Project Number: GFP 0801

A DESIGN PATTERN GENERATION TOOL

A MAJOR QUALIFYING PROJECT REPORT

SUBMITTED TO THE FACULTY OF

WORCESTER POLYTECHNIC INSTITUTE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF BACHELOR OF SCIENCE

BY

CAITLIN VANDYKE

APRIL 23, 2009

APPROVED:

PROFESSOR GARY POLLICE, ADVISOR

ABSTRACT

This project determines the feasibility of a tool that, given code, can convert it into equivalent code (e.g. code that performs the same task) in the form of a specified design pattern. The goal is to produce an Eclipse plugin that performs this task with minimal input, such as special tags.. The final edition of this plugin will be released to the Eclipse community.

ACKNOWLEGEMENTS

This project was completed by Caitlin Vandyke with gratitude to Gary Pollice for his advice and assistance, as well as reference materials and troubleshooting.

TABLE OF CONTENTS

TABLE OF ILLUSTRATIONS

INTRODUCTION

This project attempts to answer two questions:

1. Can a tool be designed that can convert a design pattern template into usable code, regardless of the purpose or structure of the code?
2. Can that tool analyze code and determine the appropriate use of a design pattern?

Design patterns help improve programmer efficiency. Several open-source tools are available to create design pattern templates for the programmer to fill in with relevant code. PatternBox (PatternBox, 2009) is one example of these tools.

This project determines whether tools of this type can be extended to apply patterns to previously written code, and automatically determine suitable patterns for the given code.

The success of the Pattern Wizard depends on the variety (in language and function) of programs which are compatible with the tool. There should be minimal constraints on the Pattern Wizard's use. Additionally, the ratio of user input in the form of code and annotation to the tool's code output should be as low as possible.

HOW DID THE PROJECT EVOLVE?

The project began as a two-person project. Galia Traub and I meant the project to explore the differences in design patterns across different programming paradigms. While design patterns started as object-oriented in nature, we believed that design patterns could be used in other paradigms as well, such as functional and procedural. To increase the scope of the project, we also attempted to write a tool demonstrating how code would change for each pattern in each paradigm and create a language to define design patterns regardless of paradigm, if it proved possible, or alternatively to define design patterns in the other paradigms individually.

The goals became two different projects due to differences between the topics. This project evolved from the code tool, reduced in scope for a single student. The number of patterns to integrate into the project caused the scope to be reduced to three representative patterns. Because of time concerns, I further reduced the project to representing the object-oriented application of the pattern. As a result, the only language currently compatible with the Pattern Wizard is Java. However, this project serves as a proof-of-concept and a starting point for further work.

WHY IS THIS HELPFUL?

I designed the Pattern Wizard with a beginner programmer in mind. Because the tool converts basic code into a form that uses a design pattern, a programmer can analyze how the pattern works without needing to write an implementation. If the process of implementing a pattern becomes easier, the focus can shift from how to write an implementation of the pattern to where in the code to use the pattern.

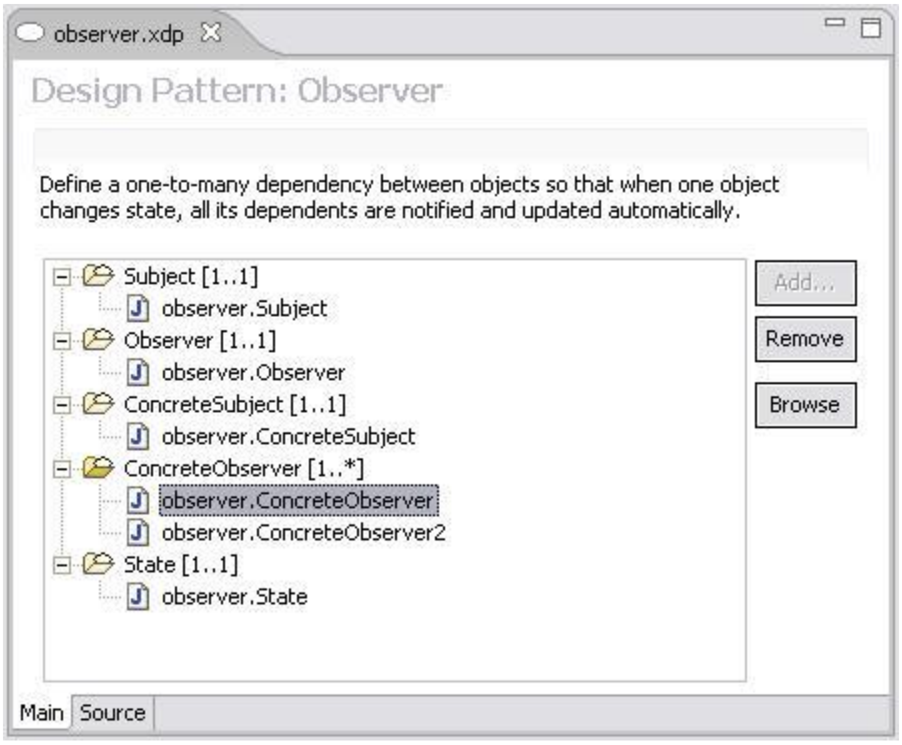
Allowing the user to write code before applying the pattern increases the number of scenarios where the tool would be appropriate; for instance, the tool could be used on third-party code and the Adapter pattern to insert it into a project in progress.

BACKGROUND

PREVIOUS WORK

For the first step in this project, I researched other approaches to this problem, both to find a starting point and to evaluate the uniqueness of my approach.

PATTERNBOX



Design Pattern: Observer

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

FIGURE 1: PATTERNBOX SCREENSHOT

This tool can be downloaded from http://patternbox.com/

According to the web site:

"PatternBox is a design pattern editor for Eclipse. It creates Java classes and interfaces that can be customized depending on your application needs. … Using PatternBox you have freedom and flexibility to insert new pattern members whenever you want.
PatternBox offers a template based code generator. The generator makes intensive use of Eclipse's Java development tooling (JDT) and the Plug-in Development Environment (PDE). Both features must be available to run the plugin. The XML-based template mechanism makes it possible to define your own code templates. The current plug-in version includes 16 design patterns of the Gang of Four (GoF)."

This tool works by generating a special file to represent the pattern. Inside the file, choosing an element of a pattern brings up a way to choose the location of the element. Once the user completes this process, the tool generates sample code of that element of the pattern.

This differs from the Pattern Wizard by only creating code, never modifying it. Modifying code requires reading and parsing the input, which increases the complexity of the program.

