TO: *Prof. David Green*
FROM: *Peter M. Corcoran, pmcorcor*
DATE: *November 22, 2015*
COURSE: *EE 433*
SUBJECT: *Design Patterns*

SIX DESIGN PATTERNS THAT EVERY DEVELOPER SHOULD KNOW WITH JAVA EXAMPLES

Developers face solving problems that occur over and over again. Often times solution to design problems are easily found using Internet search engines like Google or Bing. The ease of finding a solution to a problem is largely depended on how often the problem repeats itself. The repetitive nature of problems give rise to the patterns in solution designs. The most common problems faced will have a variety of solutions, and often developers find agreement on which solution offers the best outcomes. These design patterns allow developers to create consistent programs which meet the demands, and purpose for which they are created.

Design Patterns have been defined for a variety of problems, and therefore have a variety of purposes. In the early 90s four software developers decided to capture some of the most basic patterns they were seeing in software design. They created "*Design Patterns: Elements of Reusable Object-Oriented Software*" (also referred to as the Gang of Four or GoF book). The four authors separated out patterns based on the patterns purpose. They wrote, "We classify design patterns by two criteria. The first criterion, called purpose, reflects what a pattern does. Patterns can have either creational, structural, or behavioral purpose. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility. (Gamma, et al. 1995)." The authors went on to cover 23 design patterns, broken into three purpose driven categories: Creational, Structural, and Behavioral.

The Design Patterns book provides 23 patterns that should be studied by every developer. However, the purpose of this paper is to produce only six patterns all developers should know. It's important to note that there are complex patterns which are built using basic design patterns. Complex patterns are used in large software systems rather than reinventing individual software components. Enterprise software often takes advantage of complex patterns to decrease time-to-market, lower expenses or track value savings through reuse. Although complex patterns are useful in enterprise software systems, they require a higher level of understanding and have more complex composition. Many enterprise patterns are composed of basic design patterns defined in GoF book. Because of the complexity and composition of these patterns, they should be excluded from list of six patterns all developers should know. The list should focus on simple constructs that form the foundation of a developers understanding of design patterns. Therefore, it reasons that since the material covered in the GoF book focuses on common object-oriented design problems and the solution to those problems, then patterns selected should be limited to those defined in the GoF book.

Further, this paper assumes the reader has an understanding of important programming concepts and principles. Specifically, separation of concerns, single-responsibility, open/closed, Liskov substitution, interface segregation, and dependency inversion principles.

Each of the following sections have the same format: description, structure, sample code, and justification of the design pattern. The description section will give an overview of the pattern, what problem the pattern solves, and why it is used. The structure section generically defines the patterns in its basic form. Sample code will contain Java example implementation code of the discussed pattern and hosted on GitHub (https://github.com/corcoranp/ee433/tree/master/src/ee433) (AllAppLabs n.d.). Lastly, the justification section provides the reasoning why the pattern was chosen to be included in "*Six Design Patterns That Every Developer Should Know with Java Examples*".

CREATIONAL PATTERNS

Creational patterns are concerned with the creation of objects in a program. Many higher level languages provide a mechanism for creating objects and accomplish the task using the "new" operator. Although the operator is provided to developers, object creation is not always best completed using it. There are times when object type is not known until runtime, or when a single object is required throughout the life cycle of the application. Creational patterns exist to address the instances when the 'new' operator should not be used, or used in a limited fashion. The two patterns that every developer should know from the creational patterns are Abstract Factory, and Singleton.

Abstract Factory

Description

The Abstract Factory defines a way to create a family of interrelated or dependent objects without hard coding the specific type of the objects upfront. Clients calling the factory are isolated from the concrete class's implementation, but can interact with the object through the interface exposed by the abstract. Figure 1 shows the class structure of the Abstract Factory. Note the client is the class that receives the benefits of the pattern being described. Multiple concrete factories implement the Abstract Factory class to form a suite of objects that the Client can take advantage.

