

DP-Miner: Design Pattern Discovery Using Matrix

Jing Dong, Dushyant S. Lad, Yajing Zhao
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083, USA
{jdong, dsl044000, yxz045100}@utdallas.edu

Abstract

Design patterns document expert design experience in software system development. They have been applied in many existing software systems. However, pattern information is generally lost in the source code. Discovering design patterns from source code may help understand system designs and further change the systems. In this paper, we present a novel approach to discovering design patterns by defining the structural characteristics of each design pattern in terms of weight and matrix. Our discovery process includes several analysis phases. Our approach is based on the XMI standard so that it is compatible with other techniques following such standard. We also develop a toolkit to support our approach. An industrial size case study is conducted to evaluate our approach and tool.

KEYWORDS

Design Pattern, Reverse Engineering, Matrix, XMI, UML, Design Pattern Discovery

1. Introduction

Large computer-based systems are normally difficult to understand and change due to lack of software architecture and design documentation. After the deployment of the systems, the original architecture and design information is generally lost. Source code becomes the only source to understand the systems. However, source code is typically large in size and hard to comprehend. It is very time-consuming and error-prone to read source code manually. Understanding the systems is very important since it may help on changing them. Software systems generally should be amenable to changes due to constant changes of user requirements, platforms, technologies and environments. Change is a constant theme of computer-based system design and development.

To understand the source code of a computer-based system, we need to recover the original architectural and design decisions and tradeoffs. Software design patterns [9] document expert design experience and may help capture design decisions and record design tradeoffs.

Thus, discovering design patterns used in a software system may recover the original design decisions and tradeoffs and help on the understanding and future change of the system. Most of design patterns embed future changes that may only affect limited part of a design pattern. This evolution process can be achieved by adding or removing design elements in existing design patterns. When the pattern-related information is recovered from the source code, the developers are able to take advantage of the design patterns to change the design and thus the whole systems.

In this paper, we propose a novel approach based on matrix and weight to discover design patterns from source code. In particular, the system structure is represented in a matrix with the columns and rows to be all classes in the system. The value of each cell represents the relationships among the classes. The structure of each design pattern is also represented in another matrix. The discovery of design patterns from source code becomes matching between the two matrices. If the pattern matrix matches the system matrix, a candidate instance of the pattern is found. Besides matrix, we use weight to represent the attributes/operations of each class and its relationships with other classes. In addition to the structural aspect, our approach investigates the behavioral and semantic aspects of pattern discovery. A toolkit has also been developed to support our approach.

Different from other pattern mining approaches, our approach is based on the XML Metadata Interchange (XMI) [25] that is an interchange format for metadata in terms of the Meta Object Facility (MOF) [23]. XMI specifies how UML models [4] are mapped into a XML file. By representing a UML model in XML, the UML model can be searched for patterns. Thus, our pattern detection techniques can be naturally integrated with other techniques and tools following the XMI standard.

The remainder of this paper is organized as follows: the next section describes an overview of our approach. Section 3 presents the details of our approach in terms of several analysis phases. Section 4 discusses a case study on the Java.awt [22]. The last two sections cover related work and conclusions.

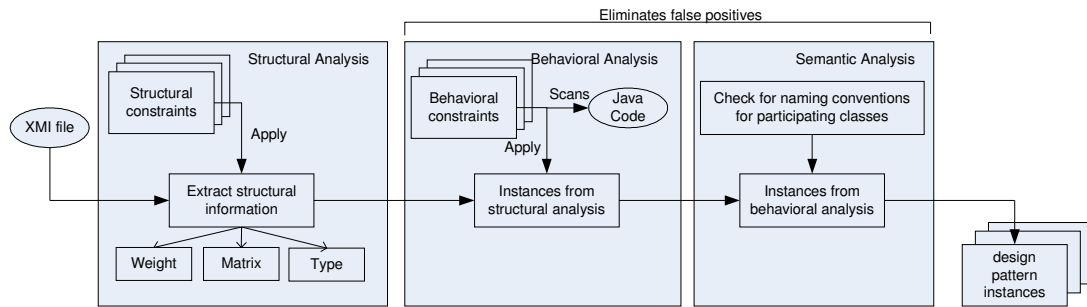


Figure 1 The Overall Architecture of the Approach

2. Overview of the approach

Figure 1 depicts the overall architecture of our approach to discovering design patterns in the source code of computer-based systems. Our approach uses the XMI standard as the intermediate representation format. The object-oriented systems can be initially reverse engineered into UML diagrams by existing UML tools, such as IBM Rational Rose [24]. Since UML diagrams are typically saved in proprietary formats of the corresponding UML tools, a standard XMI format of the UML diagrams has been defined and the plug-in of these UML tools has been developed to export UML diagrams into the XMI format. For example, the plug-in of IBM Rational Rose is called UniSys which can translate UML class diagrams into XMI format. Using the XMI plug-ins, UML class diagrams can be transformed into files in XMI format. These XMI files include all system design information, such as classes, attributes, operations, and different relationships between classes. Instead of analyzing the source code directly, our approach analyzes these XMI files.

Our approach consists of three phases: structural, behavioral, and semantic analyses. The structural analysis phase concentrates on the structural characteristics of the system, such as classes and their relationships. We introduce a novel approach that can extract structural information from software system design into weight, matrix and type. The structural aspect of each design pattern is also represented similarly by weight, matrix and type. Thus, the structural discovery of design pattern can be simplified by calculating and matching a group of numbers, instead of geometrical elements.

