-catch blocks, massive source re-write required
8	incorrect	incorrect	incomplete decompilation	missing large sections of source, missing inner classes
9	Fail	Fail	decompiler fails upon execution/produces no source output	decompiler fails to parse arbitrary bytecode

Figure 13: Decompilation Correctness Classification [20]

correctly decompiled the programs. The JD decompiler out-performed Dava, and Jad in recovering most of the source programs within the given set. SourceAgain is the only commercial decompiler that performed well, however it is no longer sold or supported.

2.2 Design patterns and Pattern detection tools

Pattern detection tools were implemented in order to detect design patterns. Software systems are built using design patterns in order to solve design specific problems. There are at least 250 existing patterns that are used in an object oriented world. The 23 GoF patterns are well known and widely used patterns. In this section, we will discuss, in brief, the 12 GoF patterns using examples, followed by

		ClassCracker3	Dava	JAD	Java Decompiler	jdec	JODE	jreversepro	Mocha	SourceAgain	SourceTec
javac	Fibonacci	8	0	0	0	0	0	9	9	0	9
	Casting	8	5	5	5	5	0	5	9	5	9
	InnerClass	8	8	2	0	8	9	8	9	8	9
	TypeInference	8	2	1	2	4	0	9	9	1	9
	TryFinally	8	4	8	0	0	4	8	9	4	9
	ControlFlow	8	1	1	5	4	1	8	9	1	9
arbitrary	Exceptions	8	0	4	4	8	9	9	9	7	9
	Optimised	8	2	4	3	4	2	3	9	3	9
	VariabeReUse	8	1	2	2	3	0	2	2	0	2
	Ada	8	3	9	4	8	9	8	4	3	4

Figure 14: Decompiler Test Results [20]

pattern detection tools and their efficiency.

2.2.1 Design Patterns

Computer science software design problems are solved using design patterns. These design patterns are reusable and use object oriented techniques that provide a design for software development in various fields. In our project we will address the 23 GoF design patterns [15]. Design patterns are grouped into three categories [15] creational patterns, structural patterns, and behavioral patterns. In this subsection we will discuss 12 design patterns, problems each pattern resolves, and when a given pattern must be applied using an example. These 12 design patterns will also be used to explain obfuscation techniques in Chapter 3. The 23 GoF design patterns are divided into categories are listed below.

Creational Patterns: These are design patterns that deal with the creation of an object according to a given situation. Creational patterns are formed by encapsulating the knowledge of given object in order to create and hide the instances of how these objects are created. There are five creational patterns listed below.

1. AbstractFactory
2. Builder
3. FactoryMethod
4. Prototype
5. Singleton

Structural Patterns: In software engineering, structural patterns solve problems of realizing relationships between different entities. A software system consists of a number of classes interacting with each other in order to complete an application. The type of structural pattern is often selected according to a given situation examples include: adapting an object, and creating complex type from simpler types. Below is the list of structural patterns:

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Facade

6. Flyweight

7. Proxy

Behavioral Patterns: These patterns solve design problems by implementing a common communication and implementation between entities. Communicating between entities involves mediating between classes, notifying the state of an object, and selecting different algorithms at run time. The list of behavioral patterns is given below:

1. Chain of Responsibility

2. Command

3. Interpreter

4. Iterator

5. Mediator

6. Memento

7. Observer

8. State

9. Strategy

10. TemplateMethod

11. Visitor

A description of commonly used and important design patterns is explained below along with examples. Obfuscation of these examples, types of obfuscation, and obfuscation testing will be explained further in Chapter 3.

2.2.1.1 FactoryMethod Pattern

This pattern solves the problem of creating objects without specifying an exact class initialization of an object. 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 and the derived type object is used further within the application [10].

Example:

Below we see a Factory Product example; it has a Factory Interface that specifies generic behavior for the products. The `Client`, when using identification details, requests a product from the `ConcreteFactory` in order to initialize the `Product` variable which uses concrete products. The `ConcreteProduct` is an implementation of the `Product` interface; there can be different implementations depending on the type of product. The UML diagram of a Factory Product example is shown in Figure 15.

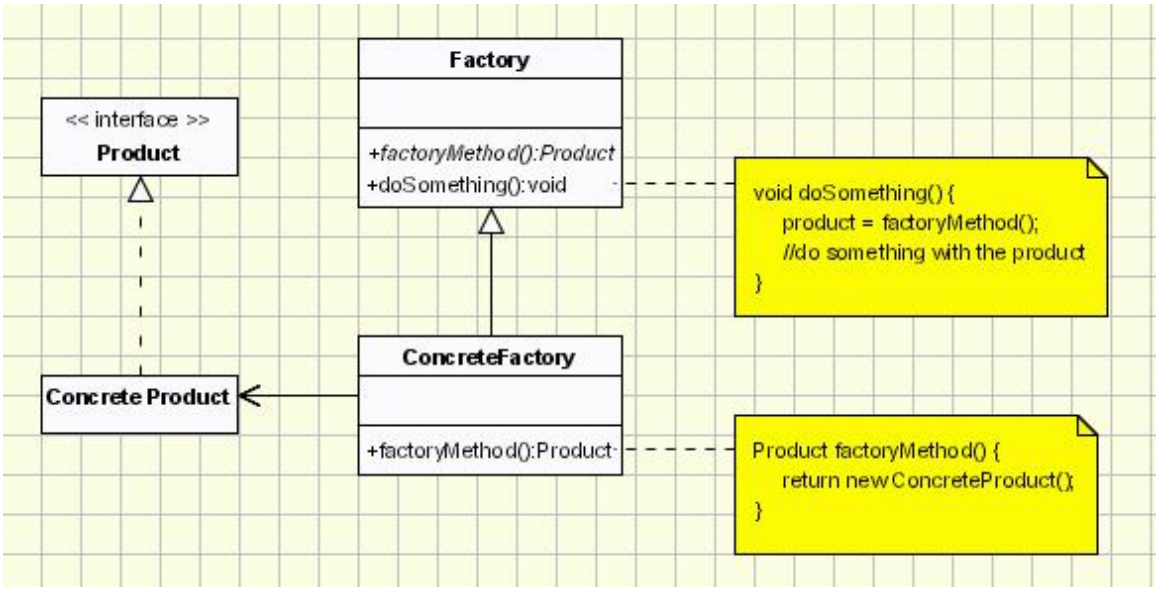

Figure 15: UML diagram of FactoryMethod Pattern [10]

2.2.1.2 AbstractFactory Pattern

The AbstractFactory is also a creational pattern used to create a family of related products without explicitly specifying their classes. Consider the example of an AbstractFactory class that can create products from two product families: AbstractProductA and AbstractProductB. The UML diagram for this program is shown in Figure 16.

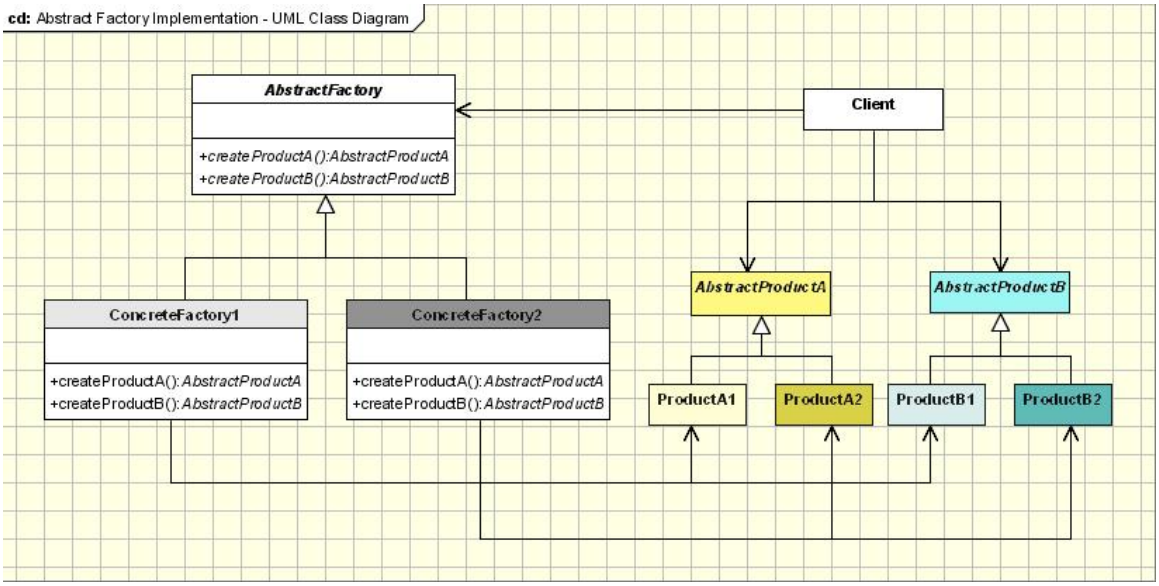

Figure 16: UML diagram of AbstractFactory Pattern [10]

Example:

An `AbstractFactory` is the abstract class that creates concrete classes where specific products are created. Product creation is accomplished through different abstraction implementations, namely `AbstractProductA` and `AbstractProductB`. When a client wants to change a product type a new concrete factory can be easily assigned to the `AbstractFactory` class and then new set of concrete products can be created.

2.2.1.3 Builder Pattern

This pattern is used to build complex products using several small objects within the application. Developing a complex application requires complex classes and objects; these complex objects can be developed using some smaller objects that follow a defined algorithm. A Builder pattern can be used to create complex objects using smaller objects according to an algorithm or procedure. Figure 17 shows the UML diagram for the Builder pattern.

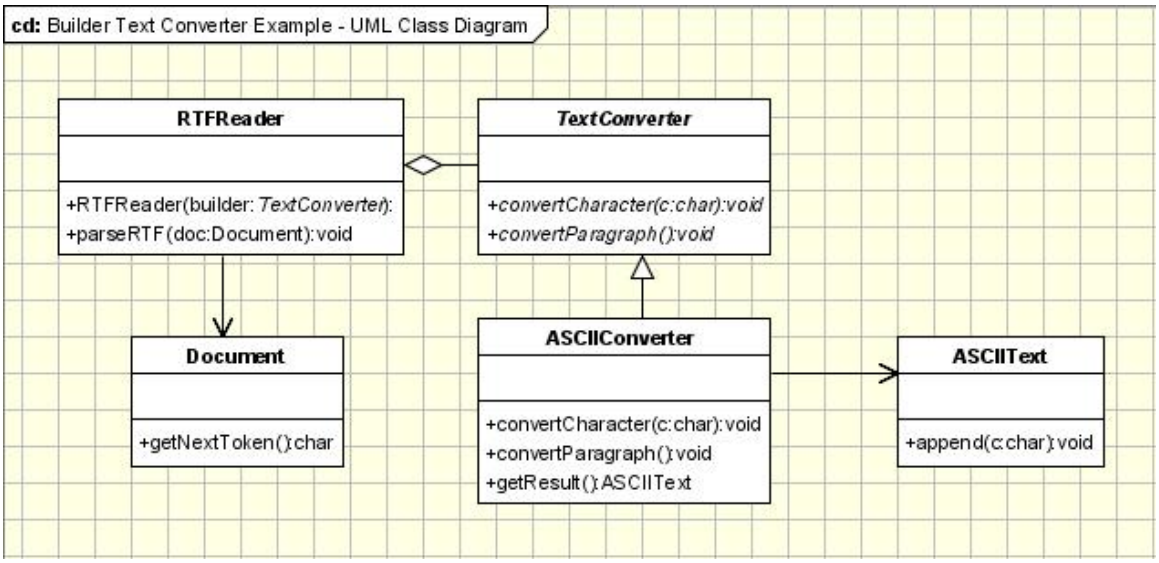

Figure 17: UML diagram of Builder Pattern [10]

Example:

The 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.

2.2.1.4 Adapter Pattern

This pattern is used to solve the problem of adapting an object to a particular operation. During software development, it is expected that the need for an object and another class that offers the same features and implements another interface will be required. Using both will diminish the need for re-implementing one of them. An Adapter pattern is used for this purpose and is used to implement required features. Figure 18, below, shows the UML diagram for this Adapter pattern.

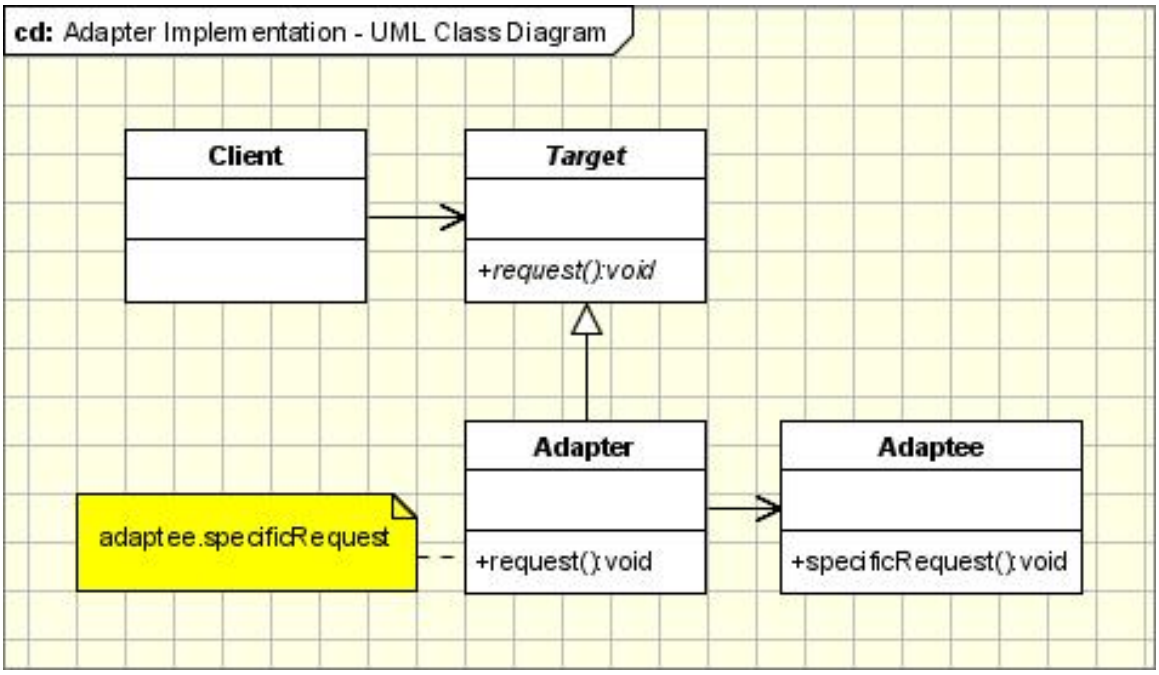

Figure 18: UML diagram of Adapter Pattern [10]

Example:

An `Adapter` class uses an `Adaptee` delegation in order to adapt to the request function overridden from the `Target` interface. A `Client` uses this `Target` interface to initiate different adapters and then uses them according to a given situation.

2.2.1.5 Bridge Pattern

A Bridge pattern is a structural pattern used for designing different abstraction implementations defined within the application. Often a single abstraction could contain different implementations. Consider the persistence of a data object over a different platform using either relational databases or file system structures (files or folders). A Bridge pattern can be used to decouple an abstraction from the implementation so that the two can vary independently. Figure 19, below, shows the UML diagram for the Persistence example using a Bridge pattern:

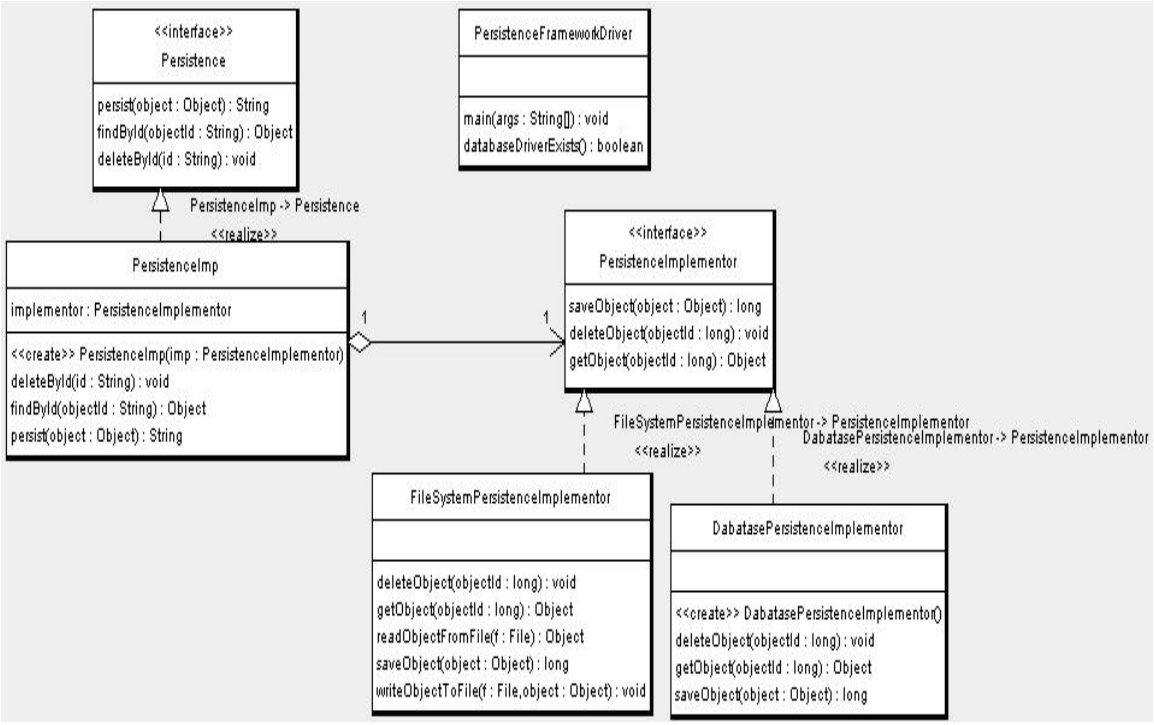

Figure 19: UML diagram of Bridge Pattern [10]

Example:

An example of a persistence API that can have many implementations for the presence of file system or relational database, is described below.

A `Persistence` Interface illustrates an `Abstraction` interface that can have

different implementations. A `PersistenceImplementor` is the `Implementor` interface can be implemented for a file system as a `FileSystemPersistenceImplementor` and for relational data base called a `DatabasePersistenceImplementor`. These `Implementor` implementations can be used as delegations to perform various functions within concrete `Persistence` implementations.

2.2.1.6 Flyweight Pattern

This design pattern is used to create a large number of objects with shared states. Some applications required a large number of objects with sharing states among them.

Example:

An example of a Wargame that needs to instantiate a large number of soldier objects is described. A soldier object has the graphical representation of a soldier, firing a weapon, and additional characteristics. Initiating a large number of soldier objects is necessary; however, this task will require a considerable amount of system memory. A Flyweight pattern can be used to create various soldier objects, by sharing states through a common function (in the present example moveSoldier function). The UML diagram for the Wargame application is shown in Figure 20.

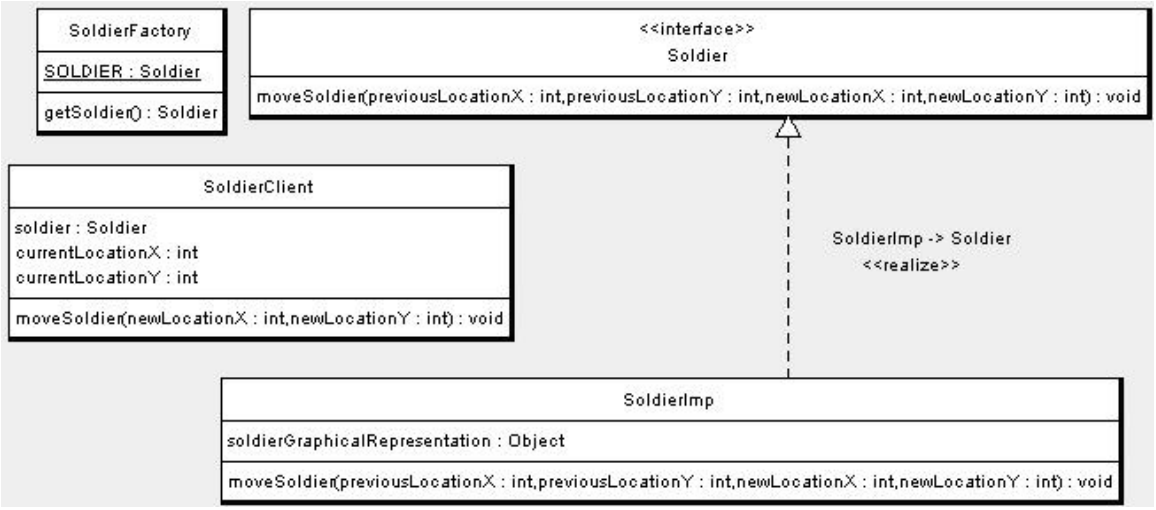

Figure 20: UML diagram of Flyweight Pattern [10]

2.2.1.7 Decorator Pattern

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

Example:

An example of a graphical window, used to create a `FrameWindow` class would decorate a `Window` class and a `FrameWindow` object created statically by the client program. This use of a `FrameWindow` needs to initiate different objects within the clients program. Decorator pattern can be used to create a `FrameWindow` dynamically, without creating objects in the clients program. The UML diagram demonstrating a Graphical Window application, using a Decorator Pattern, is shown in Figure 21.

The `Window` interface represents a component interface, and a `SimpleWindow` implements the `Window` interface in order to create a general window. A

`DecoratorWindow` is the implementation containing special and extra decorative added features. This `DecoratorWindow` can be extended by using various classes such as a `ScrollableWindow` class, that add special features to decorate a window as shown in Figure 21.