Structure

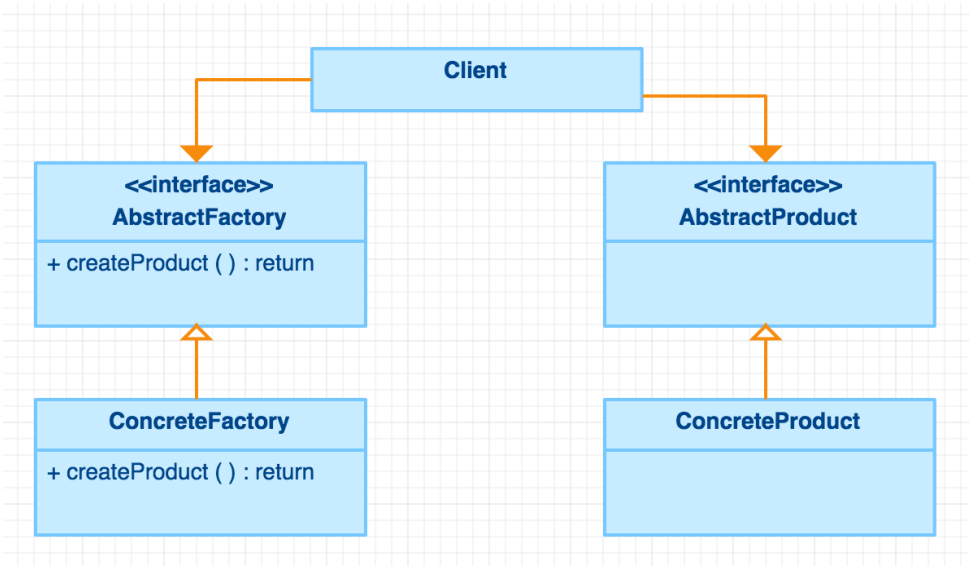

Figure 1. Abstract Factory Class Structure

Sample Code

Suppose the client (from Figure 1) was an object that displays animal sounds. The client does not concern itself with which specific animal is used, only has the responsibility of creating the animal and displaying the sound. The first class definition to create is the Abstract Animal Factory, which defines the method for creating animals.

```
public abstract class AnimalFactory {
   public abstract Animal createAnimal();
}
```

The return type for the "createAnimal" method is an abstract product called "Animal", which has 2 properties: name, and sound.

```
public abstract class Animal {
   //Animal Name
   public String name;

   //What sound the animal makes
   public String sound;
}
```

The Abstract Product would be *an* abstract animal, with the concrete product being the *specific* animal. Multiple animals can be defined and given names, and sounds, for the example two animal sub-classes are used a cat:

```
public class Cat extends Animal {
   public Cat(){
      this.name = "cat";
      this.sound = "meow";
   }
}
```

and a Dog:

```
public class Dog extends Animal {
   public Dog(){
      //default dog information
      this.name = "dog";
      this.sound = "bark";
   }
}
```

How these animal types are created and used is governed by specific factories that are sub-classes of the AnimalFactory class. In the implementation of the animal can be changed based on the factory that is creating the animal. For example, generic dog and cat factories are created that use the default implementation of the classes, however, a new factory for "Bull dog" is created to change the way the dog object is implemented.

Cat Animal Factory:

```
public class CatFactory extends AnimalFactory {
   @Override
   public Animal createAnimal() {
      return new Cat();
   }
```

```
}
```

Dog Animal Factory

```
public class Dog extends Animal {
   public Dog(){
      //default dog information
      this.name = "dog";
      this.sound = "bark";
   }
}
```

Bull Dog Animal Factory

```
public class BullDogFactory extends AnimalFactory {
   @Override
   public Animal createAnimal() {
      Dog d = new Dog();
      d.name = "Bull Dog";
      d.sound = "BARK!";
      return d;
   }
}
```

The Abstract Animal Factory is referenced by the Client but concrete animal factories are passed into the Client and used to create specific instances of animals. The client can then create the object it needs at runtime without knowing the specific type being created.

```
public class AbstractFactoryClient {
   public AnimalFactory animalFactory;

   public AbstractFactoryClient (AnimalFactory factory){
      animalFactory = factory;
   }

   public void theAnimalSays(){
      Animal a = animalFactory.createAnimal();
      System.out.println("The " + a.name + " says " + a.sound);
   }
}
```

When the client is created a specific factory is passed in, and the client object is able to create object of varying type, but still achieve the intended result.

```
   BullDogFactory bf = new BullDogFactory();
   AbstractFactoryClient afClient1 = new AbstractFactoryClient(bf);
   afClient1.theAnimalSays();
```

Justification

The pattern makes replacing specific objects, that interact with each other in a unique way, with another set of objects, which has a different implementation but expose the same interface, relatively easy. Abstract Factory is a vital pattern that every developer understands and is able to use. It helps when integrating system together or when decoupling is necessary between the creating object and the specific object being created.

Singleton

Description

The Singleton defines a way to ensure only one object is instantiated, and provides a global reference to the object so it can be used by other classes. Often there are times when creating the same object does not make sense or you have a requirement to track an object's state over the life cycle of the application no matter where the object is used.

Structure

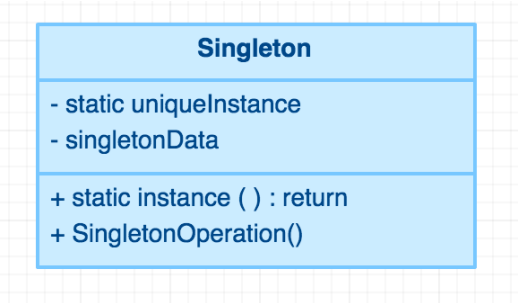

Figure 2. Singleton Class Structure.

