

Stuart Norton  
CMPS 290G, winter 2005  
Term project report

## Studying Design Pattern Evolution through Static Analysis

### 1 Introduction

Design patterns (Gamma et al. 1995) are simple templates for structuring code that conform to certain restrictions of inheritance, method invocation, etc. Software design patterns provide a window into the design of a software system, which because of their well-defined rules, can be identified (to some degree) using automated tools.

The goal of this project is to study the evolution of design patterns in software. Much of the methodology is borrowed from Heuzeroth, Holl, Högström, & Löwe (2003), “Automatic Design Pattern Detection.” Heuzeroth et al. present automatic methods for detecting design patterns in Java code. I follow these authors in using the Recoder tool, a Sourceforge project, for static analysis of Java code to identify instances of design patterns. I then use the Kenyon tool for analyzing the evolution of these design patterns.

The questions I originally hoped to address in this project are:

1. Can design patterns be detected reliably based only on static analysis?
2. Have the uses of design patterns changed in the Recoder code? If so, what kinds of change have occurred? Have design pattern instances been added, removed, or otherwise changed?
3. Has the use of design patterns changed the reliability (bug counts), complexity (lines of code) or other quantitative measurements of the software?

I have been unable to address any but the first of these questions. In the course of attempting to address these questions, I have accomplished the following:

1. I learned to use Recoder to find design pattern candidates, as described in the paper by Heuzeroth et al.
2. I created a custom FactExtractor, which uses Recoder to find code candidates which appear to use the Observer design pattern.
3. I used Kenyon to measure and store the results of this analysis, applied to the Recoder software project.
4. By examination of the graphs built by the FactExtractor and stored with Kenyon, I identified specific cases of design patterns as well as spurious cases which the algorithm identified.

One important aspect of the Heuzeroth et al. paper which I do not reproduce due to time constraints is the dynamic analysis. Heuzeroth et al. use static analysis to identify a candidate set of design pattern instances. They then use dynamic analysis to narrow the number of candidate design patterns significantly. Without these tools, I have to rely on naming conventions and manual code inspection to identify design patterns from the candidates.

## 2 Design Patterns

### 2.1 What are design patterns?

The seminal work on design patterns in software engineering was *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al. 1995). This book was the first to present a catalog of design patterns for use in software engineering. The conceptual origin, however, is with Christopher Alexander, who developed “patterns” for solving problems in architecture. Alexander wrote, “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a billion times over, without ever doing it the same way twice” (Alexander 1979). This description captures much of the meaning and purpose of design patterns. The following list of properties of design patterns attempts to expand on that description.

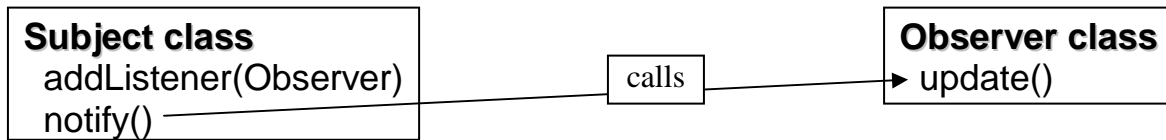
1. Design patterns are tools or “templates” for solving problems in software design. The template includes a problem definition (the goals and constraints under which the pattern can be applied) and a solution for the problem.
2. Design patterns must be reusable. If a set of design choices solves a single unique problem, it may be valuable, but it cannot be regarded as a tool for solving other problems, so it does not qualify as a design pattern.
3. Design patterns are not implementations, but they are to be applied at a relatively low level. A design pattern cannot be an implementation because it must be flexible enough to be reused in other environments. However, it must be strictly defined, in terms of the interaction between low-level components (e.g. functions, objects, and classes). A paradigm like object-oriented programming is not sufficiently constrained to be regarded as a design pattern.
4. Design patterns are sometimes seen as a means of elevating software engineering to the status of a “true” engineering discipline, in which a body of knowledge (the design patterns) can be applied systematically to solve problems which commonly arise in the discipline (Tešanović 2004).

### 2.2 An example: the Observer design pattern

This section describes a design pattern from the text by Gamma et al. (1995): the Observer pattern. This design pattern is commonly used in event-driven systems. A problem solved by the Observer pattern is separating a data model and/or business logic from the views or use of the data. For example, a database may store the price of a product. If that price is volatile, then when that price changes, it should be updated dynamically on the user’s screen. Furthermore, the user’s display may be in Euros, while the database internally stores dollar amounts. If the data and its view are not separated, then changes to one will affect the other, and maintenance will be more difficult.