The results of the structural analysis may include the detected instances that are actually not a design pattern. Although such instances satisfy the structural characteristics of a design pattern, they may not be the instances of such design pattern due to missing behavioral characteristics. Such instances are false positives. Each design pattern generally may include both structural and behavioral characteristics. Thus, the

behavioral analysis checks the results from the structural analysis for false positives. In this phase, our approach may check back the XML files or directly into the source code. Different from the structural analysis, such checks aim at verifying particular part of the system. Therefore, we do not need to go through the whole files.

It may be hard to distinguish some design pattern instances, such as the Strategy and Bridge patterns, since they have the same structural and behavioral characteristics. In such case, our approach analyzes the semantic information of the pattern instances. When a design pattern is applied into a system, the developers may follow some naming convention and include the role information of a pattern when naming the classes. Therefore, our approach checks whether the naming convention can distinguish design pattern instances.

We automated the structural, behavioral, and semantic analyses in our DP-Miner tool so that the users only need to generate the XMI input from existing tools.

3. Pattern discovery using matrix

Large computer-based systems normally include a large number of classes in the source code. These classes may relate to each other in a fairly complex manner. An instance of a design pattern is typically a group of classes that are structured and interacting with each other in a particular way. Detecting such structure and interactions in a large system can be challenging and time-consuming tasks. To efficiently discover design pattern instances in a large design, we encode the information about the classes and their relationships into weight and matrix. Our approach includes three phases: structural, behavioral, and semantic analyses.

3.1 Structural Analysis

In the structural analysis phase, we propose to use weight and matrix to represent the structural information of the system and the design patterns to be discovered.

In our approach, we take advantage of an important property of the prime number, i.e., the product of prime numbers is always a unique composite number. In other words, any given number can be broken into the multiple

of a unique list of prime numbers. Hence a single number can represent a unique combination of prime numbers. Based on this property we associate each structural element with a unique prime number. We can associate the most frequently used structural element with the lowest prime number other than 1. We exclude 1 because 1 is the identity for multiplication. We associate the prime numbers to the structural elements starting from 2 and come up with the following table:

Structural Elements	Prime number value
Attribute	2
Method	3
Association	5
Generalization	7
Dependency	11
Aggregation	13

Table 1 Prime Numbers of Structural Elements

Once we have every structural element associated with a unique prime number we can calculate the weight of each class and a matrix of the relationships between these classes. Both weight and matrix embed important information that allows us to integrate complex structural characteristics into numbers and matrices.

We define the weight of a class to be the multiples of the corresponding prime numbers to the powers of the total numbers of attributes, methods, and relationships associated with the class. The formula for calculating the weight of a class is:

$$W = w_a \times w_m \times w_{as} \times w_g \times w_d \times w_{ag}$$

$$w_a = 2^{(\text{number of attributes in the class})}$$

$$w_m = 3^{(\text{number of methods in the class})}$$

$$w_{as} = 5^{(\text{number of Association relationship of the class})}$$

$$w_g = 7^{(\text{number of generalization relationship of the class})}$$

$$w_d = 11^{(\text{number of dependency relationship of the class})}$$

$$w_{ag} = 13^{(\text{number of aggregation relationship of the class})}$$

where W represents the weight of a class. Once we compute the weight of the class we can easily get ideas about the structural elements of the class. If the weight of a class is 5670, for example, then this number is a unique combination of the prime numbers associated with the structural elements. Hence 5670 is broken into $2^1 \times 3^4 \times 5^1 \times 7^1$ which implies that there are one attribute and four methods in the class which has one association and one generalization relationships associated with it. In addition to relationships between classes, we consider the number of attributes and methods in each class because some design patterns require certain attributes and/or methods in their participating classes. For example, all participating classes of the Adapter pattern include at least one method as shown in Figure 6.

The value of W can become overflow due to potential huge value of the weight of the classes with a large

number of attributes and methods. Optimizations are done to avoid such overflow conditions in Section 3.4.

Once we calculate the weight of a class which is symbolic of all structural elements, we can check whether a class in a system design belongs to a design pattern instance based on the single value of its weight. If a design pattern requires a particular class having minimum weight of 5670 then we can consider all classes with weight of 5670 or an integral multiple of it in a system design for that role of the design pattern. Thus, we can reduce the process of structural discovery of design pattern into simple arithmetical computations.

The relationships between classes are also critical in design pattern discovery. For each design pattern, there may be some relationships that have to exist. For this purpose we encode the relationships between the classes of a system design in a similar way as weight. In particular, we build a matrix that encodes the relationships between every two classes into a value of the cell corresponding to the two classes. In this way, the system design is encoded into an $n \times n$ matrix where n is the number of classes in the system. Similarly, the relationships in a candidate design pattern are also encoded into another $m \times m$ matrix where m is the number of participating classes in the design pattern. The discovery of design pattern is further reduced into the matching of the two matrices. To construct the matrix for both a system design and a design pattern we provide the following rules:

1. Initially the matrix is $n \times n$ where n is the number of classes involved. Each row and column represents a class arranged in the symmetric order, i.e., row i and column i must have the same class name. Each cell initially has value 1.
2. For each class i , if it has association relationship with another class j then we multiply the value of cell (i,j) by 5.
3. For each class i , if it has generalization relationship with another class j then we multiply the value of cell (i,j) by 7.
4. For each class i , if it has dependency relationship on another class j then we multiply the value of cell (i,j) by 11.
5. For each class i , if there is aggregation between siblings of class i to form an aggregate class j then the value at (i,j) is multiplied by 13.