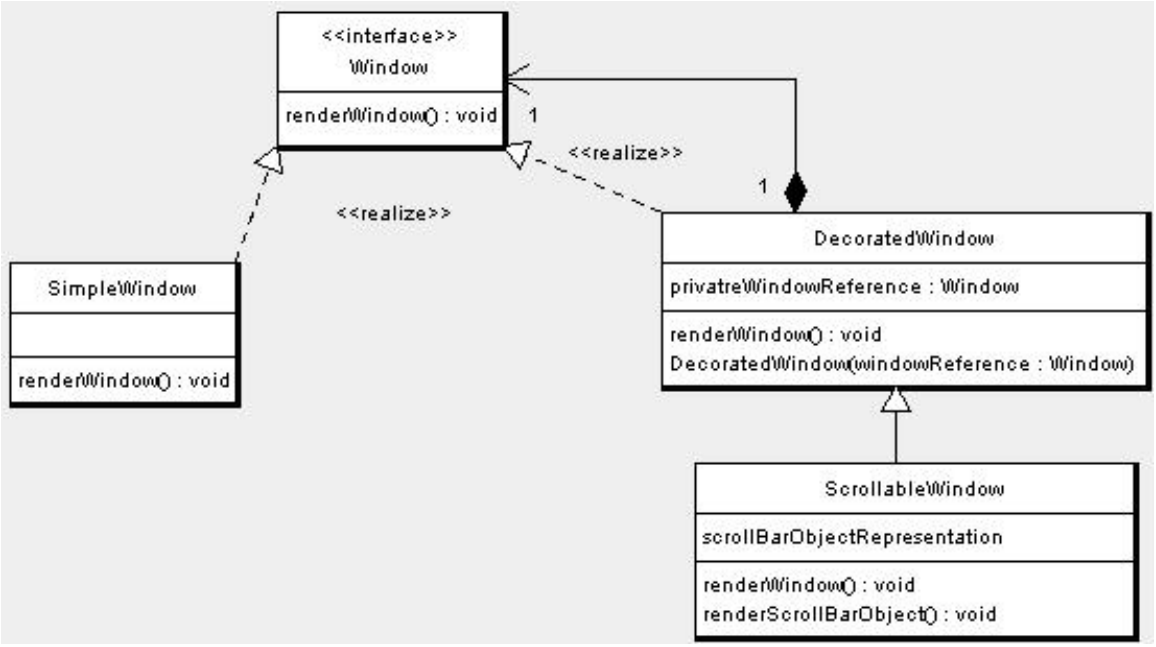

Figure 21: UML diagram of Decorator Pattern [10]

2.2.1.8 Mediator Pattern

A Mediator pattern is a behavioral pattern that aids in the interaction of large number of classes. A software project using object oriented design, will have classes that interact with each other in order to implement a particular application. If an algorithm or principle is not followed it is very difficult to understand and run the application. A Mediator pattern can be used to remove the tight coupling behavior of the above design. Figure 22, shows the UML diagram of a Mediator pattern.

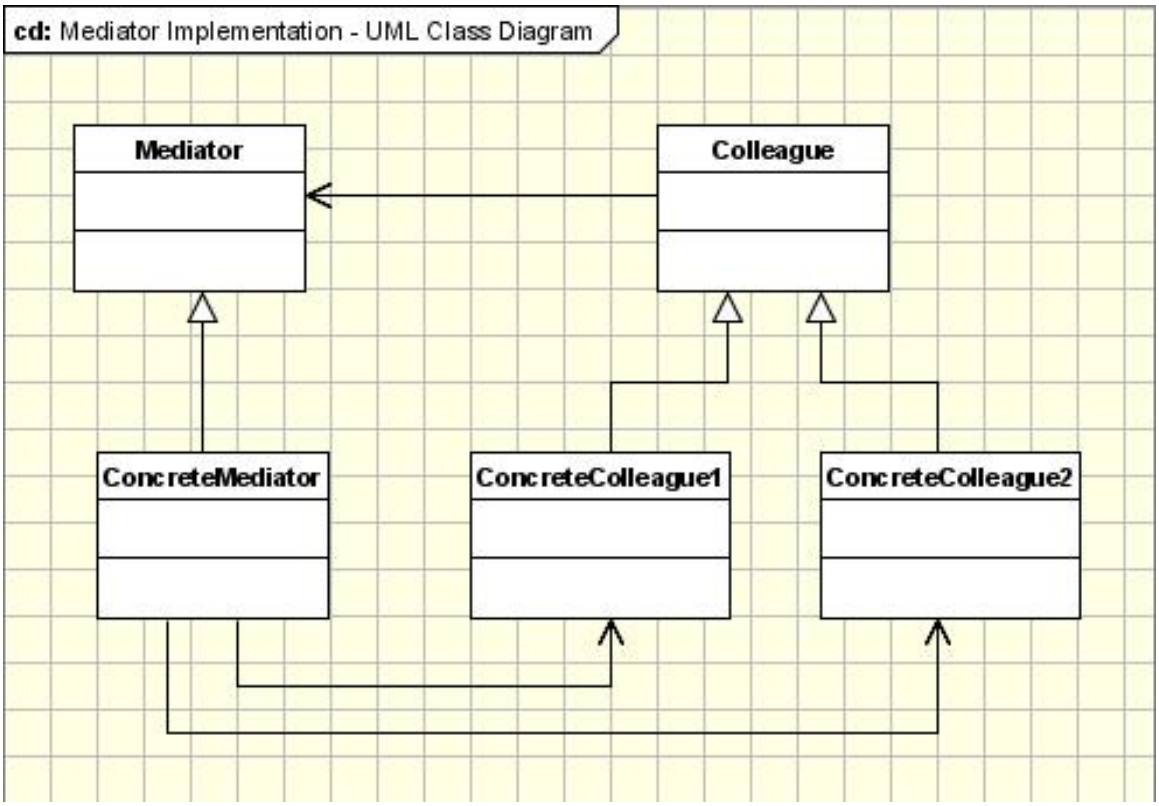

Figure 22: UML diagram of Mediator Pattern [10]

Example:

Consider an example of developing a screen that contains different controls. Various controls need to interact with other controls. For example, if a button is pressed it must determine if 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 in order to serve our purpose.

A `Colleague` is the abstraction interface that will be implemented by the concrete colleagues i.e., screens. All screens must determine the change on one screen and this information must be shared with other screens using the concrete mediator implementation from the `Mediator` abstraction.

2.2.1.9 Observer Pattern

An Observer pattern solves the problem of updating the state to certain other objects. In object oriented programming, objects have states and it is within these states changes are made within objects. In some cases, it is necessary to be informed about the changes occuring in within other objects. An Observer pattern can be used when a subject has to be observed by one or more observers.

Example:

Lets consider an example of news agency that publishes news to different subscribers, where subscribers can receive their news in different forms: Emails, and/or SMS.

A `NewsPublisher` class will act as an Observable interface and will be extended by the type of news it distributes such as business, sports, entertainment and so on. `Subscribers` (Email and SMS) will then act as Observers

A `NewsPublisher` keeps a list of all current `Subscribers` and informs them of the latest news. `NewsPublisher`, if there is change in the state of latest news, will notify all subscribers. Figure 23 shows the UML diagram of a `NewsPublisher` application.

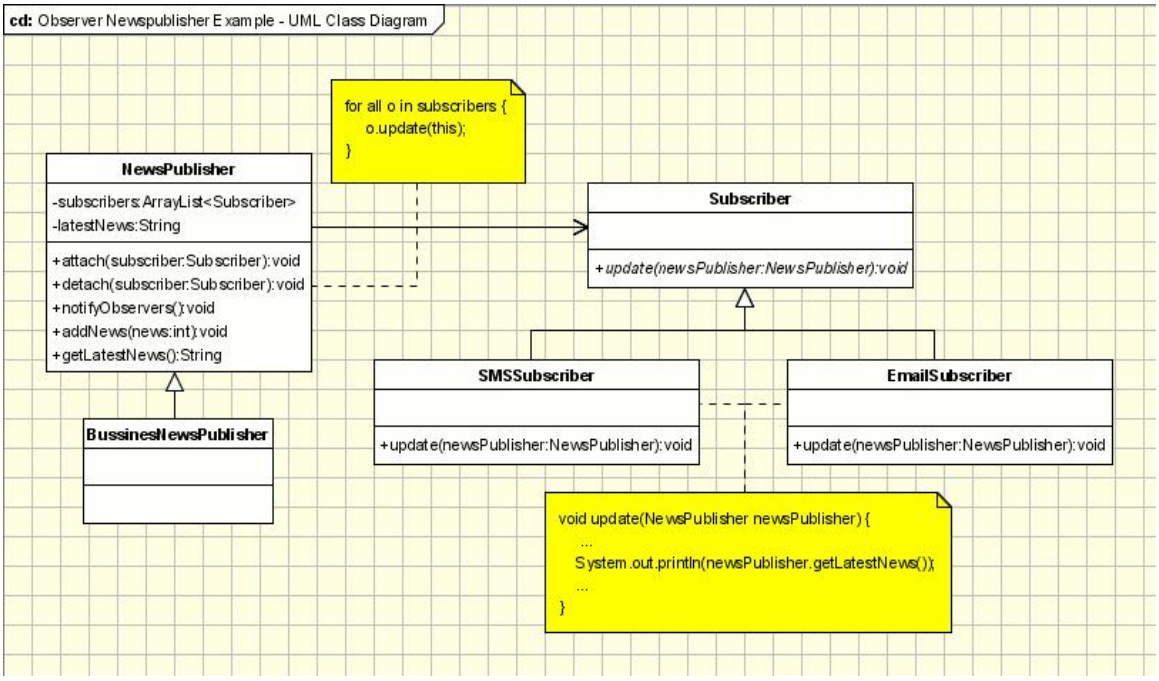

Figure 23: UML diagram of Observer Pattern [10]

2.2.1.10 Strategy Pattern

This pattern enables the user to select an appropriate class with a selected behavior at runtime. Some classes during software development differ only in their behavior. In this case, it is better to use a Strategy pattern where separate behaviors are developed into different classes, aloong with isolating algorithms enabling them to select an appropriate class at runtime.

Example:

Consider an example of the Robot simulation, this will have an `IBehavior` interface as its Strategy abstraction. This abstraction is implemented by behaviors such as the attack strategies of a robot. The UML diagram for the Robot application is shown in Figure 24. In the present example, aggressive, defensive, and normal strategies are implemented for the `IBehavior` interface.

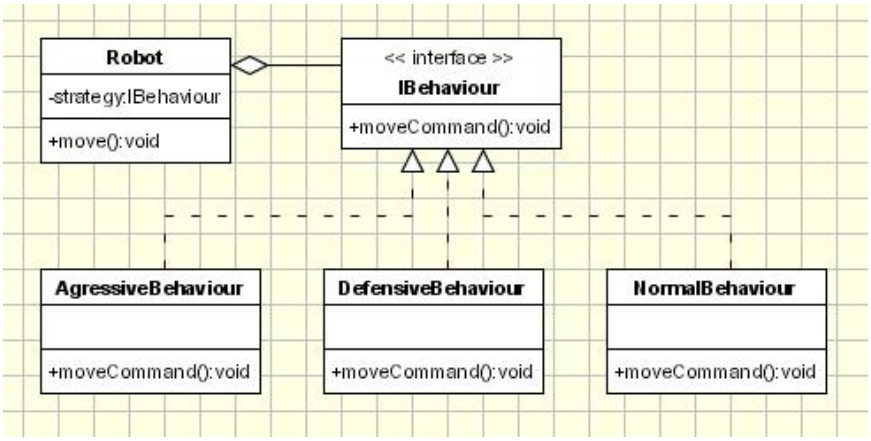

Figure 24: UML diagram of Strategy Pattern [10]

2.2.1.11 TemplateMethod Pattern

This pattern solves design problems through an abstract method that is implemented by the subclasses. In some applications it is necessary that a subclass implement all the necessary methods; these methods will be used to gain the final results. A Template method design pattern can be used for this purpose. In practice, the template method allows subclasses to override a few steps of the algorithm and the final step is performed by a Super class using implemented steps.

Example:

For example, a Travel agency has several trips to select from and to create a new class for each trip will require more memory. In this case a Template method pattern can be used, where a `Trip` abstract class has methods for an overall trip. Different trip packages must implement this `Trip` abstract class and the `Client` can call necessary functions to perform the total trip. The UML diagram for this process is shown in Figure 25. In this example the performTrip function will act as template method that has a total or a partial implementation of the entire trip. Once the abstract methods for each day are overridden by each Package, invoking a `performTrip` function will

complete the desired trip.

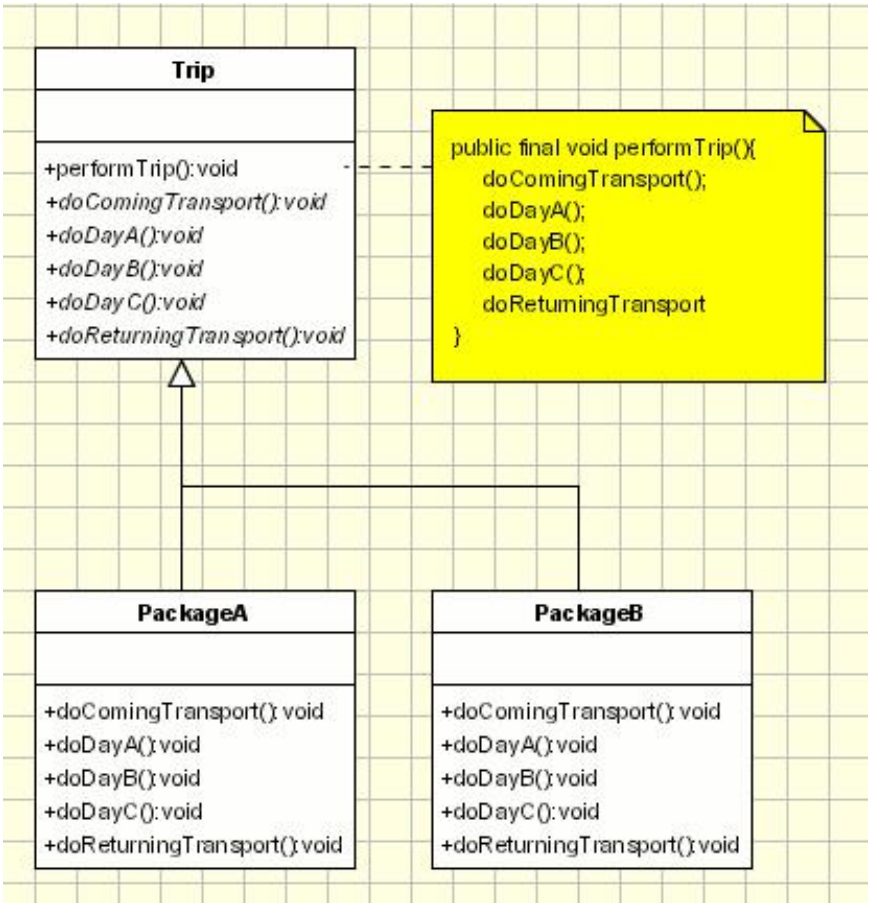

Figure 25: UML diagram of TemplateMethod Pattern [10]

2.2.1.12 Visitor Pattern

This pattern completes an operation on each element of the collection with different data types. Collections are data types widely used in all Object oriented programming languages. They often contain objects of different types. To perform a command operation on these collections we need to know the type of instances that are stored within the collection. After determining the instance types, we use instanceof function for each object to check and perform particular operations. This kind of checking is not object oriented and uses numerous if-else conditions. A Visi-

tor pattern can be used for this situation, where each object is visited and an object related operation is performed [10].

Example:

A Customers application can be considered an example of a Visitor pattern. In this application, a reporting module is created for a customer group. To gather statistics you need specific details for all the customers within a particular group. An `IVisitor` is the interface that has abstractions for all the `visit` methods within all the visitable objects. An Interface `IVisitable` should be implemented by all visitable objects and should override the `accept()` method. A `CustomerGroup`, `Customer`, `Order` and `Item` classes are considered visitable classes. The `GeneralReport` is the `IVisitor` implementation and output the customer statistics. The UML diagram of the Customers application is shown in Figure 26.

A `GeneralReport` will call visit methods for each customer present in `CustomerGroup` and obtain the number of orders for each customer and retrieve the number of items purchased for each order. Finally, a report is displayed demonstrating the Customer buying behavior.

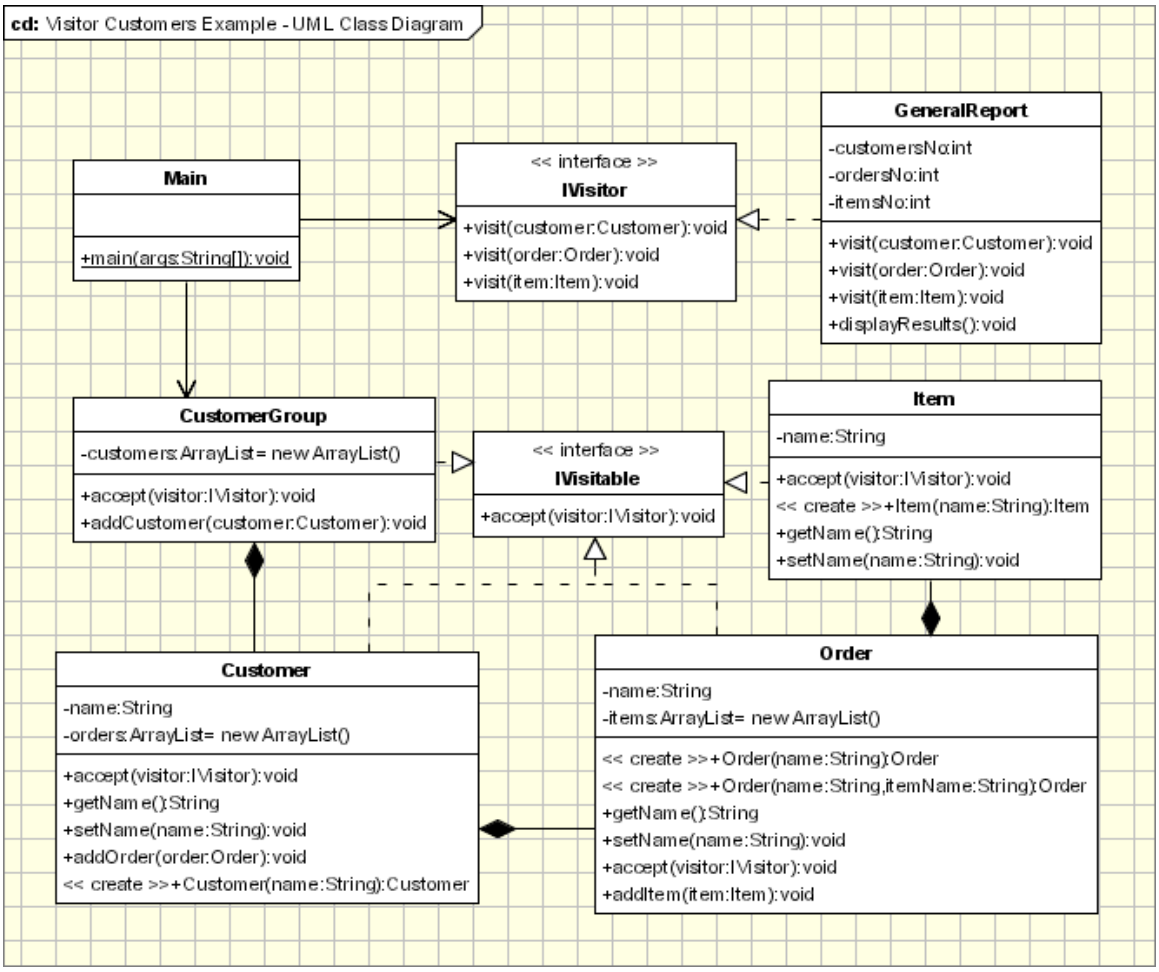

Figure 26: UML diagram of Visitor Pattern [10]

2.2.2 Pattern detection tools

A Design pattern detection is another reverse engineering technique that aids in analyzing a majority of the design patterns used within the Java source code or binaries. There are several pattern detection tools available that use different algorithms for detecting design patterns from software binaries or source. A few of the pattern detection tools are:

1. SPOOL [39]

2. Hedgehog [1]

3. Reclipse [24]

4. PINOT (Pattern INference and recOvery Tool) [33, 34]

5. Similarity Scoring [31, 32]

2.2.2.1 SPOOL and Hedgehog

A SPOOL was developed in order to reverse engineer design patterns from C++ software binaries [39]. A HEDGEHOG system was developed using pattern description language called SPINE [1]. SPINE is a language similar to Prolog and contains typed first order logic for describing patterns; it is not currently available for download. Patterns detected can be verified using a HEDGEHOG system to distinguish between patterns.

2.2.2.2 Reclipse

A Reclipse is a reverse engineering tool for automatic pattern detection from a Java source code. It uses UML2.0 diagrams from the source code in order to understand the design (i.e., object diagrams for structure based and UML sequence diagrams for behavioral based designs). Reclipse provides two graphical editors for structural and behavioral patterns. Detection of a specified pattern starts from detecting possible design pattern occurrences (candidates); once detected, they are given a percentage rating. A dynamic analysis is used to confirm or reject this percentage rating. Installation of this tool requires Eclipse IDE v3.6.1, and Eclipse Modeling Tools, version 3.6.1 [24], however, these exact versions of the software are not avail-

able for download.

2.2.2.3 PINOT

A Pattern INterference and recOvery Tool (PINOT) [34] takes a creative approach in detecting design patterns. According to [33], the authors of PINOT used their own reverse engineering techniques and characteristics that classify GoF design patterns in an attempt to detect design patterns from a Java source code. The authors discuss four reverse engineering techniques:

1. language-provided,
2. structure-driven,
3. behavior-driven,
4. and domain-specific patterns.

Prototype and Iterator patterns are classified as language-provided patterns, as they are widely used and implemented in many languages. Classes that have interclass relationships, such as Adapter and Facade patterns, are identified as structure-driven patterns; and the classes that differ in certain behavioral requirements, such as Singleton and Flyweight, are deemed as behavior-driven patterns. Finally, GoF patterns used in certain domains such as Interpreter and Command patterns are known as domain-specific patterns. PINOT focuses on detecting structure and behavior driven patterns as given in [33].

In order to detect structure-driven patterns that has interclass relationships for Adapter (adapter vs adaptee), Facade (Facade vs subparts) and Proxy (proxy vs real), we identify patterns that share a common goal of defining a new class which hide other

classes from system integration or simplification. For detection of behavioral-driven patterns, a different approach is used; their pattern detection technique is trained on GoF behavioral patterns then the given model is subjected to various inputs and the output is examined according the known behavior. For example, once a singleton pattern instantiates its data object, the same one should always be returned under multiple subsequent requests.

