

DYNAMICALLY RECOMMENDING DESIGN PATTERNS

by

SARAH RICHARDSON

Advisor

DANIEL PLANTE

A senior research paper submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science
in the Department of Mathematics and Computer Science
in the College of Arts and Science
at Stetson University
DeLand, Florida

Spring Term
2011

Contents

0.1	Abstract	1
1	Introduction	2
2	Background Information	3
2.1	Design Patterns	3
2.2	Intermediate Code Representation	4
3	Related Work	4
3.1	Design Pattern Detection	4
3.2	Anti Pattern and Code Smell Detection	7
3.3	Recommendations Systems for Software Engineering	8
4	Methods	8
4.1	Structural Matching	10
4.1.1	"K-Steps" shrinking	11
4.1.2	Permute-and-Match	12
4.1.3	Tree Matching	14
4.2	Behavioral Matching	16
4.3	Pattern Definition Format	17
4.4	Dynamic Recommendations	19
4.5	Definitions of Anti Design Patterns	19
4.5.1	Singleton	20
4.5.2	Abstract Factory	22
4.5.3	Command	24
5	Conclusions and Future Work	26

List of Figures

1	An example Abstract Syntax Tree (AST) graph	5
2	Example usage of Java's Abstract Syntax Tree	5
3	Overview of our recommendation tool	9
4	Three classes and the relationships between them, shown using UML	10
5	Matrices to represent the relationships between classes in Figure 4	10
6	The matrix for Figure 4 using prime numbers	11
7	An example pattern	13
8	An example system to which the pattern in Figure 7 is matched	13
9	The configuration for files defining anti design patterns	18
10	Example of the Singleton design pattern	20
11	Whether or not the object exists is checked before instantiating the single class	21
12	UML for Abstract Factory design pattern [7]	23
13	Poor design choice instead of using a factory method	23
14	UML for the Command design pattern [7]	24
15	Sample code for the Anti Command Pattern	25
16	Behavioral pattern definition for the Command anti design pattern	25

0.1 Abstract

Recommendation Systems for Software Engineering are created for a variety of purposes, such as recommending sample code or to call attention to bad coding practices (code smells). We have created a system to recommend the use of design patterns. Many programmers have knowledge of design patterns but when they have limited experience implementing them or are rushed to complete a product, the use of a particular design pattern may not occur to them. We have developed a tool to dynamically search for signs that a programmer would benefit by using a particular design pattern, and make the appropriate recommendations to the programmer as he or she is developing code.

1 Introduction

In today's corporate world of deadlines and managers who want to see fast results, many programmers find themselves rushed to get something done instead of writing quality code. But laying a solid foundation and making purposeful decisions is essential to producing a solid, reusable, and manageable system.

Code reuse is a common practice to improve the development process by providing well tested elements which the programmer can incorporate into his or her system. Similarly, design patterns encourage the reuse of object oriented ideas [7]. They provide design solutions to common problems, but must be implemented specifically for each project.

Anti patterns attempt to prevent common mistakes which can degrade the quality of an object oriented system [3]. Each pattern defines a bad way of structuring code, and suggests methods of refactoring. Anti patterns are not directly related to design patterns; in other words, there are not necessarily refactorings for the anti patterns that turn them into design patterns. The suggestions simply create more object oriented code.

Design pattern and anti pattern discovery assists programmers in the software development process, and is currently a popular area of research. By identifying design patterns in a developed system, future programmers are encouraged to maintain those patterns. Discovering anti patterns can alert programmers to problem areas so that they can be fixed.

We have developed a tool which, instead of just finding instances of either type of pattern, recommends the use of design patterns based on an unfinished project. We determine that a programmer is trying to solve a common problem in a way that could be improved using a design pattern, and dynamically make recommendations. We have created a framework for detection and a format of storing requirements for each of these anti design patterns. The requirements will then be processed and our tool, as an Eclipse plugin, will search for instances within the current project. The plugin will also determine what to make recommendations

about and when to make them. Although the current tool only recommends a few patterns, it is designed so that the set can be easily expanded when new “anti design patterns” are developed.

2 Background Information

2.1 Design Patterns

Design patterns were established by Gamma *et al.* [7] to offer solutions to common software design problems. These solutions are abstract and defined intuitively. Reusing these ideas (rather than concrete code) may help developers to avoid common mistakes and more quickly find solutions. There are three categories of design patterns: structural, behavioral, and creational.

Structural patterns focus on the composition of objects and relationships between them. The Adapter and Facade patterns create a new interface to simplify one structure or a set of structures, respectively. Composite and Bridge define patterns for more flexible object relationships and inheritance.

Behavioral patterns involve the responsibilities of and communication between different objects. However, they do not focus on run-time control flow but rather how objects are connected. These patterns help to reduce coupling (Mediator), define algorithms (Template Method, Strategy), manage dependency (Observer), and handle requests (Chain of Responsibility, Command), among other behavioral goals.

Creational patterns are similar to behavioral patterns but are concerned specifically with how and when objects are instantiated. Instead of an object being instantiated by the `new` operator whenever it is needed, classes are created specifically for the purpose of creating new instances of that object according to a pattern. Often, more than one creational pattern

may be used to solve the same problem. There are subtle differences between some of the creational patterns, requiring some intuitive knowledge of the code's purpose to determine the best fit. This makes it difficult to know from a programmer's code which creational pattern would best suit their needs.

2.2 Intermediate Code Representation

Rather than looking at source code directly, most pattern search algorithms use some form of intermediate code representation. The Abstract Syntax Tree (AST) is a directed acyclic graph, where each node represents a programming element and its children are the elements which are part of it [17] (See Figure 1). The Abstract Semantic Graph (ASG) is a higher level representation where nodes represent source code entities and edges represent relationships between them. It is similar to the AST but, for example, instead of a node with the name of the referenced object, the ASG contains an edge from the first node to the referenced node [4]. A matrix may be used to provide a simplified representation of the relationships between classes, as will be described later.