To calculate the weight and matrix, we can find all necessary information from the XMI file generated from source code, except the dependency and aggregation relationships. We will deal with the dependency in the behavioral analysis phase. It is generally hard to distinguish the aggregation from association in reverse engineering processes since the difference between aggregation and association is generally at semantic

level, which is hard to distinguish at code level. In our approach, rule 5 is not applicable. Discovering aggregation depends on the interpretation of the intent of the developer. It is not a syntax based aspect hence we cannot detect the aggregation relationship based on the information provided in the XMI file from source code.

We apply the first three rules for constructing the matrix based on the XMI file. Once we created the weight and matrix from the XMI file of the source code we can check whether a particular class satisfies the requirements of a particular role of a design pattern by matching the weight and matrix of a design pattern with those of a system design. If the weights and matrix of a group of classes in a system design is integral multiples of those of the corresponding classes of a design pattern, this group of classes is considered as a candidate instance of the design pattern. Examples of weight and matrix calculations are presented in 4.2.

Besides weight and matrix, we check class type, i.e., if it is an interface, an abstract or a concrete class. Some design patterns may require their participating classes to be of certain types. If we can find a group of such classes, each of which satisfies a particular role of a design pattern, we record them as an instance of that design pattern. This list of pattern instances discovered from the structural analysis contains a number of false positives that are removed in the later analysis phases.

3.2 Behavioral Analysis

Each design pattern typically includes both structural and behavioral aspects. The structural aspect describes the static organization of the classes and their relationships in the pattern. The behavioral aspect presents the dynamic collaborations among the participants in the pattern. Thus, only structural analysis is not enough for discovering design pattern. The results from structural analysis may include false positive instances. Behavioral analysis can help eliminate such false positive cases. Based on the results from our structural analysis we further check the expected behavioral (dynamic) characteristics of each instance.

The main goals of behavioral analysis include finding the dependency that exists between classes and if some class delegates the method call. Similarly to the structural analysis, each design pattern may have different behavioral characteristics. This analysis can be fairly complex based upon these characteristics of the design pattern. One of the main difficulties in behavioral analysis is that there can be various possible implementations for similar expected behavior. One of such examples is the behavioral characteristic of checking the existence of a method call from one class to an overridden method in the subclass. It can be diagrammatically represented in Figure 2. This dynamic

behavior can be implemented in the Component class in the following different ways:

1. ((FlipBufferStrategy)bufferStrategy).getCapabilities();
2. bufferStrategy = new BltBufferStrategy(numBuffers, caps);
bufferStrategy.getCapabilites();
3. BufferStrategy var=new BufferStrategy();
var=(SingleBufferStrategy)bufferStrategy;
var.getCapabilities();

In all these three different implementations the behavioral characteristics are the same. There can be several other implementations as well. Hence the knowledge is hard to grasp at the implementation level. We need to abstract the idea behind the implementation which is the same for all different implementations.

To solve this problem of implementation variations, one of the solutions is to first represent the source code in a Control Flow Graph (CFG). This would later help in construction of sequence diagram for that code accurately, independent of the ways it is implemented. Recent work [16] has been done for static control-flow analysis and would be available as a plug-in for Eclipse in near future.

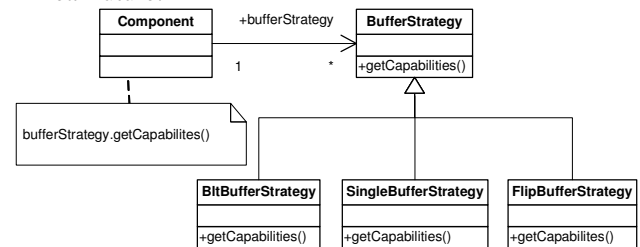


Figure 2 Instance of the Strategy pattern

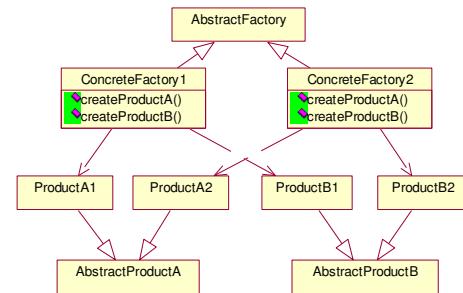


Figure 3 Abstract Factory Pattern Instance

In our approach, we try to identify these behavioral characteristics of each design pattern by scanning certain part of source code. Based on the results from our structural analysis, we have already found the candidates of design pattern instances. Thus, we can narrow down our search to certain files or classes, which can significantly reduce the scanning space.

For example, the dependency relationship is an important characteristic in some design patterns, such as the Abstract Factory pattern shown in Figure 3. This instance of the Abstract Factory pattern requires two groups of inheritance trees that are linked by

dependency between subclasses. To confirm such a pattern during behavioral analysis, hence, we only need to scan the ConcreteFactory1 and ConcreteFactory2 classes for such dependency.

3.3 Semantic Analysis

Most of the false positives can be eliminated in the behavioral analysis phase. However, there are certain closely related design patterns which are similar with respect to their structural and behavioral aspects but just different by their intent with which they were created. The Strategy and State patterns, Strategy and Bridge patterns are examples of such pairs of design patterns.

There are several ways for removing such ambiguous instances of design patterns from the results of the structural and behavioral analyses:

1. Searching the design documentation of source code.
2. Searching the in-line comments in the source code.
3. Getting clues from naming conventions of classes.