The PINOT pattern detection tool occasionally detects false positives [33, 16]; 23.75

The PINOT tool is developed using an IBM Jikes Java Compiler; using a Java compiler allows this tool the ability to compare the design pattern data using Abstract Syntax Graphs, created by Jikes. The input source files are parsed using a Jikes compiler with a PINOT back-end, then the detected patterns are output to the command-line interface. A PINOT command-line interface for testing the 23 GoF design patterns is shown in Figure 27, and Figure 30 which shows detected patterns for this test.

2.2.2.4 Similarity Scoring

Similarity scoring is a design pattern extraction tool that can be downloaded at [32]. This pattern detection tool has a unique way of building matrices for pattern detection and does not depend on behavioral characteristics. Their principle of considering only structural characteristics makes it difficult to detect patterns, such as State and Strategy which only differ in behavior.

The similarity detection of large software systems, with an increased presence of the great number of system classes and multiple roles and classes would lead to efficiency problems due to slow convergence of the algorithm [31]. An advantage to

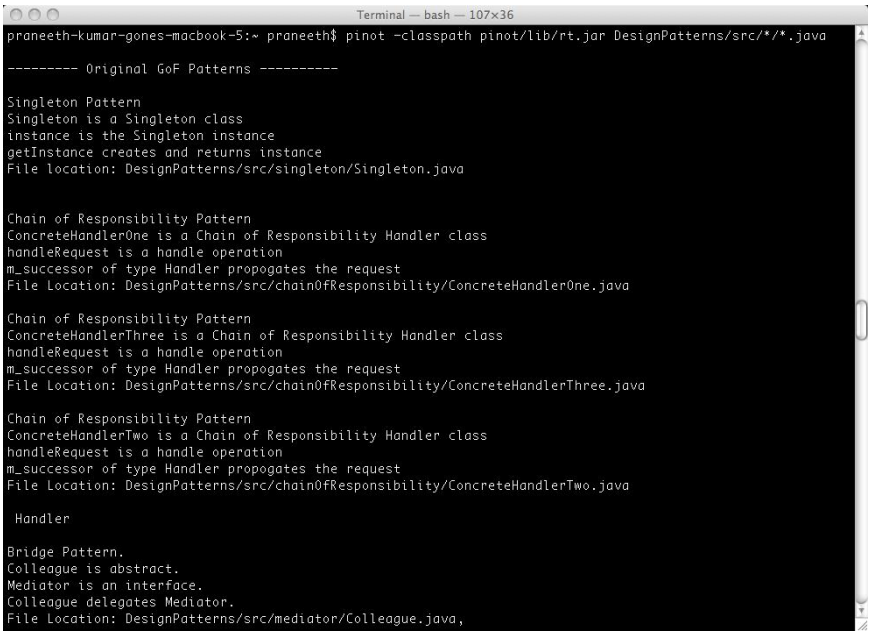

Figure 27: PINOT Command-line Interface

using this tool is that it does not follow any heuristics in detecting patterns and can be applied to any pattern once trained on that patterns input. This tool has been tested on three types of open source software: JHotDraw, JRefactory, and JUnit, each demonstrate limited false negatives and no false positives.

The process of detection follows the building of matrices from Java class files and comparing them to known matrices [31]. The rationale of a similarity scoring algorithm is from the proposed iterative algorithm in order to calculate the similarity between the vertices of two directed graphs. In the similarity matrix, each entry expressed a similarity vertex i of one matrix is to vertex j from another matrix. Two graph matching algorithms are applied to form pattern graphs: an Exact graph matching and an Inexact graph matching algorithm.

An Exact graph matching algorithm finds one-to-one mapping between the vertices of two graphs and also has the same number of nodes. Inexact graph matching

is applied when we cannot find isomorphism between two graphs and aim at finding the best possible match between two graphs. Association and Generalization matrices were created using edges from generalization and association graphs; each were formed using graph matching algorithms with connected edges such as 1 and 0, otherwise.

Methodology of detecting design patterns is accomplished in five steps as [31]:

1. Reverse engineering of the system under study;
2. Detection of inheritance hierarchies;
3. Construction of subsystem matrices;
4. Application of similarity algorithm between the subsystem matrices and the pattern matrices;
5. and Extraction of patterns in each subsystem.

The input class files are reverse engineered to obtain component information that can be used to detect the inheritance relationships. This class information is used in building two graphs Association and Generalization graphs. Matching is followed by matching two graphs using matching algorithms and creating subsystem matrices. Generalization and Association matrices are Similarity scored using a specific algorithm. The extractions of pattern instances is performed as a similarity algorithm resulting in score of 1 for subsystems and are considered to be exact matching to the patterns; for subsystems with a score of less than 1, a characteristic study and specific pattern detection process is followed to find a pattern match.

This similarity scoring is developed using Java and has a graphical user interface that shows results and we can output the results to an XML file. Usage of a

Java bytecode manipulation framework helps in static analysis of a systems structure that will help in retrieving abstraction, inheritance, class attributes, constructor signatures, method signatures, and other more advanced properties such as similar abstract method invocation, and a template method. The Adapter/Command and State/Strategy patterns are grouped for detection results; this might be because they do not check behavioral characteristics.

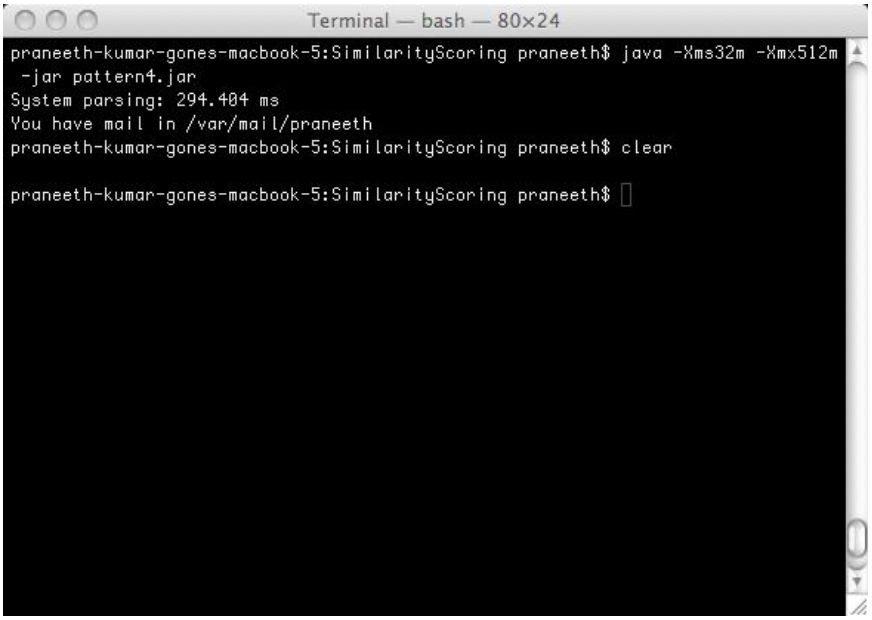

Figure 28: Similairty Scoring Command-line Interface

A Similarity scoring GUI is started through a command line as shown in Figure 28 results from a tool using a GUI are shown in Figure 29. The results window demonstrates that it cannot detect all 23 GoF patterns out of the detected patterns as Prototype and Proxy patterns are not detected. Figure 30 shows patterns detected for this test.

Using these tools, PINOT and Similarity scoring are available and can be used for testing. Therefore, for this project PINOT and Similarity Scoring were used to test an obfuscated code. The created obfuscated source is compiled in order to form

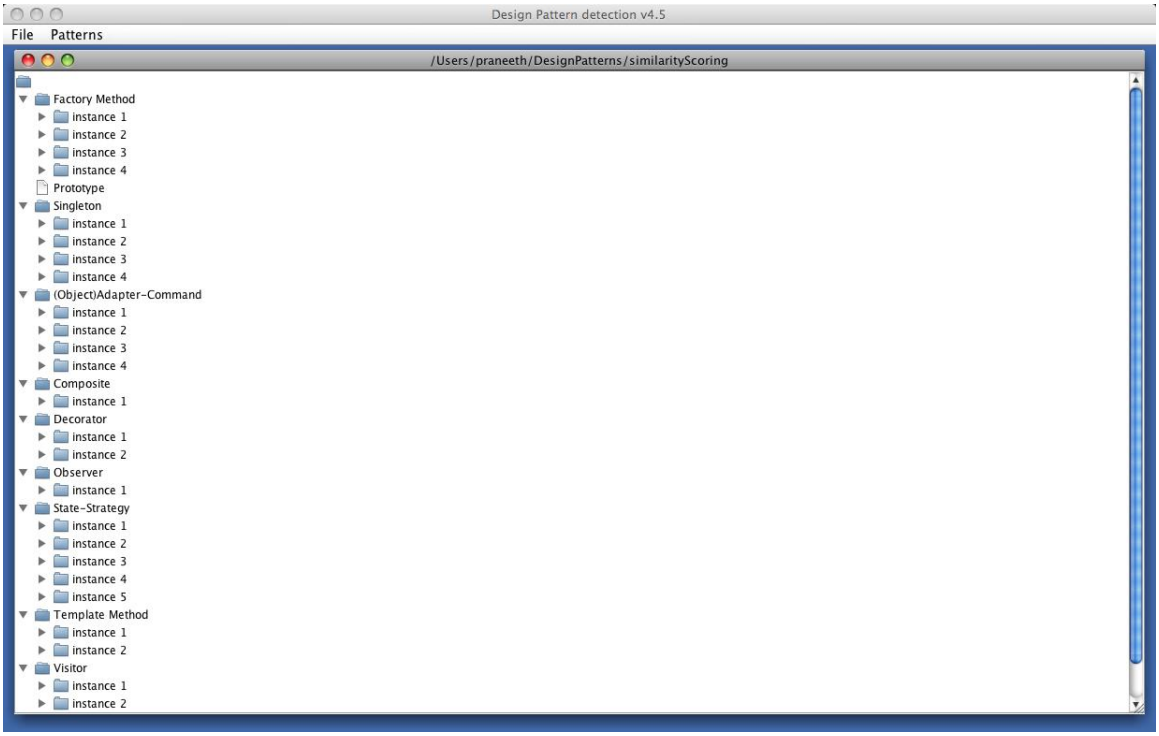

Figure 29: Similairty Scoring User Interface

class files; both source and class files are used as inputs to PINOT and Similarity Scoring respectively in an attempt to analyze patterns detected.

2.2.2.5 Pattern detection results for PINOT and Similarity Scoring

The package given as input for these tools contains all 23 GoF patterns and some patterns appear more than once. The number of patterns detected from both tools is shown in Figure 30 and Figure 31. From Figure 30 "*" represents Adapter/Command patterns detected under single pattern, and "$\tilde{}$" represents State/Strategy patterns are also detected under single pattern.

23 GoF Patterns	Actual Patterns Present	Patterns Detected	
		PINOT	Similarity Scoring
Abstract Factory	6	0	4
Builder	1	-	-
Factory Method	6	0	4
Prototype	1	-	0
Singleton	4	1	3
Adapter	4	0	6*
Bridge	1	1	-
Composite	2	2	1
Decorator	2	0	2
Façade	2	4	-
Flyweight	1	1	-
Proxy	1	1	0
CoR	1	1	-
Command	3	-	6*
Interpreter	1	-	-
Iterator	1	-	-
Mediator	5	10	-
Memento	1	-	-
Observer	3	2	1
State	3	0	5~
Strategy	4	4	5~
Template Method	1	1	2
Visitor	3	1	3

Figure 30: Patterns detected from 23 GoF patterns

The results from a PINOT tool shows less detection for creational patterns (from AbstractFactory Singleton in table) as we can see from Figure 30, only one Singleton pattern is detected out of four Singletons present in the original source. For the structural patterns (from Adapter - Proxy) tool we detected at least one instance for all patterns except Adapter and Decorator, and there are two false positives out of four detected Facade patterns, and one false positive for Flyweight and one for Bridge. The Behavioral patterns from CoR (Chain of Responsibility) to Visitor, were not able to detect Command, Interpreter, Iterator, and Memento. One false positive in the Strategy patterns were detected and two Observer patterns detected were Visitor patterns; from 10 Mediator patterns we detected two false positives and one Facade pattern detected was a Mediator pattern. Figure 32 below shows all false positives

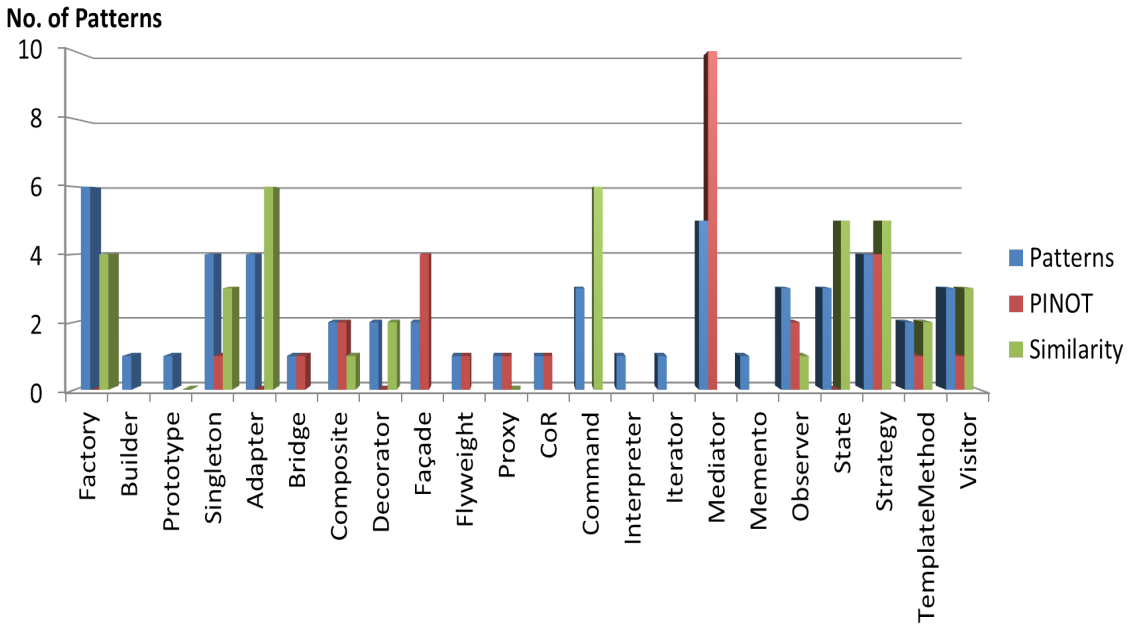

Figure 31: Graph for Patterns detected

for the PINOT tool.

False Positive No.		
	Detected Pattern	Actual Pattern
1	Façade	Builder
2	Façade	Builder
3	Flyweight	Command
4	Strategy	AbstractFactory
5	Observer	Visitor
6	Observer	Visitor
7	Mediator	Builder
8	Mediator	State
9	Bridge	Mediator

Figure 32: PINOT false positives

There are no false positives detected from the Similarity scoring detection tool, but it cannot detect all the pattern instances present within the input source; therefore only four patterns Factory patterns were detected out of six instances, out of 7 (4 Adapter and 3 Command) instances it can only detect six instances as shown in Figure 30. We ran one more comparison analysis using PINOT and the Similarity

Scoring technique is explained in [16] using JEdit and JHotDraw packages; they discuss algorithms used and do not address how exact patterns were detected.

CHAPTER 3

Obfuscation of Design Patterns

This section describes obfuscation techniques applied to the 23 GoF design patterns. Firstly, we will obfuscate a source using available obfuscation tools such as Proguard [13], SandMark [7], jarg [19], BebboSoft (bb_mug) [41], and JavaGuard [43] followed by testing detected and compared patterns before and after obfuscation. We then obfuscate design patterns using our proposed tool and detected patterns will be compared to patterns used in an actual source. These results are analyzed and discussed in Chapter 5.

3.1 Obfuscation using available tools

This section shows obfuscation of GoF patterns using jarg, JavaGuard, BeeboSoft (bb_mug), Proguard, and Sandmark. Sandmark obfuscated class files are tested using Similarity scoring as we cannot decompile class files. Class files obfuscated from other obfuscators are decompiled to source files and tested using PINOT. Then results are compared to patterns detected on an original source from Section 2.2.5.

3.1.1 jarg - Java Archive Grinder tool

The obfuscation tool jarg is used to obfuscate a Java bytecode; this tool contains features such as an optimizer, an obfuscator, a shrinker and a reducer [19]. Obfuscation starts by analyzing the Java class files, performing removal of unnecessary code such as unused functions, and debugging information; after classes, fields, methods and interfaces are renamed and optimized. This obfuscation tool can quickly obfuscate, optimize, or shrink a Java package, or jar file. We then obfuscated the original

design pattern source using jarg and then ran pattern detection tools to examine results and compare them to results before obfuscation.

3.1.1.1 Design pattern obfuscation

jarg is a command line tool that can be easily accessed using command [19]:

```
java -jar jarg.jar -nocomp abc.jar
nocomp :  no compressed output jar
```

This command will obfuscate the source abc.jar and creates new jar file abc_s.jar in the same location. An abc_s.jar file is extracted in order to obtain obfuscated class files that are used for testing a Similarity scoring algorithm. The decompiling of obfuscated source was unsuccessful so PINOT cannot be used.

3.1.1.2 Analysis of results

Obfuscation using jarg does not show changes in patterns detected; the same number of patterns are detected from the original source as shown in Figure 30 and Figure 31.

3.1.2 JavaGuard

JavaGuard is one more Java bytecode obfuscator; this tool can be included as a package within regular software development and testing processes [43]. Obfuscation of this tool follows three obfuscation techniques: class-flow, data, and layout obfuscations. We then obfuscated design patterns using JavaGuard then ran pattern detection tools in order to compare results with the normal design patterns source.

3.1.2.1 Design patterns obfuscation

JavaGuard is also a command line tool without much documentation. This tool uses a Jakarta-ORO for regular expression matching, and achieves an input parameter as a jar file that will generate an output jar as specified through the command line. A script file can also be used to configure an obfuscator in an attempt to prevent certain classes, fields, and methods from being renamed. The Command to run JavaGuard on a normal source is as shown below:

```
java -cp javaguard.jar;jakarta-oro-2.0.6.jar JavaGuard
i normalSource.jar o obfusSource.jar
```

```
-cp :  classpath
-i :  input
-o :  output
```

This command will obfuscate class files present in the normalSource.jar and store them in an output obfusSource.jar within a specified location (using this command it will be stored in the same location)

3.1.2.2 Analysis of results

Obfuscation using JavaGuard cannot hide patterns from the detection tools; the same number of patterns were detected as shwon in Figure 30. Therefore, obfuscating using JavaGuard does not help hide the design from pattern detection tools.

3.1.3 BebboSoft (bb_mug) obfuscation tool

The bb_mug is a tiny and fast Java bytecode obfuscator [41]; this tool replaces class, method and field names with shorter names, then it removes all information that is not required for execution, and specified packages are renamed.

3.1.3.1 Design patterns obfuscation

bb_mug is also a command line tool that is executed using the following command

```
USAGE: java -jar bb_mug.jar [-?]  [-l <logfile>]
[-p <package>=<newpackage>] <inpath> <outpath>
-?  display this message
-l <logfile> write mapping info into file
-p <package>=<newpackage> rename <package> to <newpackage>
```

Add bb_mug.jar to CLASSPATH, enter the folder with a normalSource.jar file and run the command:

```
java jar bb_mug.jar normalSource.jar obfusSource.jar
```

This will create obfuscated design pattern class files in jar file format, extract jar file, and run a Similarity detection tool. A decompiler is not available that can extract source from obfuscated class files so it cannot run a PINOT pattern detection tool.

3.1.3.2 Analysis of results

Obfuscating using bb_mug also did not help in hiding design patterns from Similarity detection; there was no change in patterns detected when compared to patterns detected as seen in Figure 30 and Figure 31.

3.1.4 Proguard Obfuscation tool

Proguard is an opensource Java class file shrinker, optimizer, obfuscator, and preverifier [13]. The Java file shrinker removes unused classes, fields, methods and then attributes and optimizes the bytecode removing unused instructions. This obfuscation step involves the renaming of classes, fields, and methods using short meaningless names. Proguard typically reads the input as jar, war, zip, or directories and outputs are suggested as jar, war, and zip files. For the present obfuscation we are not using shrinker and optimizer as it effects the operation of our design patterns. The Proguard tool is available in command line and GUI; we used a GUI to obfuscate class files from the 23 GoF patterns.

3.1.4.1 Design pattern obfuscation

The zip file for design pattern class files is displayed as an input to the Proguard GUI as shown in Figure 33, and obfuscation options are selected using a configuration pro file. The obfuscated class files are given as input to the JODE decompiler; decompiled source files are used to test for pattern detection for PINOT, and the obfuscated class files are used for Similarity scoring.

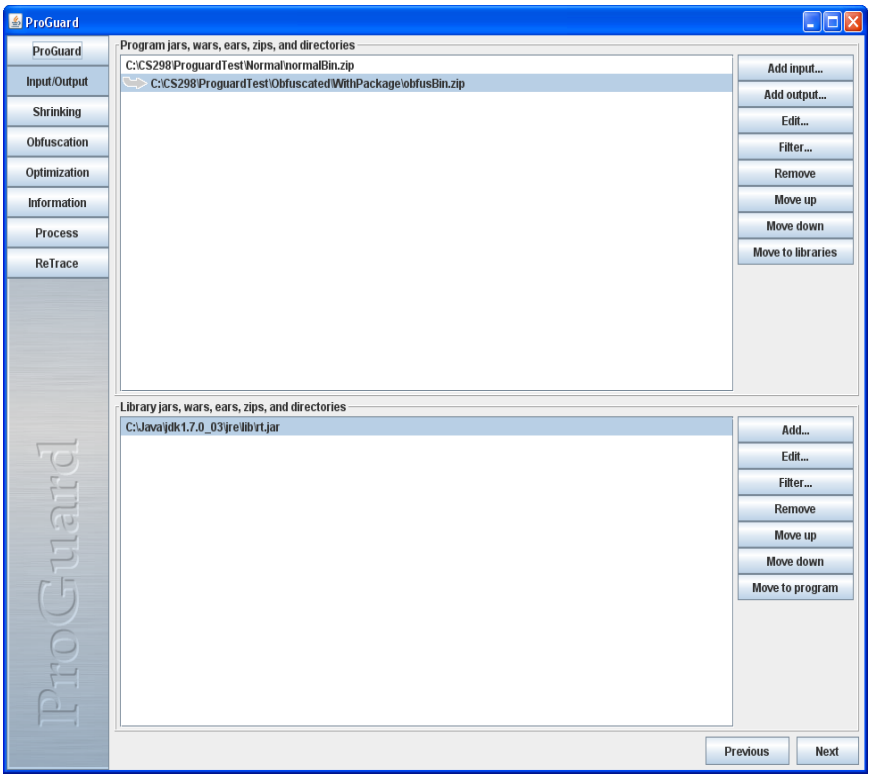

Figure 33: Proguard GUI In/Out options

3.1.4.2 Analysis of results

The obfuscated source and class files are tested using pattern detection tools. The results from PINOT and Similarity scoring are shown in Figure 34 and Figure 35.

