Software Design and Analysis
CS-3004

Design Patterns

Lecture # 37, 38, 39
6, 8, 9 Dec

Rubab Jaffar
rubab.jaffar@nu.edu.pk

Today's Outline

- An introduction to design patterns
- Why need design patterns
- Evolution of design patterns
- 3 types of design patterns
  - Creational
  - Structural
  - Behavorial
- Singleton Pattern
- Factory Pattern

What is a Design Pattern?

- Design pattern is a general reusable solution to a commonly occurring problem in software design.
- A design pattern is not a finished design that can be transformed directly into code.
- It is a description or template for how to solve a problem that can be used in many different situations.
- Helps the designer in getting to the right design faster

Design Patterns

- **A design pattern is:**
  - a standard solution to a common programming problem
  - a technique for making code more flexible by making it meet certain criteria
  - a design or implementation structure that achieves a particular purpose
  - a high-level programming idiom
  - shorthand for describing certain aspects of program organization
  - connections among program components
  - the shape of an object diagram or object model

Definitions

- A *pattern* is a recurring solution to a standard problem, in a context.
- Christopher Alexander, a professor of architecture…
  - *Why would what a prof of architecture says be relevant to software?*
  - "A 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."

Patterns in Engineering

- *How do other engineers find and use patterns?*
  - Mature engineering disciplines have handbooks describing successful solutions to known problems.
  - Automobile designers don't design cars from scratch using the laws of physics
  - Instead, they reuse standard designs with successful track records, learning from experience
  - *Should software engineers make use of patterns? Why?*
- Developing software from scratch is also expensive
  - Patterns support reuse of software architecture and design

The “gang of four” (GoF)

- Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides (Addison-Wesley, 1995)
    - *Design Patterns* book [catalogs 23 different patterns] as solutions to different classes of problems, in C++ & Smalltalk
    - The problems and solutions are broadly applicable, used by many people over many years
        - Why is it useful to learn about this pattern?
        - Patterns suggest opportunities for reuse in analysis, design and programming
    - GOF presents each pattern in a [structured format]

Evolution of Design Patterns and GoF

- The four authors of the book "Design Patterns: Elements of Reusable Object-Oriented Software" are referred to as the "Gang of Four".
- The book consists of two parts:
  - First part contains the pros & cons of OOP whereas
  - The second part consists of 23 design patterns.

The “gang of four” (GoF)

THE 23 GANG OF FOUR DESIGN PATTERNS

C	Abstract Factory
S	Adapter
S	Bridge
C	Builder
B	Chain of Responsibility
B	Command
S	Composite
S	Decorator
S	Facade
C	Factory Method
S	Flyweight
B	Interpreter
B	Iterator
B	Mediator
B	Memento
C	Prototype
S	Proxy
B	Observer
C	Singleton
B	State
B	Strategy
B	Template Method
B	Visitor

