**Software Design Patterns**

*by Deeksha Juneja, Kayler Renslow, and Nick Robbins*

**Introduction**

Design patterns are templates or re-usable solutions for commonly occurring software problems. These templates help make the code object-oriented and easy to understand and read. They can be used in any programming language, however throughout this description, Java will be used as the primary language to help describe in detail what a design pattern is. The design patterns are divided into three classes namely - structural, creational, and behavioral. Structural patterns recognize a relationship between entities, creational patterns create “objects” in a manner which works well with the situation, and behavioral patterns recognize common types of communication between entities and increase the flexibility in communication (Bauisa, 2010, p.1). These subclasses contain various design patterns. Each of the design patterns can be a model solution for a variety of problems, usually belonging to the same subclass. Some of the common design patterns are the observer pattern, the decorator pattern, and the adapter pattern. The decorator pattern will be the one that will be used as an example later in this document, which falls under the structural pattern category. Each design pattern can be further described in the following categories: structure, participants, implementation, and collaboration.

It is important for students taking 300+ level classes in software related fields to be familiar with design patterns so that they don’t struggle in their classes. Hence, this document is aimed towards individuals that have taken basic programming courses and will be taking higher level of programming courses. At Iowa State University, these basic courses include COMS 227, COMS 228, and SE/Cpre 185.

**Discussion**

To begin with, design patterns are nothing but an implementation of object oriented principles in a standardized manner. To relate design patterns to the real world, let’s consider an airplane. When building an airplane, the manufacturer gathers different parts of the airplane. Those different parts are complex “objects” themselves and each part might have been manufactured by different companies. These objects are similar to the “objects” in Java. The airplane manufacturer doesn't bother too much with the design or construction of those smaller objects, but rather concentrates on using those objects to construct the airplane. Similarly, when creating a program in Java and using objects of different classes, one doesn’t need to understand the structure of those classes to use them in their program. Subsequently, the airplane manufacturer builds the airplane based on some requirements and a design. Similarly, software engineers develop software based on the requirements of the project. They face some commonly occurring design problems when developing software. Some of these problems include: making the software object oriented, manageable, testable, expandable, and reusable. Having faced these design problems for years, engineers developed some common solutions known as the design patterns (Shubho, 2010, p.1-2). These solutions are well tested and hence don’t need to be tested again by the person using these design solutions.

To illustrate design patterns and enhance the understanding of the user, the method implemented by this paper would be to first describe the problem and then find a design solution for that problem. This paper aims to go through a general view of design patterns to a specific implementation. Hence, the first part of the paper will focus on an example from day-to-day life and then go to a specific example using the Decorator Design Pattern.

One of the most significant examples of a good design are the arrangement of the electric equipment in a house. A light bulb can be easily changed without replacing the switch corresponding to that light bulb. Similarly, the switch of the fan can easily be changed without changing the fan itself. Moreover, the connection for the fan and light bulb can be interchanged without replacing the their individual switches. In short, it is very important that even though everything is connected to each other, the connection should be made in a way that the replacement or change of one component does not affect other components of the system. To facilitate this kind of connection, it is important for things to be “loosely coupled”, which means that the connection should not be direct but via a bridge or an interface (Shubho, 2010, p. 2). Having gone through the basic problem, this is a good time to start on the structure analysis of the problem.

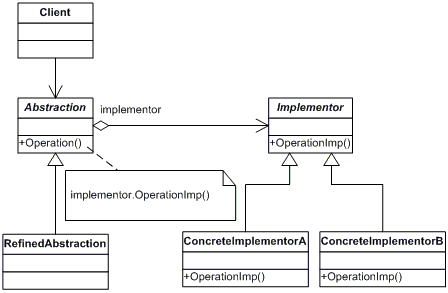
**Structure**

The structure of a design pattern is a graphical view of the process followed by it. Included in the structure are the individual components of the system as well as how the entities interact with each other.

In Figure 1, each block represents a component of the system. If two objects influence each other, they are connected. The structure of the design pattern is not meant to provide great detail on how the system operates. It should only give a brief outline on how each major part of the system operates. The three types of design patterns (structural, creational, and behavioral) will all have different structures. A structural design pattern will have its structure focused around the relationships between objects. A creational design pattern’s structure will focus on creating objects suitable for the user. Lastly, the behavioral design pattern’s structure will focus on the communication synchronization between entities. All of these structures include multiple objects which are also called participants.