We will use the Eclipse JDT's `ASTParser` to create an Abstract Syntax Tree [17]. Each Java file is parsed and each element traversed by the `ASTVisitor`. To catch elements we are interested in, we must extend `ASTVisitor` and override the `visit()` methods for each type, and store the node information in a data structure, as shown in Figure 2.

3 Related Work

3.1 Design Pattern Detection

Brown [2] made the first attempt at automatically detecting design patterns. Since then, research has focused on both defining design patterns in a manner that is programmat-

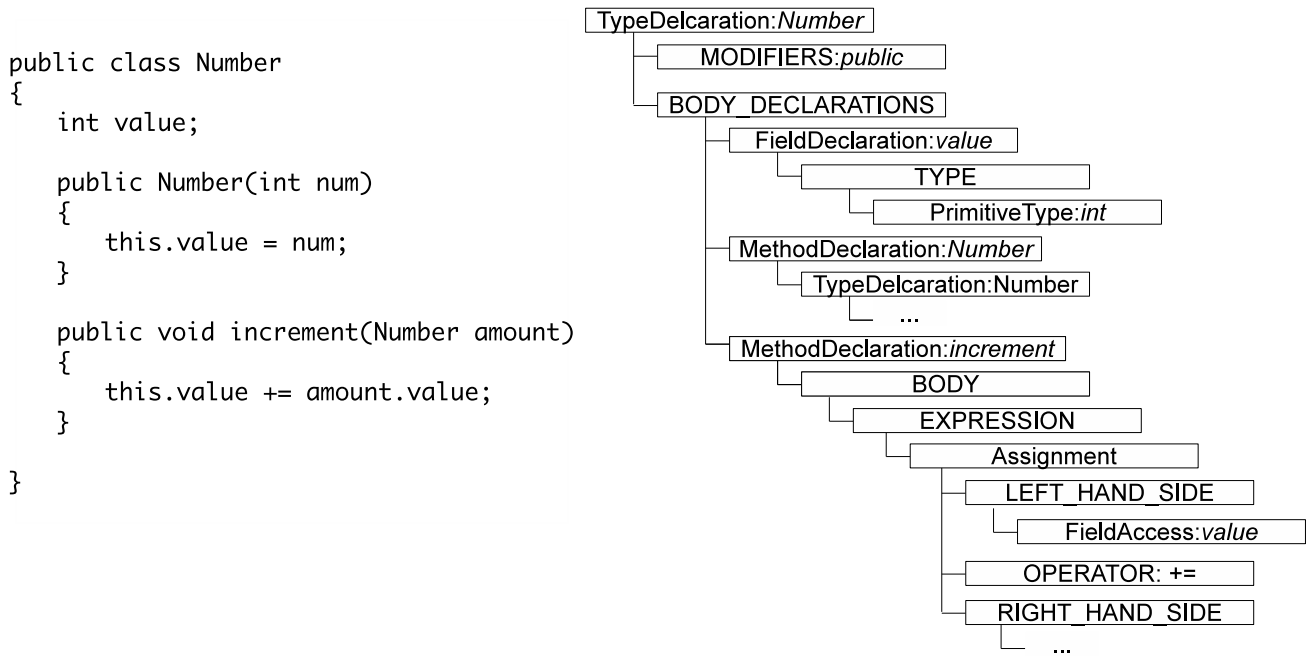


Figure 1: An example Abstract Syntax Tree (AST) graph

```

public class FieldVisitor extends ASTVisitor
{
    private List<FieldDeclaration> matches = new ArrayList<FieldDeclaration>();

    @Override
    public boolean visit(FieldDeclaration node)
    {
        matches.add(node);
        return super.visit(node);
    }

    public List<FieldDeclaration> getMatches()
    {
        return matches;
    }
}

```

Figure 2: Example usage of Java's Abstract Syntax Tree

ically useful and identifying matches in source code. Some methods only look for structural characteristics, such as the relationships between classes, while others look for more specific behavior and how the classes actually interact.

Guéhéneuc *et al.* [8] identified classes that fit certain design pattern roles through the use of metrics, which were trained with machine learning techniques on a repository of examples. If a match was found, its related classes were checked to match the entire pattern, giving an approximate result.

Blewitt *et al.* [1] developed the SPLINE language to define design patterns in Java. Instead of searching for design patterns, they verified that a certain design pattern was still intact. The SPINE language focused on behavioral aspects, including the relationships between classes.

Later research developed the use of matrices and graph theory to search for structural aspects of design patterns. Tsantalis *et al.* [28] generated matrices to represent the relationships between classes. Each type of relationship (generalization, aggregation, etc.) is represented by a unique matrix. Using a similarity score, they find approximate matches within source code. Dong *et al.* [13] combined all the relationships into one matrix, as explained in Section 4.1. In addition to relationships, each class had an associated weight for the properties of the class. Dong *et al.* [14] further explained the method of searching for a design pattern representation matrix in a source code matrix using normalized cross correlation. This method used exact matching, but incorporated variations on the patterns.

Ng *et al.* [24] analyzed behavior through dynamic analysis. Although static analysis can give a more complete model of the behavior, they claimed that dynamic analysis could better determine the actual types of object references, among other advantages. Choosing different execution scenarios based on analysis of the documentation, they generated a sequence diagram to look for behavioral characteristics and match design patterns.

3.2 Anti Pattern and Code Smell Detection

Anti patterns were originally defined by Brown *et al.* [3] as recurring solutions to common problems which have negative consequences. Similarly, Fowler [6] informally defines code smells, which are also examples of bad programming but with less complexity. Searching for anti patterns (or any type of code “smells”) is very useful to programmers who do not have the time or resources to analyze large systems.

Most anti pattern detection methods use metrics. Once the metrics are defined for a specific pattern, objects in a system are tested and potential matches are returned. Marinescu [18][19] created detection strategies for finding anti patterns. However, a strategy was created separately for each pattern, making it difficult to expand to new patterns. Munro [20] formally defined rule cards for nine design flaws and tested each one. Salehie *et al.* [26] found “hot spots,” in addition to known anti patterns, when certain metric values were outside the range found in good code. These areas could either match to specific design smells or be indicators of a problem that was not formally defined. Moha *et al.* [23] also developed a system of rule cards and tested it on four anti patterns. Vaucher *et al.* [29] considered the possibility that the creation of a god class (a class which knows or does too much) was intentional. Therefore, they looked at the history of the code to see whether the god class was created by design or by adding things over time without refactoring.