Types of Design Patterns

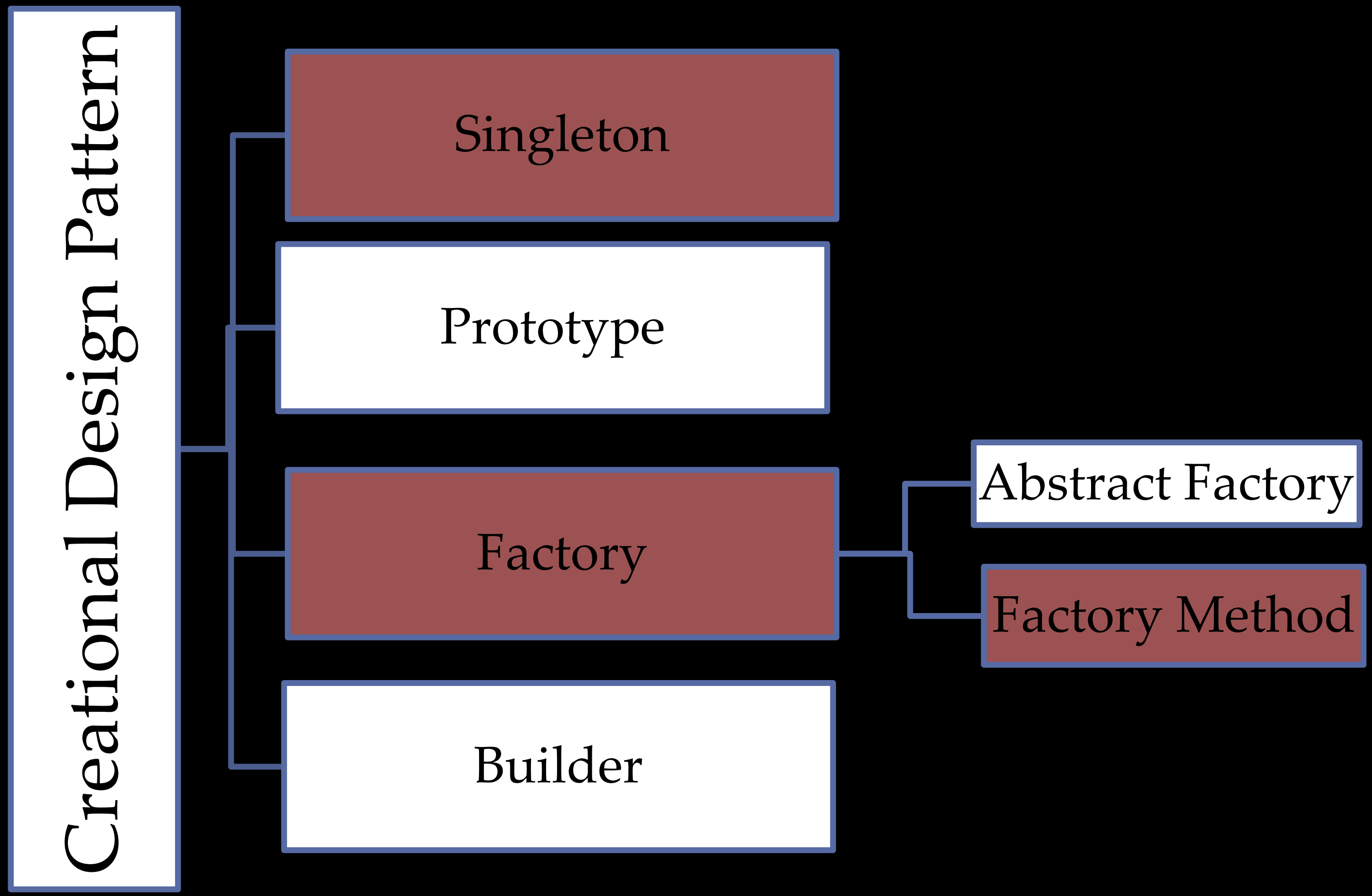
- There are three basic kinds of design patterns:
  - **Structural**
  - **Creational**
  - **Behavioral**

Creational Design Pattern

- **Creational** patterns deals with the object creation and initialization while hiding the creation logic
- It gives the program more flexibility in deciding which objects need to be created for a given case.

- E.g. Singleton, Factory, Abstract Factory

Creational Patterns



Structural Patterns

- How objects/classes can be combined to form larger structures
- **Structural** patterns generally deal with relationships between entities, making it easier for these entities to work together.
- These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
- E.g. Adapter, Bridge, façade e.t.c.

Structural Patterns

Structural Patterns

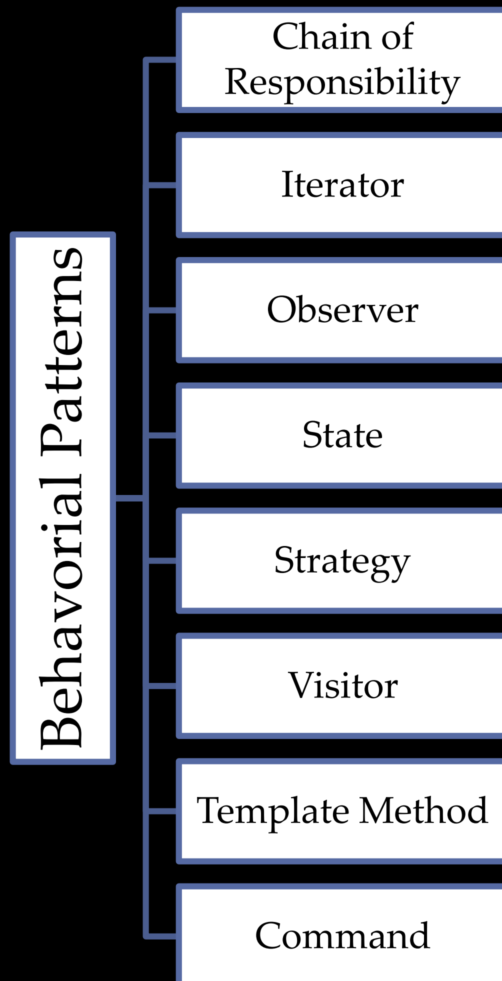
- Proxy
- Decorator
- Facade
- Adapter
- Flyweight

