Dynamically Recommending Design Patterns

by

Sarah Richardson

Advisor

Daniel Plante

A senior research proposal 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

Fall Term
2010

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 **6**
3.1 Design Pattern Detection 6
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
4.2 Behavioral Matching 12
4.3 Dynamic Recommendations 14
4.4 Definitions of Creational Patterns 14
4.4.1 Singleton 15
4.4.2 Abstract Factory 17

5 Conclusions and Future Work **18**

List of Figures

1 An example Abstract Syntax Tree (AST) graph 5
2 Example usage of Java's Abstract Syntax Tree. 5
3 Our recommendation tool 9
4 Three classes and the relationships between them, denoted using UML. 11
5 Matrices to represent the relationships between classes in Figure 4 11
6 The matrix for Figure 4 using prime numbers. 11
7 Example of the Singleton design pattern. 15
8 A boolean is checked before instantiating the single class. 16
9 UML for Abstract Factory, from http://www.vincehuston.org/dp 17
10 Poor design choice instead of using a factory method. 18

0.1 Abstract

Recommendation Systems for Software Engineering are created for a variety of purposes, such as recommending sample code or call attention to bad coding practices (code smells). We propose to create 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 get a product working, the use of a particular design pattern may not occur to them. We will develop 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 working.

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 their system. Similarly, design patterns encourage the reuse of object oriented ideas [7]. They provide design solutions to common problems, but must be implemented in a specifically to 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 method of code structure and relationships, and suggests methods of refactoring. Anti patterns are not directly related to design patterns; in other words, there are not refactorings for the anti patterns that turn them into design patterns. Instead, the suggestions 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 propose, instead of finding instances of either type of pattern, to recommend the use of design patterns based on an unfinished project. We will determine that a programmer is trying to solve a common problem in a way that could be improved using a design pattern. We will create a framework for detection and a format of storing rules for each design pattern. The rules will then be processed by the framework, and our Eclipse plugin will search for instances within the current project. The plugin will also determine what to make recommendations about and how often to make them. The initial work will only focus on recommending the

use of two or three design patterns, but the framework will be created to allow additions of new patterns.

2 Background Information

2.1 Design Patterns

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

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), 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 could be used to solve the same problem. There are also subtle differences between some of

the creational patterns, requiring some intuitive knowledge of the code's purpose. 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 it's children are the elements which are part of it [16] (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 or vector may be used to provide a simplified representation of the relationships between classes and the number of elements within each class, as will be described later.

We will use the Eclipse JDT's `ASTParser` to create an Abstract Syntax Tree [16]. A `CompilationNode` is generated from the contents of a Java file and passed to an `ASTVisitor`, which will traverse each element in the tree. 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, if we are interested in each `FieldDeclaration`, we can create an `ArrayList` to store them and override the appropriate `visit()` method.