This problem can be solved by separating the data and its view into two components: a Subject and an Observer (see Figure 1). The Subject is a component that experiences a change of some kind. In the example, the Subject may be an application on the database server that monitors prices in the database. The Observers are a set of components that must be notified about the change. In the example, the Observers may be the users’ client

applications. When a change occurs, the Subject must perform some act to notify the Observers that the change has occurred, so that the Observer can respond appropriately. In the example, the Observers may respond by multiplying the new price by an exchange rate and redisplaying the price. Each Subject can have multiple Observers, and Observers can be configured (assigned to observe a particular Subject) at run-time.



**Figure 1. Classes and methods in the observer pattern.**

Because the Observer design pattern is well defined, a definition of the Observer design pattern in object-oriented programs can be expressed relatively simply. Following Heuzeroth et al., I regard these requirements as necessary to identify an instance of the Observer design pattern:

1. The Subject component is encapsulated in a class I will refer to as **subject**. This class includes at least two methods, which I will refer to by the following names (though the actual names used in the source code are inconsequential):
  - a. **addListener**, which adds an Observer to the list of objects that must be notified
  - b. **notify**, which notifies the Observers of the event.
2. The Observer component is encapsulated in a class I will refer to as **Observer**. The **observer** class has at least one method, which I will refer to as **update**.
3. The **addListener** method must include at least one parameter of type **Observer**.
4. The **notify** method calls the **update** method on an object of type **Observer**.
5. The **notify** method call does not include a parameter of type **Observer**.
6. The **observer** class is not a superclass or subclass of the **subject** class. This would violate the separation that is produced by the Observer design pattern.
7. The **subject** stores the **observer** for later notification. To accomplish this, the **addListener** method either:
  - a. passes the **Observer** argument to another method, or
  - b. includes the **Observer** argument on the right hand of an assignment statement.

It should be noted that this definition is more restrictive than the definitions of the Observer design pattern which have been presented by Gamma et al. and others. For example, a “pull” (rather than “push”) version of this design pattern, in which the Observers poll the Subject for updates, is another possible implementation. For simplicity, I follow Heuzeroth et al. in using the more restrictive definition above.

Also, note that the above requirements make no reference to a **removeListener** method of the **subject**. The Heuzeroth et al. definition includes a **removeListener** method in the analysis, but then allows the **removeListener** to be null. The definition above will find all instances of the Observer design pattern that Heuzeroth et al. find, without the

unnecessary complexity of a `removeListener` method which is not actually a requirement of the specification.

It is asserted here, but not proven, that this definition is sufficiently restrictive to avoid including designs which do not conform to the Observer design pattern. (As will be shown in section 4, this definition is in fact insufficient to identify the relevant patterns.)

### **2.3 Why study design patterns?**

Design patterns are widely regarded as useful tools that enable inexperienced engineers to apply the time-tested techniques of their more experienced colleagues. Studying the evolution of design patterns in software will provide new insights into how designs in software evolve over time. It will also help us better understand design patterns and their impact on software projects.

Most modern studies of software evolution use data from CVS and other version control repositories to study how programmers work (see, e.g., Mockus, Fielding & Herbsleb 2002 and Fischer, Pinzger & Gall 2003). The basic facts that can be extracted relate to programmer behaviors – what is checked in, and when? What kinds of changes are made, and how widespread are they? But it is very difficult to extract information about less tangible aspects of software, such as the architecture or design of the software and how it changes. Some studies use the directory and file hierarchy to address architectural issues (Gall, Jazayeri & Krajewski 2003; Eick et al. 2001), but this is a very blunt instrument for studying such a subtle art. Design patterns are sufficiently well-defined that they can be measured automatically on source code, which is important to enable large-scale studies. Design patterns can therefore provide a window onto software design that is otherwise more difficult to quantify.

As is true in many areas of software engineering, little empirical evidence is available to test the efficacy of design patterns. Studying how design patterns affect the evolution of a software system can help to supply this empirical evidence. Questions we hope to address by studying design patterns include:

1. How widely are design patterns used?
2. Which design patterns are used most often?
3. Are design pattern instances added, removed, or otherwise changed over the course of software evolution?
4. How many users make use of design patterns? Which users are they?
5. Does the use of design patterns change the reliability of software (e.g., bug counts)?
6. Does the use of design patterns change the complexity of software (e.g., lines of code)?
7. Does the use of design patterns reduce the impact of changes in the code involved in the pattern?

These questions are not addressed in this paper, but they are worthwhile topics for study. Due to limited time, this paper focuses on the detection and analysis of design patterns, the first step in such studies.