The results show that the PINOT tool cannot detect both the CoR (Chain of Responsibility) and Facade patterns. The Factory pattern was detected in four instances along with one false positive compared to zero detected in the original source. Composite patterns also show four instances with two false positives and a false positive for Bridge, and Flyweight patterns, as shown in Figure 36. Seven instances of the Mediator pattern compared to 10 patterns detected from normal source.

23 GoF Patterns	Actual Patterns Present	Patterns Detected Normal Source		Patterns Detected Proguard Source	
		PINOT	Similarity Scoring	PINOT	Similarity Scoring
Abstract Factory	6	0	4	4	4
Builder	1	-	-	-	-
Factory Method	6	0	4	4	4
Prototype	1	-	0	-	0
Singleton	4	1	3	0	3
Adapter	4	0	6*	0	6*
Bridge	1	1	-	1	-
Composite	2	2	1	4	1
Decorator	2	0	2	2	2
Façade	2	4	-	0	-
Flyweight	1	1	-	1	-
Proxy	1	1	0	1	0
CoR	1	1	-	0	-
Command	3	-	6*	-	6*
Interpreter	1	-	-	-	-
Iterator	1	-	-	-	-
Mediator	5	10	-	7	-
Memento	1	-	-	-	-
Observer	3	2	1	2	1
State	3	0	5~	0	5~
Strategy	4	4	5~	4	5~
Template Method	2	1	2	1	2
Visitor	3	1	3	1	0

Figure 34: Patterns detected using Proguard

Similarity scoring cannot detect Visitor pattern except that there is no change with the patterns detected from the original source. igure 36 represents a bar graph represents comparing PINOT and Similarity scoring demonstrating actual patterns present.

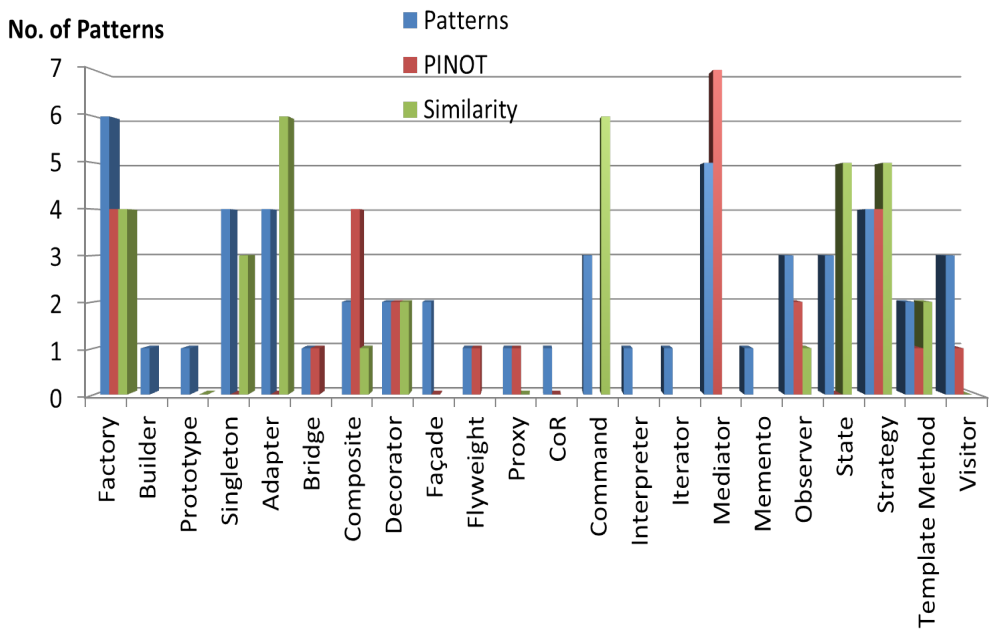

Figure 35: Detected patterns graph using Proguard

False Positive No.		
	Detected Pattern	Actual Pattern
1	Bridge	Mediator
2	Flyweight	Command
3	Factory	Memento
4	Observer	Visitor
5	Observer	Visitor
6	Composite	Visitor
7	Composite	Visitor

Figure 36: PINOT false positives

3.1.5 Sandmark Obfuscation tool

Sandmark is the tool developed for software watermarking, tamper proofing, and code obfuscation of Java bytecode [7]. This tool was developed at the University of Arizona in an attempt to study the effectiveness of software protection algorithms. The tool integrates the number of static and dynamic watermarking algorithms,a large collection of obfuscation algorithms, various code optimizers, and a tool to view and analyze the Java bytecode. The Sandmark obfuscation feature is used for

the present testing. There are 39 different algorithms available to obfuscate a Java bytecode. These algorithms are used to obfuscate the design patterns source; the obfuscated class files cannot be decompiled to source code to test using PINOT. Three algorithms: SplitClasses, Objectify and OverloadName are able to hide a few design patterns; all other algorithms show the same number of patterns as an original source. Results from these three obfuscation techniques are explained as follows.

3.1.5.1 Design pattern obfuscation

The jar file of design pattern class files is given as an input to the Sandmark GUI as shown in Figure 37 and an obfuscation algorithm is selected through the drop down present in the right side of input and output fields. The obfuscated jar file is saved at the location mentioned in output filed. This obfuscated source jar is extracted to generate class files, but cannot be decompiled using the decompiler. These extracted class files are used to test for pattern detection using a Similarity scoring tool, as there is no source available for testing using PINOT.

Three obfuscation techniques are used: Split classes, Objectify and OverloadNames. Split classes obfuscation is the obfuscation technique that splits a single class into a number of classes. Objectify is a type of obfuscation technique that creates a large number of objects of the same instance to confuse detection tools and OverloadNames will rename all classes, methods, and fields names by changing targeted access specifiers.

3.1.5.2 Analysis of results

The results of pattern detection tools using a obfuscation source from three obfuscation algorithms is shown in Figure 38. A decompiler tool is not available that

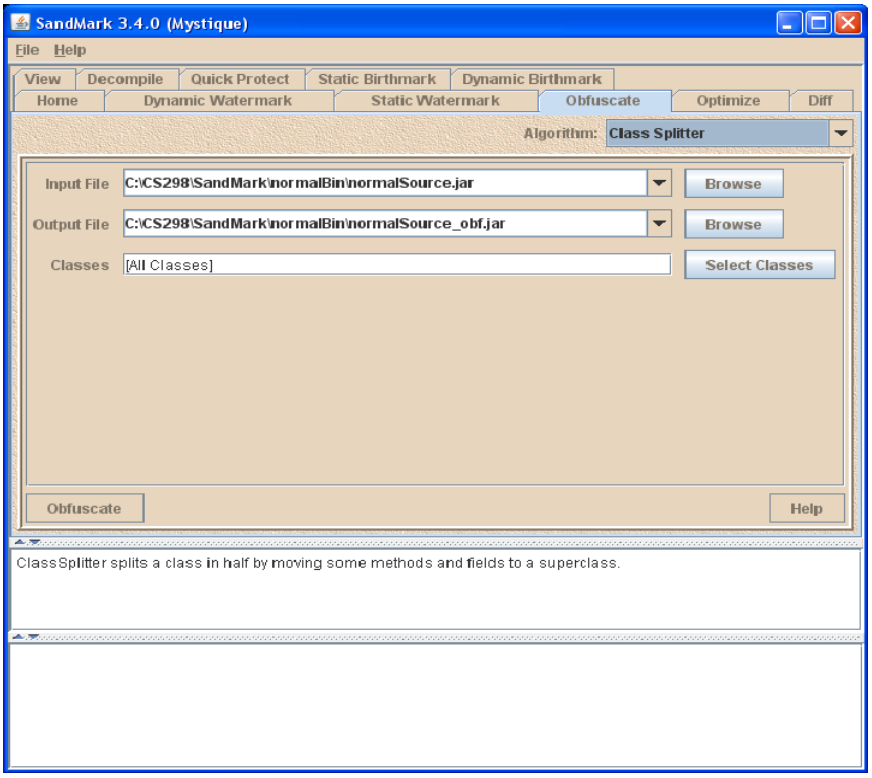

Figure 37: Sandmark GUI

can extract the source code from these obfuscated source files; therefore we cannot run a PINOT detection tool. Similarity scoring detection is used, and the patterns detected are shown in Figure 38. The graph showing the difference between actual patterns present and number of patterns detected using three obfuscation techniques using Similarity scoring is shown in Figure 39.

Results from these three obfuscation algorithms, Split class, Objectify and OverloadName demonstrate that, OverloadNames obfuscation can only hide a Visitor pattern compared to patterns detected from the original source. The SplitClass obfuscation technique hides the Adapter/Command, a Singleton and Decorator, and the Similarity algorithm detects 15 State/Strategy instances. From these 15 instances, 10 instances are false positives and five are actual instances that are present. For

23 GoF Patterns	Actual Patterns Present	Patterns Detection Normal Source		(Split-Classes) Patterns Detected	(Objectify) Patterns Detected	(OverloadName) Patterns Detected
		PINOT	Similarity Scoring	Similarity Scoring	Similarity Scoring	Similarity Scoring
Abstract Factory	6	0	4	4	4	4
Builder	1	-	-	-	-	-
Factory Method	6	0	4	4	4	4
Prototype	1	-	0	0	0	0
Singleton	4	1	3	0	0	3
Adapter	4	0	6*	0	0	6
Bridge	1	1	-	-	-	-
Composite	2	2	1	1	0	1
Decorator	2	0	2	0	0	2
Façade	2	4	-	-	-	-
Flyweight	1	1	-	-	-	-
Proxy	1	1	0	0	0	0
CoR	1	1	-	-	-	-
Command	3	-	6*	0	0	6
Interpreter	1	-	-	-	-	-
Iterator	1	-	-	-	-	-
Mediator	5	10	-	-	-	-
Memento	1	-	-	-	-	-
Observer	3	2	1	1	0	1
State	3	0	5~	15	0	5
Strategy	4	4	5~	15	0	5
Template Method	2	1	2	2	2	2
Visitor	3	1	3	3	3	0

Figure 38: Detected patterns using Sandmark

the Objectify obfuscation algorithm, except Factory, Template Method and Visitor patterns, all other patterns were hidden from the detection tool as shown in Figure 39.

3.2 Design Obfuscation

The above applied obfuscation tools completes three types of obfuscation such as control-flow to obscure flow of the program; data-flow obfuscation makes it difficult to understand fields and layout obfuscation will split the code into separate procedures. Therefore, these tools do not obfuscate structural and behavioral mechanisms pertaining to class components of the system. Obfuscating design in a class level for object oriented programs is very important in order to hide the internal architecture of the software system. From our test results jarg, BebboSoft, and JavaGuard cannot

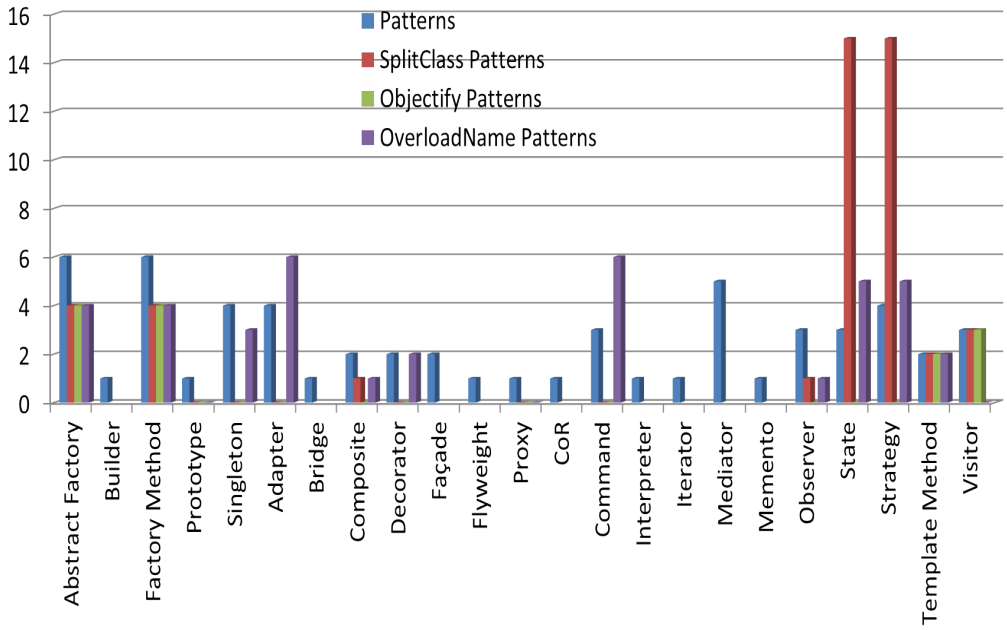

Figure 39: Detected patterns graph using Sandmark

perform design or software obfuscation, whereas Proguard can hide two patterns (Visitor and Singleton) due to additional obfuscation features such as efficient optimizing and adding dead code. A Sandmark tool using an obfuscation algorithm Objectify can hide a few design patterns and make pattern detection tools to detect many false positives for SplitClass obfuscation.

Design obfuscations described in [29] are applied on structural and behavioral characteristics of the 23 GoF design patterns. Software or design obfuscation is a new class of obfuscation techniques that are used to obscure class level design of object oriented programs. This obfuscation can be completed using three techniques: class-coalescing, class-splitting, and type-hiding.

Class-coalescing is the transformation where two or more classes within the program are replaced with a single class. At one extreme, this obfuscation can replace all the classes with a single class making an OO program into non-OO procedural

program.

Class-splitting is the transformation where one class is split into two or more classes. There are important decisions that need to be made when splitting one class. Used in addition with class-coalescing, this technique can change the program structure.

Type-hiding uses the concept of a Java interfaces, in other words it introduces a number of interfaces that are implemented by the existing classes to confuse reverse engineering from understanding the program.

In the next subsections, we describe obfuscation of 12 GoF patterns by using class-coalescing, class-splitting and type-hiding techniques. Since two techniques hide most design patterns from pattern detection tools, for our purpose we did not use type hiding obfuscation.

3.2.1 FactoryMethod pattern

Obfuscation of this FactoryMethod pattern is completed using class-coalescing (removing two or more classes using single class). By removing the `Factory` interface, plus its implementations, we included functionality into the `Client` class using a separate `createProduct` methods initiate for all products. Figure 40 below shows the `Client` class before and after obfuscation; we can see that before the `Client` class had `Factory` instantiation for creating Products. But after obfuscation, there are individual methods for each `Product` and these methods are used to create concrete products.

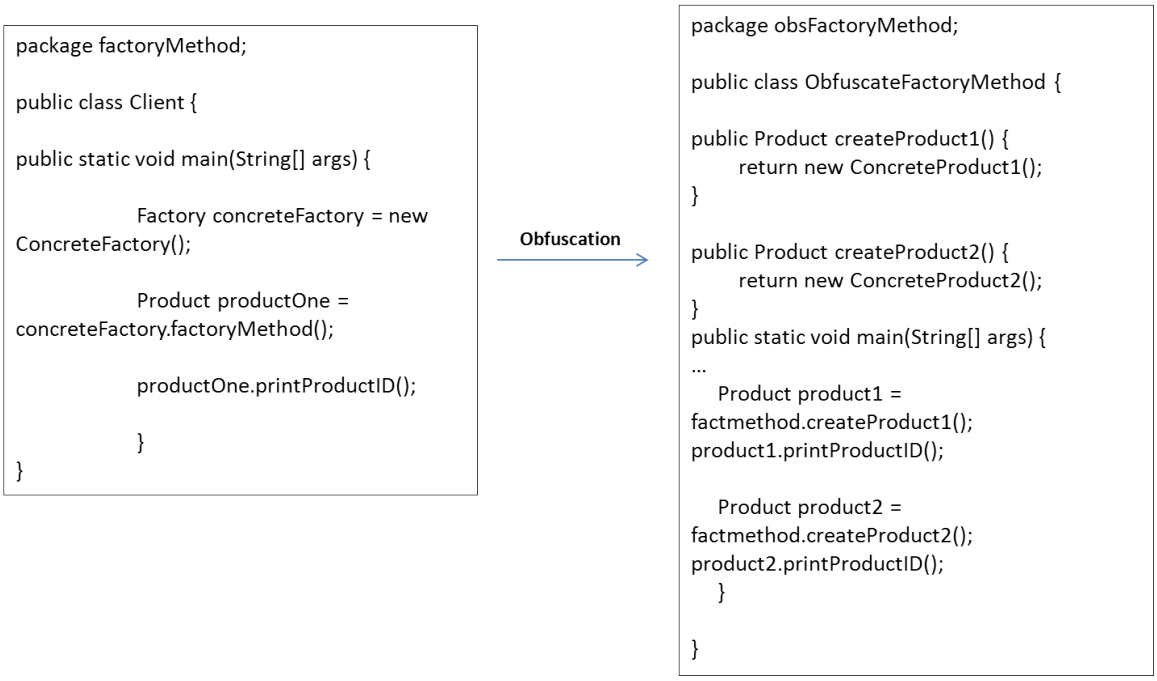

Figure 40: Obfuscate FactoryMethod pattern

3.2.2 AbstractFactory pattern

Class-coalescing `AbstractFactory`, `AbstractProductA` and `AbstractProductB` interfaces, and implementing individual classes for each Product, use them directly in the `Client` class as shown in Figure 41. For the `ObfuscateAbstractFactory` class we add two functions to create `ProductA` and `ProductB`, then initiate these classes as needed.

3.2.3 Builder pattern

Obfuscation of this pattern is completed by removing the `Builder` interface and implementing individual `ConcreteBuilder` classes for each complex object. Figure 42 shows the obfuscated version of the Builder code for the `TextConverter` example. The `TextConverter` converts an RTF text to an ASCII text. At this point the `TextConverter` is the `Builder` and an `RTFReader` is the `Director` and

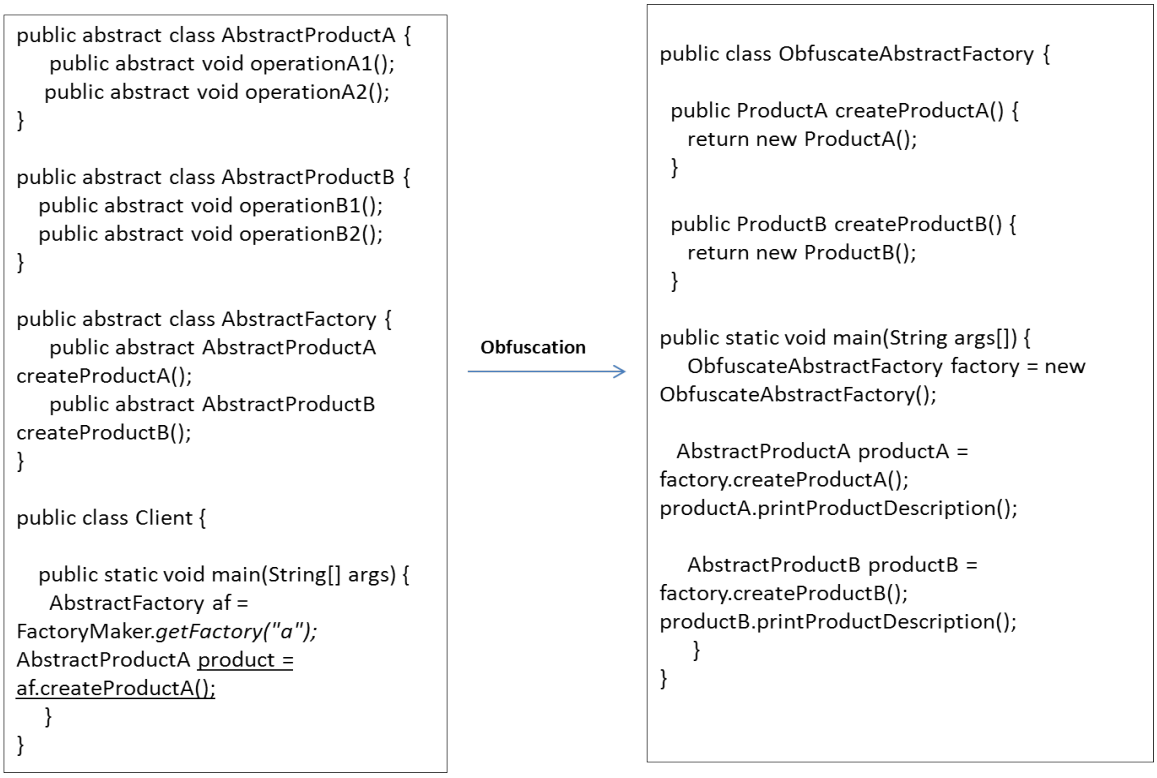

Figure 41: Obfuscate AbstracFactory pattern

an `ASCIIConverter` is a `ConcreteBuilder`. To obfuscate this application, an `ASCIIConverter` class and a `TextConverter` class are class coalesced into one class with the name `TextConvereter`. The functionality of an `ASCIIConverter` class is implemented within the `TextConverter` class in order to convert an RTF to an ASCII document.

