Design Patterns

Alan Goude

Useful resources

- **Books**

  Design patterns : elements of reusable object-oriented software, by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

  Design Patterns: AND Applying UML and Patterns, an Introduction to Object-Oriented Analysis and Design and Iterative Development: Elements of Reusable Object-oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

  Design Patterns Explained: A New Perspective on Object-Oriented Design (Software Patterns) by Alan Shalloway , James Trott

  Modern C++ Design: Applied Generic and Design Patterns (C++ in Depth Series) (Paperback) by Andrei Alexandrescu , Scott Meyers , John Vlissides.

- **Web links**
  - http://www.oodesign.com/
  - http://www.dofactory.com/Patterns/Patterns.aspx
  - http://www.developer.com/design/article.php/10925_1502691_1
  - http://en.wikipedia.org/wiki/Design_pattern_(computer_science)

The Originator of Patterns

- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." -- Christopher Alexander 1977

What are Patterns

- Current use comes from the work of the architect Christopher Alexander
- Alexander studied ways to improve the process of designing buildings and urban areas
- “Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution.”
- Hence, the common definition of a pattern: “A solution to a problem in a context.”
- Patterns can be applied to many different areas of human endeavour, including software development

Patterns in Software

- "Designing object-oriented software is hard and designing reusable object-oriented software is even harder." - Erich Gamma
- Experienced designers reuse solutions that have worked in the past
- Well-structured object-oriented systems have recurring patterns of classes and objects
- Knowledge of the patterns that have worked in the past allows a designer to be more productive and the resulting designs to be more flexible and reusable

The Gang of Four

- The authors of the DesignPatternsBook (1994) came to be known as the "Gang of Four." The name of the book ("Design Patterns: Elements of Reusable Object-Oriented Software") is too long for e-mail, so "book by the gang of four" became a shorthand name for it. After all, it isn't the ONLY book on patterns. That got shortened to "GOF book", which is pretty cryptic the first time you hear it.
- The GOF are :- Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides

Design Patterns

- *The Design Patterns* book described 23 patterns that were based on the experience of the authors at that time. These patterns were selected because they represented solutions to common problems in software development. Many more patterns have been documented and catalogued since the publishing of *Design Patterns*. However, these 23 are probably the best known and certainly the most popular.

Design Patterns

- Design patterns describe the relations and interactions of different class or objects or types.
- They do not specify the final class or types that will be used in any software code, but give an abstract view of the solution.
- Patterns show us how to build systems with good object oriented design qualities by reusing successful designs and architectures.
- Expressing proven techniques speed up the development process and make the design patterns, more accessible to developers of new system.

Classification of Design Patterns

- Design patterns were originally classified into three types
  - Creational patterns
  - Structural patterns
  - Behavioural patterns.
- A fourth has since been added
  - Concurrency patterns

Creational Patterns

- Creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.
- The basic form of object creation could result in design problems or added complexity to the design.
- Creational design patterns solve this problem by somehow controlling this object creation.

Structural Patterns

- **structural design patterns** are design patterns that ease the design by identifying a simple way to realise relationships between entities.
- These describe how objects and classes combine themselves to form a large structure

Behavioural Patterns

- Design patterns that identify common communication patterns between objects and realize these patterns.
- These patterns increase flexibility in carrying out this communication.

Structure of a Design Pattern

- Design pattern documentation is highly structured.
- The patterns are documented from a template that identifies the information needed to understand the software problem and the solution in terms of the relationships between the classes and objects necessary to implement the solution.
- There is no uniform agreement within the design pattern community on how to describe a pattern template.

Pattern Documentation

- **Pattern Name and Classification:** A descriptive and unique name that helps in identifying and referring to the pattern.
- **Intent:** A description of the goal behind the pattern and the reason for using it.
- **Also Known As:** Other names for the pattern.
- **Motivation (Forces):** A scenario consisting of a problem and a context in which this pattern can be used.
- **Applicability:** Situations in which this pattern is usable; the context for the pattern.
- **Structure:** A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.
- **Participants:** A listing of the classes and objects used in the pattern and their roles in the design.

Pattern Documentation - cont.