Exact matching using metrics does not allow for a high level of flexibility and leaves the software analyst with a list of possible design anti-patterns and no method of prioritizing which to consider first. Khomh *et al.* [5] investigated the use of Bayesian belief networks for anti-pattern detection. This produced a probability that a given class or set of classes followed an anti-pattern, which was a more realistic way to detect something which could not be exactly defined. Similarly, Oliveto *et al.* [25] used B-splines to detect anti-patterns and gave results in terms of probabilities.

These methods focus on metrics of individual classes and do not take into account the relationships between classes or specific behavior. This is logical for the anti patterns described by Brown, as the majority of them reflect a failure to follow object oriented standards, and therefore have minimal relationships to consider.

3.3 Recommendations Systems for Software Engineering

Recommendation systems for software engineering aim to assist programmers in the software development process by making recommendations based on written code and/or dynamic analysis [22]. They can make recommendations dynamically or by the request of the programmer. These systems offer assistance on a variety of topics, from what to consider changing next [30] to examples of and suggestions for what call sequence to make [11][27]. Guéhéneuc *et al.* [10] created a design pattern recommender based on words chosen to describe the programmer's needs.

4 Methods

Any code pattern can be defined in terms of three characteristics: structure, behavior, and semantics [15]. The structure refers to the types of classes and relationships between them. A UML diagram, for example, contains mostly structural information about a system, specifying the inheritance, association, and other relationships between classes. These relationships are found by looking for certain types of references; for example, a field declaration is an aggregation, an “extends” in the class signature is a generalization, and any other reference is considered to be an association.

The behavioral characteristics used to define a pattern are more complex and often abstract. They define how objects are created, methods are invoked, and information is shared. They describe what a piece of code actually does, not just the type of relationships

or objects. We match behavioral characteristics by defining specific ways they could be implemented and searched for similar code structures.

The actual choice of names for classes, methods, and other parts of a system make up the semantic data. Semantic data can be used to look for repetition of names or the use of actual words and their synonyms to name or describe parts of the system. Our system does not use semantic data in searching for matches.

Figure 3 summarizes the steps in making recommendations. In order to limit recommendations to code which the programmer is currently working on, we perform our matching algorithm when a class has been modified, using only the modified class and the classes "close" to it. We first look for a structural match to a given pattern. If we find a match, we then test the behavioral characteristics for those classes. If all the behavioral requirements are met, we make a recommendation to the programmer.

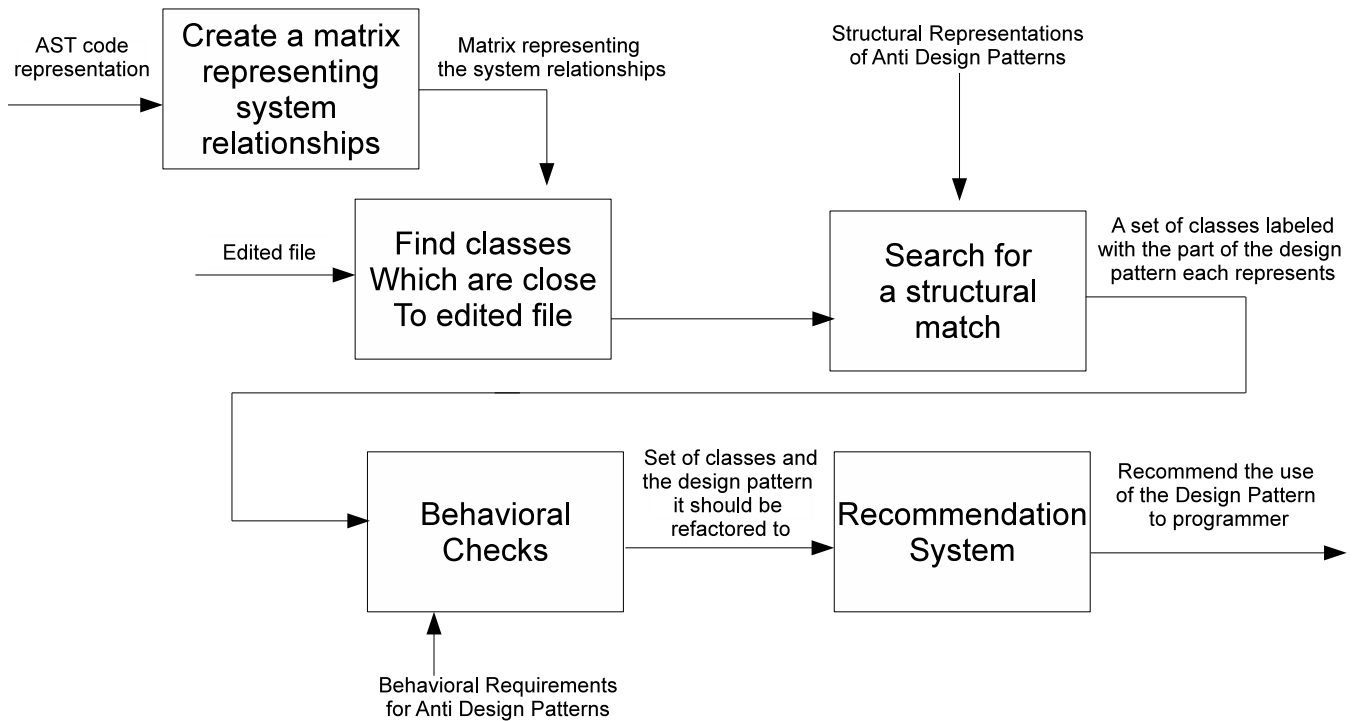


Figure 3: Overview of our recommendation tool

4.1 Structural Matching

Each pattern uses a set of classes and their relationships to define the structural requirements for a match to that pattern. Both the pattern and the system's relationships can be defined using an $n \times n$ matrix, where n is the number of classes. For example, the set of matrices in Figure 5 represents the generalizations and associations between the classes in Figure 4. However, searching for matches within the system is complicated by having multiple matrices to check. A simpler method would be to combine all the information into one matrix and then search for matches.