To illustrate a structure via an example, the problem (Switch and Electrical Equipment) described above is used. The problem mentions switches and there are many different types of switches - simple switches, fancy switches, etc. But at the end of the day, they are all switches. For instance, all switches have a basic feature of turning off and on. The problem also has other electrical components called fans and lights. These components don’t have a common design but have common functionality such as all electrical equipment can be turned on and off (Shubho, 2010, p.2-6).

As discussed in the example, an interface or bridge is required to connect the switches and electrical equipments. This type of bridge or interface building is called the Bridge Design Pattern. The technical definition of a Bridge Design Pattern is “Decouple an abstraction from its implementation so that the two can vary independently” (Gamma, 1995, p. 151) which will be explained through the example in the upcoming parts of the document.



**Figure 1: Bridge pattern structure.**

**Participants**

All of the classes and objects used throughout a design pattern are the participants in that design solution. These are essentially the building blocks that make up the design pattern’s structure. Specifically, these building blocks include the classes and interfaces that the input data is run through. Participants will not vary by a large amount between structural, creational, and behavioral design patterns. The differences come mainly from the collaboration and implementation parts of a design pattern.

Referring to the example described above, a class of switches will be constructed, which implements the turning on and off function. Now to implement, fancy switch - another class will be made which will use or “extend” the switch class. The problem also has other electrical components called fans and lights. An interface for the electrical equipment can be made which will be implemented by specific classes of fans and lights. Lights and fans are different equipments with different designs but similar functionality. Hence, an interface is made instead of a class (Shubho, 2010, p. 2-7).

**Collaboration**

Collaboration refers to how classes and objects used in the pattern interact with each other. The way these classes are used varies greatly by the type of design pattern that is used. In a structural pattern, independent classes (classes that don’t relate to one another) implement interfaces that have no relation as well. Moreover, the code API also refers to the interfaces. This is commonly known as a hierarchy composition, which differs from inheritance because with inheritance, a root class or interface is used in the API. Each class in the inheritance tree is then related, rather than being independent from each other.

A behavioral pattern’s class will instead be interested in how each class uses another. A change will occur if it is recognized that one class is in need of another. Lastly, creational design patterns use classes the way the name suggests - they create instances based on the current needs of the class.

Going back to the example above, an instance of electrical equipment needs to be added to the switch class to create a bridge between the switch and electrical component. Now any number of fan or light classes can be made by implementing the electrical equipment interface. The type of switch and electrical equipment can easily be changed as well, using encapsulation (Shubho, 2010, p. 4-7).

**Implementation**

Implementation refers to the coding of the classes and interfaces used in the pattern. The coding would strictly adhere to the structural diagram, except any number of implementor classes can be added. As the implementation of each design pattern is based on the structural diagram, the implementation for structural, behavioral and creational design patterns will be based on that diagram. The collaboration or the relation between the classes would also be based on the structural diagram as well. It is also important to notice that any number of methods can easily be added to each class or interface. It is important to follow the structural diagram because as mentioned before, it is well tested and changes in it would require additional testing by the user.

Referring to the structural diagram for the example above (Figure 1), the abstraction component in the diagram would be equal to the Switch class, RefinedAbstraction is for a specific Switch class, say the normalSwitch, the implementor would be the Electrical Equipment, and finally the ConcreteImplementorA or ConcreteImplementorB could be a Light or Fan class. An encapsulated instance of the ElectricalEquipment interface would be made in the Switch class. Lastly, the client class can easily use the switch and electrical equipment however the client wishes through making encapsulated instances of Electrical Equipment and Switch (Shubho, 2010, p. 7).