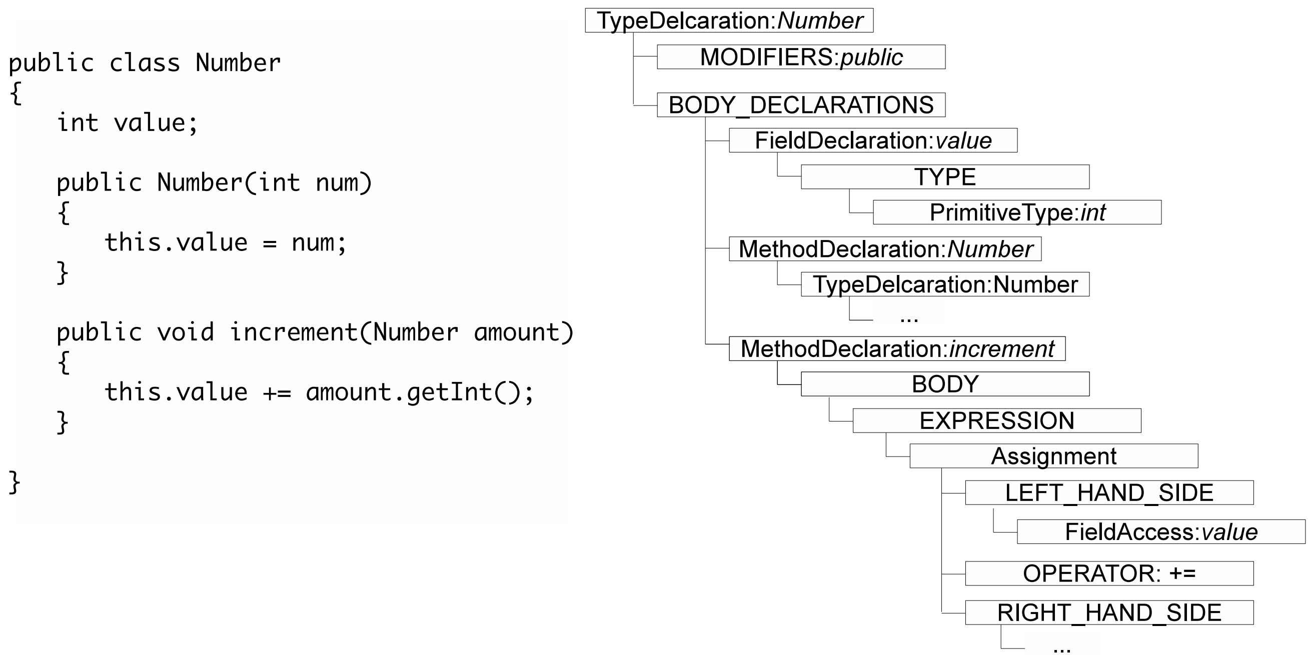

Figure 1: An example Abstract Syntax Tree (AST) graph

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

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

  public List<FieldDeclaration> getFields()
  {
     return fields;
  }
}
```

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

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 programmatically 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 focuses 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.* [27] 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.* [12] 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.* [13] further explained the method of searching for a design pattern representation matrix in a source code matrix using normalized cross correlation. This method uses exact matching, but incorporates how variations on the patterns.

In addition to structural analysis, some design pattern discovery tools analyze the behavior of the system. Ng *et al.* [23] analyzed behavior through dynamic analysis. Although static analysis can give a more complete model of the behavior, they claim that dynamic analysis can 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 [17][18] 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 [19] formally defined rule cards for nine design flaws and tested each one. Salehie *et al.* [25] 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 isn't formally defined. Moha *et al.* [22] also developed a system of rule cards and tested it on four anti patterns. Vaucher *et al.* [28] 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 produces a probability that a given class or set of classes follows an anti-pattern, which is a more realistic way to detect something which cannot be exactly defined. Similarly, Oliveto *et al.* [24] used B-splines to detect anti-patterns and gives 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 [21]. 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 [29] to examples of and suggestions for what call sequence to make [11][26]. 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 [14]. The structure refers to the types of classes and relationships between them. A UML diagram, for example, contains mostly structural information about a system,

giving the basic parameters of each class and specifying the inheritance, association, and other relationships between classes. For our purposes, the structural information for a given system could also include basic characteristics about the class, such as number of lines of code and the amount of coupling to other classes. However, if these characteristics are not necessary for a structural match, they will be considered along with behavioral characteristics.

The behavioral characteristics used to define a pattern are more complex and 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.

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.

In order to define patterns for recommending the use of design patterns, we will use a combination of structural and behavioral characteristics. Any semantic information will be incorporated into these tests. As shown in Figure 3, we will first look for a structural match. Once we have found a potential match for a particular pattern, we will test the behavioral characteristics for those classes. Given these two stages, we will determine the probability that the user would benefit from that design pattern.

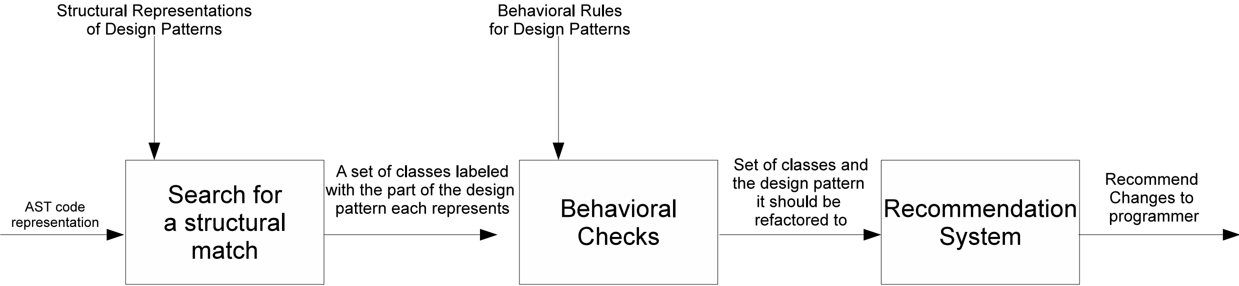

Figure 3: Our recommendation tool

4.1 Structural Matching

A programmer who would be better off using a certain design pattern will often follow a similar structure but make poor behavioral choices. This is especially true in the case of creational patterns. For each design pattern, we will define certain structural characteristics that we would expect from a programmer who should be using a particular design pattern. For example, a set of classes that should be using the Abstract Factory pattern will probably follow the same inheritance structure but will not include a factory object.