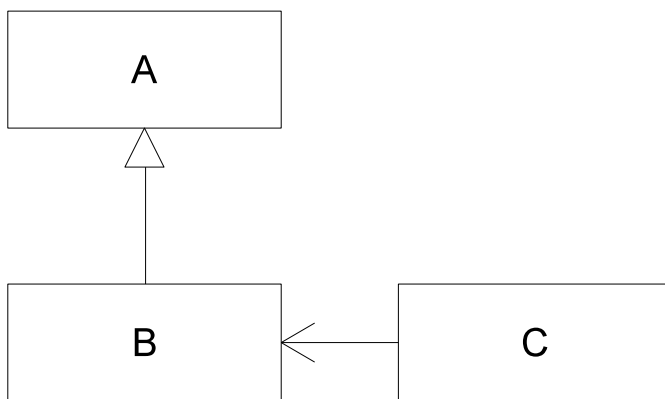


Figure 4: Three classes and the relationships between them, shown using UML

Generalizations	Associations
$ \begin{matrix} & A & B & C \\ A & \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \\ B & \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \\ C & \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \end{matrix} $	$ \begin{matrix} & A & B & C \\ A & \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \\ B & \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \\ C & \begin{pmatrix} 0 & 1 & 0 \end{pmatrix} \end{matrix} $

Figure 5: Matrices to represent the relationships between classes in Figure 4

As Dong *et al.* [13] found, we can take advantage of the fact that the product of

prime numbers is a unique composite number; in other words, it can be broken down into one specific list of prime numbers. Therefore, we can choose a prime number to represent each relationship, and each i, j location in the matrix is the product of the values representing all relationships from i to j . (And likewise each j, i position represents the relationships to i from j). The values used to represent relationships are shown in Table 1. Using these prime numbers, the matrix for the classes in Figure 4 is shown in Figure 6.

Relationship	Prime Number Value
Association	2
Generalization	3
Aggregation	5

Table 1: Prime number values for different relationships

$$\begin{array}{c}
 A \quad B \quad C \\
 A \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \\
 B \begin{pmatrix} 3 & 0 & 0 \end{pmatrix} \\
 C \begin{pmatrix} 0 & 2 & 0 \end{pmatrix}
 \end{array}$$

Figure 6: The matrix for Figure 4 using prime numbers

The process of actually searching for an instance of the pattern matrix within the (larger) system matrix is a relative of the subgraph isomorphism matching problem in graph theory. We explored both a brute force algorithm and a second algorithm which is similar to a breadth first search. Before using either of these techniques, we shrunk the size of the search space by looking only at classes "close" to the modified class.

4.1.1 "K-Steps" shrinking

Many times a programmer will only be working on a small number of classes within a large system. He will not be interested in recommended changes to a group of classes that

he is not responsible for. Additionally, to limit the complexity of matching, we want to look only at the class which was just modified and any classes which are "close" to it.

When the relationships between all classes in the system are found, we can think of the entire system as a graph where each class is a node and each relationship an edge. We define the distance between two classes as the distance in the graph, ignoring the direction of the edges. In order to limit matching, we start from the edited class and find all classes within k steps, where k is the max distance between any two classes in the pattern we want to match.

4.1.2 Permute-and-Match

We first consider a brute force method which checks for every permutation of the nodes in the graph and whether or not it matches the pattern matrix. Since the system is usually larger than the pattern, we must first find all the subsets of the graph before permuting their order to perform matching. Also recall that multiple types of relationships can be stored in this single graph by using different values for different types of relationships. So when matching the relationship values, the system value must be divisible by the pattern value in order to be considered a match. For example, if the pattern requires a generalization relationship (given the value 3) and the system relationship we are matching has a value 6, this is still considered a match because $6 = 2 * 3$ and therefore it has both a generalization and an association relationship. Because of this, a sub matrix which has extra relationships is still considered a match.

Figure 7 shows a graph and representative matrix for a pattern, and likewise Figure 8 represents the system we are matching to. (A dash for cell i, j indicates that there is no relationship from i to j . Internally this is stored as the value one). This process will find that the relationships between the set of vertices $\{U, W, V, Y\}$, in that order, will create a match

to the matrix in Figure 7.