Sample Code

It may not seem that a Singleton is a creational pattern, since no new objects are being created, but because it controls the number of objects being created it can be considered a creational pattern. The Singleton is made up of a unique instance of itself, some functions to perform on itself (SingletonOperation method), a way for other classes or objects to access it (getInstance method), and it has a private constructor (`private Singleton()`). An example of a generic singleton is provided in the next code snippet.

```
public class Singleton {
   private static Singleton uniqueInstance = new Singleton();
   private String singletonData = "Some Data";

   // Private constructor
   private Singleton(){ }

   public static Singleton getInstance(){
      return uniqueInstance;
```

```
    }

    public String SingletonOperation(String data){
        singletonData = singletonData + data;
        return singletonData ;
    }
}
```

An example of when to use a singleton would be for a keeping score in a game application. Multiple objects are able to generate points that are added or subtracted from one score object. The next code snippet uses the generic singleton pattern to define a score class that exposes a method for getting an instance of the class, and adding points to the score.

```
public class Score {
    private static Score uniqueInstance = new Score();
    private static Integer _score = 0;

    // Private constructor
    private Score(){ }

    public static Score getInstance(){
        return uniqueInstance;
    }

    public Integer addPoints(Integer points){
        _score += points;
        return _score ;
    }

    public Integer getScore(){
        return _score;
    }
}
```

Putting the example together, we have two objects that get the singleton instance and then modify the data stored in that object. The example shows that even though two different objects are used, only the one singleton is modified. The second example is taking principles of the singleton and applying them to a "score" object. Multiple objects are created by getting the reference to the score object, and new points are added to the score. Only one score is updated.

```
write("---------------------------------------------------------");
write("     Singleton Example");
write("---------------------------------------------------------");
write("");
Singleton s1 = Singleton.getInstance();
Singleton s2 = Singleton.getInstance();
write("Singleton data: " + s1.SingletonOperation(""));
write("Modify data for first reference: " + s1.SingletonOperation(" s1 "));
write("Modify data for second reference: " + s2.SingletonOperation(" s2 "));
```

```
write("");
write("Singleton Score Example");

Score scr1 = Score.getInstance();
write("Add point: " + scr1.addPoints(100).toString());
Score scr2 = Score.getInstance();
write("Add more point: " + scr2.addPoints(100).toString());
```

The results of the above code are:

```
------------------------------------------------------------
     Singleton Example
------------------------------------------------------------

Singleton data: Some Data
Modify data for first reference: Some Data s1
Modify data for second reference: Some Data s1  s2

Singleton Score Example
Add point: 100
Add more point: 200
```

Justification

Singletons make the list of the six design patterns all developers should know because of its flexibility in solving a common problem in software ranging from games to enterprise systems. By controlling the number of instances that exists of an object (to one), the pattern controls how clients access it. If a developer faces a problem that requires changing from a singleton to an instance based object, the change easier to accomplish with a singleton than with a static class or global variable. Singleton patterns help with memory management since only a single object reference is present in memory. This strategy helps with high velocity systems because it cuts down on the number of objects stored in memory as well as saving time during object creation.

STRUCTURAL PATTERNS

Where creational patterns are concerned with object creation, structural patterns concern themselves with how object and classes are combined to form new structures that meet the needs of the system. In a similar way to how wood beams and nails are used to create framed houses, structural patterns combine classes or objects to create larger "frames" or structures. Class inheritance is the most basic combining of two or more classes to form one composite class. There are two structural patterns that make the list of six patterns every developer should know, and they are the Adapter, and Composite patterns.

Adapter

Description

Just as an electrical adapter takes AC voltage and convert it into DC voltage for electronic devices, the Adapter structural pattern takes a class of a certain type (Adapted) and adapts it into a form that a Client can use. Adapters allow two classes with incompatible interfaces to work together.