Dong *et al.* [12] developed a structural search using a matrix and weight algorithm. They use this algorithm (along with some behavioral checks) to find instances of design patterns in a system. A matrix defines the relationships between classes in the system, and each class is given a weight. Each design pattern is defined in terms of a matrix and each class's weight, and they search for matching structures within a system whose matrix and weights are known.

An nxn matrix, where n is the number of classes in the system, can be used to represent the relationships between all the classes in the system. For example, the set of matrices in Figure 5 represent 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.

To do this, we take advantage of the fact that the product of prime numbers is a unique composite number; in other words, any number can be broken down into a 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 between class i and class j. The values used to represent relationships are shown in Table 1. Using prime numbers, the matrix for the classes in Figure 4 is shown in Figure 6.

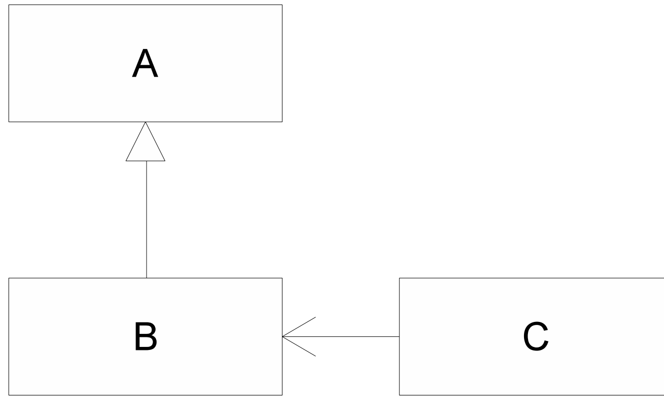

Figure 4: Three classes and the relationships between them, denoted using UML.

Generalizations

$$\begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

Associations

$$\begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{array}$$

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

.

Relationship	Prime Number Value
Association	2
Generalization	3
Aggregation	5

Table 1: Prime number values for different relationships

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

Figure 6: The matrix for Figure 4 using prime numbers.

We are also interested in knowing the types of the classes, such as whether a class is abstract or an interface. This will be tracked along with the matrix and will also help in matching. Certain forms of semantic matching will be possible, such as looking for the repetition of the name "Windows" across the implementations of several abstract classes. In the case where semantic data is tested, the structural match with the highest semantic match will be tested first for behavioral characteristics.

We also need to allow flexibility in the number of a certain class type. For example, if the pattern requires that class A have some number of inheriting classes, then our definition and search algorithm needs to account for this. We need to recognize which classes are part of this set and pass them on for behavioral matching. Therefore, our structural definition will indicate whether a certain class description should have one, zero or more, one or more, or two or more instances.

4.2 Behavioral Matching

When searching for design pattern instances, the most difficult problem in finding behavioral characteristics is that there are different ways to implement the same behavior. In addition to this difficulty, when searching for signs that point to use of a particular design pattern, there may be many different indicators. Some could be more important, therefore being stronger indicators, while others could add a small amount to the likelihood of suggesting that design pattern.

A set of behavioral rules will be defined for each class in the pattern, and a framework will be created to test their existence. The rules will be written in terms of their behavior, rather than a specific syntax, and the AST tree for each class will be preprocessed to express its behavior according to the same format. For example, `obj = new Object()` and `Object obj = new Object()` both be understood as an instantiation of `obj`. For example, one

behavioral test might be