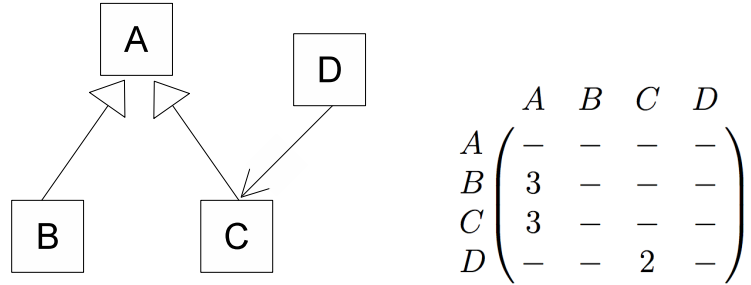


Figure 7: An example pattern

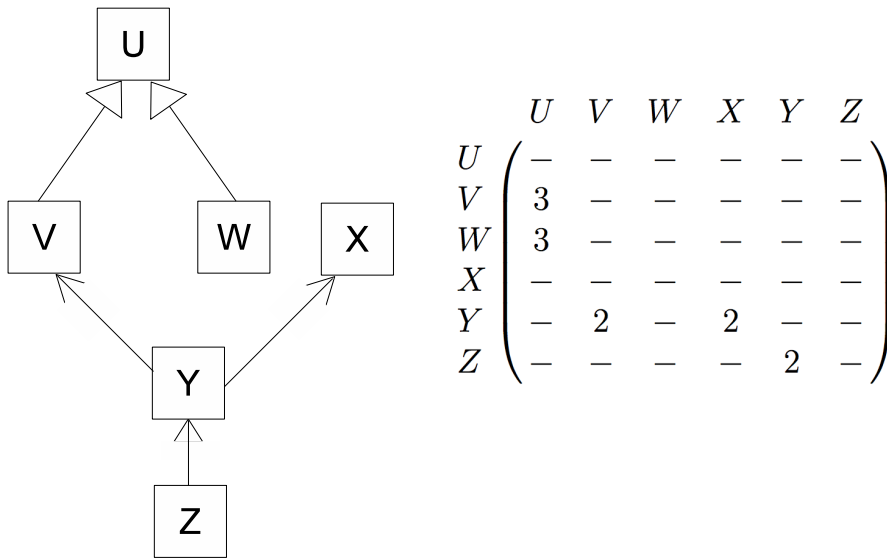


Figure 8: An example system to which the pattern in Figure 7 is matched

Finding every permutation and matching it to the pattern is very inefficient. It requires the value of $P(m, n)$ where m is the number of classes in the system and n is the number of nodes in the pattern. If our basic operation is the array comparison during matching, then in the worst case this algorithm will have a complexity of

$$\frac{m!}{(m-n)!} * n^2$$

which is of order $O(m! * n^2)$. With the help of the K-Steps algorithm (section 4.1.1), the system matrices are relatively small, but even for a system matrix cut down to 7 nodes and a pattern of size four, the worst case will require over 13,000 comparisons.

4.1.3 Tree Matching

Consider a situation where the pattern has a class with two classes inheriting from it, while the system has a matching class but with three classes inheriting from it. The permute-and-match process will find a match for each permutation of two of those three classes. Firstly, this results in multiple matches which are essentially all the same match. Secondly, we may actually want to know about all three of these inheriting classes, even if the pattern only requires two. In order to recognize which classes are part of a set defined in the pattern and pass them on for behavioral matching, we developed a new algorithm to perform matching in a way similar to a breadth first search.

We choose a node in the pattern tree and try to match it to every one of the nodes in the system tree as described in Algorithm 1. For example, in Figures 7 and 8, we would try to find a match for pattern node A by first comparing it to the system node U. The algorithm looks both at the relationships from A and to A, checking if there are a sufficient number of like relationships to and from U. If it finds a matching relationship, for example that V inherits from U, it recursively checks V for the required relationships before determining that U actually matches the pattern.

Algorithm 1: Algorithm to recursively check if there is a match between a pattern element and a particular class.

Data: *patternIdxCheck*, *uncheckedPatternIndices*, *systemIdxMatch*,
unmatchedSystemIndices, *currentMatch*

Result: A boolean representing whether there was a match found. *currentMatch* will be updated

```
1 for pattIndex in uncheckedPatternIndices do
2   if there is a relationship between patternIdxCheck and pattIndex then
3     numMatchesFound = 0
4     for sysIndex in unmatchedSystemIndices do
5       if there is a matching relationship between systemIdxMatch and sysIndex
6         then
7           aMatch = a node representing this match
8           if match(pattIndex, uncheckedPatternIndices.remove(pattIndex),
9             sysIndex, unmatchedSystemIndices.remove(sysIndex)) then
10            currentMatch.addChild(aMatch)
11            numMatchesFound++
12          end
13        end
14      end
15      if numMatchesFound >= required number of matches then
16        | continue
17      end
18      else
19        | return false
20      end
21    end
22  end
23  return true
24 end
```

4.2 Behavioral Matching

Structural information can define a large part of any pattern. However, in order to match an element of a pattern, it is often necessary to define not only the categorized relationships to other elements but also how they relate. Therefore, once a structural match is found, behavioral matching is performed on each of those elements.

Each behavioral requirement will be stored as a tree with a root node to specify the pattern element that should contain it, element nodes which match to certain types (i.e. switch statements, object instantiations, etc.), and references to other pattern elements. For example, if the pattern element named `Client` should contain a switch statement with references to all of the `Product` elements, the behavioral definition stored in the file would be

```
ROOT:Client {
  ELEM:SwitchStatement {
    REF:Product
  }
}
```

In the above case, all classes which match to `Product` must be referenced within the `SwitchStatement`. But in some cases, we may want to match a more specific behavioral structure for each `Product` match. (For example, for each `Product` match we want to find an if statement containing an instantiation of that class). Using the `for` behavioral node, one can specify a behavioral structure, and any references within it will be matched to one class at a time.

Our algorithm steps through each root node defined in the pattern and pulls out the class that is paired with it during structural matching. It then searches for the first element using an `ASTVisitor` object which stores only that type. (In the above example, the first element is a `SwitchStatement`). Each matched `ASTNode` is then revisited, using a

new `ASTVisitor`, and the process continues until the entire tree has been matched. Before beginning this process, each behavior node is passed a list of all pairings so that the references can be set to actual class names in the current system. Therefore, when a `REF` node is matched, it searches for references to the appropriate class names. This structure allows for considerable flexibility in defining elements to search for. By defining requirements in a tree like fashion, references can be searched for within constructors and method calls, or more generally within `if` and `switch` statements.

Finally, we must consider that behavioral information need not always be defined by specific syntax. We want to allow a pattern to be defined in terms of different options (such as a `switch` statement or a series of `if-then-else` statements). Therefore, after defining several different behavioral structures, the pattern also contains a statement declaring what is required and what is optional. For example, a pattern with three different behavioral structures could state that element three is required, along with either one or two:

3 AND (1 OR 2)

This logic sentence is converted into postfix using Dijkstra's Shunting-yard algorithm [12] and then evaluated with each index replaced by the boolean result of that behavioral match.