Structure

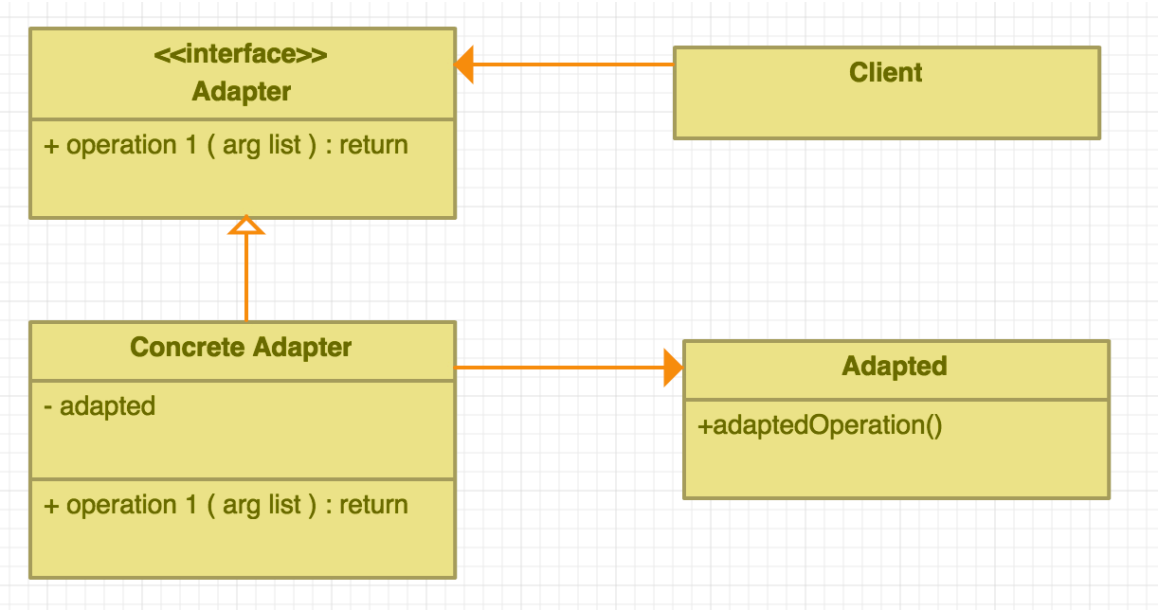

Figure 3. Adapter Structural Pattern.

Sample Code

Taking our initial example of the electrical power adapter and implementing it into code, first define an AC Socket as the Adapted object. Given an AC socket return approximately 120V AC, the object will return exactly 120V for simplicity.

```
public class ACSocket {
   public Integer getOutput(){
      return 120;
```

```
    }
}
```

Since the electrical device takes a DC input, an adapter will be needed to take the 120V signal and convert it into DC. So a client can be created, we use an adapter interface to define what to expect form a concrete DC adapter.

```
public interface IVoltageAdapter {
    public Integer getOutput();
}
```

With the interface created, the client class can be built which is expecting 5V DC.

```
public class DC5VClient {
    private IVoltageAdapter powerAdapter;

    public DC5VClient(IVoltageAdapter adapter){
        this.powerAdapter = adapter;
    }

    public String getPower(){
        Integer voltage = powerAdapter.getOutput();
        if (voltage > 5){
            return "Voltage too high!!";
        }
        if (voltage < 5){
            return "Voltage too low!!";
        }
        return "5V received from adapter";
    }
}
```

The concrete adapter is then coded to convert the AC Socket's 120V AC output to 5V DC that the DC5VClient is expecting. The concrete adapter implements the IVotageAdapter interface which has one method called "getOutput" which returns an integer output. The method in the example takes the 120V AC converts it to DC, then divides it down to the expected 5V. (The example does with with hard-coded values for the sake of simplicity.)

```
public class VoltageAdapter5V implements IVoltageAdapter {
    public static ACSocket adapted;

    public VoltageAdapter5V(ACSocket ac){
        adapted = ac;
    }

    @Override
    public Integer getOutput() {
        //Illustrates conversion of some value to make it
        //compatible for the calling client.
        Double d = (adapted.getOutput() * 0.636); //AC to DC
```

```
        Double divider = d / 15.2; // Hard coded voltage divider
        Integer output = divider.intValue(); // should be 5 volts...
        return output;
    }

}
```

When put together in an example program, an AC socket is created by using the 'new' operator. A 5-volt adapter is created which takes the AC socket as a parameter. A DC 5V client which references the DC adapter is created as well. The client calls its "get power" method which uses the adapter to get convert the AC voltage to DC voltage that the client needs.

```
write("------------------------------------------------------------");
write("     Adapter Example");
write("------------------------------------------------------------");
write("");
write("Create AC Socket");
ACSocket ac = new ACSocket();
write("Create 5V adapter ");
VoltageAdapter5V va = new VoltageAdapter5V(ac);
write("Plug adapter into client");
DC5VClient client = new DC5VClient(va);
write(client.getPower());
write("------------------------------------------------------------");
```

When the above code is run, the result is:

```
------------------------------------------------------------
     Adapter Example
------------------------------------------------------------

Create AC Socket
Create 5V adapter
Plug adapter into client
5V received from adapter
------------------------------------------------------------
```

Justification

The Adapter pattern is a key addition to the list of patterns every developer should know because of its role domain-specific interfacing. The separation-of-concerns principles encourages separating domain specific knowledge into separate components. Often these components do not support direct integration into non-domain objects, as a result, adapters are used to connect domains at the domain's boundaries. With the rise of service-oriented-architectures connecting loosely coupled system with adapters are common place in enterprise type systems. Knowledge of adapters is a fundamental for developers involved in integrating systems.

