Dynamically recommending design patterns

S. Smith, D. R. Plante
Department of Mathematics and Computer Science
Stetson University
421 N. Woodland Boulevard
DeLand, FL 32723, USA
Email: sarahgrace89@gmail.com, dplante@stetson.edu

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. While many programmers have knowledge of design patterns, whether rushed to meet deadlines, inexperienced in their implementations, or unaware of a particular pattern, pattern implementation may be overlooked. 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 during code development.

I. INTRODUCTION

Code reuse is a common practice used 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 [6]. 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 [2]. Each pattern defines a “bad way” of structuring code and suggests methods for refactoring. Anti patterns are not directly related to design patterns in that there are not necessarily refactoring methods for refactoring. The suggestions simply create more object oriented code.

Design pattern and anti pattern discovery assists programmers in the software development process and is currently an active 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 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, developed 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 included.

II. RELATED WORK

A. Design Pattern Detection

Brown [1] made the first attempt at automatically detecting design patterns. Since then, research has focused on both defining design patterns in a manner that is programmatically useful and identifying matches in source code. Some methods only look for structural characteristics, such as the relationships between classes, while others focus more on specific behaviors and how the classes actually interact. Dong et al. [11] provide an excellent review of design pattern mining techniques presently implemented. As noted previously, we focus on matching anti patterns.

B. Anti Pattern and Code Smell Detection

Anti patterns were originally defined by Brown et al. [2] as recurring solutions to common problems which have negative consequences. Similarly, Fowler [5] 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 [14][15] 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 [16] formally defined rule cards for nine design flaws and tested each one. Salehie et al. [21] 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. [19] also developed a system of rule cards and tested it on four anti patterns.

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. [4] 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. [20] 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.

C. 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 [18]. 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 [23] to examples of and suggestions for what call sequence to make [8][22]. Guéhéneuc et al. [7] created a design pattern recommender based on words chosen to describe the programmer’s needs.

III. METHODS

Any code pattern can be defined in terms of three characteristics: structure, behavior, and semantics [11]. The structure refers to the types of classes and relationships between them. A UML class 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 as we are not matching design patterns but rather anti patterns.

In order to limit recommendations to code which the programmer is currently working on, our matching algorithm is triggered when a class has been modified, using only the modified class and the classes “close” to it. Here we define classes to be “close” to the modified class if they may be reached by traversing at most \( N \) relations from it, where \( N \) is chosen by the developer as a configuration setting. We first look for a structural match to a given anti 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.

A. Intermediate Code Representation

As noted by Dong et al. [11], rather than working with 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 [13]. 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 [3]. A matrix may be used to provide a simplified representation of the relationships between classes, as will be described later.

For this work, we use the Eclipse JDT’s ASTParser to create an Abstract Syntax Tree [13]. 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.

B. Structural Matching

For structural matching of anti patterns, we use the matrix representation developed by Dong et al. [10], with prime numbers encoding relationships between classes. The use of prime numbers allows multiple relations to be encoded and decoded unambiguously. With the mapping defined in Table I, we note that the UML class diagram shown in Figure 1 may be represented by the matrix given in Figure 2.

![UML Diagram](image)

Fig. 1. Three classes and the relationships between them, shown using UML.

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Prime Number Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Association</td>
<td>2</td>
</tr>
<tr>
<td>Generalization</td>
<td>3</td>
</tr>
<tr>
<td>Aggregation</td>
<td>5</td>
</tr>
</tbody>
</table>

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 explore both a brute force algorithm and a second algorithm which is similar to a breadth-first search. Before using either of these techniques, we reduce the size of the search space by including only classes “close” to the modified class.

1) “K-Steps” shrinking: A developer working with a large system comprised of many classes will often only work with a few classes at a time. Only those classes being modified, along with those “close” to them, should be examined for pattern matches. 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.

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 anti 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 \( 6 = 2 \times 3 \) 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 3 shows a graph and representative matrix for a pattern, and likewise Figure 4 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 3.

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 require \( \frac{m!}{(m-n)!} \times n^2 \) operations, which approaches order \( O(m! \times n^2) \) for \( m \approx n \). With the help of the K-Steps algorithm (Section III-B1), the system matrices are relatively small, but even for a system matrix reduced to 7 nodes and a pattern of size 4, the worst case will require over 13,000 comparisons.

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 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 an 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 3 and 4, 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.

C. 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
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

for pattIndex in uncheckedPatternIndices do
  if there is a relationship between patternIdxCheck and pattIndex then
    numMatchesFound = 0
    for sysIndex in unmatchedSystemIndices do
      if there is a matching relationship between systemIdxMatch and sysIndex then
        aMatch = a node representing this match
        if match(pattIndex, uncheckedPatternIndices.remove(pattIndex), sysIndex, unmatchedSystemIndices.remove(sysIndex)) then
          currentMatch.addChild(aMatch)
          numMatchesFound++
        end
      end
    end
    if numMatchesFound ≥ required number of matches then
      continue
    end
  else
    return false
  end
end

return true

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
  }
}
```

Our algorithm steps through each root node defined in the pattern and pulls out the class that is paired with it during structural matching. In our case, using an AST representation, it then searches for the first element, a SwitchStatement, using an ASTVisitor object which stores only that type. Each matched ASTNode is then visited, 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 [9] and then evaluated with each index replaced by the boolean result of that behavioral match.

D. Pattern Definition Format

Each of the patterns created is defined in its own file and stored with the other patterns. As shown in Figure 5, a series of parameters are set, followed by labels for each element. If a particular class can be matched to multiple elements (e.g., 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 5, 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.

E. Dynamic Recommendations

Happel et al. [12] 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
Name of Pattern
NUM_CLASSES = <number classes in pattern> 
NUM_BEHAVIOR = <number behavioral elements>
MAX_STEPS = <max number 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.

Fig. 5. The configuration for files defining anti design patterns.

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 con-
stantly 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. [17]. 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.

F. Sample Definition of Singleton Pattern

In this section we provide the definition of the Singleton 
creational pattern. We explain the pattern and also the indica-
tors that a programmer should use it.

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.

A programmer who is not following this design pattern 
may attempt alternative implementations 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.

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

Fig. 6. 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 in Figure 6.

IV. 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 recom-
mand, future work requires the determination and inclusion 
of more anti patterns to provide recommendations for more 
design patterns. One possibility for doing so would be to
obtain the input of experts on design patterns who have the experience to know what the common design mistakes are that developers make. Of particular utility would be a corpus of examples to test and perfect not only our system but those of other researchers in this field. We hope that future research in this area will encourage the development of such 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 edge types in our model.

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).

Finally, an advanced version of this plugin could help programmers refactor their 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**


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