DESIGN PATTERN AUTOMATION TOOLKIT

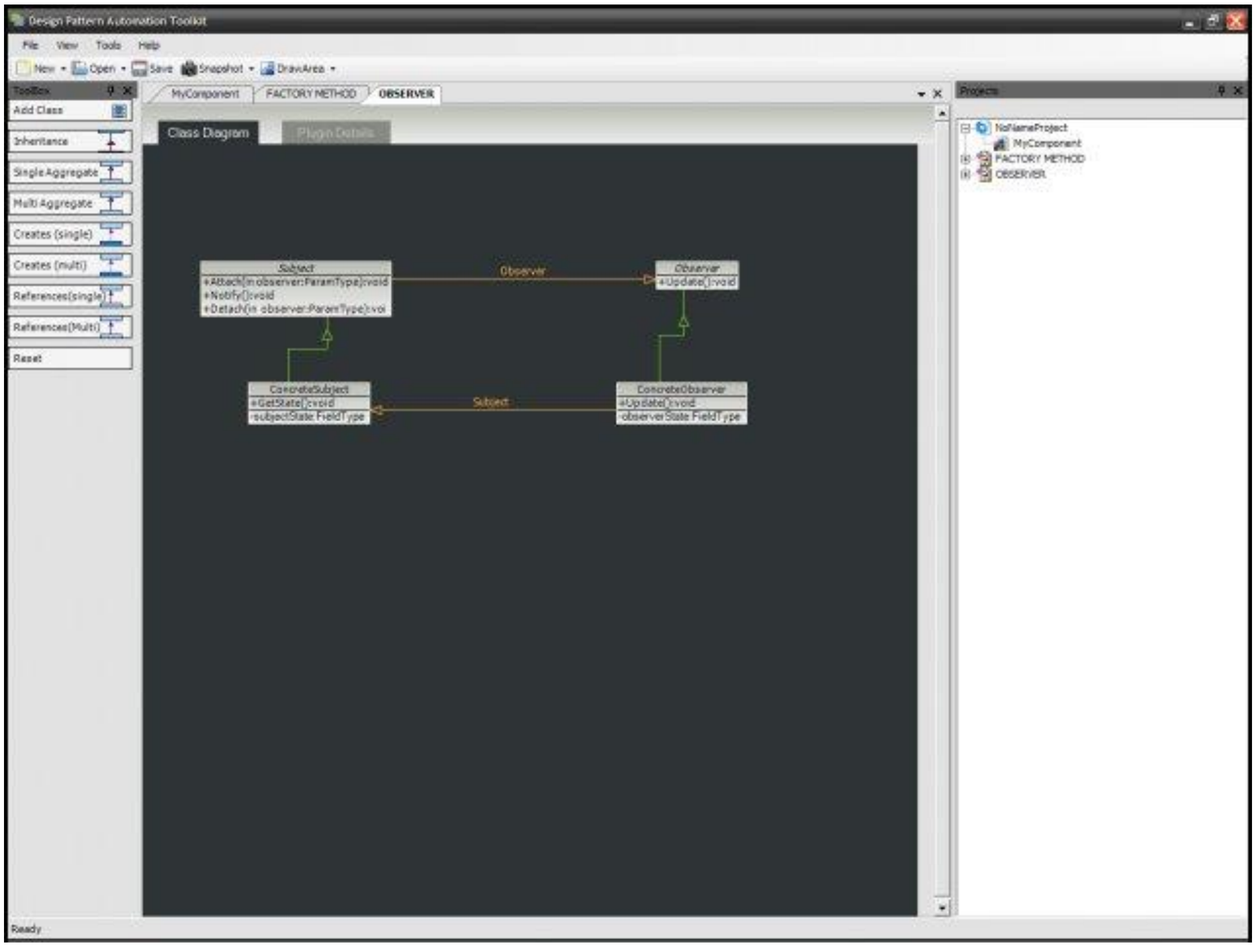


FIGURE 2: DPA TOOLKIT SCREENSHOT

This tool can be downloaded from http://dpatoolkit.sourceforge.net/.

According to the web site:

"DPAToolkit is a tool to help in software development via design patterns. The design can be visualized via class diagrams and design patterns can be incorporated into the design easily…Dpatoolkit comes with all the 23 Gang of Four design patterns which can be easily added in the design. Apart from these users can develop their own design pattern plugins and add them in their designs or share with others. The design patterns are saved in XML format…Dpatoolkit comes with the following code generators : C# , C++ , Java , VB.NET. New code generators can be easily plugged into the application."

This tool displays the use of design patterns in multiple languages—an idea I hoped I could implement with the Pattern Wizard. It also has a way to create new design patterns (other than the GoF patterns) from the client-side using a pattern description language. However, this tool does not modify existing code to fit the pattern, it only generates it.

ALPHAWORKS DESIGN PATTERN TOOLKIT

This tool can be downloaded from http://www.alphaworks.ibm.com/tech/dptk/download

This is an Eclipse plugin along the same lines as PatternBox. Again, this tool does not modify code, but it does more than generate a blank pattern template. From the web site:

"The pattern can be thought of as a transformation between the input application definition and the complete set of generated source artifacts. There are template tags that let you transform the application definition into an intermediate XML form and then navigate that XML to drive the generation of source artifacts. Each source artifact is generated as a result of applying the modified application definition to a specific pattern template. Each pattern template contains a set of static, or boilerplate, code interspersed with template tags that direct and define the merging of the application definition with the template."

DESIGN PATTERNS

ORIGINAL USE

Design patterns were originally meant as handy solutions in (physical) architecture. Christopher Alexander first published a book on the subject, describing useful patterns for designing and building. This included patterns to design an entire community, from farmland to shopping centers to places of worship and cemeteries (Alexander, 1977).

In addition, this book explained similar patterns to design groups of buildings, like a mansion or school; arrange rooms in a house according to their purpose; and landscaping outside the building (Alexander, 1977).

DESIGN PATTERNS IN COMPUTER SCIENCE

Alexander's work became useful in the growing field of computer science. A design pattern defines a general solution to a commonly occurring problem, written as a template or as a set of relationships. In computer science, this translates to a coding template or an object diagram representing a known solution to a common task.

Like algorithms, design patterns are a tried and accepted solution; as such, people develop and adapt them instead of discovering them. They both represent an approach to a problem more than the specific implementation of the solution. However, they differ in purpose: algorithms optimize the cost of a solution, while design patterns optimize its clarity.

Architectural patterns are also similar to standard design patterns; however, architectural patterns focus more on entire systems of programs as opposed to just one. This is a subjective difference; in some cases design patterns have been used as architectural patterns. They both look at the structure of code in the context of relationships. (Avgeriou, 2005)

CLASSIC DESIGN PATTERNS

The first pattern language in computer science was developed by Kent Beck and Ward Cunningham. It was small, containing only five patterns, but it was new at the time (Beck & Cunningham) .

Later pattern languages expanded the list to include more languages. One ambitious project listed and categorized 23 patterns, completed by a group who would become known as the Gang of Four: Erich Gamma, Richard Helm, Ralph Johnson, and Paul Vlissides. The patterns were categorized as:

- creational patterns; that is, patterns that deal with class instantiation,
- structural patterns; that deal with class and object composition, and
- behavioral patterns, that deal with communication between objects (Erich Gamma, 1995).

OTHERS

As an analogy, a book on pattern language is simply a dictionary. Software designers constantly add new patterns, just as speakers of a language constantly add new words. So, while the Gang of Four patterns remain the 'classic' patterns, far more exist. In particular, the GoF misses an entire category of patterns: concurrency patterns. These patterns offer solutions to scenarios involving multiple threads (Schmidt, Stal, Rohnert, & Buschmann, 1996).