Composite

Description

Composite objects create hierarchal relationships between itself and other composite objects of the same type. They are tree like structures that allow clients to treat single objects or groups of the same object uniformly. Composites are made up of a component object, leaf object, and the interface that represents both. A simple composite structure to recognize is a file system object. File system objects have components (directories) that contain other file system objects (files or directories). The structure in Figure 4 shows the basic object relationships of the composite objects.

Structure

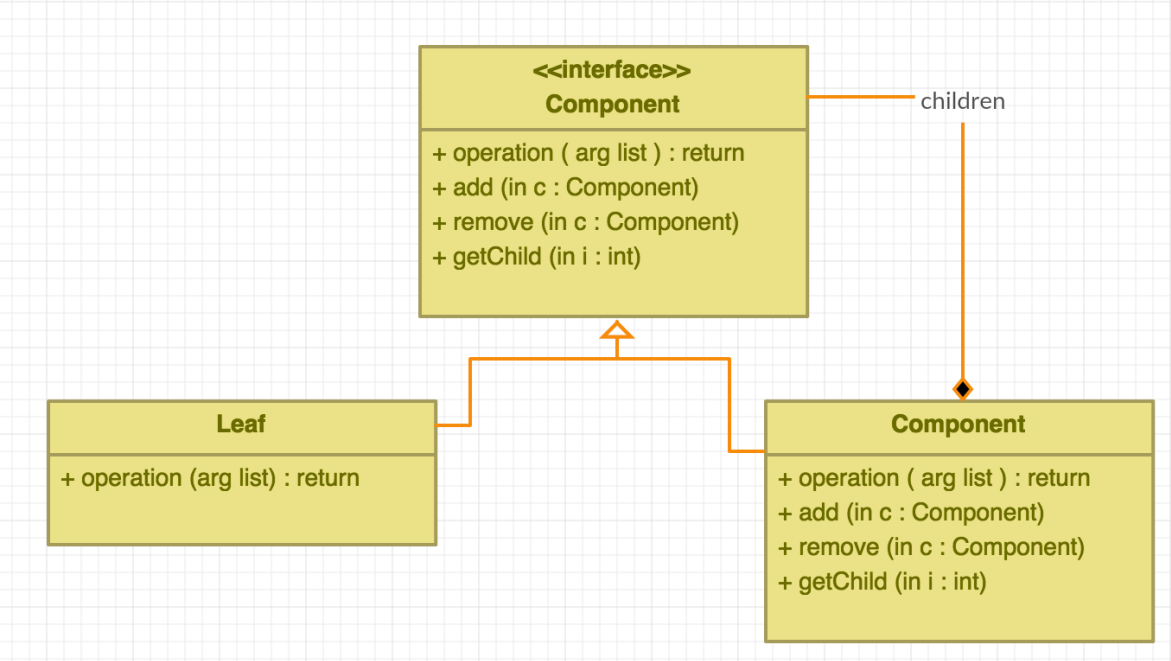

Figure 4. Composite Structural Pattern.

Sample Code

Using the file system object example, an interface is defined for a file system object. The interface should define a file system object name, the ability to add/remove other file system objects, and a method for enumerating through the object's children. IFileSystemObject below defines this interface.

```
public interface IFileSystemObject {
   public String getFileName();
   public void add(IFileSystemObject fso);
   public void remove(IFileSystemObject fso);
   public IFileSystemObject getChild(Integer i);
}
```

Next, the concrete file system object is defined for the "leaf" type objects. Leafs are objects that do not contain other bojects so do not require implementation of all the methods in the file system interface.

```
public class File implements IFileSystemObject {
   public  String name = "File";
   public File(String filename){
      name = filename;
   }
   @Override
   public String getFileName() {
      // TODO Auto-generated method stub
      return name;
   }

   ...
}
```

Directories are file system objects that can contain other file system objects. By using the file system object interface, the object is able to add/remove other objects that implement the same interface, namely files and other directories. The class definition for a directory contains logic that implements each of the interface's methods.

```
import java.util.Vector;

public class Directory implements IFileSystemObject {
   public String name = "directory";
   public Vector<IFileSystemObject> subitems = new
Vector<IFileSystemObject>();

   @Override
   public String getFileName() {
      return name;
   }

   @Override
   public void add(IFileSystemObject fso) {
      subitems.add(fso);
   }

   @Override
   public void remove(IFileSystemObject fso) {
      subitems.remove(fso);
   }

   @Override
   public IFileSystemObject getChild(Integer i) {
      return subitems.get(i);
   }
}
```