- **Collaboration:** A description of how classes and objects used in the pattern interact with each other.
- **Consequences:** A description of the results, side effects, and trade offs caused by using the pattern.
- **Implementation:** A description of an implementation of the pattern; the solution part of the pattern.
- **Sample Code:** An illustration of how the pattern can be used in a programming language
- **Known Uses:** Examples of real usages of the pattern.
- **Related Patterns:** Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

Creational Patterns

- Abstract Factory - Creates an instance of several families of classes
- Builder - Separates object construction from its representation
- Factory Method - Creates an instance of several derived classes
- Prototype - A fully initialized instance to be copied or cloned
- Singleton - A class of which only a single instance can exist

Structural Patterns

- Adapter - Match interfaces of different classes
- Bridge - Separates an object's interface from its implementation
- Composite - A tree structure of simple and composite objects
- Decorator - Add responsibilities to objects dynamically
- Facade - A single class that represents an entire subsystem
- Flyweight - A fine-grained instance used for efficient sharing
- Proxy - An object representing another object

Behavioral Patterns

- Chain of Resp. - A way of passing a request between a chain of objects
- Command - Encapsulate a command request as an object Interpreter A way to include language elements in a program
- Iterator - Sequentially access the elements of a collection
- Mediator - Defines simplified communication between classes
- Memento - Capture and restore an object's internal state
- Observer - A way of notifying change to a number of classes
- State - Alter an object's behavior when its state changes
- Strategy - Encapsulates an algorithm inside a class
- Template Method - Defer the exact steps of an algorithm to a subclass
- Visitor - Defines a new operation to a class without change

Factory Pattern

- This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

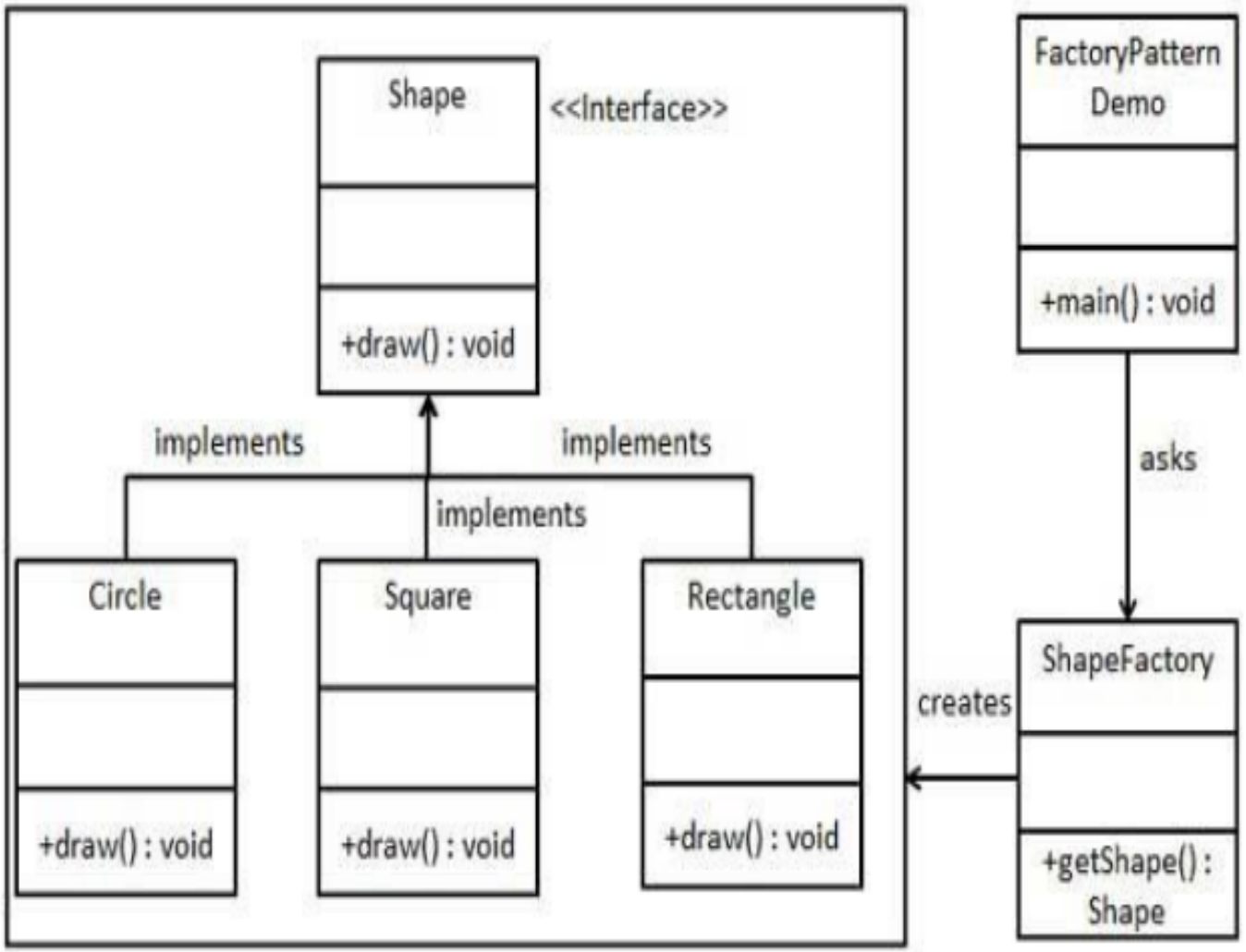
Implementation

- We're going to create a ***Shape interface*** and concrete classes implementing the Shape interface. A factory class ***ShapeFactory*** is defined as a next step.