```
conditional
    instantiation
```

Additionally, it will at times be necessary to specify that certain objects are involved in multiple places in the same rule or even that classes found during structural matching are called within a rule. Therefore, each of these behavioral elements will allow parameters, and anywhere that the parameter exists must use that same object or method. For example, the rule above could be a check on the existence of some object followed by the instantiation of the same.

```
conditional objName
    instantiation objName
```

Once we find all rule matches in the set of classes, we need a method for determining the probability that these structures should be refactored into a design pattern. For many of these rules, we are interested in knowing how many times they are matched. Therefore, a simple probability table with the inputs being each rule (true if a match is found and false otherwise) may not suffice to model this situation. We are still developing a system for measuring these probabilities. It is possible we will need probability functions with distributions dependent on the number of matches, or an additive ranking in which the more rules matched and the more of each rule that is matched, the higher the rank.

The method most often used for finding the probabilities in general is to use machine learning on a corpus of examples. Because this particular work has not been done, there is no readily available corpus of examples of code which is poorly designed and have known improvements using design patterns. We hope to construct a framework for using machine learning and begin to develop a corpus of examples.

4.3 Dynamic Recommendations

Happel *et al.* [15] 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 bombarded with suggestions is likely to begin ignoring them or turn off the tool. Therefore, once a recommendation has been ignored, it is important to remember and not revisit it.

Our tool will only make recommendations within packages which the user has edited. Additionally, it will focus on making recommendations which involve files which the programmer has modified or at least viewed during the current session. The tool will present recommendations without the user having to go through a series of steps to request them, as suggested by Murphy-Hill *et al.* [20]. However, we will develop an unobtrusive mechanism which does not distract the user from his or her work. Finally, the recommender will explain the recommendation and the advantages of refactoring the code to fit the design pattern.

4.4 Definitions of Creational Patterns

This section contains the definitions of two of the creational patterns we will make suggestions for. We explain the pattern and also the indicators that a programmer should use this particular pattern. Future work involves developing a framework to define these indicators and decide whether to make a recommendation.

4.4.1 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 7 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 Single getInstance() {
     if(this.instance == null) {
          this.instance = new Single(0);
     }
     return this.instance;
  }
}
```

Figure 7: Example of the Singleton design pattern.

A programmer who is not following this design pattern but does not want multiple instances of a particular class 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 singleton class (the class which should have only one instantiation) and the using class or classes.