With the components of the example completed, a recursive method for writing out the path of the object is created.

```
public static void writeFso(IFileSystemObject fso, String path){
     Directory root = (Directory) fso;
     Vector<IFileSystemObject> list = root.subitems;

     if (list == null) return;

     for ( IFileSystemObject f :  root.subitems) {
         if ( f instanceof Directory ) {
          write(path + f.getFileName() );
              writeFso(f, path  + f.getFileName()+ "\\");
                          }
         else {
              write(path + f.getFileName() );
         }
     }
   }
```

Finally, the composite example code to demonstrate how a composite class is created.

```
write("-------------------------------------------------------------");
write("     Composite Example");
write("-------------------------------------------------------------");
write("");

Directory root = new Directory();
root.name = "root";
write("Create root directory: " + root.name);

write("Create file1, add to root");
File f1 = new File("file1");
root.add(f1);
write("Create directory 2");
Directory dir2 = new Directory();
dir2.name = "directory2";
write("Create file2, add to directory 2");
dir2.add(new File("file2"));
root.add(dir2);
write("Recursively walk the root object...");
writeFso(root, "root\\");
```

The demonstration shows the creation of each object, and the recursive crawling of the objects in the composite.

```
-------------------------------------------------------------
```

```
     Composite Example
-------------------------------------------------------------

Create root directory: root
Create file1, add to root
Create directory 2
Create file2, add to directory 2
Recursively walk the root object...
root\file1
root\directory2
root\directory2\file2
```

Justification

The composite structural pattern is included in the list of patterns every developer should know because of how common the pattern is used in almost all object-oriented systems. According to the GoF book, the composite pattern is used in many toolkits, frameworks, and higher level patterns like Model/View/Controller. Composites are highly flexible and useful in data aggregation. Developers must have ways to relate objects, or create hierarchies, composites provide a mechanism accomplishing these common challenges.

BEHAVIOR PATTERNS

"Behavioral patterns are those which are concerned with interactions between the objects. The interactions between the objects should be such that they are talking to each other and still are loosely coupled. The loose coupling is the key to n-tier architectures. In this, the implementation and the client should be loosely coupled in order to avoid hard-coding and dependencies.'

Observer

Description

An observer, like its suggests, is an object that watches for a change to occur before taking action. Observer patterns create a one-to-many dependency between objects so when the subject being observed changes, the observer can take action. Observers help in object-oriented systems by providing a mechanism in which objects can communicate state-changes efficiently to the objects needing to be notified.

Structure

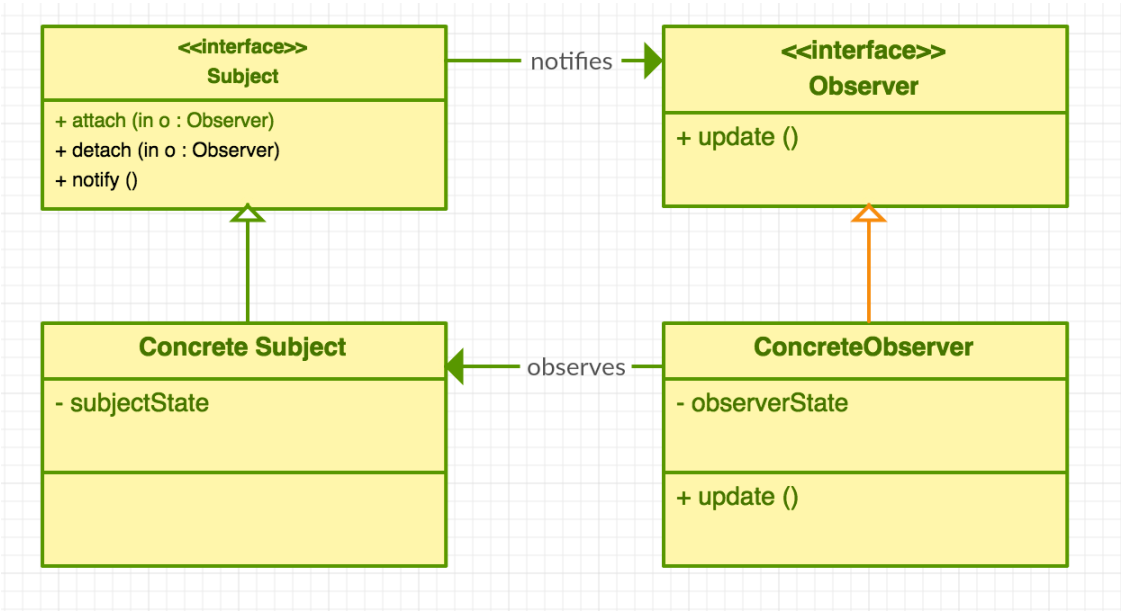

Figure 5. Observer Behavior Pattern.

Sample Code

An observer pattern is made of four items, an interface for both the subject and the observer, and a concrete for both as well. The subject is the object being watched, and the observer is the object that takes action when the subject changes in some way. The pattern is simple in implementation so a generic sample will be used to demonstrate the pattern.