3.2.4 Adapter pattern

Obfuscation of this pattern can be completed by removing an Adaptee interface and implementing an Adapter class for each request method that needs to be implemented. An example for drawing a shape such as Line and Rectangle, to implement an Adapter pattern we should use Shape interface with abstract draw method; Line and Rectangle class implements a Shape interface and uses an Adaptee classes

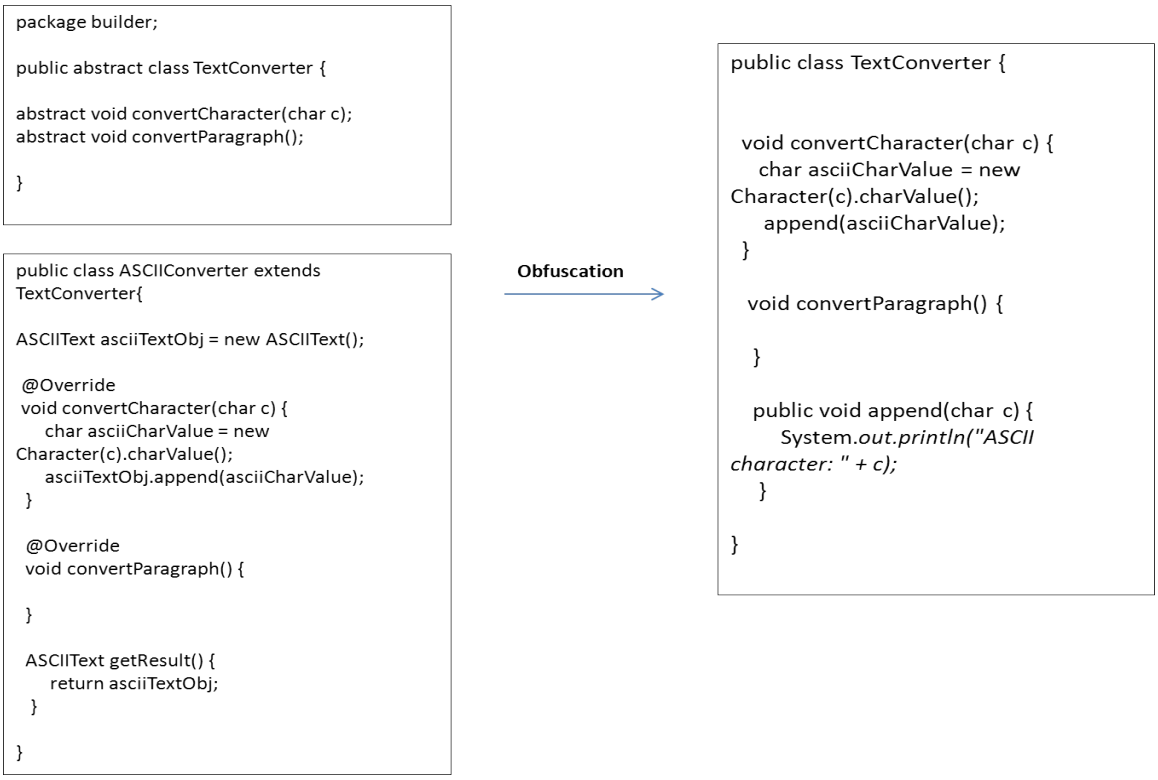

Figure 42: Obfuscate Builder pattern

LegacyLine and LegacyRectangle as instances. Obfuscation of this pattern can be completed by removing the Line and Rectangle interface and using objects from the LegacyLine and LegacyRectangle classes. Obfuscation of a source level code is shown in Figure 43, and Figure 44. Figure 43 demonstrates the removing of the Adaptor implementation Rectangle and adding all functionality to the LegacyRectangle class.

Figure 44 shows the Client class that uses direct instantiation of the LegacyLine and LegacyRectangle after obfuscation instead of Line and Rectangle. Some minor modifications are implemented to the main method as shown in Figure 44.

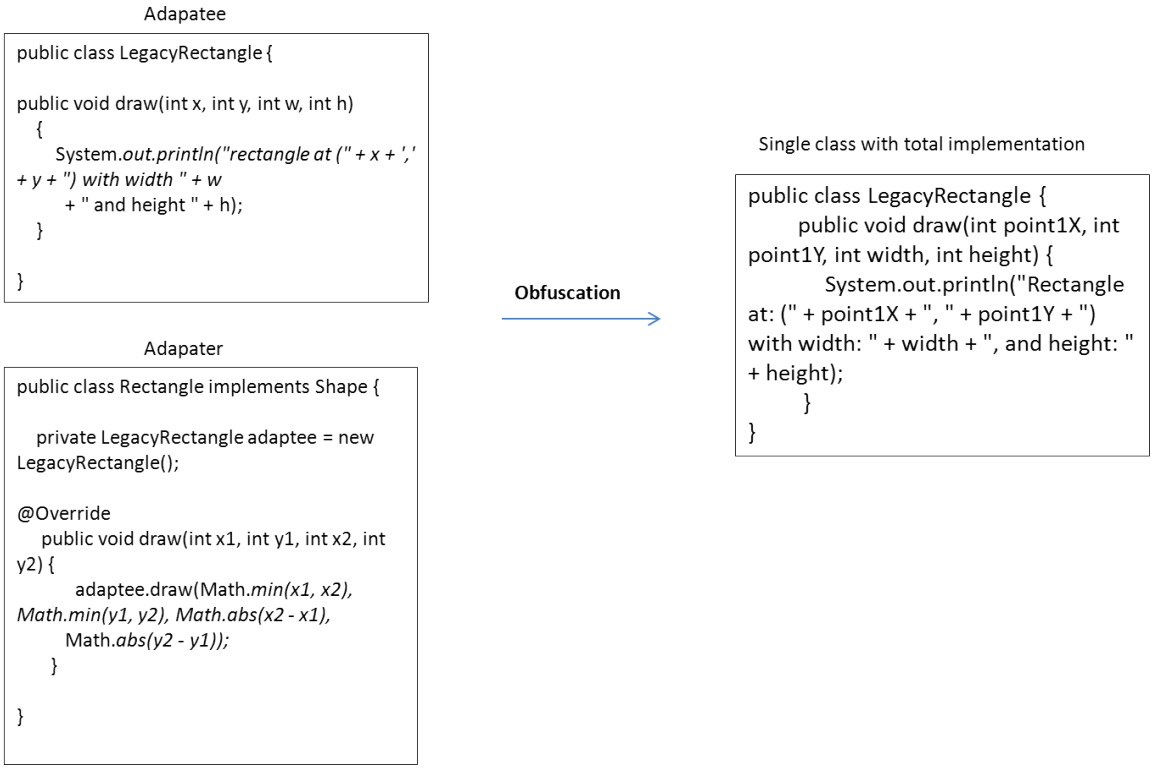

Figure 43: Obfuscate Adapter pattern

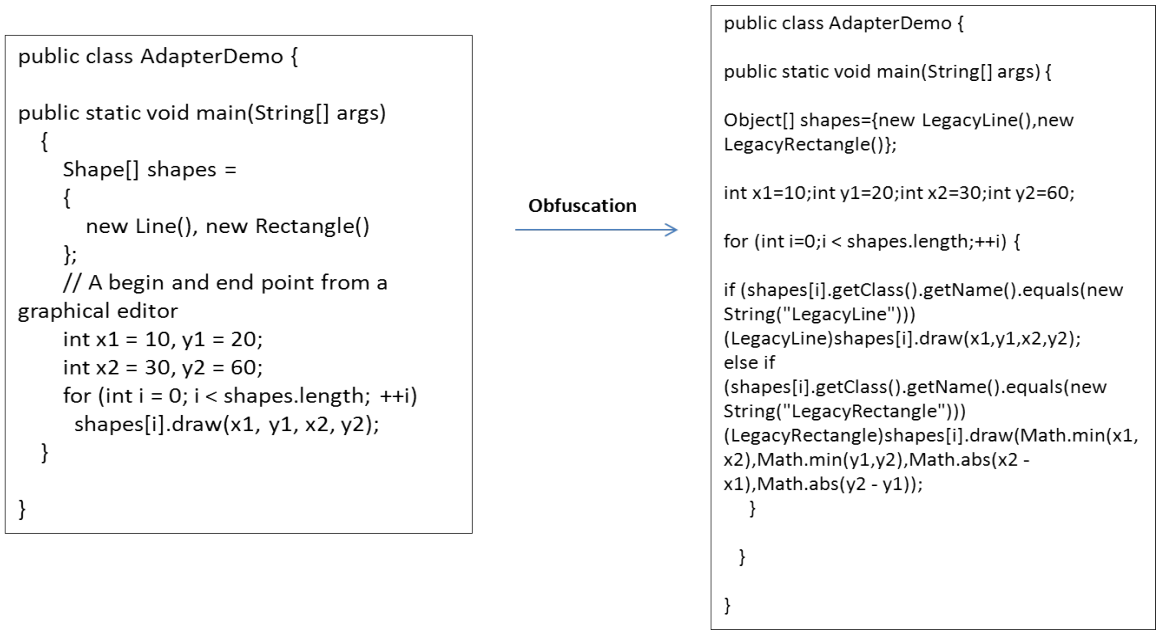

Figure 44: Obfuscate Adapter Client

3.2.5 Bridge pattern

Obfuscation of this design pattern is achieved by class-coalescing of the `PersistenceImplementor` interface and `Persistence` interface from the application. In this pattern we see that leaving an interface (i.e., only obfuscating one interface) will remove a Bridge pattern which will result in detecting other patterns. Figure 45 shows the obfuscation by removing an Abstraction and Implementation interface. We see in Figure 45 `Persistence` and `PersistenceImplementor` are removed and their functionality is added to the respective implementation, i.e., `FileSystemPersistenceImp`, `DatabasePersistenceImp`, `FileSystemPersistenceImplementor`, and `DatabasePersistenceImplementor` classes.

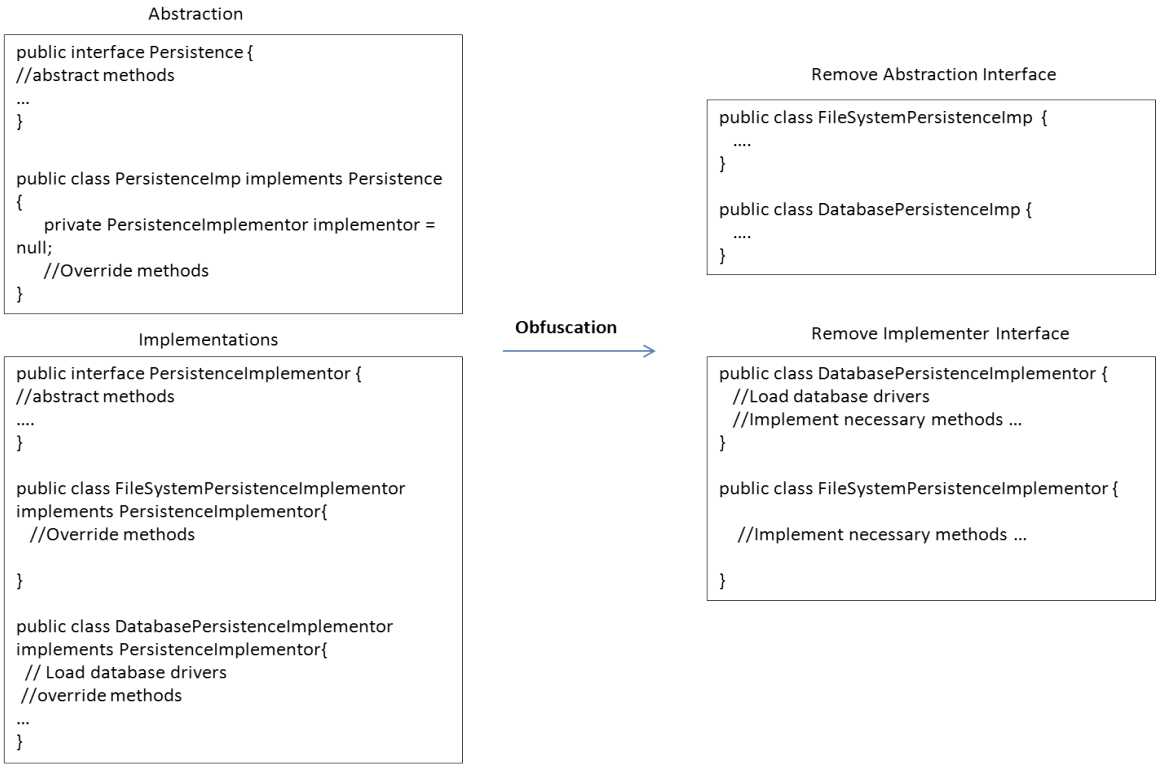

Figure 45: Obfuscate Bridge pattern

3.2.6 Flyweight pattern

To obfuscate this pattern class-coalesce, a `Soldier` and `SoldierFactory` class makes the pattern hide from pattern detection tools. Removing the `Soldier` interface makes it necessary to implement all `Soldier` objects individually. As shown in Figure 46 `SoldierImp1`, and `SoldierImp2` are the two soldier classes needed to implement all soldier objects. Implementing the number of classes for a Wargame will increase the total number of classes and require a large amount of memory. We can see after obfuscation, the `SoldierClient` class has objects from `SoldierImp1`, and `SoldierImp2`.

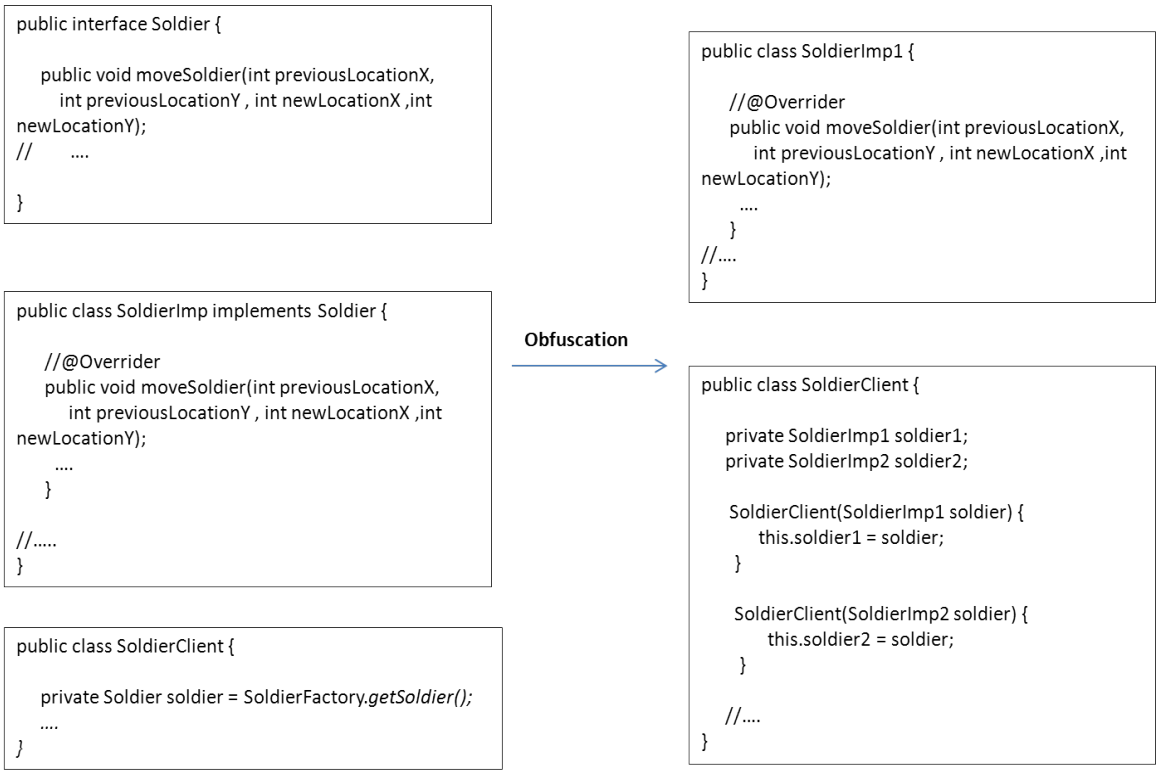

Figure 46: Obfuscate Flyweight pattern

3.2.7 Decorator pattern

Obfuscation of a decorator pattern can be achieved by class-coalescing a Window interface from the application and designing each decorator window individually. Within the client class you need to create each object statically in order to hide the decorator pattern. Obfuscation showing client class and other Windows implementations are shown in Figure 47. A `GUIDriver` client class initiates a `ScrollableWindow` class without defining a super object `DecoratorWindow` class; we can determine that a `Windows` interface is class coalesced from the `Decorator` pattern.

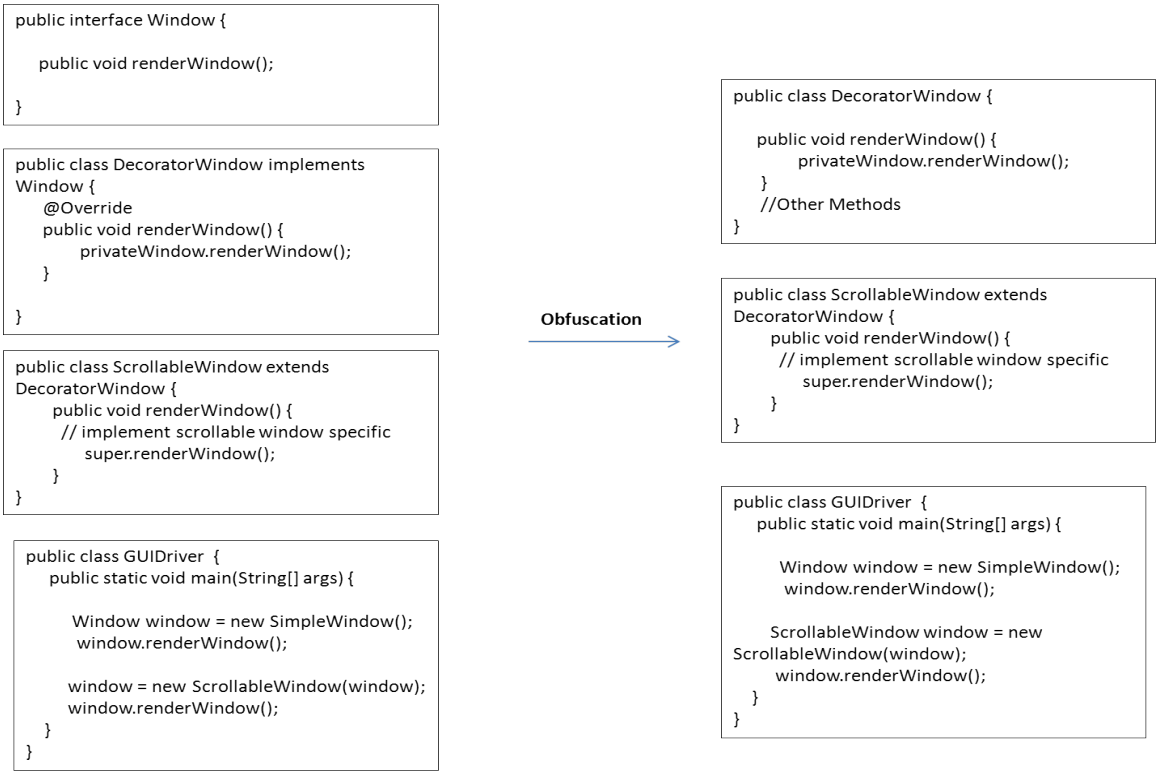

Figure 47: Obfuscate Decorator pattern

3.2.8 Mediator pattern

Obfuscation of a mediator pattern can be achieved using classcoalescing (removing `Mediator` Interface) from the project. We need to replace `Mediator`

references to respective mediator implementations. For this example it should be `ApplicationMediator`, Figure 48 shows the obfuscation of the `Mediator` pattern through its source code; a `Mediator` interface is removed, and an `ApplicationMediator` is implemented as an ordinary class with all the implementation. This `ApplicationMediator` is to be instantiated into a `Colleague` class; all Mediators must be implemented and be used in respective `Colleague` classes by instantiating.

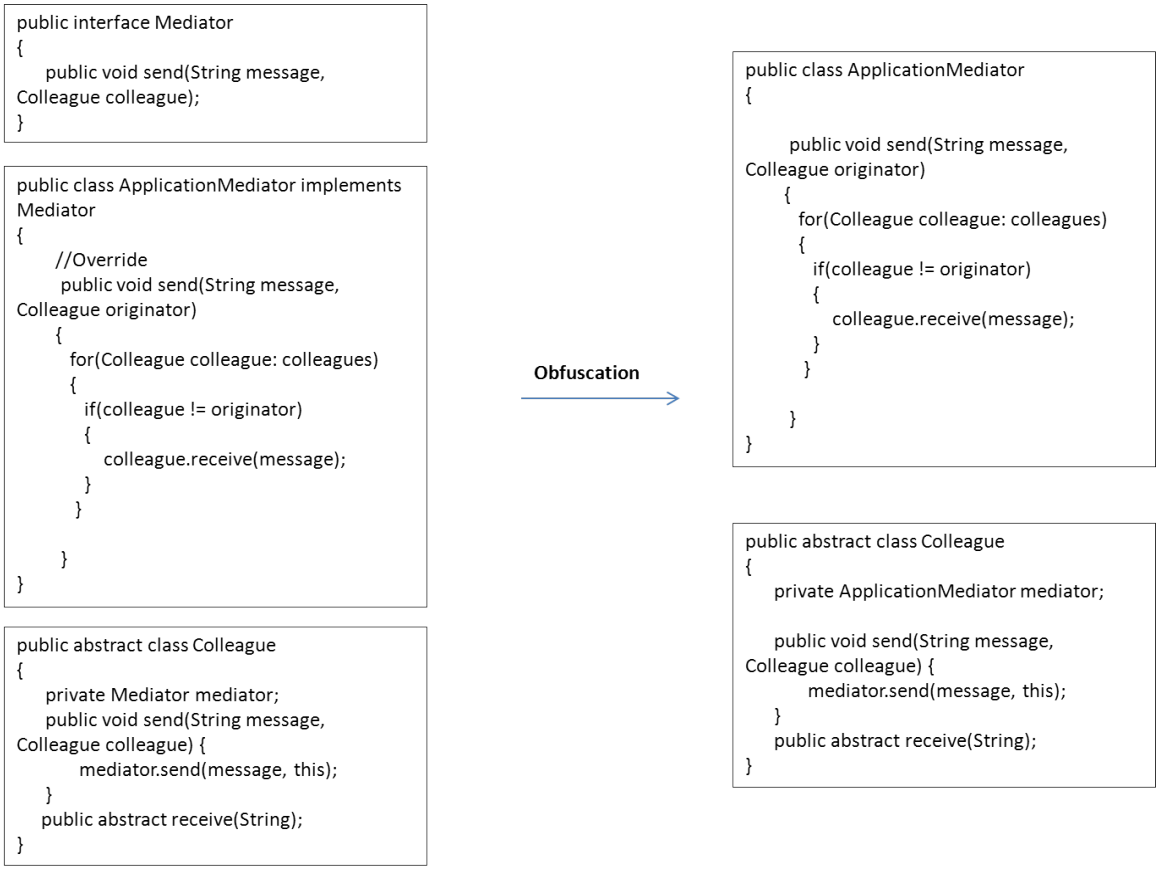

Figure 48: Obfuscate Mediator pattern

3.2.9 Observer pattern

Class-coalescing obfuscation is used for hiding the Observer pattern; we removed the `NewsPublisher` interface and `Subscriber` interface and developed separate news classes for each news type: business, sports and others. A Single subscriber class will contain all the methods needed to update various types of subscribers such as SMS, Email, and all other forms. In Figure 49, a `NewsPublisher` interface is removed and a common `NewsPublisher` class is implemented that will update to all subscribers by calling respective methods for each subscriber. A `Subscriber` abstract class is also removed following the same technique and updating each subscriber using methods such as `emailUpdate` and `smsUpdate`.