The first method generally works the best. However, the design documentation is not always available. It is quite common that the design documentation is lost, which is one of main reasons for reverse engineering. In the second method, in-line documentation generally describes the functionalities of a class, instead of pattern-related information. In addition, both the design and in-line documentation is likely to remain unchanged even when the corresponding code has been changed. The third method seems to be pretty relevant since the designers/developers are typically required to follow some convention when naming classes, attributes, and operations, whose names are generally not changed over time. Such naming convention sometimes may give us clues on their originally intent. For example, Figure 2 presents a pattern instance that we discovered from Java.awt based on our structural and behavioral analyses. Initially, our approach cannot distinguish whether it is an instance of the Bridge or Strategy pattern. After we investigated the case closely, we found the naming convention actually indicates it is an instance of the Strategy pattern since several class names include “Strategy” as substring. Thus, the Component class plays the role of Abstraction. The BufferStrategy class plays the role of Implementor. The BltBufferStrategy, FlipBufferStrategy and SingleBufferStrategy classes are the concrete implementors. Based on this study, we provide additional semantic analysis in our tool to check whether the naming convention has any clues to distinguish pattern instances that are similar in their structures and behaviors.

The limitation with this approach is that not all classes presented in a design pattern may be named with pattern-related information. In addition, the reverse engineering process is often applied to the

implementation which is part of legacy code that might be developed without the pattern knowledge in consideration. Hence the naming conventions of classes might not always contain any clue for design pattern.

In our case study on the Java.awt package we found some classes include “Adapter” in their names. These classes are ContainerAdapter, ComponentAdapter, DragSourceAdapter, DropTargetAdapter, FocusAdapter, HierarchyBoundAdapter, KeyAdapter, MouseAdapter and WindowAdapter. Although these class names contain “Adapter”, they are not qualified to be instances of the Adapter pattern due to two main reasons. First, the above Adapter classes are abstract classes instead of concrete classes. Second, there is no Adaptee presented in these instances. These Adapter classes are actually partial instances of the Adapter pattern, which are left open for future development or left for the users to provide their code for implementation of, e.g., the Adaptee class. As a result, we conduct the semantic analysis after structural and behavioral analyses.

3.4 Optimizations

In the previous sections, we present our novel approach on reducing pattern matching problem into weight and matrix calculations. In this section, we introduce some optimizations of our approach to deal with large cases. With a large number of attributes, methods and relations, the weight of a class can be a very large number that may cause overflow in practice. For software packages like Java.awt that contains 458 classes and 111 interfaces, the matrix construction time takes 28 seconds in our approach, which is not very good. For this purpose, we optimize our approach to improve scalability and performance in the following processes: (1) Weight calculation for class; (2) Matrix construction process.

Our original approach on calculating the cumulative weight of each class works fine for the applications with a small number of classes which have a small number of attributes, method and simple relationships among them. When we consider large software system, such as Java.awt, however, it has the scalability problem. The weight variable suffers an overflow. Consider the KeyEvent class with more than 150 attributes in Java.awt. Its weight is over 2^{150} which definitely results in overflow. To solve this problem, we optimize our weight calculation method.

Consider the design patterns to be identified; the number of attributes in any class participating in a design pattern is typically a small number. For example, the 23 patterns presented in [9] have at most one attribute in any class of each design pattern. Similarly, the number of operations in any class participating in a design pattern is also a small number, e.g., less than five

in all GoF patterns. If a class in an application has more than five operations, thus, it is not meaningful to calculate its weight in terms of all operations. Based on this observation, we optimize our weight calculation method by considering at most five attributes and operations in each class. If a class has more than five attributes/operations, we consider this class has five attributes/operations in our weight calculations. Hence the maximum weight of a class after calculating the attributes and methods is $2^5 \times 3^5 = 7776$.

For the relations between classes, we also optimize our methods based on the observation that two classes normally have only one instance of each relationship in a design pattern. For example, two classes may have at most one instance of generalization relationship. Although two classes in a software system may have multiple association and dependency relationships, there is generally only one instance of each kind of relationship between classes in a design pattern. Thus, it is not relevant to consider more than one instance of each kind of relationships in our weight calculations. Based on Table 1, the weight bases of the generalization, association, and dependency relationships are 5, 7, and 13, respectively. Hence, the maximum weight of a class becomes: $2^5 \times 3^5 \times 5 \times 7 \times 13 = 31352832$. This solves the problem of class weight overflow.

Similarly to the weight, we optimize the matrix calculation which represents the inter-class relationships. Since our matrix only encodes the relationships between classes, we consider only one instance of each kind of relationships when we calculate each cell of the matrix. Even though two classes of a software system may have more than one instance of the same kind of relationships, we only consider one instance of such relationship. For example, suppose ClassA and ClassB have three association relations starting from ClassA to ClassB and one generalization from ClassA to ClassB. Then the resulting value of the cell in the row of ClassA and the column of ClassB is $7 \times 5 = 35$ instead of $7 \times 5^3 = 875$. This optimization is based on the similar facts as we restricted the number of attributes, methods and relationships during class weight calculation. Hence our approach restricts the maximum value of any cell of the matrix to $7 \times 5 \times 13$, i.e., 455.