LEPUS3

Continuing the analogy of a pattern language to a dictionary, the patterns needed a meaningful definition. LePUS3 is a design description language especially designed to model the classic design patterns. The language is flexible enough to also define other patterns, and similarly code not involved in any pattern.

The premise of the language represents each object with rectangles. The rectangle representing the object lists all the methods in that object. Dotted lines represent relationships such as 'calls', 'creates', and other soft connections. A solid arrow pointing in the direction of the parent represents inheritance (LePUS3 and ClassZ Reference).

ABOUT THE PATTERNS USED IN THE PATTERN WIZARD

ADAPTER

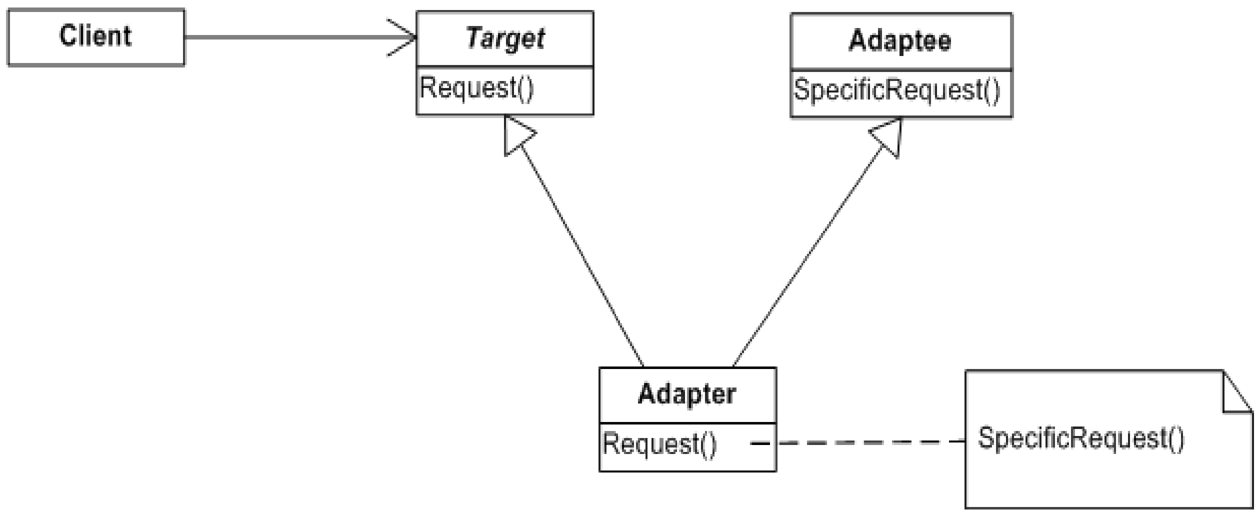


FIGURE 3: ADAPTER PATTERN (ADAPTER (CLASS) PATTERN)

The adapter pattern translates the interface of a class into one useful to the program. It breaks the connection between the calling method and the implementation. Practically, this would adapt code quickly to previously written classes. This diagram shows how to do this. The calling method (Client), would ordinarily directly make the SpecificRequest() in the Adaptee. Instead, it places a request through the Adapter, which determines the appropriate target and makes the request to them.

One example of an application of the Adapter is a weather tool that uses the Fahrenheit scale. Different functions on (Fahrenheit) temperatures are a separate class from the weather client. The adapter pattern works here to apply previously written code that returns temperatures in Celsius. Then, the weather client would call the general Temperature interface. If the appropriate method uses Fahrenheit, it simply returns the value of the desired method. If the appropriate method uses Celsius, it would convert the return value into Fahrenheit before returning it.

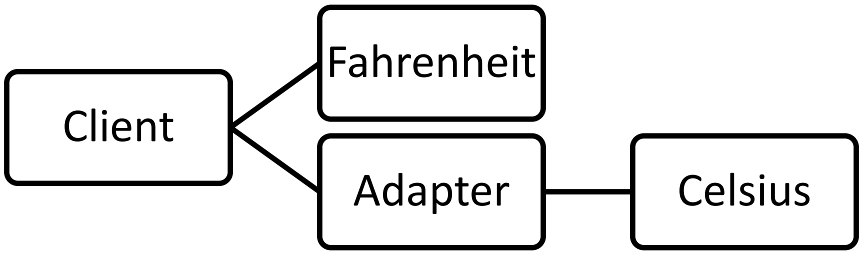


FIGURE 4: TEMPERATURE EXAMPLE

ABSTRACT FACTORY

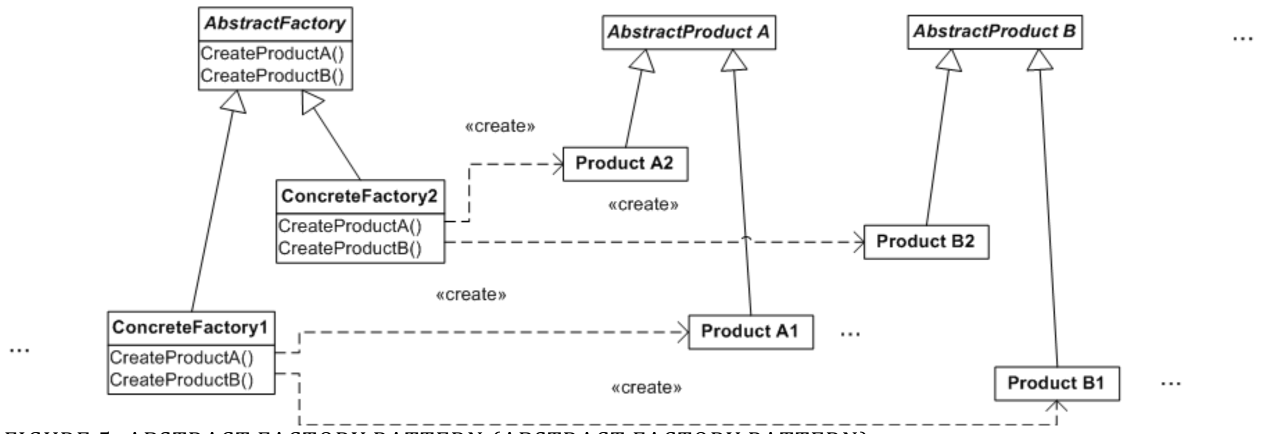


FIGURE 5: ABSTRACT FACTORY PATTERN (ABSTRACT FACTORY PATTERN)

The abstract factory pattern allows the creation of multiple objects with identical interfaces. Like the adapter pattern, it separates implementation details from their usage. However, this applies to objects specifically instead of methods.

The diagram shows the structure of this pattern. The AbstractFactory defines the objects that the user wants to create. The clients call on the AbstractFactory for ProductA or ProductB. Either ConcreteFactory1 or ConcreteFactory2 will act on that request, and return an AbstractProductA or

AbstractProductB. However, the pattern keeps the client unaware of which factory – and therefore which particular implementation – created the product.