Behavioral patterns

- **Behavioral** patterns are used in communications between entities and make it easier and more flexible for these entities to communicate.

- E.g. Chain of Responsibility, command, Interpreter etc..

Behavorial Patterns



Types of design patterns

CREATIONAL

- how objects can be created
  - maintainability
  - control
  - extensibility

STRUCTURAL

- how to form larger structures
  - management of complexity
  - efficiency

BEHAVIOURAL

- how responsibilities can be assigned to objects
  - objects decoupling
  - flexibility
  - better communication

Elements of Design Patterns

The pattern's name	The problem	The solution	The consequences
The name of the pattern is a one or two word description that pattern-literate programmers familiar with patterns can use to communicate with each other. Examples of names include "factory method" "singleton", "mediator", "prototype", and many more. The name of the pattern should recall the problem it solves and the solution.	The problem the pattern solves includes a general intent and a more specific motivation or two. For instance, the intent of the singleton pattern is to prevent more than one instance of a class from being created. A motivating example might be to not allow more than one object to try to access a system's audio hardware at the same time by only allowing a single audio object.	The solution to the problem specifies the elements that make up the pattern such as the specific classes, methods, interfaces, data structures and algorithms. The solution also includes the relationships, responsibilities and collaborators of the different elements. Indeed these inter-relationships and structure are generally more important to the pattern than the individual pieces, which may change without significantly changing the pattern.	Often more than one pattern can solve a problem. Thus the determining factor is often the consequences of the pattern. Some patterns take up more space. Some take up more time. Some patterns are more scalable than others.

Why need design patterns?

- **Patterns provide object-oriented software developers with:**
  - Reusable solutions to common problems.
  - Names of abstractions above the class and object level.
  - Use the experiences of software developers.
  - A shared library/lingo used by developers.
  - “Design patterns help a designer get a design right faster”.

Singleton Pattern
Type: Creational

Singleton Design Pattern

- **Singleton pattern** restricts the instantiation of a class and ensures that only one instance of the class exists.
- Example:
  - President of the US/ Government
  - Java.lang.System
- Encapsulated "just-in-time initialization" or "initialization on first use".

Problem/Solution pair - Singleton

- The Singleton design pattern solves problems like:
  - How can it be ensured that a class has only one instance?
  - How can the sole instance of a class be accessed easily?
  - How can a class control its instantiation?
  - How can the number of instances of a class be restricted?

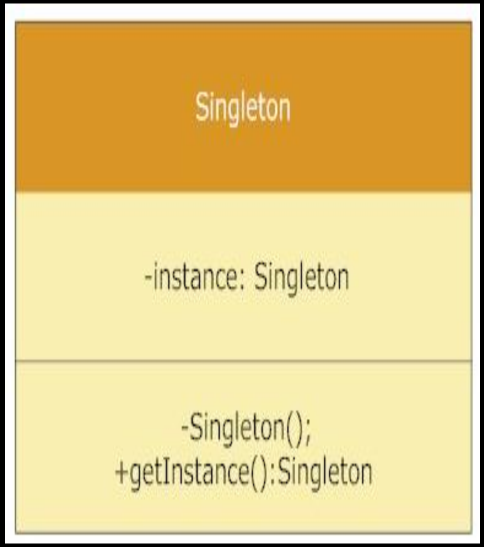
- The Singleton design pattern describes how to solve such problems:
  - Hide the constructor of the class.
  - Define a public static operation (getInstance()) that returns the sole instance of the class.