Another optimization on calculating the matrix is to change the way we scan the XMI file. Initially, we scanned the XMI file by relationships, i.e., we look for all occurrences of one relationship between classes and multiply the corresponding cell before those of next relationship. Although it may save some time on calculating the matrix, it requires scanning the XMI file several times. In practice, we found that it is more critical to save time on scanning the XMI file than on calculating the matrix. Thus, we revised our algorithm to

get the list of all possible relationships in which a class involves and then set the value of all cells in the row of the particular class depending on that list. In this way, we can calculate the matrix with a single iteration on the XMI file. These optimizations reduce the matrix construction time for Java.awt to 15 seconds.

4. Case Study

In this section we present a case study on the Java.awt package in JDK 1.4 using our approach to discovering design patterns. AWT stands for Abstract Windowing Toolkit, a powerful and flexible Java class library for the development of Graphical User Interface (GUI). It contains rich user components like Frames and Panels which ease GUI development and can run on any platform. The definition of each such component also contains a range of actions which can be performed on a specific event associated with the component. The AWT package also contains different layouts for placing components like Scrollbar, Buttons, etc., onto containers like Frame or Panel. These layouts do not depend on window size or screen resolution [22]. The Java.awt package consists of 346 files. These files contain a total of 485 classes (including inner classes) and 111 interfaces. It is large open source software.

In this case study on the Java.awt package we used a PC with 3.4 GHz Pentium processor, 1 GB of RAM and Windows XP operating system.

We developed a toolkit, DP-Miner, to support our approach described in the previous section. In this case study, we use DP-Miner to first calculate the weight of all classes and the matrix describing their relationships in the Java.awt package. We then describe the discovery of some design pattern instances based on our structural, behavioral, and semantic analyses.

4.1 Weight and Matrix

As discussed in the previous section, the weight of each class represents the number of attributes/operations in the class and the relationships with other classes. Figure 4 displays a screen shot of DP-Miner showing the weight of each class. The rightmost column lists all classes in the Java.awt package with their corresponding weights in the second rightmost column. These weights are computed based on the number of attributes, methods, outgoing associations, and outgoing generalization relationships, which are shown on the same row. For example, the weight of class SelectiveAWTEventListener is 34020, which contains 2 attributes, 5 methods, 1 association, and 1 generalization relationship. Only 5 methods are counted in the weight although Figure 4 shows it has 6.

Some classes may have more than one outgoing generalization relationships because our current

approach treats the following two scenarios to be equivalent: (1) class *i* extends class *j*, (2) class *i* implements interface *j*. Therefore, a class may have multiple parent classes that are either an abstract class or interfaces.

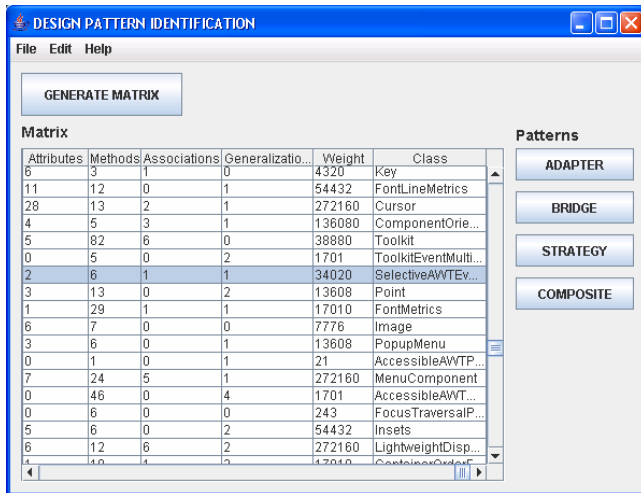


Figure 4 Weight of all classes in Java.awt

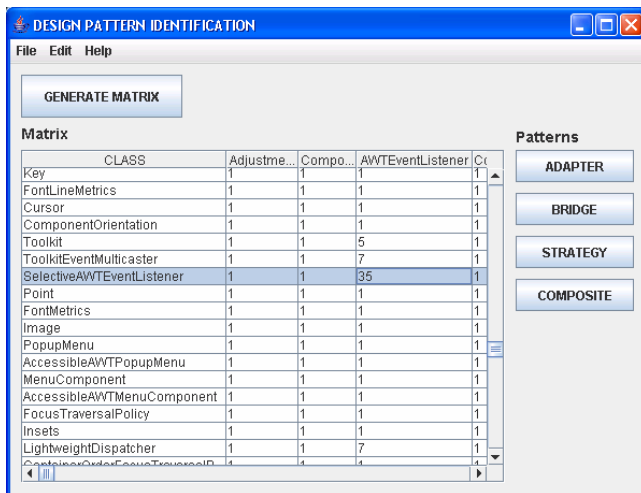


Figure 5 Matrix of Java.awt

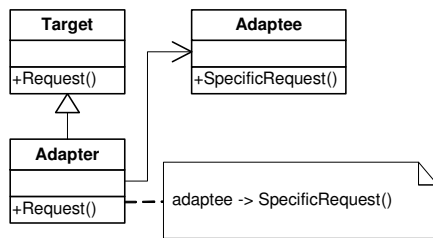


Figure 6 The Adapter pattern

Class	Target	Adapter	Adaptee
Target	1	1	1
Adapter	7	1	5
Adaptee	1	1	1

Table 2 The Adapter pattern matrix

Figure 5 presents the matrix that encodes the relationship information between all pairs of classes. All classes and interfaces are listed both at the top row and the leftmost column of the matrix. Each cell (i,j) represents the value of relationship from class *i* to class *j*. If there exists at least one association relation from class *i* to class *j*, the value of the cell (i,j) is set to 5. Similarly, if class *j* is a generalization of class *i* then the value of cell (i,j) is 7. Currently the aggregation and dependency relationships are not represented in the matrix thus the highest value of a cell can be $5^1 \times 7^1 = 35$.