For example, if the client needed a paper, it would call on a PaperFactory to produce it. Either EssayFactory or ResearchPaperFactory will actually create the Document, and then return it. However, the client has no knowledge of which factory created its paper.

OBSERVER

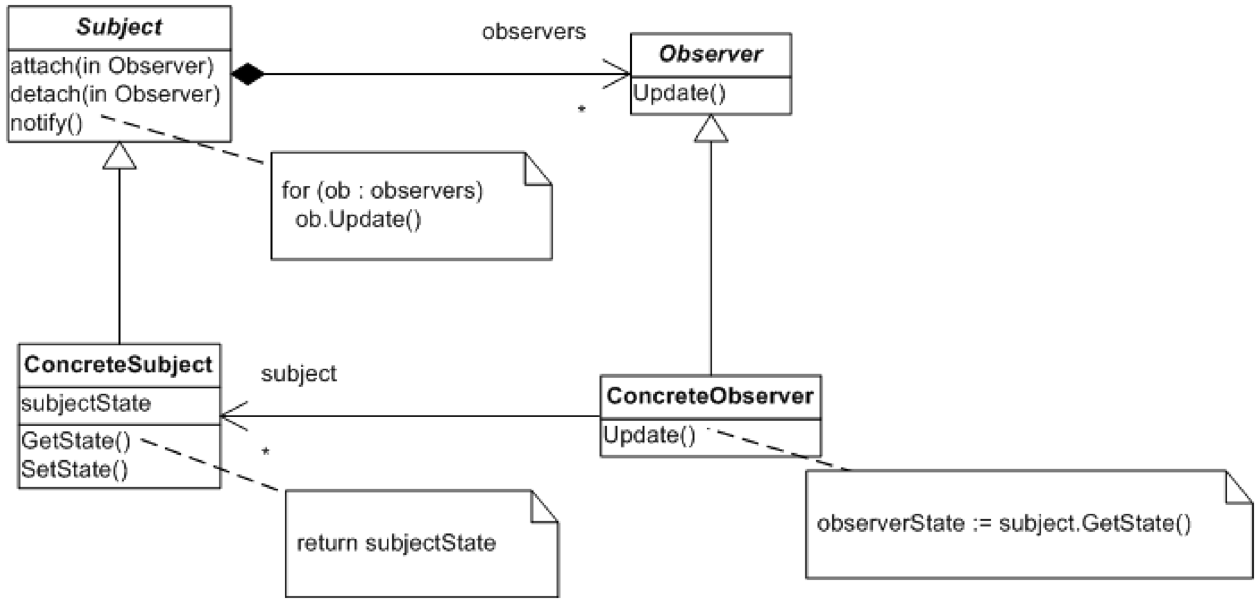


FIGURE 6: OBSERVER PATTERN (OBSERVER PATTERN)

This pattern works if some object needs to know about the state of another object. Continually polling the object wastes processor time, so the object waits for the other object to notify it. This pattern improves on that by allowing the notification to go out to many waiting objects at once.

As the diagram shows, the Subject has some way for the Observer to subscribe to it. In this case, the attach() method will perform the subscription. When an event occurs, the Subject starts to notify its list of subscribers. To notify a specific subscriber, it runs that subscriber's Update() method. The Update() method defines what the subscriber does after the event.

For example, assume a Bear object should eat Girl when the Girl's location equals 'woods'. To use this pattern, the Bear would subscribe to the Girl, and the Girl would notify all its subscribers, including the Bear, when its location changed. When the process reached the Bear, the Girl would call the Bear's Update() method, which would include the action of checking the Girl's location and, depending on the location, eating the Girl.

ABOUT ECLIPSE PLUGINS

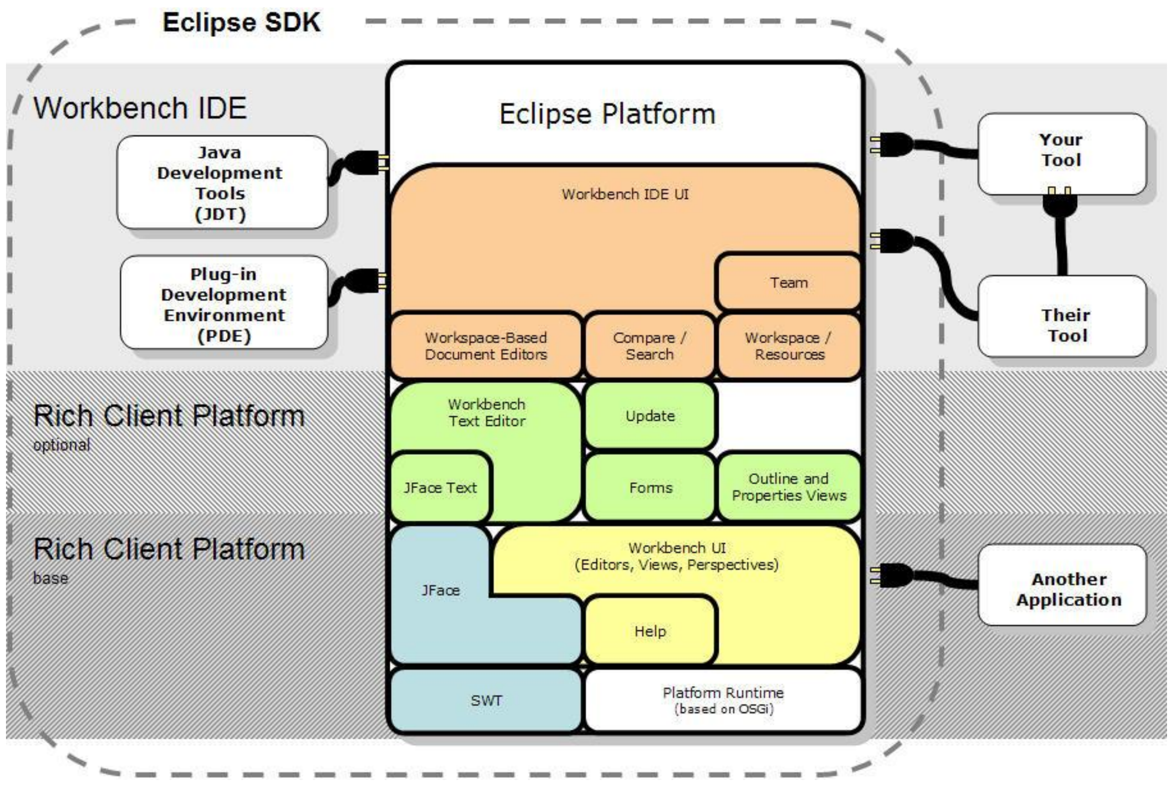


FIGURE 7: ECLIPSE ARCHITECTURE

Eclipse is a software development platform supporting Java. It is intriguing in the fact that it supports user-developed plugins to add functionality. Since the purpose of this project relates to adding functionality on an IDE level, this is an appropriate form for the Pattern Wizard.