Figure 49: Obfuscate Observer pattern

3.2.10 Strategy pattern

Obfuscation of this pattern is completed using class-coalescing of the `IBehavior` interface by implementing all behaviors as methods for a `Robot` class. Each robot behavior is implemented a separate function and is called when a particular movement is required. Obfuscation of a source is shown in Figure 50 and demonstrates that `AggressiveBehavior, DefensiveBehavior,` and `NormalBehavior` classes, which implement an `IBehavior` interface, are removed. Add methods to a `Robot` class such as `moveAggressiveCommand`, and `moveDefensiveCommand`, are called in the `Client` class according to the behavior needed.

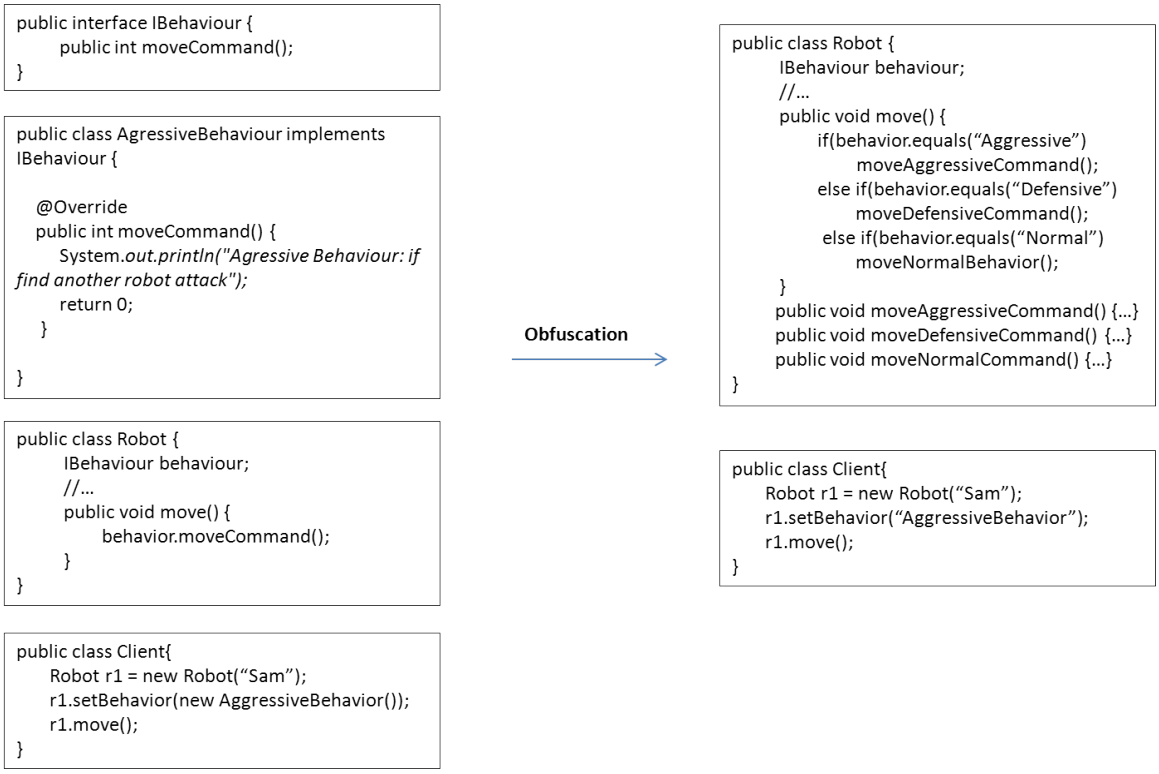

Figure 50: Obfuscate Strategy pattern

3.2.11 TemplateMethod pattern

Obfuscation of this pattern can also be completed by removing the `Trip` interface and individually creating classes for each trip package. These package classes are initiated to perform the trip as shown in Figure 51. A `Trip` interface is removed from the code after obfuscation of individual `Package` classes are implemented and Packages are instantiated within the client class to call a `performTrip` function.

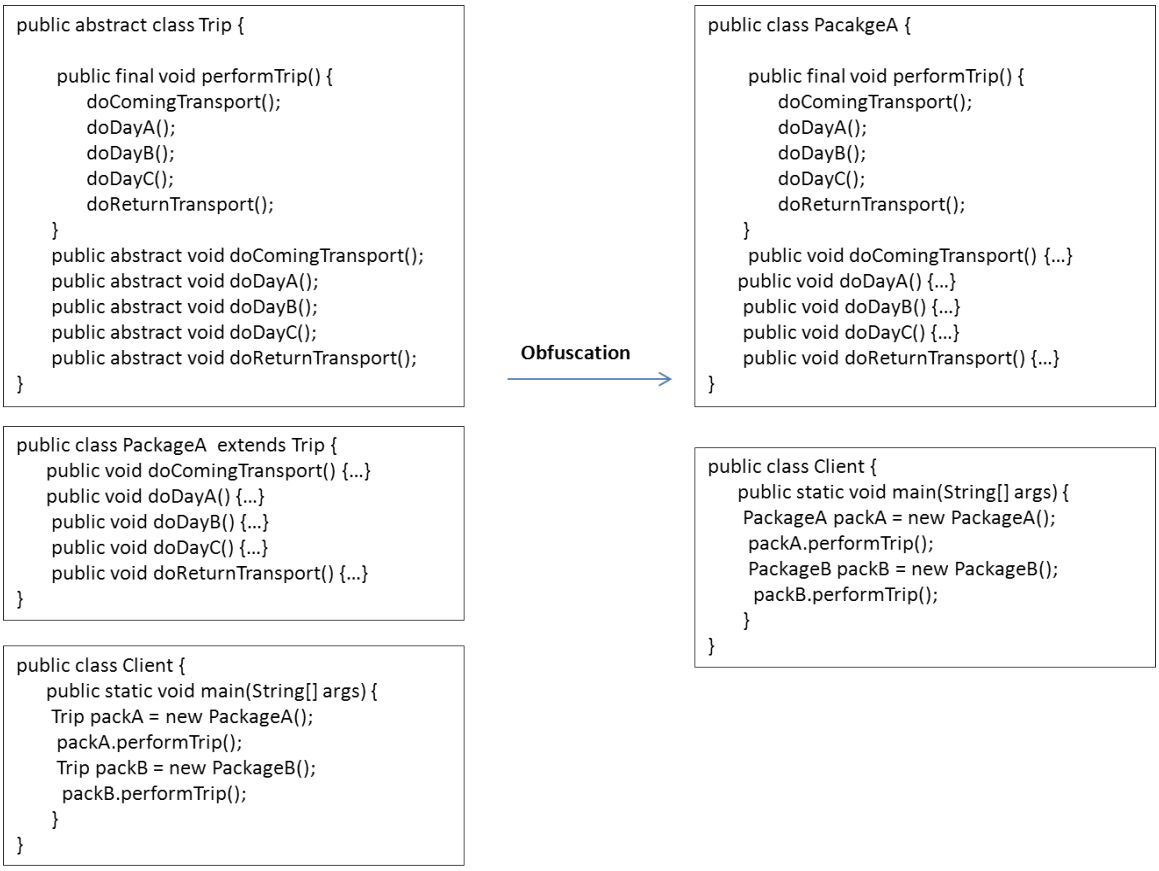

Figure 51: Obfuscate TemplateMethod pattern

3.2.12 Visitor pattern

Obfuscation of the Visitor Pattern can be completed through class-coalescing `IVisitor` and `IVisitable` interfaces from the application. Implementing all the in-

dividual functions, with respective class related statistics in a `GeneralReport` class is shown in Figure 52. We can see that the `IVisitor` and `IVisitable` interfaces are removed; individual classes for `Customer, Order, Item` and `GeneralReport` are implemented. Still a `GeneralReport` will have visit methods and `Customer, Order` and `Item` classes will have `accept` methods in order to work according to the application. This pattern is obscure by simply removing the `IVisitor` and `IVisitable` interfaces.

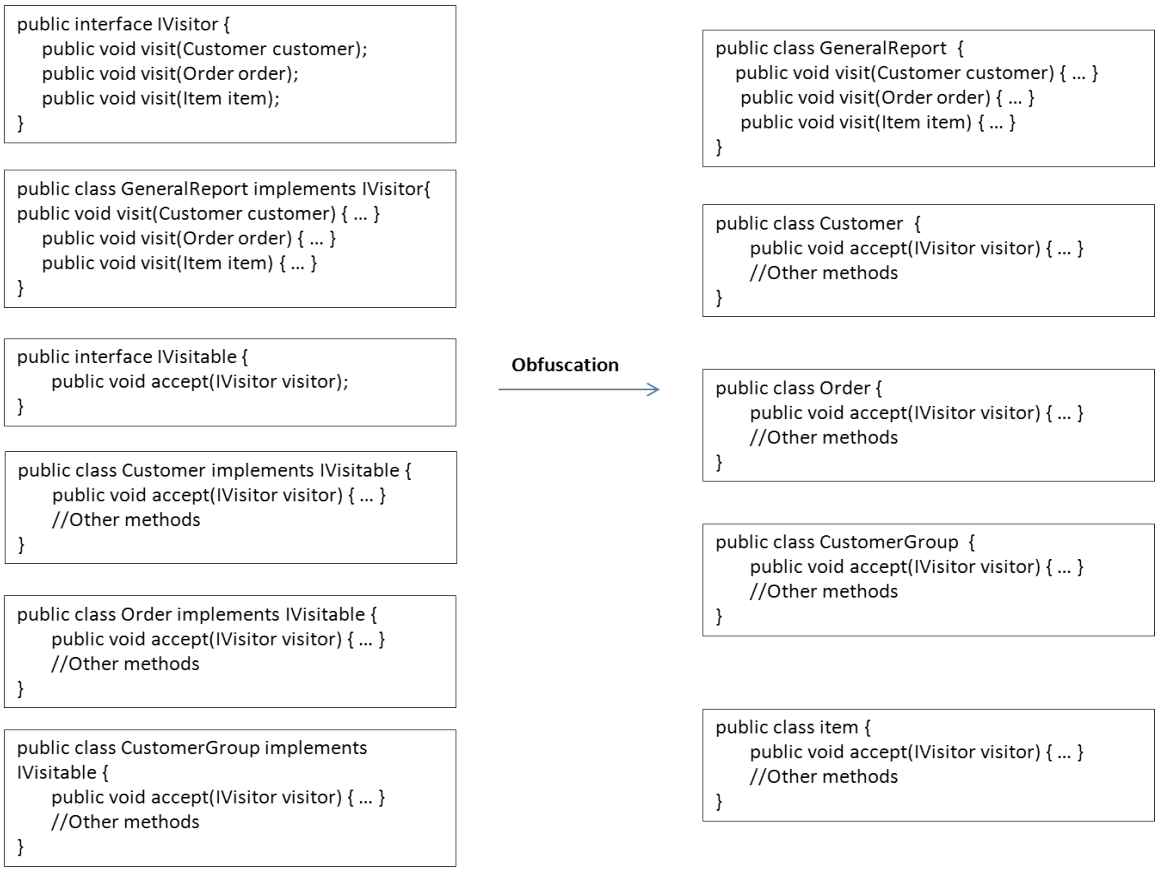

Figure 52: Obfuscate Visitor pattern

Obfuscation techniques applied to all design patterns are listed in the Figure 53 below:

23 GoF Patterns	Design Obfuscation Technique	
	Class Coalescing	Class splitting
Abstract Factory	✓	x
Builder	✓	x
Factory Method	x	✓
Prototype	✓	x
Singleton	x	✓
Adapter	✓	x
Bridge	✓	x
Composite	✓	✓
Decorator	✓	✓
Façade	✓	x
Flyweight	✓	x
Proxy	✓	x
CoR	✓	x
Command	✓	x
Interpreter	✓	x
Iterator	✓	x
Mediator	✓	✓
Memento	✓	x
Observer	✓	x
State	✓	x
Strategy	✓	x
Template Method	x	✓
Visitor	✓	x

Figure 53: Obfuscate design patterns

CHAPTER 4

Tool Implementation

In our project we implemented an obfuscation tool, DesignObfuscationEngine, using techniques described in Chapter 3. This tool must implement class-coalescing, class-splitting and type-hiding within the Java source code in order to hide design patterns. An Obfuscation tool is developed using:

1. Abstract Syntax Tree (AST) Eclipse API
2. PINOT command line
3. Similarity Scoring command line to start GUI

4.0.12.1 Abstract Syntax Tree (AST) Eclipse

A tool is developed in Java using an Eclipse Abstract Syntax Tree (AST) API. An AST API is used to create source files of the obfuscated version of patterns. An AST is a tree representation of an abstract syntactic structure of the source code from any programming language [46]. An Abstract Syntax Tree is the base framework for many tools in Eclipse including refactoring, QuickFix, and QuickAssist [42]. Eclipse IDE looks at your code using an AST as shown in Figure 54; every Java source file is entirely represented as a tree of AST nodes. An ASTNode is the parent class of all these nodes. Each element in a Java source file is a node; an example would be: the node for method declarations (`MethodDeclaration`) and for a variable declaration (`VariableDeclarationFragment`). A DesignObfuscationEngine tool uses AST nodes to create source files.

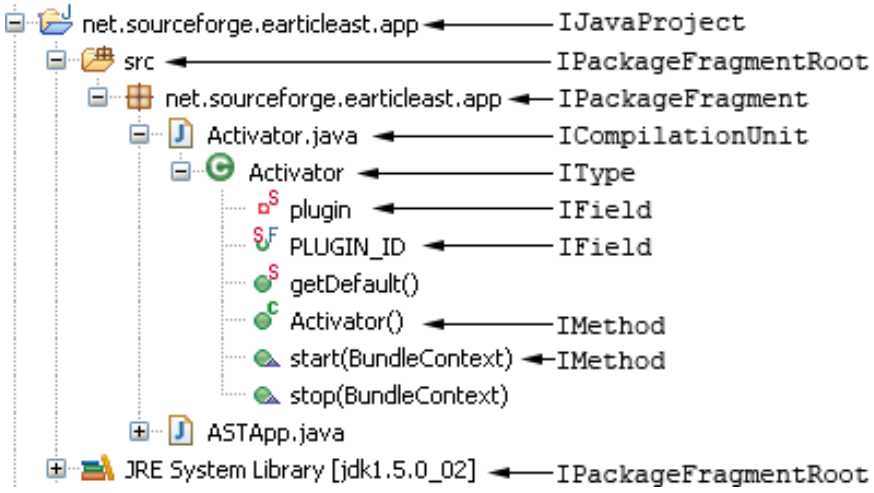

Figure 54: Java Model Overview [42]

4.0.12.2 PINOT Command Line

The PINOT tool does not have a GUI, therefore command `pinot` is used to run the tool giving `rt.jar` as classpath argument and source files path. The command used to run PINOT is:

```
pinot classpath pinot/lib/rt.jar <sourcePath>
```

This command is run using a Java program using a `Runtime` class from the package `org.eclipse.jdt.internal`. A Method invocation statement used to run PINOT command is:

```
Process p = Runtime.getRuntime().exec("pinot classpath
pinot/lib/rt.jar <SourcePath>");
```

4.0.12.3 Similarity Scoring Command

A Similarity scoring tool is developed using Java language, runs through the jar file `pattern4.jar` and command used to start the Similarity Scoring GUI

```
java -Xms32m -Xmx512m -jar pattern4.jar
```

and command to run a Similarity Scoring tool through a command line is

```
java -Xms32m -Xmx512m -jar pattern4.jar
target <sourcepath> -output <xml>
```

The same method invocation statement using Runtime class is used to run a Similarity Scoring tool.

For Command line:

```
Process p = Runtime.getRuntime().exec("java -Xms32m -Xmx512m
-jar pattern4.jar -target j̈HotDraw-̈output jhotdraw_v0_0.xml");
```

To start GUI:

```
Process p = Runtime.getRuntime().exec("java -Xms32m -Xmx512m -jar
pattern4.jar");
```

4.1 Design and Functionality

Designing the tool can be explained using a sequence diagram shown in Figure 55 and also briefly explaining the sequence of steps within the tool in order to complete

total obfuscation and testing using detection tools.

As shown in the sequence diagram, the user starts the DesignObfuscationEngine tool that first creates an obfuscated design pattern code and saves it to the system. A `DesignObfuscationEngine` class uses a `Compiler` object to compile this obfuscated code and save the binaries, followed by tests using pattern detection tools through a `RunCommand` object. Then results from both detection tools will be displayed on the users computer screen.

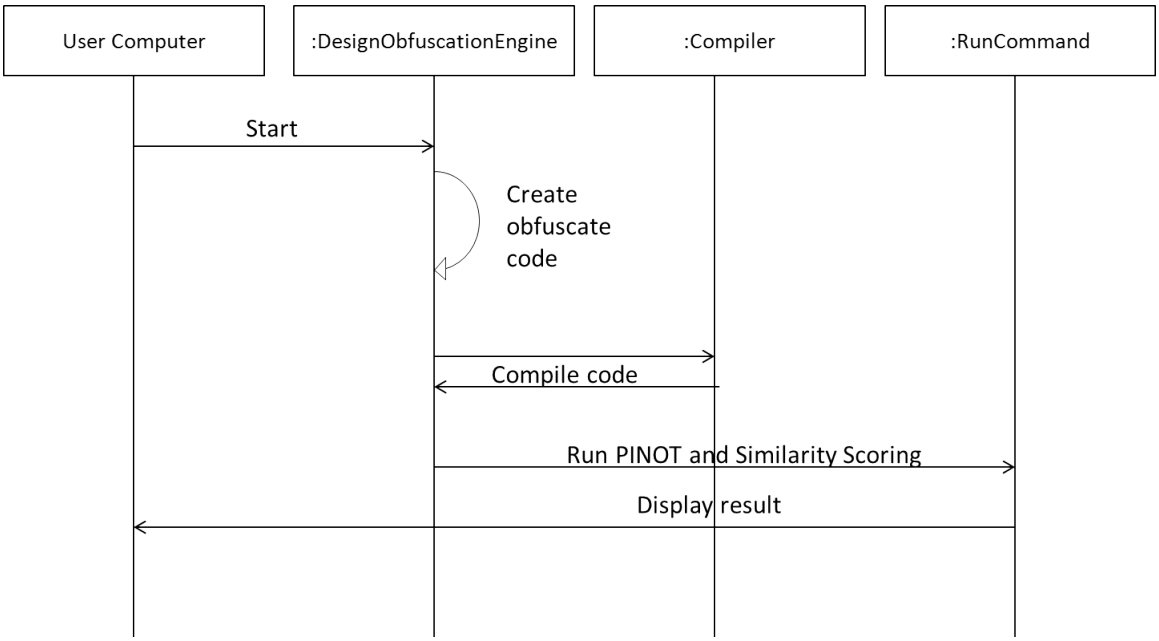

Figure 55: Sequence diagram DesignObfuscationEngine

Functionality of a DesignObfuscationEngine tool can be explained in three steps:

1. Obfuscate example programs that use 23 GoF design patterns
2. Compiles these obfuscated source program to get class files
3. Run pattern detection tools PINOT and Similarity Scoring

A DesignObfuscationEngine first creates obfuscated Java source files of example programs that use 23 GoF patterns using an Eclipse AST, code in order to generate a `Product` class file. Obfuscated source files are placed under respective packages and all source files are stored at a known location. In the next step, these files are compiled using a `Compiler` class from an `org.eclipse.jdt.internal` package. After successful compilation, PINOT tool is run on the source files using pinot command programmatically and then a Similarity Scoring tool is run in a similar way as PINOT. A Block diagram of the obfuscation tool is shown in Figure 56.

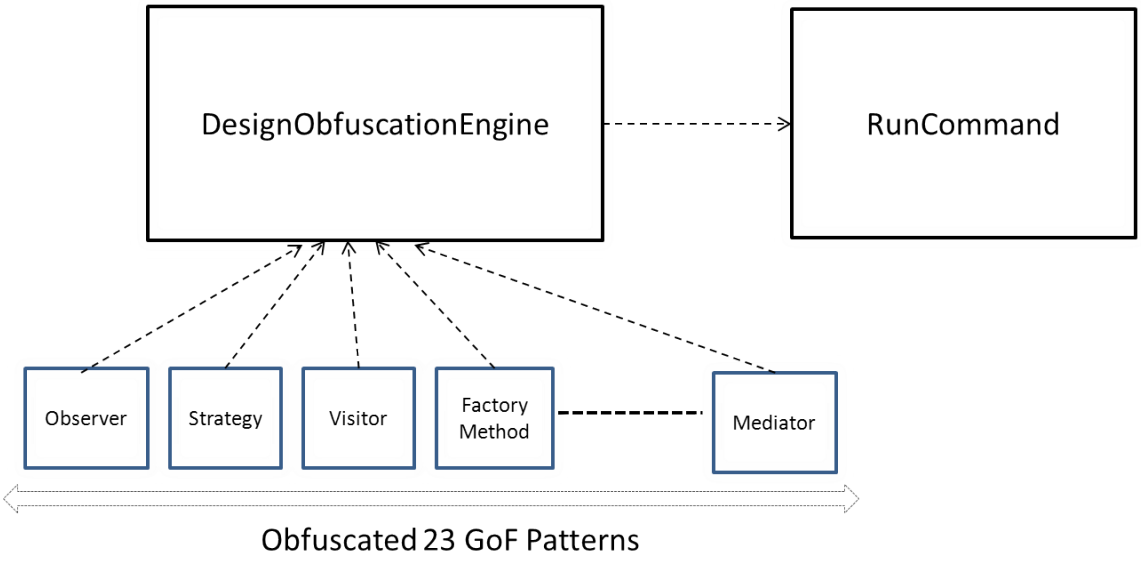

Figure 56: Block diagram DesignObfuscationEngine

4.2 Implementation Platform

The DesignObfuscationEngine is developed in Java SDK v1.7.0 revision 3 using Eclipse AST on Windows OS. An IDE used for development is Eclipse v3.6.2 with necessary plugins installed. A Similarity scoring pattern detection tool works on Windows and Linux OS, whereas the PINOT tool was developed to work on Linux or any UNIX based systems. Therefore, a developed DesignObfuscationEngine must be

platform independent and able to run on machines with Linux OS. The DesignObfuscationEngine is implemented using Java to make it platform independent in order to run both pattern detection tools.