Since the value of each cell in the matrix is the product of several prime numbers, we can decode the corresponding relationships from this single value of each cell. For instance, the value of the cell (SelectiveAWTEventListener, AWTEventListener) is 35 that equals to 5×7 in Figure 5. Thus, we know the SelectiveAWTEventListener class inherits from AWTEventListener class. There is also an association relationship between them.

4.2 Pattern Discovery

Consider the discovery of the Adapter pattern whose intent is to convert the interface of a class into another interface clients expect. The class diagram of the Adapter pattern is shown in Figure 6.

In our structural analysis, we need to calculate the weight of each class in the Adapter pattern and the matrix describing the relationships between these classes. Based on our approach described in the previous section, the weight of Target is 3 (1 method), the weight of Adapter is 105 (1 method, 1 generalization and 1 association), and the weight of Adaptee is 3 (1 method). The matrix of the Adapter pattern is shown in Table 2.

Once we constructed the weights and matrices of both the Java.awt package and the Adapter pattern, we can find whether the package contains instances of the Adapter pattern following our approach described in the previous section. In particular, we first identify instances of Adapter based on the structural analysis, and then we can eliminate false positives based on the behavioral analysis. There are no ambiguity issues for Adapter hence the third phase is not applied.

The structural analysis becomes simply calculating whether any weights of a group of classes in Java.awt are integral multiples of those of the Adapter pattern as well as whether the corresponding cells of the Java.awt matrix are integral multiples of those in the Adapter matrix. The Adapter pattern matrix does not have to be exact sub-matrix of the Java.awt matrix, matching all corresponding cells.

CL	INSTANCE	TARGET	ADAPTER	ADAPTEE
BASIC	Adapter[0]	AWTEvent	InputMethodEvent	TextHitInfo
FillAd	Adapter[1]	RenderableImage	RenderableImageOp	ParameterBlock
Borde	Adapter[2]	WritableRendered...	BufferedImage	WritableRaster
Comp	Adapter[3]	BufferedImageOp	AffineTransformOp	AffineTransform
Acces	Adapter[3]	RasterOp	AffineTransformOp	AffineTransform
Acces	Adapter[4]	ImageProducer	FilteredImageSource	ImageFilter
Acces	Adapter[5]	PathIterator	ArcIterator	AffineTransform
AWTT	Adapter[6]	PathIterator	CubicIterator	AffineTransform
FlipBU	Adapter[7]	PathIterator	EllipseIterator	AffineTransform
BitBuf	Adapter[8]	PathIterator	GeneralPathIterator	GeneralPath
Single	Adapter[9]	PathIterator	GeneralPathIterator	AffineTransform
Native	Adapter[10]	PathIterator	LinIterator	AffineTransform
Dimen	Adapter[11]	PathIterator	QuadIterator	AffineTransform
Conta	Adapter[12]	PathIterator	RectIterator	AffineTransform
Mouse	Adapter[13]	PathIterator	RoundRectIterator	AffineTransform
DropT	Adapter[14]	ActionListener	DropTargetAutoScroller	Rectangle
Acces	Adapter[15]	BufferStrategy	FlipBufferStrategy	BufferCapabilities
Acces	Adapter[16]	BufferStrategy	BitBufferStrategy	BufferCapabilities
Acces	Adapter[17]	BufferStrategy	SingleBufferStrategy	BufferCapabilities
Acces	Adapter[18]	Component	Container	LightweightDispat...
Acces	Adapter[19]	AWTEventListener	LightweightDispatcher	Container

Figure 7 The Adapter pattern instances discovered

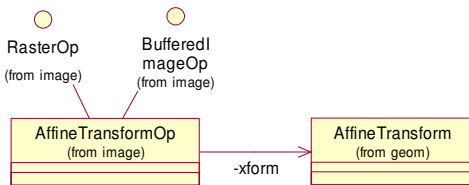


Figure 8 An Adapter pattern instance discovered

In addition to weight and matrix, we consider class type. In summary, we check the following structural characteristics in our structural analysis:

1. The Target class can be of type Abstract or Interface and its weight should be a multiple of 3.
2. The Adapter class should be of type Concrete and its weight should be multiple of 105. It should be the specialization of the Target class and have an outgoing Association relationship with Adaptee.
3. The Adaptee class should be a concrete class and its weight should be multiple of 3.

After the structural analysis, we conduct the behavioral analysis to remove false positive results in two steps. First our tool checks whether there exists any method that plays the role of the Request method, i.e., a method that is common in the Target and Adapter classes. The Request method is an important method that needs to be presented in the Adapter pattern. Without such method, the candidate instance cannot be considered as an instance of the Adapter pattern. If such method cannot be found in a candidate instance, that instance of Adapter is removed. Second our tool finds the occurrence of an invocation of a method in Adaptee from this Request method in the Adapter class. Even if a single such instance is found, the process of verification for that instance is terminated and moved on to the next instance. Otherwise, the candidate instance is removed since it is a false positive.