Eclipse provides the extension points to support plugin development. Among these is the Wizard class, which generates a set of pages with prompts that the wizard displays if a particular action is completed. Eclipse also supports adding new types of file, visual input, and new menu actions.

Eclipse maintains information on its plugins through a manifest file specific to each plugin. This manifest file is stored in xml format in Eclipse's plugin folder. So, Eclipse has a list of available plugins at all timeswhich is referenced on starting Eclipse.

The manifest file stores information about each plugin, including a unique identifier and a list of extension points, which reference ways the user accesses the plugin's functionality. These can also be parameterized, to distinguish between user calls in different contexts (Bolour, 2003).

PATTERN WIZARD

For practical information on the Pattern Wizard, see Appendix A, the User Manual.

STRUCTURE

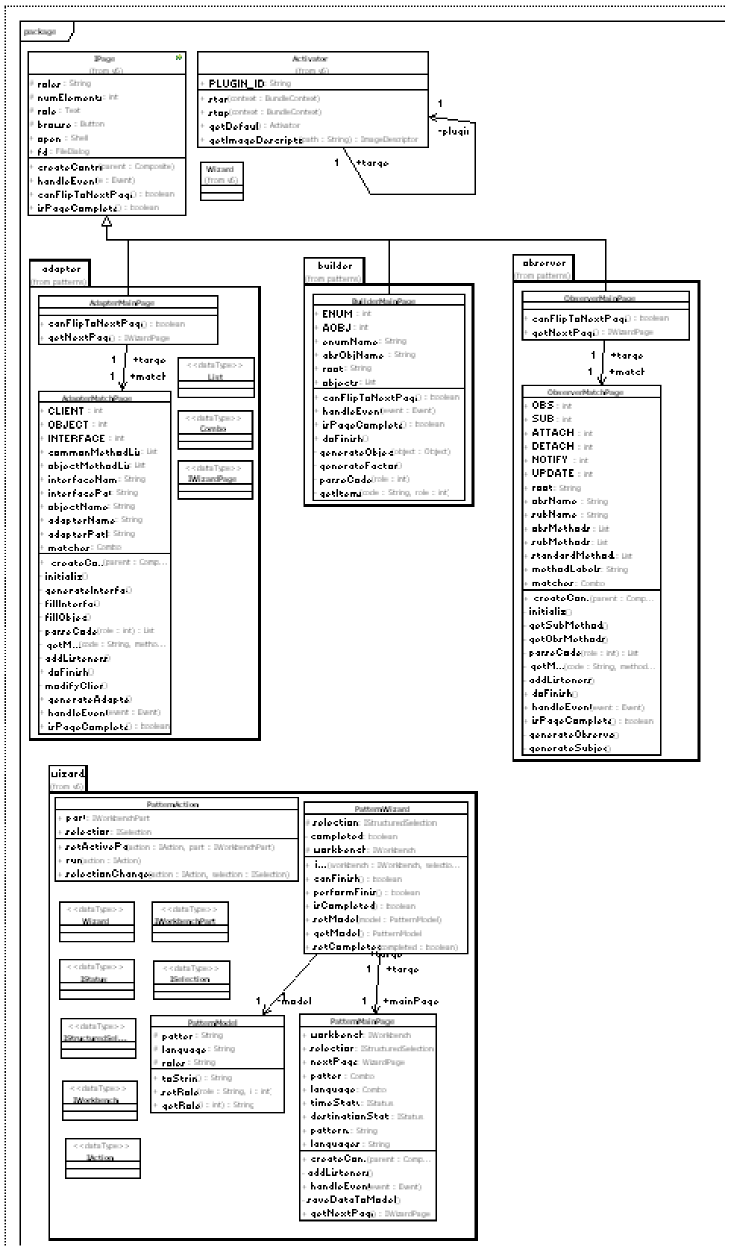


FIGURE 8: PATTERN WIZARD STRUCTURE

Control starts with the PatternWizard object. It passes control to the main page, which displays a choice of patterns and languages. Based on that input, the wizard generates a new page and presents it to the user on clicking the ‘next’ button.

This new page prompts the user to add files to the pattern, in different roles depending on the specific pattern. This page is named <Pattern>MainPage, where <Pattern> was the specific name chosen.

If the wizard requires more information to generate the pattern, control then passes to another page of the wizard. This page is named <Pattern>MatchPage, in the sense that the user matches information to the pattern.

In either case, the last page makes the required modifications to the code and returns control to the PatternWizard to close the tool.

HOW TO ADD MORE PATTERNS

To add a new pattern, consider what a beginner programmer would write in a situation where the pattern would be appropriate. With this in mind, write a document for what parts of the code would be used to implement the pattern. Also note whether any judgment calls would be required by the user.

Create a new package edu.wpi.patternmaker.<Version>.patterns.<Pattern Name>. In that package, create a <Pattern>MainPage object, which extends IPage. Depending on whether the pattern requires another page to take additional information, define:

- Int numElements, the number of source files required from the user,
- String roles[numElements], the labels for inputting the source code locations, and

- Text role[numElements], the locations of the source code

if there only needs to be one page and also

- <Pattern>NextPage nextpage

if the pattern needs more than one page.

You will also need to implement:

- canFlipToNextPage, indicating when enough information has been entered for the wizard to continue to the next page (false if there is no next page)
- handleEvent, checking whether the user fully completed the current page and updating the wizard
- isPageComplete, checking whether all of the required elements have been filled in (supports handleEvent and canFlipToNextPage, if the pattern uses multiple pages)
- doFinish, which parses the given code and rewrites it as the design pattern

and also

- getNextPage, which initializes the next page
- createControl, which describes the next page

if the pattern uses multiple pages

WHERE DOES IT GO FROM HERE?

Further study on this topic can progress in several directions. Researchers can increase the number of patterns supported by the Pattern Wizard. More information can be found below for instructions on how to do this. Additionally, other projects can cover the multi-paradigm approach, or the description language, or the pattern catalog.

ADDITIONAL INFORMATION

VERSION HISTORY

Version 1:	PatternBox source code
Version 2:	Generic Eclipse wizard
Version 3:	Retrieves code from user
Version 4:	Uses generic IPage to convert code
Version 5:	Parses user code to implement pattern
Version 6:	Parses user code to implement three representative patterns
Version 7:	Clean code and documentation