This version of the algorithm is somewhat different from that presented by Heuzeroth et al., because I have chosen not to include the **removeListener** method in my analysis, as mentioned in section 2.2.

The above pseudocode returns a set of tuples, where each tuple represents an instance of the Observer design pattern. The algorithm was implemented in Java, in the `PatternFinder` class, and the code is available upon request.

### 3.3 Using Recoder with Kenyon

Kenyon can be used to track the evolution of design patterns over time. Because the design pattern detection algorithm, Recoder, and Kenyon were all written in Java, it was relatively simple to link the three together. In the completed processing pipeline, there are four main steps:

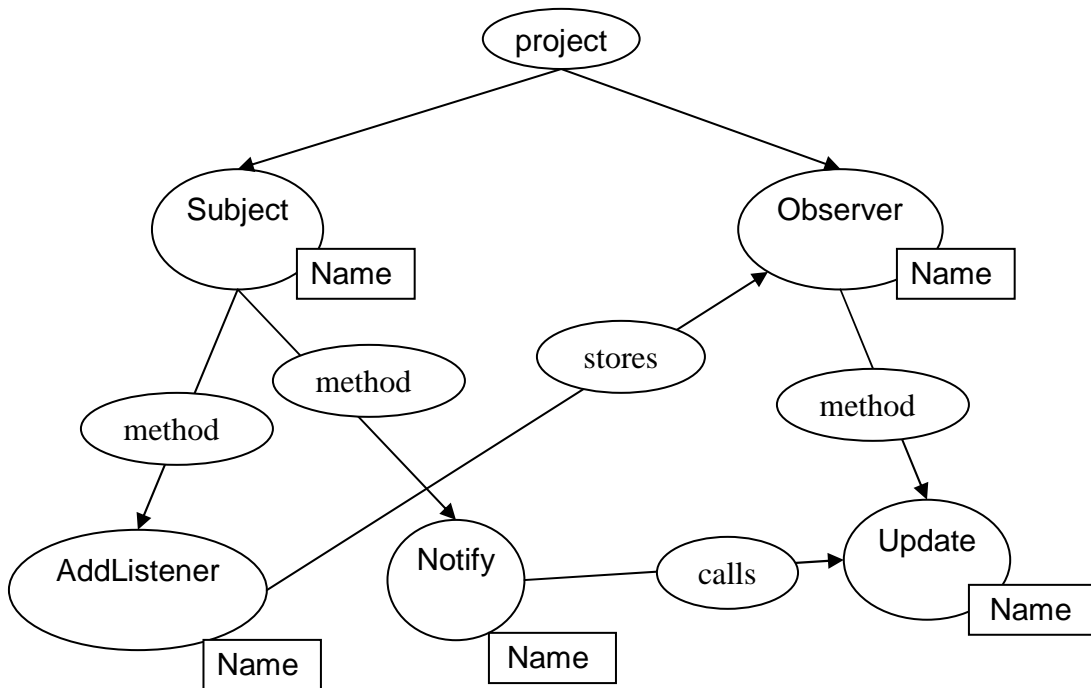
1. Kenyon downloads a revision of the project to be studied from a source control repository.
2. A fact extractor, `PatternExtractor`, is executed on the root directory of the checked-out project.
3. `PatternExtractor` executes `PatternFinder` to detect any pattern instances in the project.
4. `PatternExtractor` constructs a graph to represent the patterns that were found and saves the graph in a Hibernate database.

For analysis with Kenyon, the tuples returned by the pseudocode in section 3.2 must be stored in a graph. The structure of the Kenyon's graph should depend on the methods and objectives of the analysis. For this study, it would be sufficient to use a very simple model, where each node of the graph represents a single instance of the observer pattern. However, to combine these data with other analysis tools, it would be more useful to have a more flexible structure that matches the granularity of data provided by other methods. For example, to see how the design patterns contribute to data or control flow, it is useful to have methods represented individually, rather than including all methods and classes in a single node for each design pattern. For this reason, I have chosen to store the results from the pattern extractor in the graph shown by Figure 2. In this graph, the following nodes are included:

- Project: root node of the graph, representing the project as a whole
- Subject: represents the Subject class
- Observer: represents the Observer class
- AddListener: represents the `addListener` method
- Notify: represents the `notify` method
- Update: represents the `update` method

The following edges are also included in the graph:

- Three edges labeled “method” indicate the relationship between the Subject/Observer and their methods
- An edge from AddListener to Observer indicates that the former “stores” the latter
- An edge from Notify to Update indicates that the former “calls” the latter



**Figure 2. Graph used to store data about observer pattern instances.**  
Node and edge names are indicated by ovals, and attributes by boxes.

