CECS 277 – Lecture 19 – Design Patterns

What are Design Patterns?

Design Patterns are general solutions to common problems that come up when designing software. They are also used as a means of communication (like idioms) between programmers about the design. Being familiar with them can help you efficiently build applications that are easier to maintain by using clearly defined approaches rather than reinventing a solution from scratch.

In 1994, four authors, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (more commonly known as the Gang of Four (GoF)), published a book describing 23 different design patterns with descriptions, examples, template code, and UML diagrams. They were interested in reusable object-oriented design and noticed that programmers often came up with similar solutions to commonly recurring problems and thought it would be useful if programmers everywhere knew about these patterns.

These patterns are general solutions to common problems, they do not consist of code that you can copy and paste into your program and have them work. The patterns must be adapted to fit your problem. Do not try to adapt your problem to fit the pattern.

Types of Design Patterns

Creational – these patterns provide a systematic way of creating objects without directly using the keyword new. They use interfaces and polymorphism to allow the construction process to be more dynamic and flexible of which, when, and how objects are initialized.

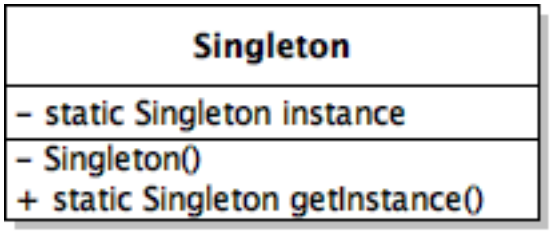
Structural – these patterns are used to define and build relationships between classes to form them into a larger structure that is flexible and extensible. They use inheritance and interfaces to get classes to work together to form a single working item. They help make sure that if one part changes, the rest of the structure doesn't need to be changed as well.

Behavioral – these patterns make for better interactions and communication between objects by abstracting actions out of a class.

Singleton – a creational design pattern.

Singleton is the smallest, simplest, and most commonly used design pattern. It is used to make sure that there is only a single instance of the class and provides a global access point to it. It's primarily used when you need global access to a single shared resource, like a file or a database (especially if it is resource intensive to open).

Singleton UML – In order to ensure that there is only one instance of the class created, access to the constructor is removed by making it private. This way, only a method of the class may call the constructor. This method will do a check to see if the object has already been created, if it hasn't, then it makes one, if it has, then it returns the original instance, which is stored as a private static member of the class.



Singleton Template Code:

```
public class Singleton {
   //other data members of class
   private static Singleton instance = null;
   private Singleton( ) { // no parameters
      //set default values of data members
   }
   public static Singleton getInstance( ) {
      if( instance == null ) {
         instance = new Singleton( );
      }
      return instance;
   }
   //class methods
}
```

Singleton Example – A Logger Class – a logger can be helpful in tracking errors in the execution of a program. All of the logs are written to a file (or set of files) that can be accessed anywhere in the program. It would be impractical for each method and class to reconstruct the Logger object, since they would either create new separate files for each one, or overwrite logs that weren't supposed to be overwritten yet. Creating it as a Singleton, we can ensure that the Logger is a single shared resource usable throughout the program.

```
public class FileLog {
   private PrintWriter writer;
   private static FileLog instance = null;
   private FileLog( ) {
      try{
         writer = new PrintWriter( "log.txt" );
      }catch( FileNotFoundException e ) {
         System.out.println( "FNF" );
      }
   }
   public static FileLog getInstance( ) {
      if( instance == null ) {
         instance = new FileLog( );
      }
      return instance;
   }
   public void writeMsg( String msg ) {
      writer.println( msg );
      writer.flush( );
   }
}
```

```
public class Main {
   public static void main( String [] args ) {
      FileLog logger = FileLog.getInstance();
      logger.writeMsg( “Program Start” );
      method1( );
      method2( );
      logger.writeMsg( “Program End” );
   }
   public static void method1( ) {
      FileLog.getInstance().writeMsg( “Method1” );
   }
   public static void method2( ) {
      FileLog.getInstance().writeMsg( “Method2” );
   }
}
```