Implementation Details

- Static member : This contains the instance of the singleton class.

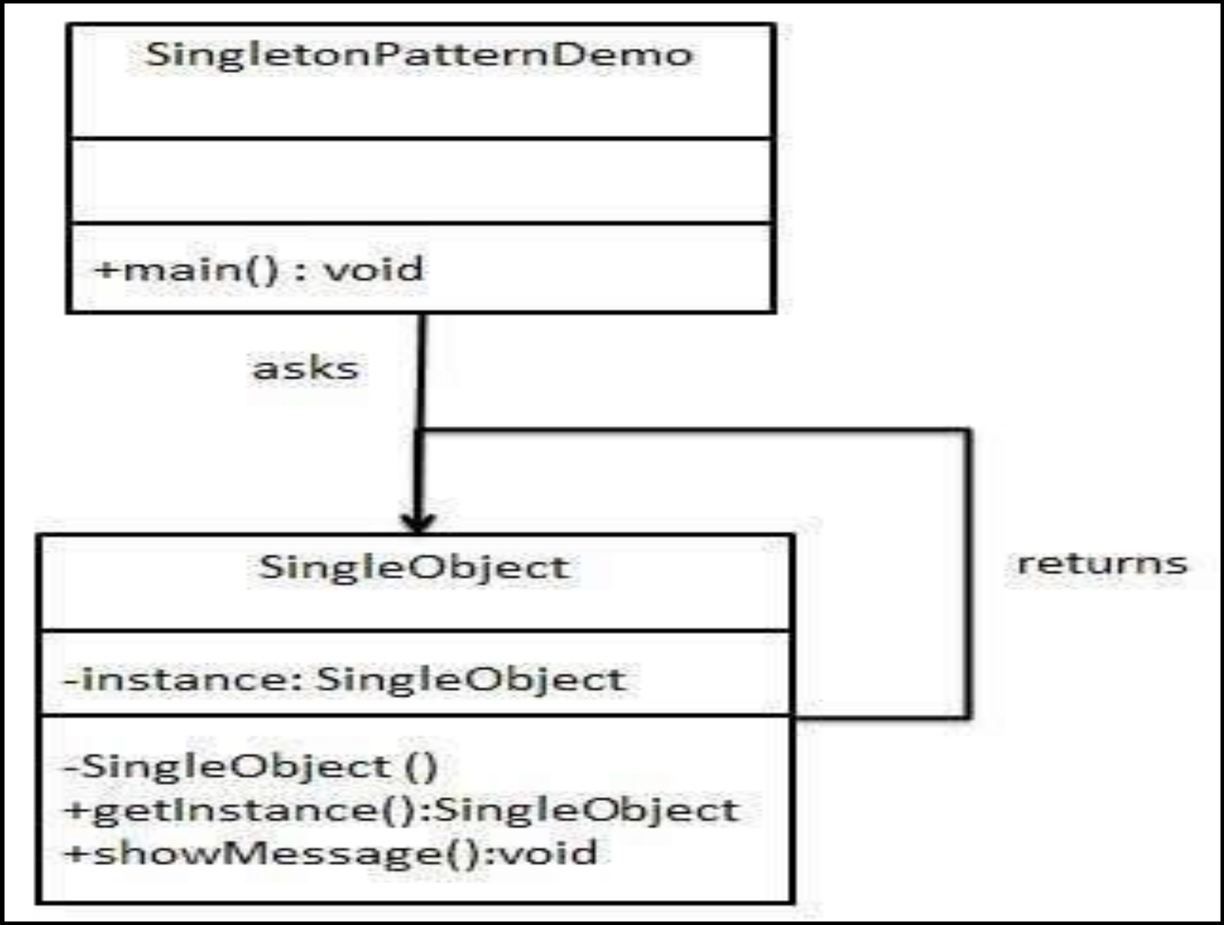
- Private constructor : This will prevent anybody else to instantiate the Singleton class.

- Static public method : This provides the global point of access to the Singleton object and returns the instance to the client calling class.



- **Singleton pattern** is used for logging, drivers objects, caching and thread.