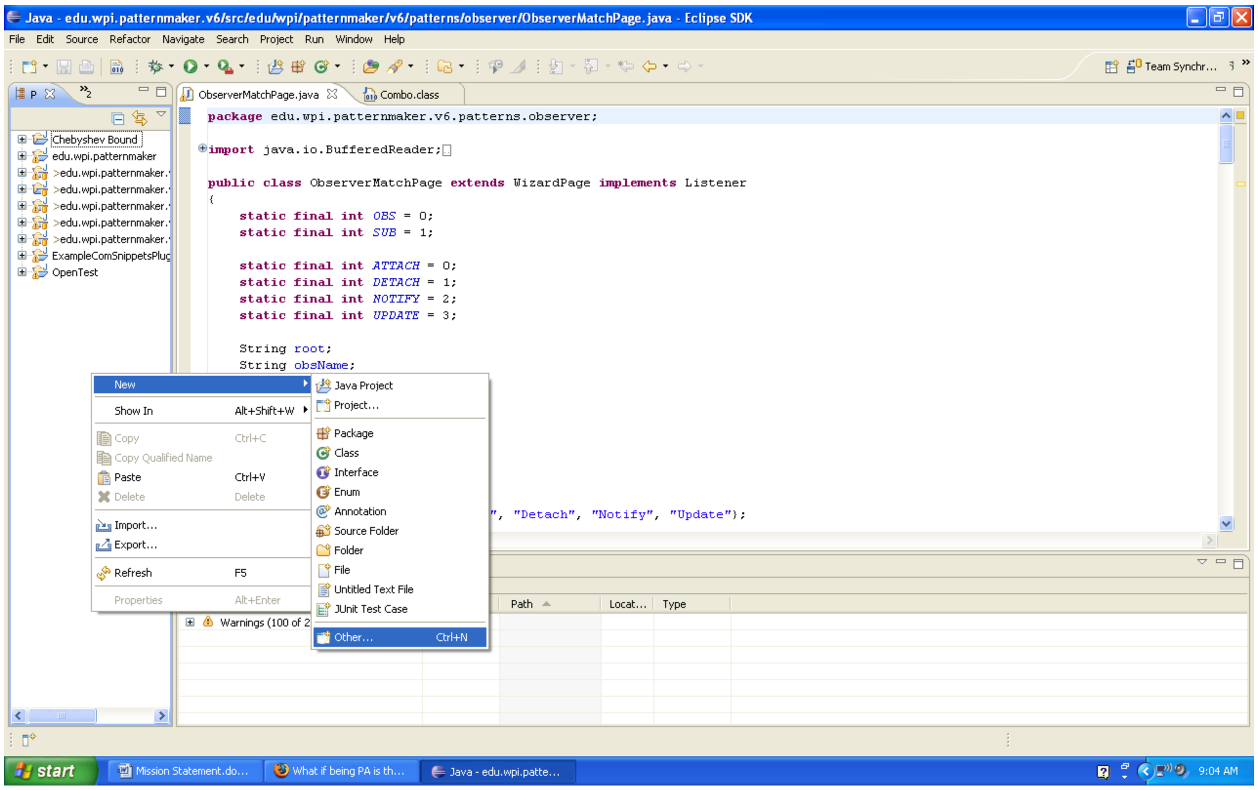
PROJECT TIMETABLE

September / October:	Research on Design Patterns and previous attempts
October:	Created versions of the Adapter pattern in all paradigms
	Researched code parsers
November:	Researched Eclipse plugins and how to make them
December/January:	Wrote Pattern Wizard framework
February:	Added Adapter pattern to Pattern Wizard
March:	Added Abstract Factory and Observer patterns

April:	Wrote report and documented tool

APPENDIX A: USER MANUAL

I designed this tool for use with Eclipse IDE. Once Eclipse loads, right-click in the File Explorer, then click New > Other. Alternatively, press Ctrl+N.



```
package edu.wpi.patternmaker.v6.patterns.observer;

import java.io.BufferedReader;

public class ObserverMatchPage extends WizardPage implements Listener
{
    static final int OBS = 0;
    static final int SUB = 1;

    static final int ATTACH = 0;
    static final int DETACH = 1;
    static final int NOTIFY = 2;
    static final int UPDATE = 3;

    String root;
    String obsName;

", "Detach", "Notify", "Update");
```

FIGURE 9: CLICK NEW > OTHER

When the New Dialog appears, click on Design Pattern > New Design Pattern. Then click Next.

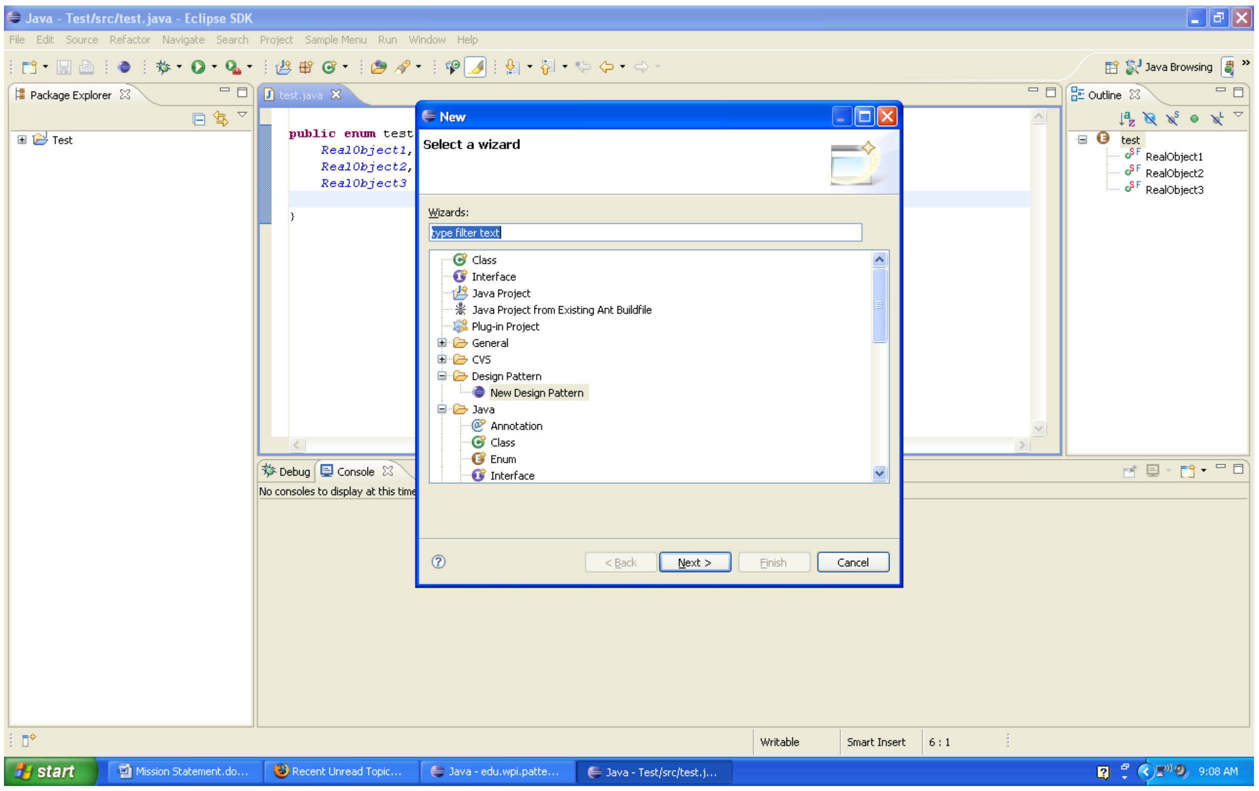
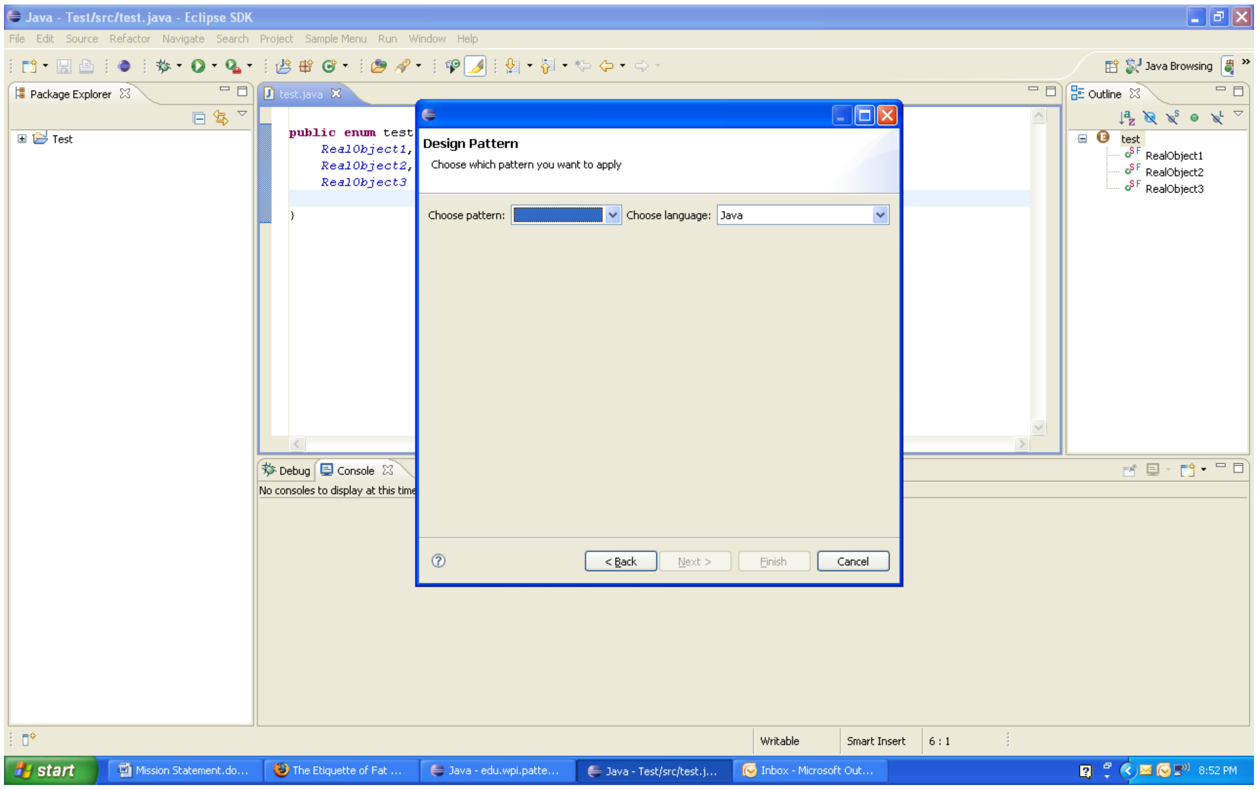

```
public enum test
    RealObject1,
    RealObject2,
    RealObject3

}
```