4.3 Program Flow

Basic working and functionality of the DesignObfuscationEngine are explained till now. This section of the report explains the flow of the program using Figure 57. Running the tool starts `DesignObfuscationEngine` class that will initially create Java source files and that are stored in known location. Then `Compiler` class will be delegated and used to compile these source files, the class files are also stored in known location. These Java source and class files are used by `RunCommand` object to run PINOT command for source files and SimilarityScoring command for class files. The pattern detection results from PINOT are shown on command prompt and Similarity Scoring results are given as XML file.

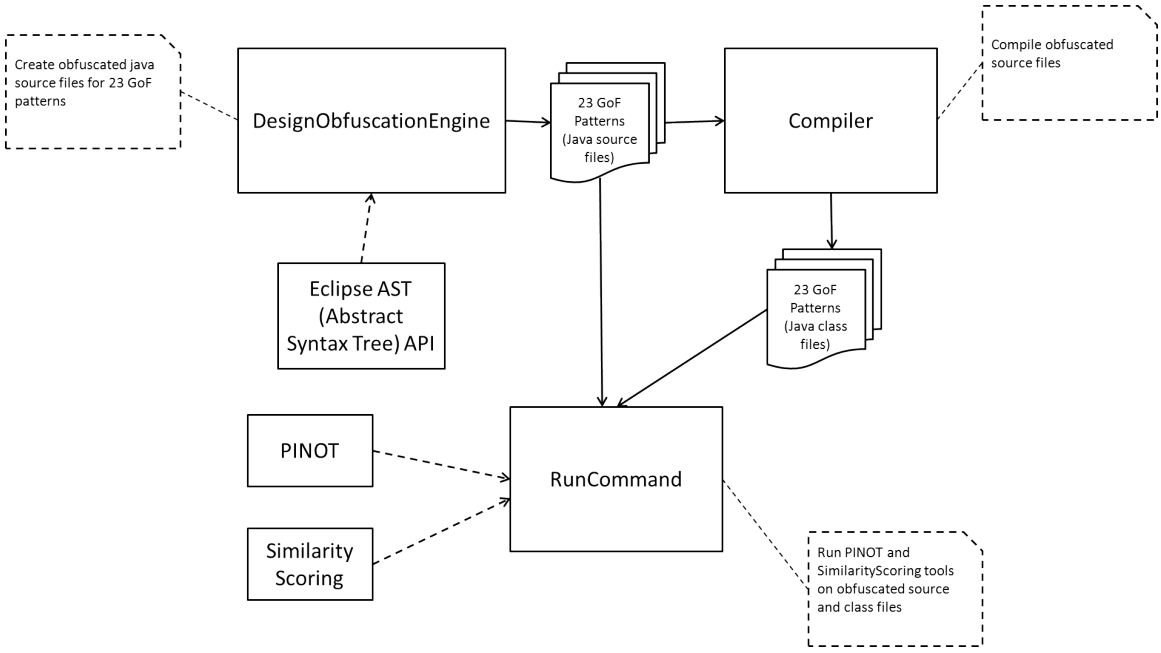

Figure 57: Program flow DesignObfuscationEngine

Code snippets for creating obfuscated files are shown below with a brief explanation of its working:

For creating any Java source file first we need to instantiate a language parser for creating an abstract syntax tree (ASTs) as it decodes the parameter of a language specification (JLS2). This parser is used to create a `CompilationUnit` as shown in Figure 58 below.

```
ASTParser parser = ASTParser.newParser(AST.JLS2);
parser.setSource("".toCharArray());

unit = (CompilationUnit)parser.createAST(null);
              unit.recordModifications();
              AST ast = unit.getAST();
```

Figure 58: Create AST and CompilationUnit

A `CompilationUnit` contains elements that need to be opened before they can be navigated or manipulated. The children of a `CompilationUnit` are a type

of `PackageDeclaration` that declares Packages as shown in Figure 59 and an `ImportDeclaratoin` is for importing necessary packages.

```
PackageDeclaration packageDeclare = ast.newPackageDeclaration();
      unit.setPackage(packageDeclare);
packageDeclare.setName(ast.newSimpleName(className));
```

Figure 59: Create Import and Package Declaration

A `TypeDeclaration` and `MethodDeclaration` are classes used to define class and declare methods, respectively, as shown in Figure 60 below. A `MethodDeclaration` class contains methods such as `setConstructor` and `setModifier` in order to create methods as needed.

```
TypeDeclaration classType = ast.newTypeDeclaration();
classType.setInterface(false);
classType.setModifiers(Modifier.PUBLIC);

MethodDeclaration cp1Declaration =  ast.newMethodDeclaration();
            cp1Declaration.setConstructor(false);
            cp1Declaration.setModifiers(Modifier.PUBLIC);
```

Figure 60: Class and Method declaration

The instance and local variables in a class are created using a `VariableDeclarationFragment` class applied to the `CompilationUnit`. A `SingleVariableDeclaration` is used to create parameters for the methods added to the class. These two classes can be instantiated as shown in Figure 61 below.

```
SingleVariableDeclaration svd = ast.newSingleVariableDeclaration();
svd.setModifiers(org.eclipse.jdt.core.dom.Modifier.NONE);
svd.setType(ast.newArrayType(ast.newSimpleType(ast.newSimpleName("String"))));
svd.setName(ast.newSimpleName("args"));


VariableDeclarationFragment vdf = ast.newVariableDeclarationFragment();
vdf.setName(ast.newSimpleName("obsAbstractFactory"));
```

Figure 61: Variable and Parameter declaration

These child classes are added to a `CompilationUnit`, and these statements were placed in the order they were added to the `CompilationUnit`. These classes taken from the package `org.eclipse.jdt.core.dom`, were used to add different types of statements, such as assignments, and method invocations for developing our tool.

CHAPTER 5

Results and Observations

Obfuscated design pattern tools were scored using two pattern detection tools PINOT and Similarity Scoring. Results from each tool will be discussed separately:

5.1 Test of 23 GoF patterns

Similarity Scoring:

The Similarity scoring tool obtains input .class files, and was unable to detect any design information from the obfuscated source generated using the DesignObfuscationEngine tool. Results of the pattern detection tool are shown in Figure 62. This figure demonstrates that no patterns were detected through a Similarity Scoring, and the design is completely obfuscated from the Similarity algorithm. This similarity algorithm depends on the class information retrieved from the Java class files and uses this information to create Association and generalization graphs. With class-coalescing and class-splitting applied to the design patterns, finding an exact class relationship that matches known graphs is difficult. These also result in no or fewer edges connected within the graph. These graphs are not scored to match known scores. This result in poor similarity score and concludes in no exact matches within any given patterns were discovered.

23 GoF Patterns	Actual Patterns Present	Patterns Detected Normal Source		Patterns Detected Design Obfuscation Source	
		PINOT	Similarity Scoring	PINOT	Similarity Scoring
Abstract Factory	6	0	4	0	0
Builder	1	-	-	-	-
Factory Method	6	0	4	0	0
Prototype	1	-	0	-	0
Singleton	4	1	3	0	0
Adapter	4	0	6*	0	0
Bridge	1	1	-	0	-
Composite	2	2	1	0	0
Decorator	2	0	2	0	0
Façade	2	4	-	1	-
Flyweight	1	1	-	0	-
Proxy	1	1	0	1	0
CoR	1	1	-	0	-
Command	3	-	6*	-	0
Interpreter	1	-	-	-	-
Iterator	1	-	-	-	-
Mediator	5	10	-	4	-
Memento	1	-	-	-	-
Observer	3	2	1	2	0
State	3	0	5~	0	0
Strategy	4	4	5~	0	0
Template Method	2	1	2	1	0
Visitor	3	1	3	0	0

Figure 62: Detected patterns with Obufscation

PINOT:

The PINOT tool runs pattern detection with the source code of the obfuscated design patterns. The PINOT tool was unable to detect patterns, for most of the design patterns, within the source. For patterns that include more abstractions such as Visitor, Bridge, Command, Chain of Responbility (CoR) we need to remove all abstractions in order to hide patterns. Figure 62 shows the results from running the PINOT tool on the obfuscated design pattern source. Results show that it still detects

one Facade, Proxy, TemplateMethod, and CoR pattern that are false positives. There were four false positives, detected as a Mediator pattern. These two observer patterns are false positives that are detected Visitor patterns shown as Observer patterns.

Figure 63 below shows patterns detected on obfuscated design patterns for PINOT and Similarity scoring tools.

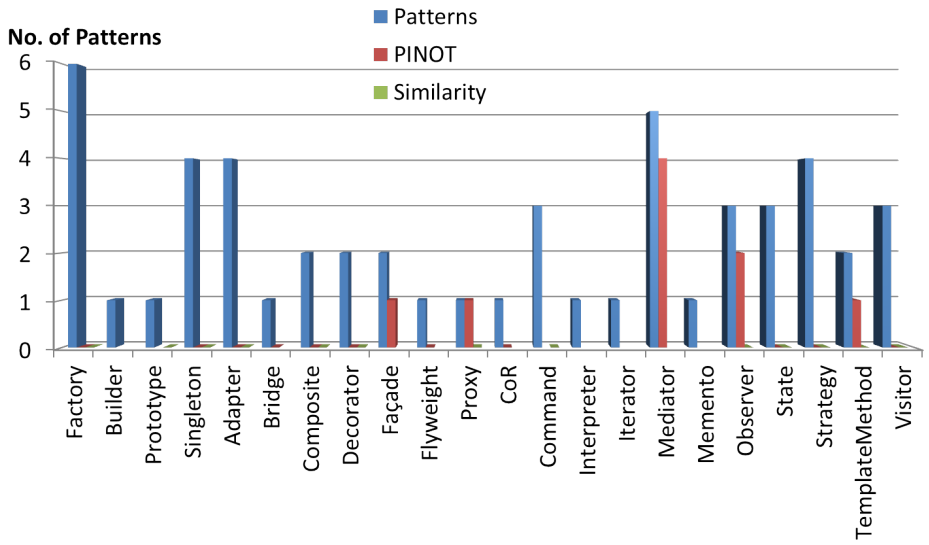

Figure 63: Detected patterns graph with Obufscation

5.1.1 Runtime Analysis

The runtime analysis of obfuscated design patterns demonstrates faster execution times when compared to normal design patterns. This increase in execution time is expected due to a class-coalescing mechanism applied to most of the patterns. In class-coalescing obfuscation, we replace two or more classes with one class; this technique will reduce instantiation of two or more objects and improves the runtime of obfuscated design patterns. Runtime analysis of a large software application will be affected due to class-coalescing; but when only applied to the design patterns

source, it demonstrated an improved runtime in most cases. The running time of each pattern, before and after obfuscation, is shown in Figure 64 and Figure 65 graphs the compared runtime.

23 GoF Patterns	Runtime (ms)	
	Normal	Obfuscate
Abstract Factory	13.24	1
Builder	1.92	1.86
Factory Method	2.37	1.81
Prototype	4.67	2.56
Adapter	4.45	7.24
Bridge	8.97	2.48
Composite	5.7	4.76
Decorator	2.72	3.08
Façade	5.84	4.61
Flyweight	4.51	2.85
Proxy	1.92	2.52
CoR	1.99	6.5
Command	5.7	5.3
Interpreter	5.24	4.69
Iterator	3.94	2.27
Mediator	1.6	1.57
Memento	2.94	2.75
Observer	1.12	2.72
State	2.99	1.86
Strategy	4.5	2.06
Template Method	0.8	3.12
Visitor	5.15	4.86

Figure 64: Runtime Analysis for Normal and Obfuscated patterns

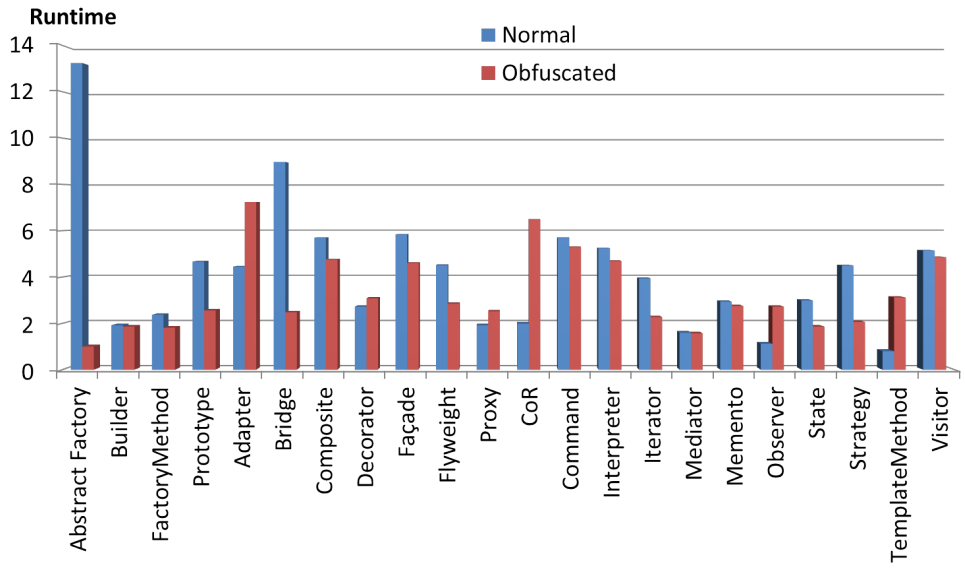

Figure 65: Runtime analysis graph

5.2 Tests on Grand GoF patterns from [26]

Further application testing for GoF patterns from the Patterns in Java book is needed; these patterns are developed using object oriented structures such as inner classes, multilevel inheritance, including multi-patterns such as Composite and TemplateMethod patterns that are present in most of the patterns. Patterns detected using PINOT and Similarity scoring for these patterns are shown in Figure 66 and Figure 67.

Once these patterns are obfuscated using the proposed tool, we can see that patterns are hidden from a Similarity scoring tool; with the exception of one false positive: Prototype pattern. The PINOT tool that uses a source code for detecting patterns can detect a few patterns that are not obfuscated, since our tool does not obfuscate inner classes, multilevel inheritances, and multiple patterns inside a single source. PINOT also detects patterns through method invocation details from ob-

23 GoF Patterns	Actual Patterns Present	Patterns Detected	
		PINOT	Similarity Scoring
Abstract Factory	2	2	1
Builder	2	-	-
Factory Method	2	2	1
Prototype	2	-	1
Singleton	2	1	2
Adapter	2	0	3*
Bridge	2	1	-
Composite	3	2	2
Decorator	1	1	1
Façade	1	2	-
Flyweight	2	0	-
Proxy	2	3	0
CoR	1	5	-
Command	1	-	3*
Interpreter	0	-	-
Iterator	1	-	-
Mediator	0	2	-
Memento	0	-	-
Observer	1	1	1
State	5	0	4~
Strategy	1	5	4~
Template Method	3	1	2
Visitor	1	0	0

Figure 66: Detected patterns without Obfuscation

jects, as does Singleton pattern getInstance() (even after obscuring method name) method. The number of detected patterns through PINOT and Similarity scoring after obfuscation is shown in Figure 68 and Figure 69.

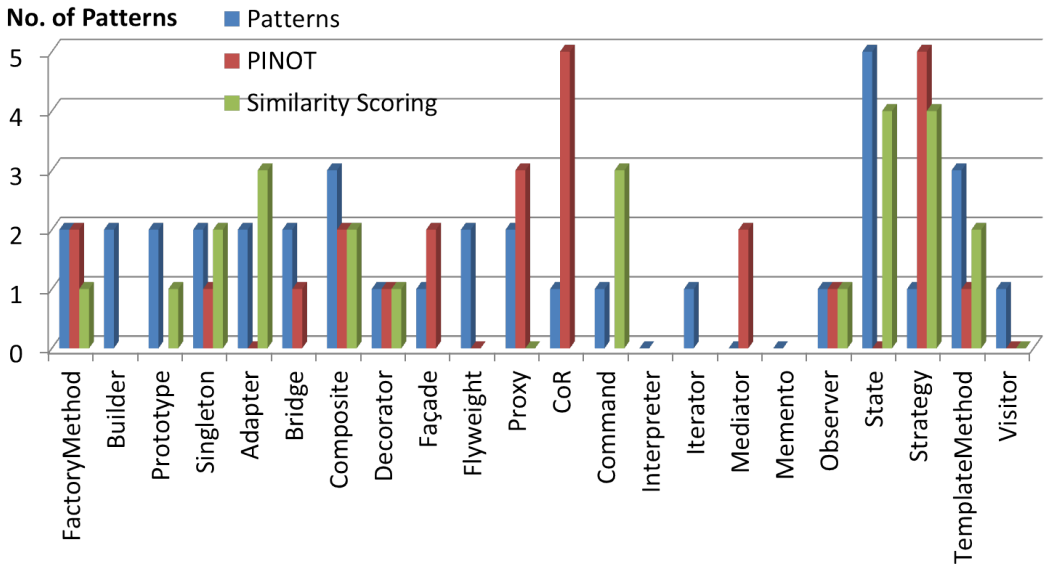

Figure 67: Detected patterns without obfuscation graph

23 GoF Patterns	Actual Patterns Present	Pattern Detected Normal Source		Patterns Detected Obfuscated Source	
		PINOT	Similarity Scoring	PINOT	Similarity Scoring
Abstract Factory	2	2	1	1	0
Builder	2	-	-	-	-
Factory Method	2	2	1	1	0
Prototype	2	-	1	-	1
Singleton	2	1	2	0	0
Adapter	2	0	3*	0	0
Bridge	2	1	-	0	-
Composite	3	2	2	3	0
Decorator	1	1	1	0	0
Façade	1	2	-	3	-
Flyweight	2	0	-	0	-
Proxy	2	3	0	2	0
CoR	1	5	-	5	-
Command	1	-	3*	-	0
Interpreter	0	-	-	-	-
Iterator	1	-	-	-	-
Mediator	0	2	-	3	-
Memento	0	-	-	-	-
Observer	1	1	1	1	0
State	5	0	4~	0	0
Strategy	1	5	4~	0	0
Template Method	3	1	2	0	0
Visitor	1	0	0	0	0

Figure 68: Detected patterns with Obfuscation

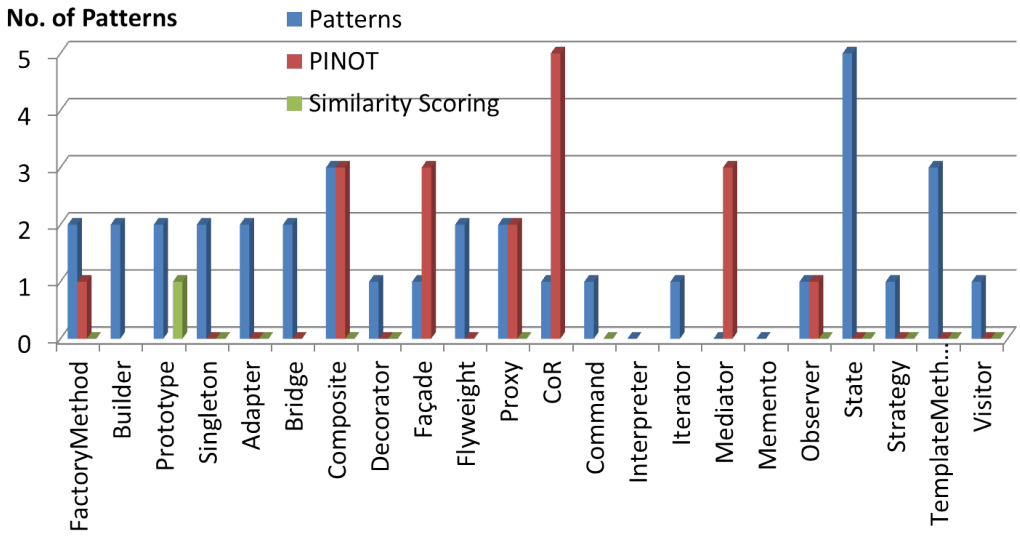

Figure 69: Detected patterns with obfuscation graph

5.3 Tests for Vince Huston patterns [44]

The design patterns from Vince Hustons website also use high object orientation structures with the addition of an interface for several implementations and multilevel inheritance. These patterns also have multipatterns, i.e., pattern within patterns. Detection of patterns, without obfuscation of Hustons patterns is shown in Figure 70 and Figure 71.

23 GoF Patterns	Actual Patterns Present	Patterns Detected	
		PINOT	Similarity Scoring
Abstract Factory	2	2	1
Builder	1	-	-
Factory Method	2	2	1
Prototype	0	-	1
Singleton	0	0	0
Adapter	1	0	0
Bridge	1	1	-
Composite	1	1	1
Decorator	1	1	1
Façade	0	3	-
Flyweight	1	0	-
Proxy	1	0	0
CoR	1	1	-
Command	1	-	0
Interpreter	0	-	-
Iterator	1	-	-
Mediator	1	9	-
Memento	0	-	-
Observer	1	2	2
State	10	0	4~
Strategy	1	0	4~
Template Method	3	0	4
Visitor	3	0	3

Figure 70: Detected patterns without Obfuscation

Obfuscation of these patterns demonstrates false positives for PINOT and a Similarity scoring technique can detect a false positive template method. Detection of patterns with is shown in Figure 72 and Figure 73.