**Decorator Pattern Example**

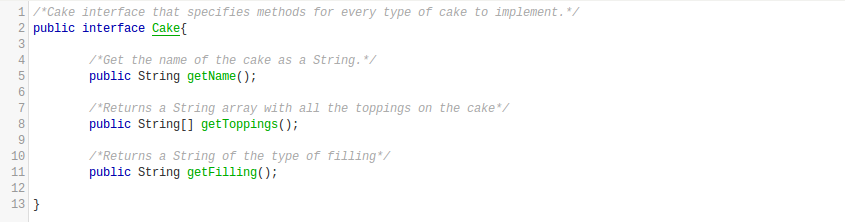
When it costs millions of dollars to get a project done, time is very limited, so older code can’t be rewritten. The decorator pattern is one way around rewriting old code. A technical description of Decorator Pattern would be “A Decorator Pattern is a structural design pattern which enables us to add new or additional behavior to an object during runtime, depending on the situation.” When the goal of the project is to make “extended functions” applied to a specific new instance and still be able to create an instance of the original functions which don’t have the characteristics of the new function, the decorator pattern is used. It can combine multiple decorators for one instance (Bauisa, 2010, p.10). Code being restructured that still follows the previous standard will be demonstrated throughout this example.

To show an example implementation of the decorator pattern, this document will use a sample implementation of cake. This implementation has 3 classes and 1 interface. The cake decorator pattern sample code is made to be as simple and objective as possible to avoid confusion. As mentioned before, this example uses Java as the programming language, but decorator patterns and design patterns themselves are not limited to Java.

Now the document will go into detail about the interface and classes to describe their implementation.

**Participants**

**Cake Interface**



**Figure 2: Code for Cake interface.**

Figure 2 is the Java code for the interface **Cake** which has methods getName(), getToppings(), and getFilling(). As described in the comments, the **Cake** interface is what specifies the minimal implementation every type of cake; every subclass of **Cake** must implement the methods specified to be classified as a cake.

The method getName() is used to return a String that is a human friendly name of the cake. With getToppings(), it will return a String[] (String array) of every topping the cake has. Finally, getFilling() will return a String with the type of filling the cake has.

With just these few methods, we can describe any type of cake. A real implementation of a cake interface can possibly have double the amount of methods, but once again, this code sample is supposed to be brief and to the point.

**ChocolateCake Class**



**Figure 3: Code for ChocolateCake class.**

The **ChocolateCake** class is a sample implementation of a cake based upon **Cake**’s method requirements. The annotation @Override means, the method will override/replace the implementation of the superclass. In this case, the superclass of **ChocolateCake** is the interface **Cake**. It is important to remember that interfaces have no implementation and they are just method stubs. Overriding on class implementations will be demonstrated later in the code sample.

The method getName() for the class **ChocolateCake** simply returns a String literal “chocolate cake”. The method getToppings() returns an empty String[] since this chocolate cake doesn’t have toppings. The final method, getFilling(), returns the String literal “chocolate”, which means the chocolate cake has a chocolate filling.

For this class, we are making the fields filling and toppings final because they should never change. If the fields were not final, the implementation would still work, but there needs to be clear intentions that a cake’s filling and toppings doesn’t change (they’re constant). It can easily be asserted that a cake’s toppings can change in the real world, but with the current **Cake** interface, there is no specification that a cake’s toppings can be removed.

**CakeDecorator Class**



**Figure 4: Code for CakeDecorator class.**

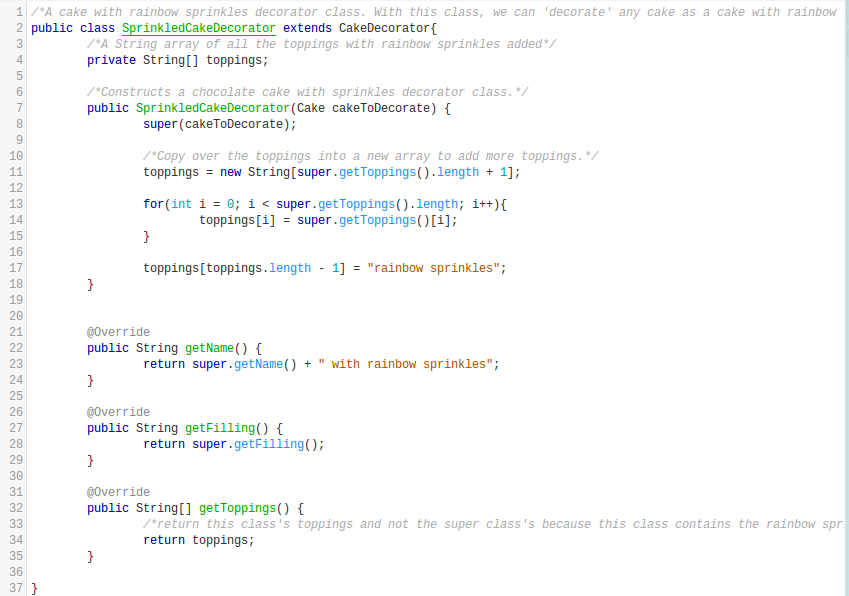
The **CakeDecorator** class is the decorator class for **Cake** interface implementations. Using the **CakeDecorator**’s constructor, we can pass in any instance of **Cake** and store it. The field cakeToDecorate is declared protected and not private so that subclasses have access to it.