One alternative to using the Singleton pattern is to make all of the properties of the singleton class static. In this way, for each instantiation, the values will remain the same. Many times a Singleton pattern is needed when the singleton class may or may not be instantiated, depending on whether certain code fragments are reached. In order to guarantee that it is only instantiated once, the programmer may set up a conditional check on some boolean value before instantiating the singleton with the `new` operator, as shown in Figure 8. Alternatively, they may initially set the object to `null` and later instantiate if it is not `null`.

```
public class SingleUser {
  private Single obj;
  private boolean isInstantiated;

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

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

Figure 8: A boolean is checked before instantiating the single class.

The structural tests are very simple; if one ore more objects have an aggregation with another object (the singleton), it fits the structural requirements. The behavioral checks will look for either mostly static variables or conditionals, as described above. Either of these being found will result in some probability that Singleton should be recommended, and seeing

both will increase that probability.

4.4.2 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 9. For example, consider a situation where products are various menu 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.

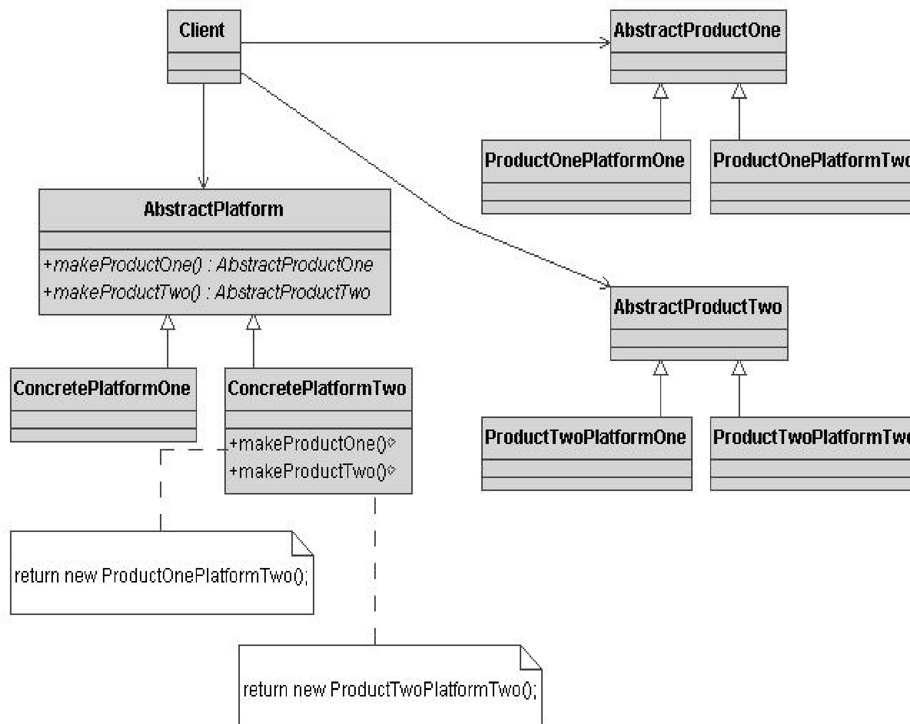

Figure 9: UML for Abstract Factory, from http://www.vincehuston.org/dp

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 same

logic every time it is needed. As shown in Figure 10, 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 will look for switch statements in which multiple objects are being instantiated, 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 Widget instantiations.

Multiple ocurences of these instantiation structures will increase the likelihood that the Abstract Factory pattern would be useful to the programmer.

```
void display_window()
{
   Widget[] w;
   switch(this.platform)
   {
      case MAC:
         w[] = { new MacButton, new MacMenu };
      case WINDOWS:
         w[] = { new WindowsButton, new WindowsMenu };
   }
w[0].draw();
w[1].draw();
}
```

Figure 10: Poor design choice instead of using a factory method.

5 Conclusions and Future Work

We have proposed a system to recommend the use of design patterns based on a programmer's poor design decisions. This work will be implemented in the Spring of 2011. We will focus on developing a framework for detecting signs that a programmer should make use of a particular design pattern. This will involve determining what and how to represent

these patterns and signs programmatically, how to search for them, and how a dynamic tool can be used to make recommendations.

In order to make progress towards this framework, we will develop a flexible and generic format for representing structure and behavioral rules. We will consider the need to look for one or more instances of rules and increasing the probability accordingly. To this aim, we will develop a method of additive ranking or probability in order to determine what is recommended to the programmer.

A corpus of examples is necessary to make progress in properly detecting these situations. Using a corpus, it would be possible to appropriately train our framework to suggest design patterns. We will build our framework in such a way that a future corpus could be incorporated, and we hope that future research in this area will encourage the development of these examples.

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, emphAnti 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. 952Ð970.

[12] 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).

[13] 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).

[14] 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.

[15] 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).

[16] 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`.

[17] 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).

[18] 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).

[19] 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).

[20] 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).

[21] M. Robillard, R. Walker, T. Zimmermann, "Recommendation Systems for Software Engineering," *IEEE Software*, vol. 27, no. 4, pp. 80-86, July/Aug. 2010.

[22] 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.

[23] 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.

[24] 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.

[25] 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).

[26] 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. 204Ð213).

[27] 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.

[28] 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).

[29] 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. 429Ð445.