The observer only has to implement a method that should be executed upon some update in the subject's state. Therefore, a simple 'update' method will be used as the utility for updating the observer.

```
public interface IObserver {
   public void update(ISubject subject, Object arg);
}
```

The Subject exposes methods for adding and removing observers that will need to be notified upon state change.

```
public interface ISubject {
   public void attach(IObserver o);
   public void detach(IObserver o);
   public void notifyObservers();
}
```

The concrete implementation of the IObserver interface writes out what subject triggered the update.

```
public class Observer implements IObserver {
   public String name = "";
   @Override
   public void update(ISubject subject, Object arg) {
      System.out.println(name + " was updated from : " + arg);
   }
}
```

The Subject code is more complex because it implements the methods for adding and removing observers in a collection. In the sample code, an ArrayList of IObservers is used to track the observer objects that need to be notified when changes occur. The notifyObservers method is used to cycle through the collection and execute the 'update' method on each one. Other implementations of this pattern will sometime disconnect the subject and observer using a centralized broker (which is a more complex pattern and therefore not covered).

```
public class Subject implements ISubject {
   private ArrayList<IObserver> observers = new ArrayList<IObserver>();
   public String name = "name";
   @Override
   public void attach(IObserver o) {
      observers.add(o);
   }

   @Override
   public void detach(IObserver o) {
      observers.remove(o);
   }

   @Override
   public void notifyObservers() {
      for(IObserver o : observers){
         o.update(this, name);
```

```
        }
    }
}
```

Sample code to demonstrate the observer pattern is created which a single subject is created, and multiple watchers are created and bound to the subject. Other subjects could be created and bound to the concrete observers. This creates a seemingly many-to-many relationship between the subject and observers.

```
write("-------------------------------------------------------------");
write("     Observer Example");
write("-------------------------------------------------------------");

Subject sub = new Subject();
sub.name = "subject1";

Observer watcher = new Observer();
watcher.name = "watcher";
Observer watcher1 = new Observer();
watcher1.name = "watcher1";
Observer watcher2 = new Observer();
watcher2.name = "watcher2";
sub.attach(watcher);
sub.attach(watcher1);
sub.attach(watcher2);
sub.notifyObservers();
write("-------------------------------------------------------------");
```

As a result of the above code, the each water was updated from the single subject is was registered with.

```
-------------------------------------------------------------
     Observer Example
-------------------------------------------------------------

watcher was updated from : subject1
watcher1 was updated from : subject1
watcher2 was updated from : subject1
-------------------------------------------------------------
```

Justification

Observers are added to the list of patterns every developer should know because in object-oriented programming object state changes occur very often, and some mechanism must exist to track changes, and take action. The observer pattern is a critical pattern to understand for inter-object communication. Many event based systems rely on observer patterns, and a number of complex patterns build on the observer pattern to create more useful programs.

Command

Description

The command pattern encapsulates the process used to make a request, or instruct an object what to do next. The pattern is often used in menu based systems where view elements interactions result in command execution. Web services are often viewed as command like interfaces in enterprise systems. A client calls an invoker which knows of the commands interface. After the command executes, another object receives the benefit, or takes action based on the command execution. The interface provided to invoker means concrete command classes can be substituted which allows for flexibility and separation in the design.

Structure

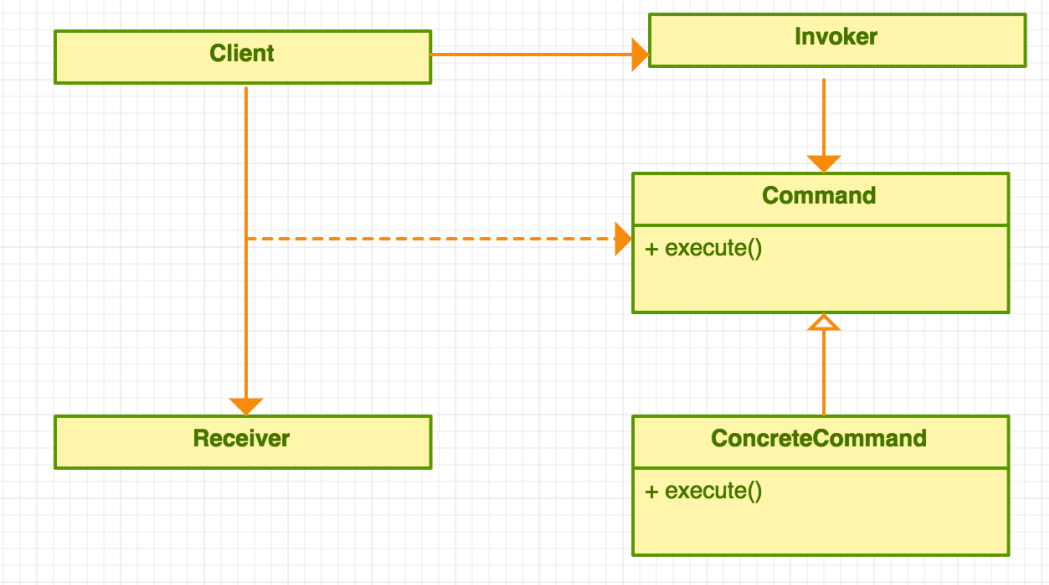

Figure 6. Command Behavior Pattern.