## 4 Results

I have tested the PatternFinder class by running it on a sample set of source files: a typical observer published online at <http://www.javaworld.com/javaworld/javaqa/2001-05/04-qa-0525-observer.html>. The algorithm is successful at detecting this observer pattern and correctly identifying all of the components in the pattern.

After testing the PatternFinder, I used Kenyon to analyze the source code of the Recoder project itself. Recoder was a good choice for testing because in the Recoder project, all observers have been named according to strict conventions (where all Observer class names include the string “Listener”). This makes it easy to identify where the developers believe they have used the observer pattern as a test of the algorithm.

I used a 1-day search window for revisions, and using this search window there were a total of 13 revisions of the Recoder project, all between 2001 and 2004. The Recoder project consists of 84,000 LOC (including whitespace and comments) in its current version.

The static analysis on the full Recoder project detected over 1,000 instances of the design pattern using the criteria described in section 2.2 (the search halted at 1,000 instances because I had only allocated space for 1,000 instances in the array). This is more than one instance of the Observer pattern per 100 LOC, possibly much more. Most of these instances are clearly spurious, for example:

Observer: java.lang.String

Subject: recoder.io.DefaultClassFileRepository

addListener: recoder.io.DefaultClassFileRepository.findClassFile

notifyMethod: recoder.io.DefaultClassFileRepository.propertyChange

updateMethod: java.lang.String.equals

This instance does conform to all of the static criteria for an instance of the Observer design pattern, but it is clear that the java.lang.String class does not represent an observer!

After getting these results, I narrowed the focus of the search by limiting the search for Subject instances to the recoder.io package, which contains only 2411 LOC, but there were still 166 instances found. I then required that the Observer class belong to the recoder package, which will eliminate cases where the Java SDK classes are found to be observers. This reduced the number of instances found substantially, but there were still 71 instances found in the recoder.io package. In the recoder.services package, which has 9345 LOC, there were 1336 instances of the pattern detected, more than one instance per 10 lines of code.

I was very surprised at the number of Observer pattern instances detected, but random checking seems to confirm that the algorithm is working properly. Of course, there are systematic reasons for the huge number of misidentified patterns. I have not made a detailed study, but an informal investigation shows that a large number of the findings appear to be setters and getters. Getter and setter methods are common in the Recoder API, and because the requirements on the individual methods in the Observer pattern are not very stringent, getter and setter methods often fit the requirements. That is, a setter method can be identified as the AddListener method, and any method in the same class which calls a method of the class being set can be identified as the Notify method. Also, overloaded methods (which are counted multiple times) can greatly add to the number of instances found. I would like to assess how much the data can be trimmed by cutting out method overloads, but time does not permit further investigation.

I have not developed the proper analysis tools to deal with this volume of data; analyzing these data robustly with Kenyon is not a trivial problem. In any case, since the vast majority of the data are spurious findings, it makes such an analysis unproductive.

## **5 Conclusions**

### **5.1 Evaluation of the method**

The pattern detection methods used here are relatively new, first presented by Heuzeroth et al. in 2003. Due to limited time, I have not been able to add a significant amount to previous results by adding Kenyon to the process. The volume of spurious data makes

such an analysis unwieldy and lacking in insight. However, to state the result more positively, I have confirmed the assertion by Heuzeroth et al. that static analysis is insufficient for the study of design patterns. The magnitude of the problem was not described by Heuzeroth et al., but I have found that only a fraction of a percent of the Observer pattern candidates that are found using this static method are likely to be actual instances of the pattern.

The Recoder tool could be used to automatically instrument the code for candidate instances of a design pattern. This allows a dynamic analysis to be performed by the instrumented code. This is apparently the method used by Heuzeroth et al. to narrow the list of candidates to only 4. (Incidentally, a simple grep on the source code reveals 5 classes or interfaces with the word “listener” in their names: `ProgressListener`, `PropertyChangeListener`, `ChangeHistoryListener`, `ModelUpdateListener`, and `ASTIteratorListener`. It would be interesting to determine why there is a mismatch between the naming convention and the candidate list, but it may simply be that one of these entities is not in use or not instantiated in a class.)

This discovery, a confirmation of the assertion by Heuzeroth et al. that dynamic analysis is needed to further trim the results, in (unfortunately) the most reliable and important finding to come out of this effort.