4.3 Pattern Definition Format

Each of the patterns created is defined in its own file and stored with the other patterns. As shown in Figure 9, a series of parameters are set, followed by labels for each element. If a particular class can be matched to multiple elements (i.e., we are looking for all the inheriting children of a particular class), the minimum number of required elements is listed with the class. The next part of the file specifies the matrix representing the relationships between elements. Following this is a list of behavioral descriptions, made into tree structures and

indexed in order starting at zero, and a logic phrase to specify which are required (the & symbol is used for AND and | for OR). Finally, each file contains a paragraph to display to the user about the recommended Design Pattern. Note that in Figure 9, ElementLabel1 should match to one element, ElementLabel2 to at least one element, and ElementLabel3 to two or more. Also, when ElementLabel3 is listed as a reference in the behavioral requirements, this means that all matches to ElementLabel3 should be referenced.

```

Name of Pattern
NUM_CLASSES = <number of classes in pattern>
NUM_BEHAVIOR = <number of behavioral elements>
MAX_STEPS = <max number of steps between any two classes>
ElementLabel1
ElementLabel2(1)
ElementLabel3(2)
...
ElementLabelN
1 1 2 ... 1
3 1 1 ... 1
1 1 1 ... 1
  :   . . .  :
1 1 1 ... 1
ROOT:ElementLabel1 {
    ELEM:<ASTNode type> {
        REF:ElementLabel3
    }
}
ROOT:ElementLabel1 {
    ...
}
ROOT:ElementLabel2 {
    ...
}
...
(0|1)&2
Paragraph explaining design pattern that will be recommended to the user.

```

Figure 9: The configuration for files defining anti design patterns

4.4 Dynamic Recommendations

Happel *et al.* [16] discuss the issues of what and when to recommend in a recommendation system for software engineering. For a programmer working on a large system, it would be useless to recommend the use of a design pattern in a package they are not working on and are not responsible for. Therefore, in addition to being able to detect these patterns, it is important to consider where to search for them and how often to make recommendations. A programmer who is constantly interrupted with suggestions is likely to begin ignoring them or turn off the tool. Therefore, once a recommendation has been ignored, a dynamic recommendation system must remember and not revisit it.

Every time a user makes a modification to a file, our tool checks to see if there has been a change in relationships between that class and the others in the system. Because our tool only looks for matches when there is a change, the user will only receive recommendations about the part of code which he is currently editing. Once a match is discovered, the tool presents it to the user automatically and without requiring the user to make any requests, as suggested by Murphy-Hill *et al.* [21]. The plugin has its own window, where current recommendations will be displayed. When the user sees that there is a recommendation, he can click on it to obtain more information, which is in the form of a popup describing the recommended pattern and pointing out which files are involved.

4.5 Definitions of Anti Design Patterns

This section contains the anti design pattern definitions for three common design patterns. We explain the design pattern and also the structural and behavioral indicators that a programmer should use it. Our tool currently understands and looks for these patterns.

4.5.1 Singleton

The Singleton design pattern is used when only one instance of an object is to be created during a particular execution. The singleton object keeps track of one instance of itself. Instead of instantiating it with the *new* keyword, a `getInstance()` method is called each time it is needed by another object. Figure 10 shows an example of a correct implementation of the Singleton design pattern.

```
public class SingleUser {
    public void foo() {
        Single.getInstance.setVal(1);
    }
    public void bar() {
        Single.getInstance.setVal(2);
    }
}

public class Single {
    private static Single instance = null;
    private int value;

    private Single(int v) {
        this.value = v;
    }

    public void setVal(int v) {
        this.value = v;
    }

    public static Single getInstance() {
        if(this.instance == null) {
            this.instance = new Single(0);
        }
        return this.instance;
    }
}
```

Figure 10: Example of the Singleton design pattern

A programmer who is not following this design pattern may do other things to accomplish the same goal, which are the rules we are looking for in order to recommend the use of the Singleton design pattern. To discuss these "bad" solutions, we will refer to the "single"

class (the class which should have only one instantiation) and the using class or classes.

Many times a Singleton pattern is needed when the single class may or may not be instantiated, depending on whether certain code fragments are reached. In the anti pattern, in order to guarantee that it is only instantiated once, the programmer may set up a conditional check before instantiating the single class with the `new` operator, as shown in Figure 11.

```
public class SingleUser {
    private Single obj;

    public void foo(){
        if(!obj)
            obj = new Single();
        ...
    }

    public void bar() {
        if(!obj)
            obj = new Single();
        ...
    }
    ...
}
```

Figure 11: Whether or not the object exists is checked before instantiating the single class

The structural tests are very simple; if two or more objects have an aggregation with another object (the singleton), it fits the structural requirements. The behavioral check involves looking for an existence of a conditional containing an instantiation which refers to the single class. This is specific but flexible enough to allow for either checking if the object equals `null` or some boolean value set up by the programmer. The structural and behavioral definition is shown below.

Singleton

```

NUMCLASSES = 2
NUMBEHAVIOR = 1
MAXSTEPS = 2
User(2)
1,1
5,1
ROOT:User {
  ELEM:IfStatement {
    ELEM:ClassInstanceCreation {
      REF:Single
    }
  }
}
}

```

4.5.2 Abstract Factory

The Abstract Factory design pattern assists in the creation of families of related objects. Rather than each class which uses these objects knowing how to instantiate them, the factory methods handle instantiation using the `new` keyword. A common use is for a set of platform-specific products, as shown in Figure 12. Consider a situation where products are various windowing items and the platforms are different operating systems. The client only needs to know what product it needs (such as a menu or button), and a factory method will produce the appropriate platform-specific product.

A programmer who should be using an abstract factory is likely to follow the same class structure but without the factory classes. Instead, he or she will reproduce the creational logic every time it is needed. As shown in Figure 13, one way to do this is to store a variable with the platform and use a switch statement in which the set of objects is created. The behavioral checks for this situation look for a switch statement with references to all of the products, or a behaviorally equivalent string of if-then-else statements. Instead of creating the Button and Menu type directly, there should be a WindowsFactory and MacFactory (which both inherit from Factory) to handle the instantiations.