Sample Code

A basic example of a command pattern is a switching mechanism for a computer. Computers require a sequence of events to occur prior to being shut completely off from power. Therefore, when a switch is tripped, a more elegant power off system needs to take effect. The same is true with the powering on of a computer. For the sample code, two commands will be created: Computer On Command, and Computer Off Command. First, a generic interface for the commands is created.

```
public abstract class Command {
   public abstract void execute();
}
```

Since the system exists to power off a computer, an object should be defined for a computer with two methods: turn off and turn on. These methods define the process for turning on/off the computer.

```
public class Computer {
    //...
    public void turnOff(){
        System.out.println("turning off computer");
    }

    public void turnOn(){
        System.out.println("turning on computer");
    }
}
```

The command objects encapsulate the command execution so if multiple steps need to occur pre or post powering off the computer, the command object can complete the process. In our sample code the commands will only call the respective on/off functions of the computer object.

```
public class ComputerOffCommand extends Command {
    public Computer computer;

    public ComputerOffCommand(Computer comp){
        computer = comp;
    }

    @Override
    public void execute() {
        //Do other items related to shutdown...
        computer.turnOff();
    }
}

public class ComputerOnCommand extends Command {
    public Computer computer;

    public ComputerOnCommand(Computer comp){
        computer = comp;
    }

    @Override
    public void execute() {
        //Do other items related to turn on
        computer.turnOn();
    }
}
```

Next, the Switch object encapsulates the process related to pressing the power button on the computer. The switch is not concerned with the details of the commands is calls, its only concern is providing a way to switch between states. The objects passed into the switch must inherit from the abstract command object.

```
public class Switch {
   public Command onCmd;
   public Command offCmd;

   public Switch (Command oncmd, Command offcmd){
      onCmd = oncmd;
      offCmd = offcmd;
   }

   public void PressOn (){
      onCmd.execute();
   }

   public void PressOff(){
      offCmd.execute();
   }
}
```

The example execution is straightforward, a computer, switch, and concrete commands are created. The switch is 'pressed' on which causes the ComputerOnCommand object to execute the "turnOn" method of the computer (as seen in the code output). The Switch object doesn't know anything about the computer that is turned on or off, it is completely separated from the computer's implementation. A simpler implementation may have been just to call the computer 'turnOn' method directly from the switch, however, this would tightly couple the switch to the computer. The Command pattern allows the switch to behave independently of the object being switched. This results in the ability to reuse the switch in other applications.

```
write("------------------------------------------------------------");
write("     Command Example");
write("------------------------------------------------------------");
write("");
write("Create Computer Object");
Computer c = new Computer();
write("create new switch and concrete command objects");
Switch s = new Switch(new ComputerOnCommand(c), new ComputerOffCommand(c));
s.PressOn();
s.PressOff();
write("------------------------------------------------------------");
```

```
------------------------------------------------------------
     Command Example
------------------------------------------------------------
```

```
Create Computer Object
create new switch and concrete command objects
turning on computer
turning off computer
-------------------------------------------------------------
```

Justification

Command patterns are vital to the the list of the six design patterns every developer should know because they allow request from one object to be issued without the originating object knowing anything about the operation being requested or the receiver of the request. From large distributed systems to any object-oriented code, object segregation is extremely important for maintenance and support. Commands are on of the most basic functions of any computer system, to encapsulate them allows developers the ability to implement a time proven model of system execution.

CONCLUSIONS

The six design patterns every developer should know are the Abstract Factory, Singleton, Adapter, Composite, Observer, and Command patterns. These basic design patterns make up a strong base for developer's understanding of patterns, and will lead developers toward more complex patterns. These six patterns expose important object-oriented principles that would help developers learn more about the craft of developing software. Patterns are vital to solving design problems that commonly occur in software development, and the six patterns chosen here are key to helping developers solve those problems.

REFERENCES

AllAppLabs. *Java Design Patterns.* http://www.allapplabs.com/java_design_patterns/ (accessed 11 2015).

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissidaes. *Design Patterns: Elements of Reusable Object-Oriented Software.* Indianapolis, IN: Pearson Education Corporate Sales Division, 1995.