FIGURE 10: OPEN DESIGN PATTERN > NEW DESIGN PATTERN

The Design Pattern Wizard will appear. Choose the appropriate pattern and language and click 'Next'. Bear in mind that only Adapter, Abstract Factory, and Observer in the pattern selection and Java in the language selection will display the appropriate behavior.


Java - Test/src/test.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Sample Menu Run Window Help
Package Explorer
Test
test.java
Outline
Debug
Console
< Back
Next >
Finish
Cancel
Writable
Smart Insert
6 : 1
start
8:52 PM

```
public enum test
    RealObject1,
    RealObject2,
    RealObject3

}
```

Design Pattern

Choose which pattern you want to apply

Choose pattern: Choose language: Java

No consoles to display at this tim

FIGURE 11: SELECT PATTERN AND LANGUAGE

Next, the wizard will request pointers to previously existing code. Click ‘Browse’ to bring up a dialog. Navigate to the file containing the code, click on the file, and click ‘Next’. If the file is blank, the wizard will write a template for the pattern into that file. Otherwise, it will read in the code and make the appropriate changes. Some files are optional. When enough information has been gathered, either the ‘Next’ button or the ‘Finish’ button will be unblurred. Click either one to continue.

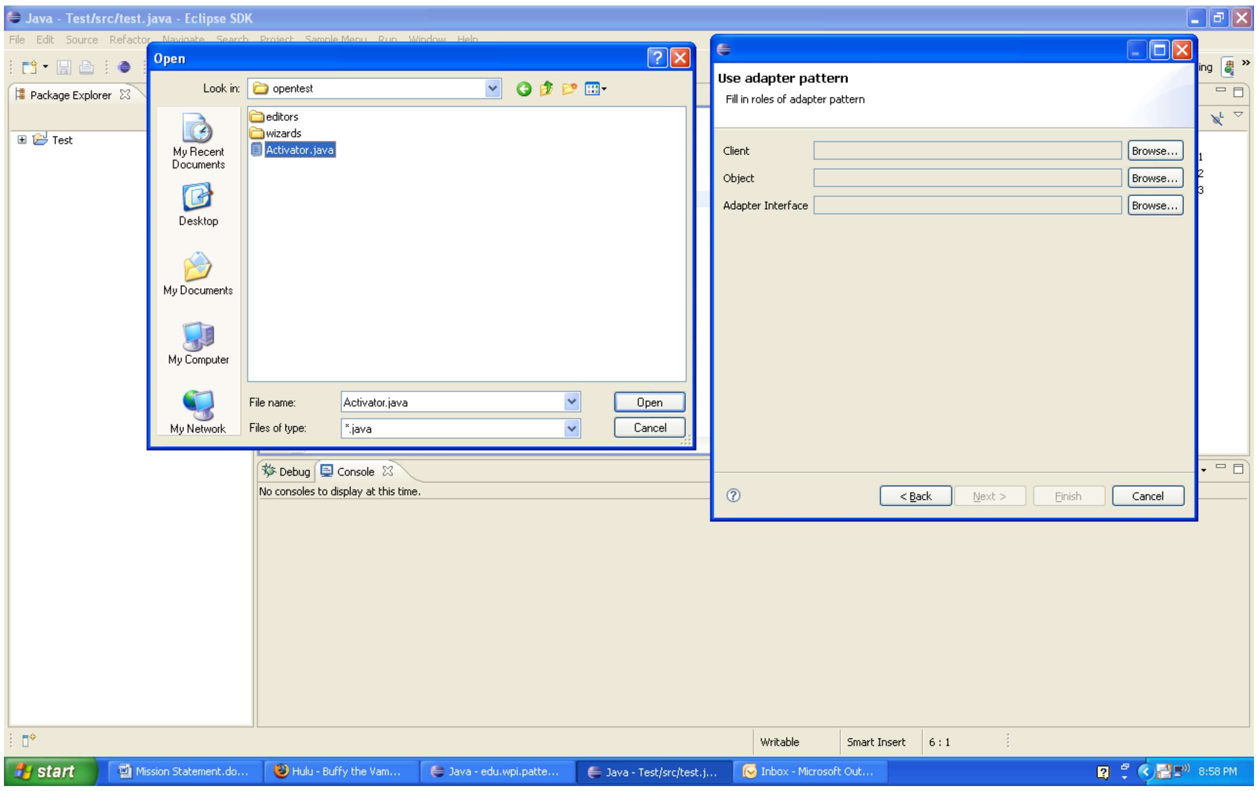


FIGURE 12: CHOOSE CODE FILES

Some patterns, such as the Adapter pattern, need to identify methods as well as classes. There will be another page in this case with a list of methods in one file, which will need to be matched to methods from another file. Do this by selecting the method from the drop menu, then click 'Finish'.