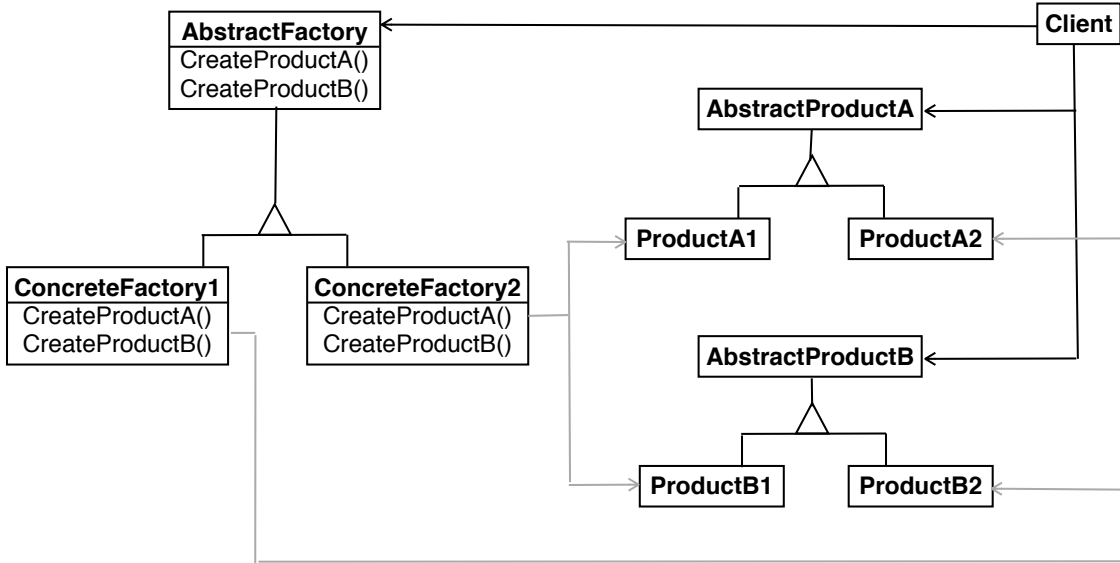


Figure 12: UML for Abstract Factory design pattern [7]

```

void displayWindow()
{
    Menu menu;
    Button button;
    switch(this.platform) {
        case MAC:
            button = new MacButton();
            window = new MacMenu();
            break;
        case WINDOWS:
            button = new WindowsButton();
            window = new WindowsMenu();
            break;
    }
    menu.draw();
    button.draw();
}
  
```

Figure 13: Poor design choice instead of using a factory method

Behavioral matching for this pattern requires either the switch statement or the if statement. Therefore, following the two behavioral definitions, the pattern file will also

contain the line (0 | 1) to indicate that either the first or the second requirement must find a true match.

4.5.3 Command

The Command design pattern encapsulates a request as an object, allowing the client to execute methods specific to actions without actually knowing what occurred. This is a very common pattern used in designing GUI toolkits in which actions are received for various buttons, menus, and other forms of interaction. The controller does not need to know which button was pressed if the button is associated with its own execution method.

Figure 14 shows the structure of the Command design pattern. In our GUI example, each GUI item would have an associated `ConcreteCommand` object, which would be passed to the `Client` when an event occurs.

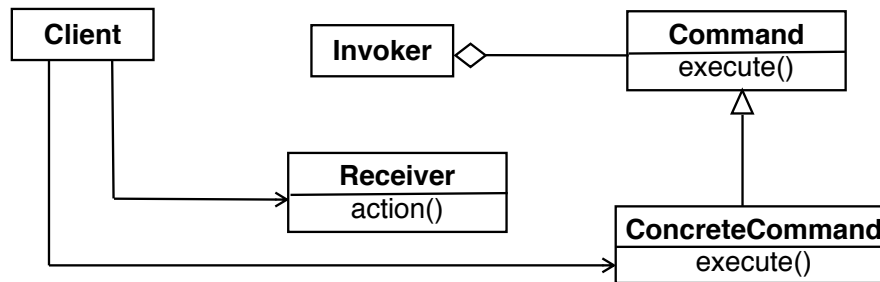


Figure 14: UML for the Command design pattern [7]

In the anti design pattern, we assume that the `Client` aggregates to set of objects, perhaps representing different GUI elements, which are labeled `Function` in the pattern. The behavioral portion of the pattern, as shown in Figure 15, requires a method which is passed an `Action` object representing the event that occurred. The `Client` then requests the source of the action, and chooses which method to execute based on the source. This is considered a bad design because for every additional `Function` type that is added, this method must be modified and execution methods must be added.

```

void actionPerformed(Action event)
{
    Object o = event.getSource();
    if(o instanceof OpenFile)
        this.openFile();
    if(o instanceof ExitProgram)
        this.exitProgram();
}

```

Figure 15: Sample code for the Anti Command Pattern

In order to search for the behavioral portion of this anti design pattern, we must use the `for` node. For each object matched to an element labeled `Function`, we look for an `if` statement containing a method invocation.

```

ROOT:Client {
    ELEM:MethodDeclaration {
        ELEM:SingleVariableDeclaration {
            REF:Action
        }
        FOR:Function {
            ELEM:IfStatement {
                REF:Function
                ELEM:MethodInvocation
            }
        }
    }
}

```

Figure 16: Behavioral pattern definition for the Command anti design pattern

We assume that the `Client` aggregates to various `Function` objects. It is also likely that instead of checking if the source object is an instance of each `Function` class, the programmer would check if it equals one of its `Function` members. However, due to limitations of the AST parser and our system, there is not a simple way of knowing the class of a named type. Therefore, we can only match this anti design pattern when the programmer directly

checks the object's class against the `Function` classes.

5 Conclusions and Future Work

We have focused on developing a framework for defining and detecting anti design patterns and making dynamic recommendations to the programmers. We have developed a format for representing both the structural and behavioral requirements of these patterns.

In order to expand the patterns that our system can recommend, future work may require the input of an expert on design patterns who has the experience to know what people tend to do wrong. We also need a corpus of examples to test and perfect our system. We hope that future research in this area will encourage the development of these examples.

As we are working from the idea of proof of concept, our format is flexible but somewhat limited in the types of behaviors and even structures that it can check. Due to the tree nature of our matching algorithm, we are not able to deal with cyclical structures. Subgraph isomorphism matching is a similar problem and research in this area may provide more insight, but any solution to the basic graph problem would need to be modified for our situation, due to the existence of different types of edges.

Since behavioral matching can only find a matching node and then drill down into it, there are certainly more complicated structures which cannot be defined. It would also be useful to understand the uncertainty of a particular match, and tell the user this information. Better behavioral matching and uncertainty would require looking for multiple matching to certain behavioral constructs and a method of probabilistic reasoning (such as Bayes reasoning).