A few other notes on working with Recoder may be useful for future researchers:

- My experience with Recoder was positive overall. The tool is quite powerful, and as mentioned above, its ability to automatically make changes in source code could prove useful for dynamic analysis methods.
- The Recoder API is not very well documented. There are a few high-level diagrams showing some of the structure, and fortunately javadoc is provided. But the specifics of how the lower-level pieces of the API fit together are not documented. As a result, to find out how to extract a given piece of information from Recoder can require a lot of combing through the javadoc, and even then, you must often rely on naming conventions to determine what a class in the API actually represents.
- Recoder suffers from many of the instabilities seen with other static analysis tools – though successful compilation is not required to run Recoder, the tool does rely on the assumption that the code is “well-formed”. If there are errors in the code, it will often cause an exception. Fortunately, most exceptions can be caught reliably and the analysis can continue despite such problems.

## **5.2 Threats to validity**

I regard the main finding of this paper, that dynamic analysis is required to identify pattern instances with any precision, as quite reliable. Aside from substantial errors in my code or the Recoder project, I can see two main threats to the validity of this finding:

- This study only addresses the Observer pattern, which is a relatively simple example of a design pattern. It is possible that other, more complex design patterns can be identified more reliably based on static analysis alone.

- It is possible that more ingenious methods can be used to narrow down the list of candidate patterns through static means. For example, it may be possible to statically determine whether every Observer instance that is passed to the Subject instance (with the `addListener` method) will have its `notify` method called by the Subject's update method. This is a challenging objective for static analysis, however. A first step in this direction might be to add a requirement that a single execution of the update method must be able to call `notify` multiple times, by requiring that the call to `notify` is enclosed in a loop. We could also require that the variable representing the Observer instance is assigned to a value inside the loop, so that more than one observer can have its `notify` method called. These are challenging requirements; a simpler way to narrow the candidates would be to use static methods to identify getter and setter methods. It is uncertain what the full impact of such requirements would be, but they would certainly help.

The larger question of the validity and value of studying the evolution of design patterns is not addressed significantly here. To conclude this paper, I present a list of threats to the validity of such research that have come to light during this project.

- The kind of analysis done here can only be performed when the types of all variables involved in the design pattern are known at compile-time. Less strongly-typed languages than Java, non-object-oriented frameworks, and less well-constrained code can significantly limit the effectiveness of the methods used here.
- There is a great deal of variability among design patterns, and even some variability in how they are defined by different authors. Even if these methods are successful in analyzing one design pattern, they may not be sufficient to constrain another.
- The details of how the evolutionary analysis should proceed are still uncertain. To answer the questions listed in section 2.3, sophisticated tools will be required. Problems such as linking lines of source code with other activities (e.g., Froehlich and Dourish 2004), measuring the reliability and maintainability of source code in response to changes (e.g., Eick et al. 2001), and visualizing the evolution (e.g., Gall et al. 1999) are all open areas of research on which such an analysis would rely.

The author of this paper is indebted to the authors of the Recoder project, whose software was remarkably effective, and to Heuzeroth, Holl, Högström, & Löwe, who inspired this work.

## References

- Alexander, C. *The Timeless Way of Building*. Oxford University Press, 1979.
- Eick, S., Graves, T., Karr, A., Marron, J., and A. Mockus, *Does Code Decay? Assessing the Evidence from Change Management Data*, IEEE Trans. on Software Engineering, Vol. 27, No. 1, January 2001, pp. 1-12.

- Fischer, M., Pinzger, M., and H. Gall, *Populating a Release History Database from Version Control and Bug Tracking Systems*, in Proc. 2003 Int'l Conference on Software Maintenance (ICSM'03), September, 2003, pp. 23-32.
- Froehlich, J. and P. Dourish. *Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams*, in Proc. International Conference on Software Engineering (ICSE 2004), Edinburgh, Scotland, pp. 387-396.
- Gall, H., Jazayeri, M., and J. Krajewski, *CVS Release History Data for Detecting Logical Couplings*, in Proc. Sixth Int'l Workshop on Principles of Software Evolution (IWPSE'03), 2003.
- Gall, H., Jazayeri, M., and C. Riva, *Visualizing Software Release Histories: The Use of Color and Third Dimension*, in Proc. International Conference on Software Maintenance (ICSM), Oxford, England, August/September 1999, pp. 99-108.
- Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.
- Heuzeroth, D., Holl, T., Högström, G., and W. Löwe. *Automatic Design Pattern Detection*, in Proc. IEEE International Workshop on Program Comprehension, 2003, Vol. 11, p. 94.
- Mockus, A., Fielding, R., and J. Herbsleb, *Two Case Studies of Open Source Software Development: Apache and Mozilla*, ACM Trans. on Software Engineering and Methodology, Vol. 11, No. 3, July 2002, pp. 309-346.
- Tešanović, A. "What is a pattern?" Course notes, Linköping University, Sweden, 2004.