APPENDIX B: ADAPTER PATTERN USE CASES

CASE 1: INITIAL IMPLEMENTATION

```
Object
{
      do1(...) {...}
      do2(...) {...}
}
Client
{
      Object obj;
      obj.do1(...);
      obj.do2(...);
}
```

becomes

```
Object
{
      do1(...) {...}
      do2(...) {...}
}
iObject
{
      do1(...) {...}
      do2(...) {...}
}
```

```
Adapter implements iObject

{

      Object obj;

      Adapter(Object newObj) {

            obj = newObj;

      }

      do1(...) {

            obj.do1(...);

      }

      do2(...) {

            obj.do2(...);

      }

}

Client

{

      Object obj;

      Adapter adapter = new Adapter(obj);

      adapter.do1(…);

      adapter.do2(…);

}
```

CASE 2: USE EXISTING INTERFACE

```
NewObject

{

      makeOne(...) {...}

      makeTwo(...) {...}

      makeThree(…) {…}

}

Client

{

      NewObject obj;

      obj.makeOne(...);

      obj.makeTwo(...);

}

iObject

{

      do1(...) {...}

      do2(...) {...}

}
```

becomes

```
NewObject

{

      makeOne(...) {...}

      makeTwo(...) {...}

      makeThree(…) {…}

}

iObject
```

```
{
      do1(...) {...}
      do2(...) {...}
}
Adapter implements iObject
{
      Object obj;
      Adapter(Object newObj) {
            obj = newObj;
      }
      do1(...) {
            obj.makeOne(...);
      }
      do2(...) {
            obj.makeTwo(...);
      }
}
Client
{
      Object obj;
      Adapter adapter = new Adapter(obj);
      adapter.do1(…);
      adapter.do2(…);
}
```

APPENDIX C: ABSTRACT FACTORY USE CASE

```
class RealObject1 {
    private double dub = 8.5;
    public double do1() {
        return dub;
    }
}
class RealObject2 {
    private double dub = 10.5;
    public double do1() {
        return dub;
    }
}
class RealObject3 {
    private double dub = 11.5;
    public double do1() {
        return dub;
    }
}
class Main {
     public static void main (String args[]) {
        RealObject1 obj1 = new RealObject1();
        RealObject2 obj2 = new RealObject2();
        RealObject3 obj3 = new RealObject3();
    }
}
```

becomes

```
public abstract class AbstractObject {
    public abstract double do1();
}
class RealObject1 extends AbstractObject {
    private double dub = 8.5;
    public double do1() {
        return dub;
    }
}
class RealObject2 extends AbstractObject {
    private double dub = 10.5;
    public double do1() {
        return dub;
    }
}
class RealObject3 extends AbstractObject {
    private double dub = 11.5;
    public double do1() {
        return dub;
    }
}
public class AbstractFactory {
    public enum ObjectType {
        RealObject1,
        RealObject2,
        RealObject3
    }
    public static AbstractObject create (ObjectType type) {
        switch (type) {
            case RealObject1:
                return new RealObject1 ();
            case RealObject2:
                return new RealObject2();
            case RealObject3:
                return new RealObject3();
        }
        throw new IllegalArgumentException("The type " + type + " is not
recognized.");
    }
}
class Main {
     public static void main (String args[]) {
        for (ObjectType type : ObjectType.values()) {
            System.out.println(ObjectType + ": ″ +
ObjectFactory.create(type).do1());
        }
    }
}
```

APPENDIX D: OBSERVER USE CASE

```
interface Observer

{

	void update(Subject sub);

}

interface Subject

{

	void attach(Object o);

	Object detach(Object o);

	void notify();

}
```

becomes

```
import java.util.ArrayList;

interface Observer

{

	void update(Subject sub);

}

interface Subject

{

	void attach(Object o);

	Object detach(Object o);

	void notify();

}

public class RealObserver implements Observer
```

```
{
	private RealSubject sub;
	public RealObserver(RealSubject sub)
	{
		this.sub = sub;
		sub.attach(this);
	}
	public void update(Subject sub)
	{
		//TODO: React to change
	}
}
public class RealSubject implements Subject
{
	private ArrayList observers = new ArrayList();
	public void attach(Object o)
	{
		observers.add(o);
	}
	public Object detach(Object o)
	{
		observers.remove(o);
	}
	public Iterator iterator()
	{
		return list.iterator();
	}
	private void notify()

	{

		Iterator i = observers.iterator();

		while(i.hasNext())

		{

			Observer o = (Observer) i.next();

			o.update(this);

		}

	}

}
```

BIBLIOGRAPHY

Abstract Factory Pattern. (n.d.). Retrieved March 19, 2009, from The LePUS3 and ClassZ Companion to the GoF: http://www.lepus.org.uk/ref/companion/AbstractFactory.xml

Adapter (Class) Pattern. (n.d.). Retrieved March 19, 2009, from The LePUS3 and ClassZ Companion to the GoF: http://www.lepus.org.uk/ref/companion/Adapter(Class).xml

Alexander, C. (1977). *A Pattern Language.* Oxford University Press.

Avgeriou, P. (2005). *Architectural Patterns Revisited.*

Beck, K., & Cunningham, W. (1987). *Using Pattern Languages for Object-Oriented Programs.* OOPSLA-87.

Bolour, A. (2003, July 3). *Notes on the Eclipse Plugin Architecture*. Retrieved April 20, 2009, from Eclipse Corner: http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html

Gamma, E., et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading: Addison-Wesley.

LePUS3 and ClassZ Reference. (n.d.). Retrieved April 20, 2009, from LePUS3 and ClassZ: http://www.lepus.org.uk/about.xml

Observer Pattern. (n.d.). Retrieved March 19, 2009, from The LePUS3 and Class-Z Companion to the GoF: http://www.lepus.org.uk/ref/companion/Observer.xml

PatternBox. (2009, February 8). Retrieved April 23, 2009, from PatternBox: http://patternbox.com/

Schmidt, D. C., Stal, M., Rohnert, H., & Buschmann, F. (1996). *Pattern-Oriented Software Architecture.* Wiley.