Our behavioral matching is limited by our structural matching and lack of uncertainty. There are many cases where there may be many structural matches to a particular element, but only a subset fit the behavioral requirements. In many cases, those that fit are the actual

matches, and overall there exists a match to the pattern. In our current system, all structural matches to an element must fit the behavioral requirements of that element. More advanced matching would reject certain structural matches but still be able to determine if a pattern match exists.

An advanced version of this plugin could help the programmer to refactor his code into the recommended design pattern. However, this would require a much more extensive programmatic understanding of each design pattern and its anti pattern.

References

- [1] A. Blewitt, A. Bundy, and I. Stark, “Automatic verification of design patterns in Java”, in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, November 07-11, 2005, Long Beach, CA, USA.
- [2] K. Brown, “Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk,” Technical Report TR-96-07, Dept. of Computer Science, North Carolina State Univ., 1996.
- [3] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1st edition, March 1998.
- [4] P.T. Devanbu, D. S. Rosenblum, A.L. Wolf. “Generating Testing and Analysis Tools with Aria”, *ACM Transactions on Software Engineering and Methodology*, vol 5 no. 1, January 1996.
- [5] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, “A Bayesian Approach for the Detection of Code and Design Smells,” in *2009 Ninth International Conference on Quality Software*, qsic, pp.305-314, 2009.
- [6] M. Fowler, *Refactoring – Improving the Design of Existing Code*, 1st ed. Addison-Wesley, June 1999.
- [7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [8] Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi, “Fingerprinting Design Patterns”, in *Proceedings of the 11th Working Conference on Reverse Engineering*, p.172-181, November 08-12, 2004.
- [9] Y.-G. Guéhéneuc and G. Antoniol, “DeMIMA: A Multilayered Approach for Design Pattern Identification”, in *IEEE Transactions on Software Engineering*, vol.34 no.5, p.667-684, September 2008.

- [10] Y.-G. Guéhéneuc and R. Mustapha. “A simple recommender system for design patterns”, in *Proceedings of the 1st EuroPLOP Focus Group on Pattern Repositories*, July 2007.
- [11] R. Holmes, R.J. Walker, and G.C. Murphy, “Approximate Structural Context Matching: An Approach for Recommending Relevant Examples,” in *IEEE Trans. Software Eng.*, vol. 32, no. 1, 2006, pp. 952D970.
- [12] E. W. Dijkstra, “Making a Translator for ALGOL 60,” in *APIC-Bulletin*, no. 7, 1961.
- [13] J. Dong, D. S. Lad, and Y. Zhao, “DP-Miner: Design Pattern Discovery Using Matrix,” in *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS’07)*. (Tuscon, Arizona, March 26-29, 2007, pp.371-380).
- [14] Jing Dong , Yongtao Sun , Yajing Zhao, “Design pattern detection by template matching”, in *Proceedings of the 2008 ACM symposium on Applied computing*, (Fortaleza, Ceara, Brazil, March 16-20, 2008, pp.765-769).
- [15] J. Dong, Y. Zhao, and T. Peng, “A review of design pattern mining techniques,” *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, vol. 19, no. 6, pp. 823-855, 2009.
- [16] H.-J. Happel and W. Maalej, “Potentials and challenges of recommendation systems for software development”, in *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, (Atlanta, Georgai, November 09-15, 2008, pp.11-15).
- [17] T. Kuhn and O. Thomann, “Abstract Syntax Tree,” *Eclipse Corner Articles*, 20 Nov 2006. [Online]. Available WWW:http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html.
- [18] R. Marinescu, “Detection Strategies: Metrics-Based Rules for Detecting Design Flaws,” in *20th IEEE International Conference on Software Maintenance (ICSM’04)*, (September 11-14, 2004, pp.350-359).

- [19] R. Marinescu, "Measurement and Quality in Object-Oriented Design," *21st IEEE International Conference on Software Maintenance (ICSM'05)*, (Budapest, Hungary, September 25-30, 2005, pp.701-704).
- [20] M. J. Munro, "Product metrics for automatic identification of 'bad smell' design problems in java source-code," in *Proceedings of the 11th International Software Metrics Symposium*, (September 19-22, 2005, pp.15).
- [21] E. Murphy-Hill and A. P. Black, "Seven habits of a highly effective smell detector", in *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, (Atlanta, Georgia, November 9-15, pp.36-40).
- [22] M. Robillard, R. Walker, T. Zimmermann, "Recommendation Systems for Software Engineering," *IEEE Software*, vol. 27, no. 4, pp. 80-86, July/Aug. 2010.
- [23] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "DECOR: A method for the specification and detection of code and design smells," *Transactions on Software Engineering (TSE)*, vol.36, no.1, pp.20-36, 2009.
- [24] J. K.-Y. Ng and Y.-G. Guéhéneuc, "Identification of behavioral and creational design patterns through dynamic analysis," in *Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis*, Delft University of Technology, October 2007.
- [25] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on b-splines". In *Proceedings of the 14th Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, March 2010.
- [26] M. Salehie, S. Li, L. Tahvildari, "A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, (Athens, Greece, June 14-15, 2006, pp.159-168).

- [27] S. Thummalapenta and T. Xie, "PARSEWeb: A Programming Assistant for Reusing Open Source Code on the Web," in *Proc. IEEE/ACM International Conference on Automated Software Eng. (ASE 07)*, (Atlanta, GA, November 5-9, 2007, pp. 204D213).
- [28] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design Pattern Detection Using Similarity Scoring," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 896-909, Nov. 2006.
- [29] S. Vaucher, F. Khomh, and N. Moha, Y.-G. Guéhéneuc, "Tracking Design Smells: Lessons from a Study of God Classes," in *16th Working Conference on Reverse Engineering*, (Lille, France, October 13-16, 2009, pp.145-154).
- [30] T. Zimmermann, P. Weißgerber, and S. Diehl "Mining Version Histories to Guide Software Changes," in *IEEE Trans. Software Eng.*, vol. 31, no. 6, 2005, pp. 429D445.