Object Model for Singleton



Method 1: Classic Implementation

```
// Classical Java implementation of singleton
// design pattern
class Singleton
{
    private static Singleton obj;

    // private constructor to force use of
    // getInstance() to create Singleton object
    private Singleton() {}

    public static Singleton getInstance()
    {
        if (obj==null)
            obj = new Singleton();
        return obj;
    }
}
```

The main problem with above method is that it is not thread safe. Consider the following execution sequence.

Method 2: Make getInstance() synchronized

```
// Thread Synchronized Java implementation of
// singleton design pattern
class Singleton
{
	private static Singleton obj;

	private Singleton() {}

	// Only one thread can execute this at a time
	public static synchronized Singleton getInstance()
	{
		if (obj==null)
			obj = new Singleton();
		return obj;
	}
}
```

Method 3: Eager Instantiation

```
// Static initializer based Java implementation of
// singleton design pattern
class Singleton
{
        private static Singleton obj = new Singleton();

        private Singleton() {}

        public static Singleton getInstance()
        {
                return obj;
        }
}
```

Method 4 (Best): Use “Double Checked Locking”

```
// Double Checked Locking based Java implementation of
// singleton design pattern
class Singleton
{
	private volatile static Singleton obj;

	private Singleton() {}

	public static Singleton getInstance()
	{
		if (obj == null)
		{
			// To make thread safe
			synchronized (Singleton.class)
			{
				// check again as multiple threads
				// can reach above step
				if (obj==null)
					obj = new Singleton();
			}
		}
		return obj;
	}
}
```

Java Code Example for Singleton Pattern

```java
public class SingletonExample {

    // Static member holds only one instance of the
    // SingletonExample class
    private static SingletonExample singletonInstance;

    // SingletonExample prevents any other class from instantiating
    private SingletonExample() {
    }

    // Providing Global point of access
    public static SingletonExample getSingletonInstance() {
        if (null == singletonInstance) {
            singletonInstance = new SingletonExample();
        }
        return singletonInstance;
    }

    public void printSingleton(){
        System.out.println("Inside print Singleton");
    }
}
```

Factory Pattern
Type: Creational

Intent

- “Define an interface for creating an object, but let subclasses decide which class to instantiate”
- It lets a class defer instantiation to subclasses at run time.
- It refers to the newly created object through a common interface.
- The factory method pattern is a design pattern that define an interface for creating an object from one among a set of classes based on some logic.
- Factory method pattern is also known as virtual constructor pattern.
- Factory method pattern is the most widely used pattern in the software engineering world

AKA

- Virtual Constructor
  - The main intent of the virtual constructor idiom in C++ is to create a copy of an object or a new object without knowing its concrete type and this is exactly the Factory Method of initialization.