Notice that every method in **CakeDecorator** is returning what the field cakeToDecorate would return. Since every method does it, it is ensured that **CakeDecorator**’s implementation doesn’t hide cakeToDecorate’s implementation. Afterall, the decorator class is used as a wrapper for **Cake** instances and doesn’t represent an actual type of cake.

More important characteristics to notice is **CakeDecorator** implements **Cake** and **CakeDecorator** is abstract. **CakeDecorator** is made abstract because it is never allowed to be instantiated. Since **CakeDecorator** implements **Cake**, **CakeDecorator** instances can be stored in variables of the type **Cake** without casting.

Moreover, any implementation of **CakeDecorator** can be passed into another **CakeDecorator** instance. A decorated cake can be decorated again essentially. For example, a chocolate cake can be decorated with sprinkles with one implementation of **CakeDecorator** and then decorated again with chocolate syrup with an entirely new and different implementation of **CakeDecorator**.

**SprinkledCakeDecorator Class**



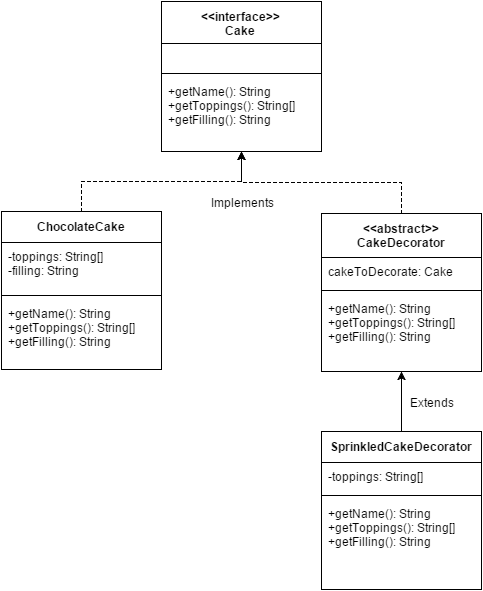
**Figure 5: Code for SprinkledCakeDecorator class.**

The class **SprinkledCakeDecorator** is used to decorate cakes with rainbow sprinkles. Notice that **SprinkledCakeDecorator** extends **CakeDecorator**. This means that every instance of **SprinkledCakeDecorator** is of the type **CakeDecorator** and **Cake**. It also means that **SprinkledCakeDecorator** must call **CakeDecorator**’s constructor via super.

After super(cakeToDecorate) is invoked, the String literal “rainbow sprinkles” is appended to toppings. A new String[] with size equal to the previous toppings array size +1 is used. Since it is undetermined whether or not toppings from the superclass is final or not, a new array is used.

The method getName() returns the superclass’s name with the String literal “ with rainbow sprinkles” appended to it. Finally, getToppings() returns the new toppings array which has rainbow sprinkles.

**Collaboration**



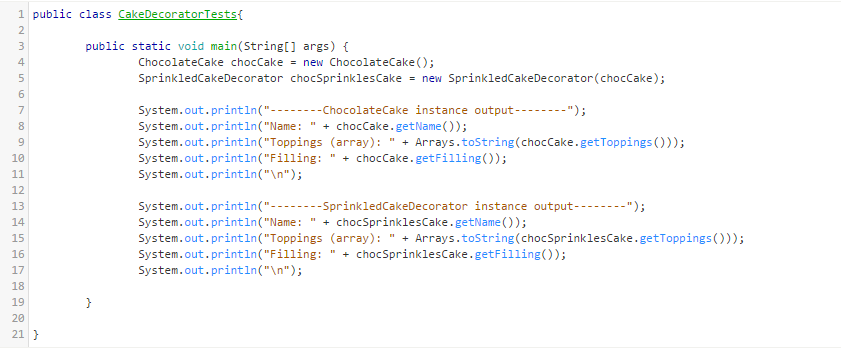
**Figure 6: Diagram of inheritance for cake example.**

Above is the class inheritance diagram. This diagram shows how **SprinkledCakeDecorator** is of type **CakeDecorator** and the interface **Cake**. This diagram also shows how methods are being overwritten. **SprinkledCakeDecorator** is overriding **CakeDecorator**’s methods for example. This diagram also shows how easy it is to build upon previous interfaces or classes, without changing any original code.