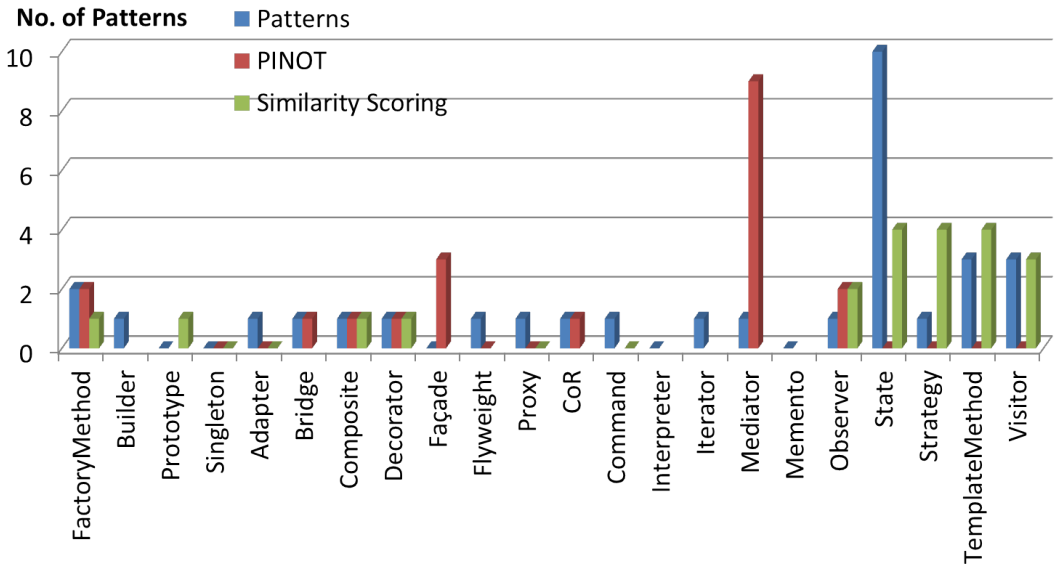

Figure 71: Detected patterns without obfuscation graph

23 GoF Patterns	Actual Patterns Present	Pattern Detected Normal Source		Patterns Detected Obfuscated Source	
		PINOT	Similarity Scoring	PINOT	Similarity Scoring
Abstract Factory	2	2	1	1	0
Builder	1	-	-	-	-
Factory Method	2	2	1	1	0
Prototype	0	-	1	-	0
Singleton	0	0	0	0	0
Adapter	1	0	0	0	0
Bridge	1	1	-	0	-
Composite	1	1	1	3	0
Decorator	1	1	1	0	0
Façade	0	3	-	9	-
Flyweight	1	0	-	0	-
Proxy	1	0	0	0	0
CoR	1	1	-	0	-
Command	1	-	0	-	0
Interpreter	0	-	-	-	-
Iterator	1	-	-	-	-
Mediator	1	9	-	8	-
Memento	0	-	-	-	-
Observer	1	2	2	2	0
State	10	0	4~	0	0
Strategy	1	0	4~	0	0
Template Method	3	0	4	0	3
Visitor	3	0	3	0	0

Figure 72: Detected patterns with Obfuscation

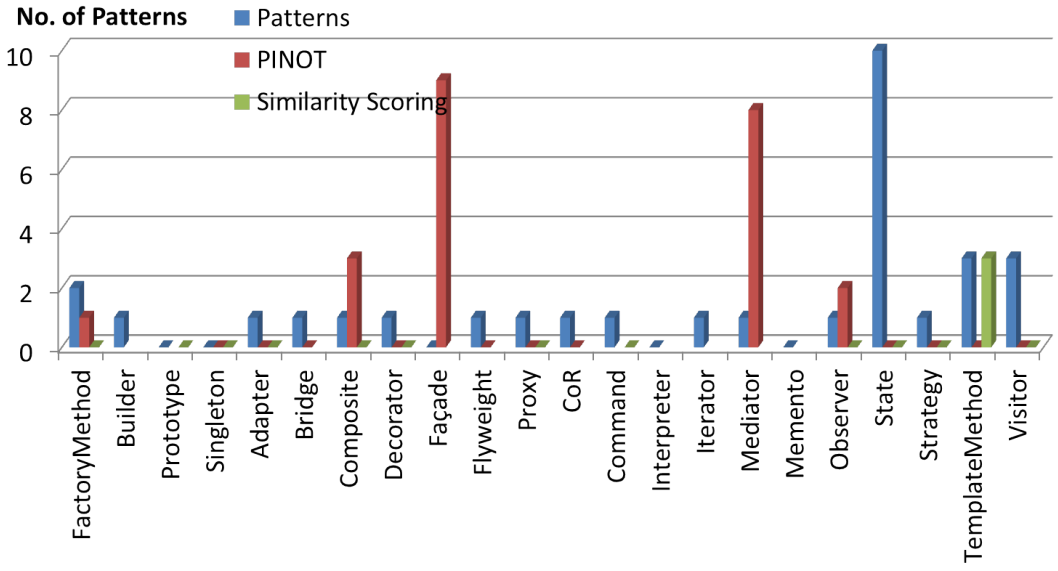

Figure 73: Detected patterns with obfuscation graph

5.4 Observations

These pattern detection tools use different techniques for analyzing source or class files. The Similarity scoring algorithm uses a symmetry scoring approach that was unable to detect patterns from the obfuscated class files. The PINOT tool collects information from blocks such as class hierarchies, method invocations, and relationships between classes. This information is analyzed through a behavior mechanism or examined for related patterns, as well as structural aspects used to detect patterns. As PINOT gathers all this information, it finds relationship between classes that matches patterns and sometimes arrives at a false positive. Results from present tests demonstrated that PINOT detects false positives even after implementing class-coalescing and class-splitting obfuscation.

The proposed tool must be extended in order to work on the source code adding more complex object oriented structures such as inner classes, multilevel inheritance,

and codes with more abstractions. Tests on patterns from Patterns in Java, demonstrates that patterns with complex structures cannot be completely obfuscated. The present obfuscator cannot perform a complete obfuscation and should be extended to work on different cases.

5.5 Comparison to Proguard and Sandmark

In this section we shall compare pattern detection results for the obfuscate patterns from available obfuscators Proguard, and Sandmark to DesignObfuscationEngine. Figure 74 & Figure 75 show graphs of detected patterns from these three obfuscators.

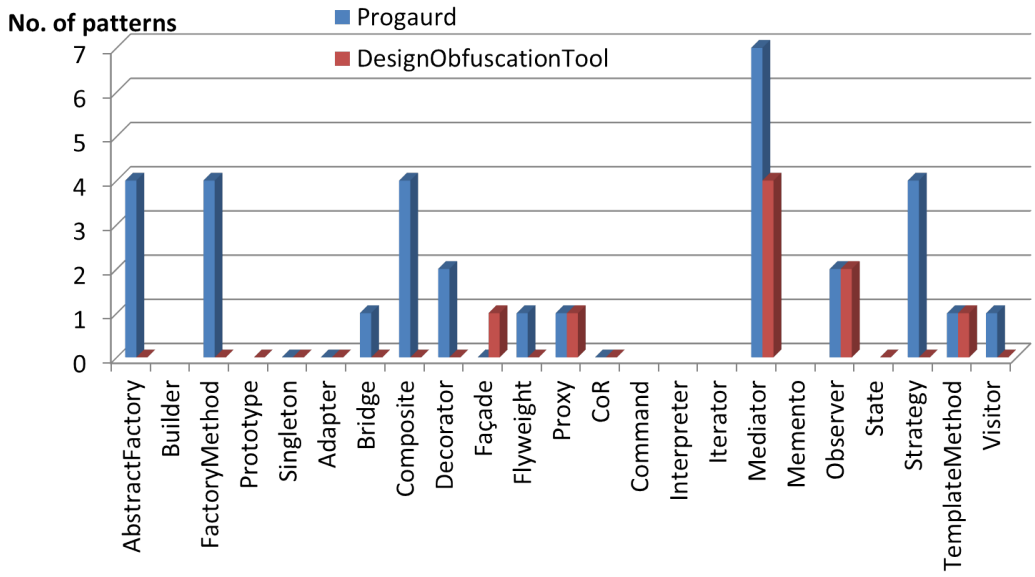

Figure 74: Patterns for Proguard and DesignObfuscationEngine

Firstly comparing patterns detected using PINOT detection tool, Figure 74, demonstrates using Proguard obfuscated code can hide three patterns Singleton, Adapter, and CoR patterns. The obfuscated source from the DesignObfuscationEngine obscures all patterns, and the detected patterns are false positives.

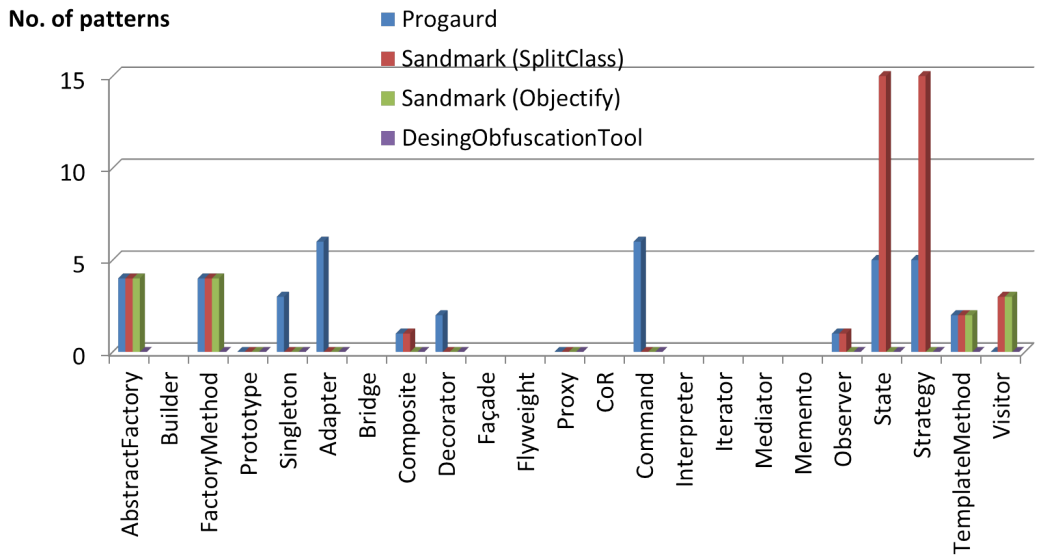

Figure 75: Patterns for Proguard, Sandmark and DesignObfuscationEngine

A comparison using the Similarity scoring detection tool are given in Figure 75, this demonstrates that obfuscation, using Proguard and Sandmark can hide few design patterns. However, the DesignObfuscationEngine completely obscure design patterns from the Similarity scoring tool.

CHAPTER 6

Conclusion and Future Work

Reverse engineering Java applications are simple as binaries will be in bytecode that represent an intermediate symbols between program and machine code. With just a little knowledge regarding the use of bytecode anyone can easily understand details of code such as method, field names, and additional architecture related information. Binaries reverse engineering gives unauthorized access to source code ordering in understanding the architecture and internal structure of a given application. In 2008, the reported loss to the software industry due to software piracy in general was $47.809 billion (Business Software Alliance, May 2008). This loss increased to $51.41 billion by May 2010 (Business Software Alliance, May 2010) [8] and for 2011 that loss is expected to be more than $59 billion. Creating a tool or mechanism that can eradicate reverse engineering is needed, although we cannot completely stop reverse engineering, we can develop tools that obfuscates binaries in order to consume a reverse engineer's time and while not revealing the details of the software.

Obfuscation of the software binary, using available obfuscation tools such as jarg, bb_mug, JavaGuard, do obscure binaries. Well-known tools such as Proguard and Sandmark, use many features to obfuscate code such as inserting dead code, and layout obfuscation. Obfuscation, using these tools can hide patterns and cause detection tools to detect false positives. These tools follow three types of obfuscation control-flow, data obfuscation, and layout obfuscation. Three obfuscation types cannot totally obscure the design of the softwares binary code, as it is easy for detection tools to still detect patterns [29].

In our project, we defined a tool that obfuscates the 23 GoF design patterns and analyzed it for obfuscation. Design obfuscation techniques from [29], class-coalescing, class-splitting, and type-hiding are used for obfuscation. This prototype tool is able to obfuscate different example programs that are designed using 23 design patterns. The obfuscation mechanisms are applied to patterns such that minimum modifications are needed; these modifications include removing interface abstractions from the patterns and adding some methods to implement these functions. The obfuscated patterns are examined using both tools and the results show improved obfuscation over available obfuscation tools. The PINOT tool detects false positives and no patterns are detected for the Similarity scoring detection. One main observation is to remove all abstractions within a software package so that pattern detection would be difficult. In order to hide the design or internal architecture from detection tools, we need to remove or combine all abstractions from software binaries.

Testing the proposed tool by using patterns and object oriented approaches, such as inner classes, multilevel inheritance, using multiple patterns in single code, and blocks of code can break the present obfuscator tool. As shown in Section 5.1, there are patterns detected using PINOT after obfuscation. For future work, this prototype tool should be extended to work on complex object oriented approaches.

The current prototype tool can be used as base work and applied to binaries or packages. Future work for this tool would be to test an obfuscated source from our prototype tool on other available pattern detection tools. Functionality can be extended to work on medium-to-large Java binaries. In order to add and create functions for each Product in Factory pattern for `Factory` class; the problem is to search for Products required, and then create functions in `Client` for initializing each `Product`. This is just one approach to create obfuscated code for extending the

proposed tool.

LIST OF REFERENCES

[1] Alex Blewitt, Alan Bundy, Ian Stark (2001). Automatic verification of Java design patterns. Proceedings 16th Annual International Conference on Automated Software Engineering, 2001. (ASE 2001) [Electronic Version]
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.9.9295

[2] Antoniol, G., Casazza, G., Penta, M.D., & Fiutem, R. (2001, November 15). Object-oriented design patterns recovery, Journal of Systems and Software, Volume 59, Issue 2, Pages 181-196, ISSN 0164-1212, DOI: 10.1016/S0164-1212(01)00061-9 [Electronic version]
http://www.sciencedirect.com/science/
article/B6V0N-449TJ06-J/2/1194eca49fa9a9d8dbafde4af2041130

[3] Atanas Neshkov, DJ Java Decompiler
http://www.neshkov.com/dj.html

[4] Benedusi, P., Cimitile, A., & Carlini U.D. (1992, November). Reverse engineering processes, design document production, and structure charts. Journal of Systems and Software, Volume 19, Issue 3, Pages 225-245

[5] Canfora, G., Cimitile, A., Lucia, A. De, & Lucca, G. A. Di (2000). Decomposing legacy programs: a first step towards migrating to client-server platforms. Journal of Systems and Software, 54(2):99-110.

[6] Chikofsky, E.J.; Cross II, J.H. (1990). Reverse Engineering and Design Recovery: A Taxonomy in IEEE Software. IEEE Computer Society: 1317. [Electronic version]
http://seal.ifi.uzh.ch/fileadmin/User_Filemount/
Vorlesungs_Folien/Evolution/SS05/chikofsky90.pdf

[7] Christian Collberg, Department of Computer Science, The University of Arizona, SandMark: A Tool for the study of Software Protection Algorithms
http://sandmark.cs.arizona.edu/index.html

[8] Cipresso, T. (2009). Software Reverse Engineering Education. Masters thesis, San Jose State University, CA. [Electronic version] Retrieved December 3, 2010, from Software Reverse Engineering (SRE) Web supplement to Masters thesis:
http://reversingproject.info/wp-content/uploads/
2008/10/cipresso_teodoro_cs299_report.pdf

[9] Deepti Kundu. (2011) JShield: A Java Anti-reversing Tool. Masters thesis, San Jose State University, CA. [Electronic version] May 2011, Web supplement to Masters thesis
http://scholarworks.sjsu.edu/etd_projects/161/

[10] Design Patterns [Online]
http://www.oodesign.com/

[11] Eastridge Technology, JShrink
http://www.e-t.com/jshrink.html

[12] Emden, E.V. & Moonen, L. (2002, November). Java quality assurance by detecting code smells. In Ninth Working Conference on Reverse Engineering (WCRE 2002), Richmond, VA, USA, pages 97-107

[13] Eric Lafortune, ProGuard
http://proguard.sourceforge.net/

[14] Eric Lafortune , Proguard Alternative shrinkers, optimizers, obfuscators, and preverifiers
http://proguard.sourceforge.net/index.html#alternatives.html

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software text book
http://c2.com/cgi/wiki?DesignPatternsBook

[16] George Spanogiannopoulos (January 2007) An Analysis of Design Pattern Detection Using PINOT AND Similarity Scoring. Pattern Detection Project, York University [Electronic Version]
www.cse.yorku.ca/~spano/reports/report.pdf

[17] H. A. Mller, J. H. Jahnke, D. B. Smith, M. Storey, S. R. Tilley, and K. Wong, Reverse engineering: a roadmap, in Proc. Conf. Future of Software Engineering, Limerick, Ireland, 2000, pp. 47-60.

[18] Hanpeter van Vliet, Mocha, The Java Decompiler
http://www.brouhaha.com/~eric/software/mocha/

[19] Hidetoshi Ohuchi, jarg Java Archive Grinder
http://jarg.sourceforge.net/

[20] James Hamilton, Sebastian Danicic An Evaluation of Current Java Bytecode Decompilers (2009). Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2009) [Electronic Version]
http://jameshamilton.eu/sites/default/files/JavaBytecodeDecompilerSurvey.pdf

[21] Java Decompiler
http://java.decompiler.free.fr/

[22] Jochen Hoenicke, JODE decompiler and optimizer
http://jode.sourceforge.net/

[23] JSRs (Java Specification Requests) detail JSR# 202
http://www.jcp.org/en/jsr/detail?id=202

[24] M. von Detten, M. Meyer, D. Travkin, Reclipse Reverse Engineering for Eclipse
http://www.fujaba.de/no_cache/projects/reengineering/reclipse.html

[25] Mariano Ceccato, Thomas Roy Dean, Paolo Tonella, Davide Marchignoli, (April 2010) Migrating legacy data structures based on variable overlay to Java, in Journal of Software Maintenance and Evolution, vol. 22, n. 3, pp. 211-237

[26] Mark Grand, Patterns in Java: a catalog of reusable design patterns illustrated with UML 1998, Wiley Computer Publishing

[27] Mark Stamp, Chapter 12: Insecurity in Software (October 2010)
Information Security: Principles and Practices Text book

[28] Merlo, E., Gagne, P.-Y., Girard, J.-F., Kontogiannis, K., Hendren, L.J., Panangaden, P., & Mori, R. de (1995). Reengineering user interfaces. IEEE Software, 12(1):64-73

[29] Mickail Sosonkin, Gleb Naumovich, Nasir Memon (2003) "Obfuscation of design intent in object-oriented applications" DRM03 Proceedings of the 3rd ACM Workshop on Digital rights management

[30] Mickail Sosonkin, Gleb Naumovich, Nasir Memon Design Obfuscator for Java (DOJ)
http://users.rowan.edu/~tang/courses/
ref/metrics/Design%20Obfuscator%20for%20Java%20%28DOJ%29.htm

[31] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. T. Halkidis, (November, 2006). "Design Pattern Detection Using Similarity Scoring", IEEE Transactions on Software Engineering, vol. 32, no. 11, pp. 896-909. [Electronic Version]
http://java.uom.gr/~nikos/publications/TSE_2006.pdf

[32] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. T. Halkidis, Design pattern detection Similarity scoring tool
http://java.uom.gr/~nikos/pattern-detection.html

[33] Nija Sji and Ronald A. Olsson (2006) Recovery of design patterns from Java Source code - 21st IEEE/ACM International Conference on Automated Software Engineering [Electronic Version]
http://www.cs.ucdavis.edu/~shini/research/pinot/reverseJavaPatterns.pdf

[34] Nija Sji and Ron Olsson, (2006) PINOT (Pattern INterference and recOvery Tool) Detection the GoF Patterns,
http://www.cs.ucdavis.edu/~shini/research/pinot/

[35] Open Source obfuscators in Java
http://java-source.net/open-source/obfuscators

[36] Pavel Kouznetsov, JAD Java Decompiler Download Mirror
http://www.varaneckas.com/jad

[37] Prashant Chandrakar, Java Decompilers (.class to .java file)
http://yuvadeveloper.blogspot.com/2009/03/java-decompilers-class-to-java-file.html

[38] Robert Lie, Java class decompiled: HelloWorld.class
http://www.mobilefish.com/download/java/HelloWorld.html

[39] Rudolf K. Keller, Reinhard Schauer, Sbastien Robitaille, and Bruno Lagu (2002) Pattern-Based Design Recovery with SPOOL, Advances in Software Engineering. Comprehension, Evaluation, and Evolution, chapter 6, pages 113-135. Springer. [Electronic Version]
http://www.iro.umontreal.ca/~keller/Publications/Papers/2002/cbook-2002-dpr.pdf

[40] Smardec, Allatori Obfuscator
http://www.allatori.com/features.html

[41] Stefan Bebbo Franke, bb_mug a Java class obfuscator
http://www.bebbosoft.de/\#java/mug/index.wiki

[42] Thomas Kuhn, Eye Media GmbH, Olivier Thomann, IBM Ottawa Lab, Java Code Manipulation AST
http://www.eclipse.org/articles/
article.php?file=Article-JavaCodeManipulation_AST/index.html

[43] Thorsten Heit, JavaGuard bytecode obfuscator
http://sourceforge.net/projects/javaguard/

[44] Vince Huston Design Patterns
http://www.vincehuston.org/dp/

[45] Viral Patel, Java Virtual Machine, An Inside story
http://viralpatel.net/blogs/2008/12/java-virtual-machine-an-inside-story.html

[46] Wiki - Abstract Syntax Tree (AST)
http://en.wikipedia.org/wiki/Abstract_syntax_tree

[47] Wiki - Java Class file
http://en.wikipedia.org/wiki/Java_class_file

[48] Wiki - Reverse Engineering of Software
http://en.wikipedia.org/wiki/Reverse_engineering#Reverse_engineering_of_software

[49] yWorks, yGuard Bytecode Obfuscator and Shrinker
http://www.yworks.com/en/products_yguard_about.html

[50] Zelix, KlassMaster
http://www.zelix.com/klassmaster/features.html