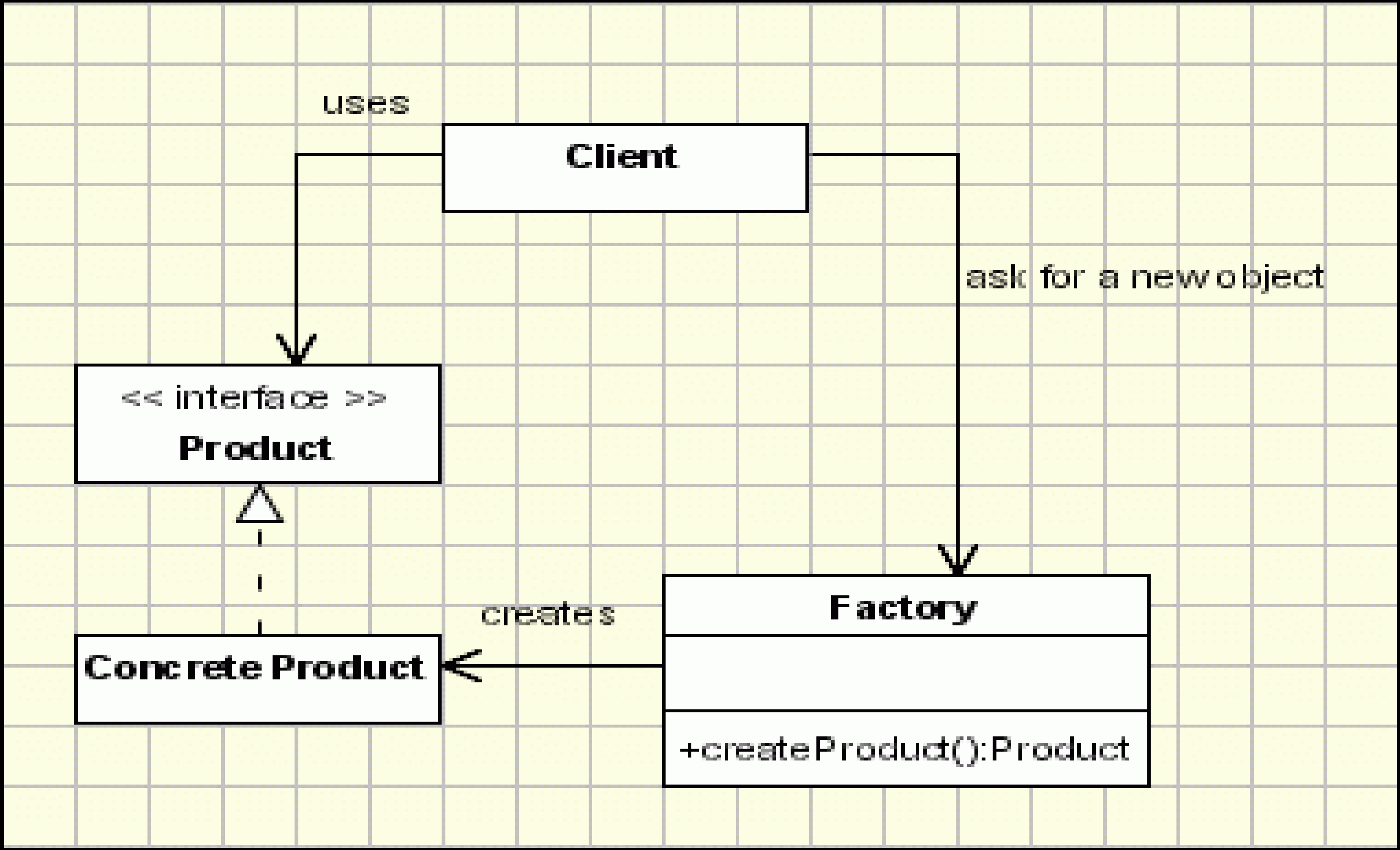
Applicability

- Use the Factory Method pattern when
  - a class can´t anticipate the class of objects it must create.
  - At times, application only knows about the super class (may be abstract class), but doesn't know which sub class (concrete implementation) to be instantiated at compile time.
  - a class wants its subclasses to specify the objects it creates.
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Participants

- Product
  - o Defines the interface of objects the factory method creates
- ConcreteProduct
  - o Implements the product interface
- Creator
  - o Declares the factory method which returns object of type product
  - o May contain a default implementation of the factory method
  - o Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate Concrete Product.
- ConcreteCreator
  - o Overrides factory method to return instance of ConcreteProduct