4.3 Results

Using DP-Miner, we discovered 21 Adapter pattern instances. The time taken for discovering these instances was 2.44 seconds after the matrix was generated. Figure 7 shows the snap shot of our tool displaying results of the Adapter pattern. Figure 8 shows the UML class diagram of the third instance of the Adapter pattern in Figure 7 found by our tool, where the RasterOp, AffinTransformOp and AffineTransform play the roles of the Target, Adapter and Adaptee, respectively. All operations in Interface RasterOp are implemented by AffineTransformOp. Hence they are candidates of the Request method of this instance. The “xform” variable is for the association relation between AffineTransform and AffineTransformOP. The following statement:

```
xform.transform(pts, 0, pts, 0, 4);
```

is defined in the method named “get2DBounds” in AffinTransformOp. Thus, the “get2DBounds” method plays the role of the Request method in the Adapter class, whereas method “transform” in AffineTransform plays the role of the SpecificRequest method in Adaptee.

We found Figure 8 actually contains two instances of the Adapter pattern with different Target classes but sharing the same Adapter and Adaptee classes. Hence another interface BufferedImageOp playing the role of Target is implemented by AffineTransformOp playing the role of Adapter. The following statement:

```
int type = xform.getType();
```

is defined in the “filter” method in AffineTransformOp. Thus, the “filter” method plays the role of Request, whereas the “getType” method plays the role of SpecificRequest.

One of the critical aspects of our discovery is the signature of the methods. The AffineTransformOp class implements two interfaces: BufferTransformOp and RasterOp. Both interfaces define the methods with the same name (called “filter”) but different return types and parameters. The signature of the “filter” method defined in the BufferedImageOp interface is:

```
public final BufferedImage filter(BufferedImage src, BufferedImage dst)
```

The signature of the “filter” method defined in the RasterOp interface is:

```
public final WritableRaster filter(Raster src, WritableRaster dst)
```

Hence we need to take the return types and parameters of methods into consideration during the behavioral analysis. When confirming the second instance, DP-Miner checks if the “filter” method with BufferedImage, instead of WritableRaster, as return type invokes a method defined in AffineTransform. If the invocation does not exist in the proper “filter” method, the second instance is a false positive.

In a similar fashion, we applied our tool and found 3, 76, and 65 instances of the Composite, Strategy, and Bridge patterns, respectively, in the Java.awt package. We have manually checked the source code and document of Java.awt and found all detected patterns have correct pattern characteristics.

5. Related Work

Tsantalis *et al.* [18] applies an existing similarity score algorithm to detect design patterns. Each class relationship, such as generalization, creation, and delegation, is represented by a matrix. Every pattern characteristic is also presented by an $n \times n$ matrix. Thus, many matrices are used in their representations. The detection of design patterns is achieved by calculating the similarity matrices using the existing similarity score algorithm. In contrast to their multi-matrix approach, we encode all design information into a single matrix using prime numbers. Thus, our detection task can be reduced to simple arithmetic rather than complex matrix computations. Besides, our approach uses exact match instead of approximate match.

Balanyi and Ferenc [3] introduce a method to recover design patterns using a XML-based language, called Design Pattern Markup Language (DPML). The structural information of a design pattern, including the participating classes, their attributes and operations, and their relationships, is specified in DPML. The Abstract Semantic Graph (ASG) is constructed to represent the source code. Therefore, finding a design pattern in their approach is to find sub-structures in ASG which match its DPML descriptions. Their approach is different from ours in that they have two representations, ASG and XML for source code and pattern specifications, respectively. Pattern matching in their case is from one kind of representation to another. Our approach uses XMI as the intermediate representation for both source code and patterns without any issues with different representations. We also introduce weight and matrix.

Asencio *et al.* [2] develop a pattern recovery system called Osprey. C/C++ source code is parsed using Imagix. The information is stored in XML file since XML helps validating the output of their Imagix scripts. Their pattern recognizer, developed using a structural specification language based on set theory, can discover patterns from their internal representation of source code. Although XML is used, the main goal is to help validate the output of Imagix scripts, not as intermediate representation for pattern discovery since XML is converted into Osprey internal representation. In contrast, our approach directly uses XMI as the internal representation in our pattern recovery process.

Antoniol *et al.* [1] use the Abstract Object Language (AOL) as the intermediate representation for pattern

discovery. To improve the recovery process, they extract class metrics, such as the numbers of attributes, operations, and relations, from AOL Abstract Syntax Tree (AST). In addition to the class metrics, we introduce weight and matrix that can be calculated from these metrics. In this way, we can integrate several metrics into a single value. Thus, our pattern discovery process is reduced into arithmetic computations.

Gueheneuc *et al.* [10] propose to fingerprint design pattern from source code using machine learning techniques. Size, filiations, cohesion, and coupling are their metrics. Classes that do not match these metrics are eliminated from the candidate set so as to reduce the search space. Our approach also uses metrics similar to size and filiations, but not cohesion degree and coupling degree. Instead of applying machine learning techniques, we reduce the searching problem into simple computation problem.

A top-down-bottom-up recovery approach based on FUJABA [21] platform and AST internal representation is presented in [14][15]. Both top-down and bottom-up search on AST work independently to check each other's search results to speed up the search speed. In an extension of FUJABA work, Wendehals [19] suggests using runtime data to check the behavioral aspect of patterns dynamically. The runtime information is gathered during program execution using a debugging tool. We use different intermediate representation and search process. Our search process consists different phases, each of which narrows down the search scope based on the input candidate set. Our behavior analysis does not need to manually collect runtime information. It is realized automatically.

Heuzeroth *et al.* [11] emphasize the importance of both structure and behavior of design pattern because neither structure nor behavior analysis alone is sufficient. AST is used as an intermediate representation of source code. Structure specification contains Prolog predicates, whereas dynamic constraints are defined as Prolog procedure based on temporal logic of actions (TLA) [12]. Our approach, on the other hand, uses different intermediate representation and considers not only structural and behavioral but also semantic aspects.