In other implementations of a decorator, entirely new methods can be added to the decorator classes. Some situations won’t allow the base interfaces to be modified, and so a decorator pattern becomes very useful. In this example, it isn’t necessary. All that needed to be done was add sprinkles to the toppings array.

**Implementation**

Now since the code part is laid out, a sample implementation of the cake decorator pattern can be created. Below is the Java code and then the direct console output.



**Figure 7: Code for testing the cake example.**

Console output:

|  |
| --- |
| --------ChocolateCake instance output--------  Name: chocolate cake  Toppings (array): []  Filling: chocolate  --------SprinkledCakeDecorator instance output--------  Name: chocolate cake with rainbow sprinkles  Toppings (array): [rainbow sprinkles]  Filling: chocolate |

The **ChocolateCake** instance outputs the name of “chocolate cake”, an empty array since it has no toppings, and the filling of “chocolate”. The **SprinkledCakeDecorator** instance outputs the name of “chocolate cake with rainbow sprinkles”, toppings array with one element: “rainbow sprinkles”, and a filling of “chocolate”. Notice how the same instance of chocolate cake is used for both cases, but running getToppings() on the decorator instance adds on “rainbow sprinkles”.

**Conclusion**

To summarize the document, design patterns are structured solutions to commonly occurring design problems. The knowledge of design patterns is essential for people in software related fields to assist them in creating manageable software. Design patterns also saves the time a coder spends on testing because they are already well tested. This document goes from a general view of design patterns to a specific implementation of a design pattern. The general view explained the components of a design pattern using the bridge pattern as an example. Furthermore, a specific implementation of the decorator pattern using cake was demonstrated in the document to help students taking 300+ level classes to get a better understanding of design patterns. In the end, it is important to remember that every problem can be broken down to fit into a design pattern.

**Sources**

Bautista, N. (2010, July 7). A Beginner’s Guide to Design Patterns. Retrieved August 6,

2015, from http://code.tutsplus.com/articles/a-beginners-guide-to-design-patterns

--net-12752

Computerworld's Smart Salary Tool 2015. (2015). Retrieved September 20, 2015, from

http://www.computerworld.com/salarysurvey/tool/2015

Derek Banas. (2012, August 19). Design Pattern Video Tutorial [Video File]. Retrieved

from https://www.youtube.com/watch?v=vNHpsC5ng\_E&list=PLF206E90

6175C7E07&index=1

Fowler, M., & Beck, K. (1999). Refactoring: Improving the design of existing code.

Reading, MA: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995): Design Patterns (p. 151).

Addison-Wesley

Mitra, S. (2015, August 31). Design Patterns. Lecture presented at Coms 319 in

Lagomarcino Hall, Ames.

Shalloway, A., & Trott, J. (2010). Design patterns explained: A new perspective on

object oriented design (2.nd ed.). Boston, Mass. [u.a.: Addison-Wesley].

Shubho, A. (2010, August 9). How I explained Design Patterns to my wife: Part 1.

Retrieved September 6, 2015 from http://www.codeproject.com/Articles/98598/

How-I-explained-Design-Patterns-to-my-wife-Part

Shvets, A. (n.d.). Design Patterns (G. Frey, Ed.). Retrieved from

https://sourcemaking.com/design\_patterns

Figure 1: Bridge pattern structure (Captured by Shubho, 2010)

Figure 2: Code for Cake interface (Captured by Kayler Renslow, 2015)

Figure 3: Code for ChocolateCake class (Captured by Kayler Renslow, 2015)

Figure 4: Code for CakeDecorator class (Captured by Kayler Renslow, 2015)

Figure 5: Code for SprinkledCakeDecorator class (Captured by Kayler Renslow, 2015)

Figure 6: Diagram of inheritance for cake example (Captured by Kayler Renslow, 2015)

Figure 7: Code for testing the cake example (Captured by Kayler Renslow, 2015)