- ***FactoryPatternDemo***, our demo class will use ***ShapeFactory*** to get a Shape object.

- It will pass information (CIRCLE / RECTANGLE / SQUARE) to ShapeFactory to get the type of object it needs.

Class Diagram



Shape.java

- Step 1
  - Create an interface(Shape.java)
    - Public interface Shape{

      void draw();

      };

Rectangle.java

- Step 2
  - Create concrete classes implementing the same interface
  - //Rectangle.java

```
public class Rectangle implements Shape
 {
      @Override
      public void draw()
                  {
                  System.out.println("Inside Rectangle::draw()
method.");
                  }
};
```

Circle.java

```
public class Circle implements Shape {
        @Override
public void draw()
        {
        System.out.println("draw method in circle class");
        }
};
```

Square.java

```
public class Square implements Shape{
        @Override
        public void draw()
          {
                System.out.println("Draw method in Square class");
          }
};
```

ShapeFactory.java

```java
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}
```

FactoryPatternDemo.java

```java
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of circle
      shape3.draw();
   }
}
```

Singleton Design Pattern

- Singleton pattern is one of the simplest design patterns in Java.
- This type of design pattern comes under creational pattern as this pattern provides one of the best way to create an object.
- This pattern involves a single class which is responsible to creates own object while making sure that only single object get created.
- This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

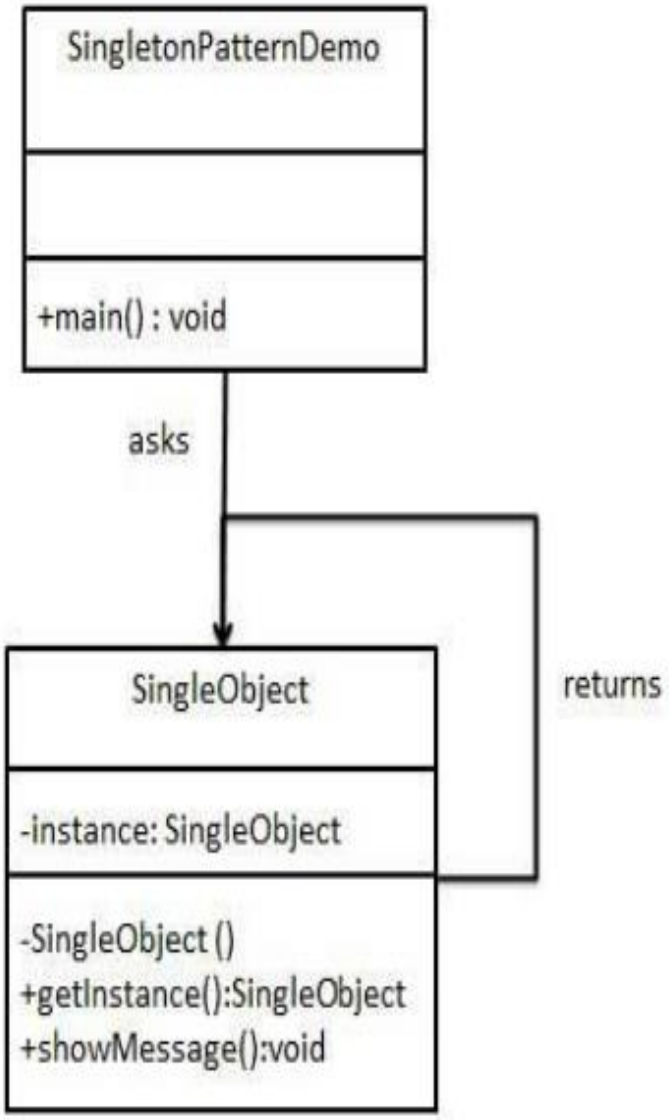
Implementation

- We're going to create a ***SingleObject*** class. ***SingleObject*** class have its constructor as private and have a static instance of itself.

- ***SingleObject*** class provides a static method to get its static instance to outside world.

- ***SingletonPatternDemo***, our demo class will use SingleObject class to get a SingleObject object.

Class Diagram



Step 1:

Create a Singleton Class.

SingleObject.java

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

Step 2

Get the only object from the singleton class.

SingletonPatternDemo.java

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not
visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Step 3

Verify the output.

```
Hello World!
```