Keller *et al.* [13] use the SPOOL, an environment for reverse engineering of design components based on structural information extracted and represented using both UML metamodel and CDIF Intermediate Source Model. Their pattern identification is through query. The System for Pattern Query and Recognition (SPQR) is proposed by Smith and Stott in [17]. They do not hard-code pattern definitions, but infer pattern variants dynamically during code analysis with the help of a theorem prover. Both approaches use different techniques from ours.

6. Conclusions

In this paper, we presented a novel approach to reverse engineering design patterns from source code. We introduce weight and matrix that facilitate the pattern matching process as arithmetic computations. Different from other pattern mining approaches, our intermediate representation is based on XMI standard, which allows our techniques and tools to seamlessly integrate with other tools following such industry standard, such as IBM Rational Rose [24] and ArgoUML [20]. Our approach includes different analysis phases: structural, behavioral, and semantic analyses to reduce false positives. We also conduct a real-world case study to evaluate our approach and tool.

Our future work includes applying data-mining techniques on matching pattern weight and matrix to those of the systems and applying sparse matrix algorithms for storage and computation. Our approach currently can detect four patterns. Other patterns can be detected similarly, which will be included in our tool. In addition, we plan to integrate this work with our other work on design pattern compositions [5][6] and evolutions [7] to achieve round-trip pattern engineering. Furthermore, the detected design pattern instances can be visualized using our VisDP tool [8] to display the identified patterns in the context of the whole class diagram of the system. We plan to include our detection results as stereotypes in the input XMI file of VisDP because DP-Miner provides all necessary information.

References

- [1] G. Antoniol, R. Fiutem, and L. Cristoforetti, "Design pattern recovery in object-oriented software." *Proceedings of the 6th IEEE International Workshop on Program Understanding (IWPC)*, pp 153-160, 1998.
- [2] A. Asencio, S. Cardman, D. Harris, and E. Laderman, "Relating expectations to automatically recovered design patterns." *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*, 2002.
- [3] Z. Balanyi and R. Ferenc, "Mining design patterns from C++ source code." *Proceedings of the 19th IEEE International Conference on Software Maintenance (ICSM)*, pp. 305-314, September, 2003.
- [4] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.
- [5] J. Dong, P. Alencar, and D. Cowan, "Ensuring Structure and Behavior Correctness in Design Composition," *Proceedings of the 7th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS)*, pp279-287, Edinburgh UK, 2000.
- [6] J. Dong, P. Alencar, and D. Cowan, *Automating the Analysis of Design Component Contracts, Software - Practice and Experience (SPE)*, Wiley, Volume 36, Issue 1, pages 27-71, January 2006.
- [7] J. Dong, S. Yang and K. Zhang, "A Model Transformation Approach for Design Pattern Evolutions," *Proceedings of the Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS)*, pp 80-89, Germany, March 2006.
- [8] J. Dong, S. Yang and K. Zhang, "VisDP: A Web Service for Visualizing Design Patterns on Demand," *Proceedings of the IEEE International Conference on Information Technology Coding and Computing (ITCC)*, pp385-391, USA, April 2005.
- [9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [10] Y. Gueheneuc, H. Sahraoui, and F. Zaidi, "Fingerprinting design patterns." *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE)*, 2004.
- [11] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, "Automatic design pattern detection." *Proceedings of the 11th International Workshop on Program Comprehension (IWPC)*, pp 94-103, 2003.
- [12] D. Heuzeroth, S. Mandel, and W. Lowe, "Generating design pattern detectors from pattern specifications." *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE)*, 2003.
- [13] R. K. Keller, R. Schauer, S. Robitaille, and P. Page, "Pattern-based reverse-engineering of design components." *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pp 226-235. 1999.
- [14] J. Niere, W. Schafer, J. P. Wadsack, L. Wendehals, and J. Welsh, "Towards pattern-based design recovery." In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pp 338-348, 2002.
- [15] J. Niere, J. P. Wadsack, L. Wendehals, "Handling large search space in pattern-based reverse engineering." *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC)*, pp. 274-279, 2003.
- [16] A. Rountev, O. Volgin, M. Reddoch, *Static Control-Flow Analysis for Reverse Engineering of UML Sequence Diagrams*, 6th ACM Workshop on Program Analysis for Software Tools and Engineering, Lisbon, Portugal, 2005.
- [17] J. M. Smith and D. Stotts, "SPQR: Flexible automated design pattern extraction from source code." In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE)*, 2003.
- [18] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design Pattern Detection Using Similarity Scoring." *IEEE transaction on software engineering*, Vol. 32, No. 11, November 2006.
- [19] L. Wendehals, "Improving design pattern instance recognition by dynamic analysis." *Proceedings of the ICSE workshop on Dynamic Analysis (WODA)*, 2003.
- [20] ArgoUML, <http://argouml.tigris.org/>
- [21] Fujaba User Documentation <http://wwwcs.uni-paderborn.de/cs/fujaba/documents/user/manuals/FujabaDoc.pdf>
- [22] Java.awt resource information, September 2006, <http://java.sun.com/j2se/1.5.0/docs/guide/awt/index.html>.
- [23] Model Driven Architecture. <http://www.omg.org/mda/>
- [24] Rational Rose website. <http://www.rational.com/>
- [25] W3C, Extensible Markup Language (XML), <http://www.w3.org/>