UML class diagram of Factory Design Pattern

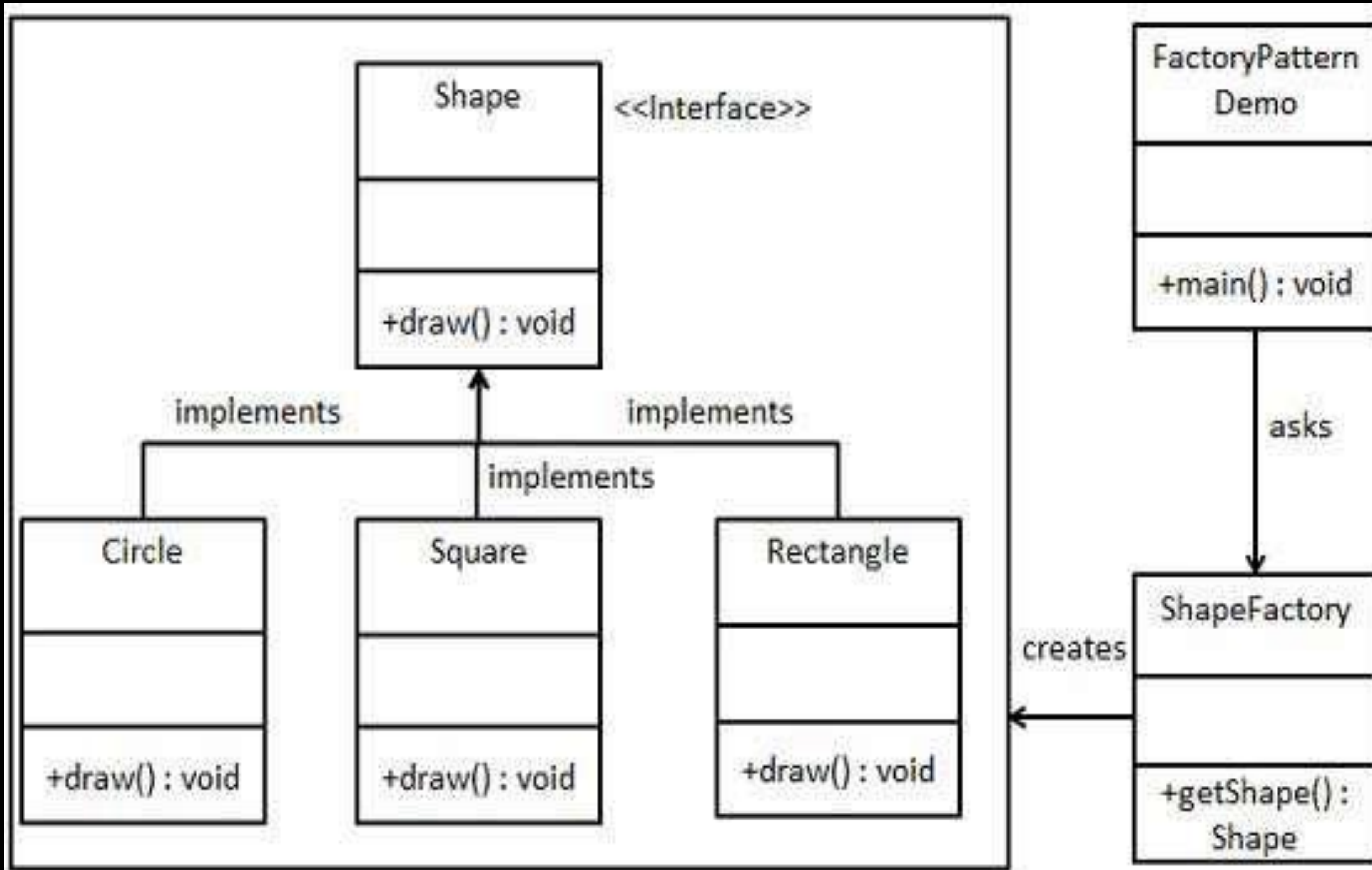


Implementation

- The implementation is really simple the client needs a product, but instead of creating it directly using the new operator, it asks the factory object for a new product, providing the information about the type of object it needs.
- The factory instantiates a new concrete product and then returns to the client the newly created product (casted to abstract product class).
- The client uses the products as abstract products without being aware about their concrete implementation.

Example

- 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.*


Shape
<<Interface>>
+draw() : void
implements
implements
implements
Circle
+draw() : void
Square
+draw() : void
Rectangle
+draw() : void
FactoryPattern Demo
+main() : void
asks
ShapeFactory
+getShape() : Shape
creates

Step 1

- Create an interface. (*Shape.java)*

```
public interface Shape {
void draw();
}
```

Step 2

- Create concrete classes implementing the same interface.

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

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

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

Step 3

- Create a Factory to generate object of concrete class based on given information. *(ShapeFactory.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;
} }
```

Step 4

- Use Factory to get object of concrete class by passing information such as type. (*FactoryPatternDemo.java)*

```
public class FactoryPatternDemo {
  public static void main(String[] args) {
    ShapeFactory shapeFactory = new ShapeFactory();
    Shape shape1 = shapeFactory.getShape("CIRCLE");
    shape1.draw();
    Shape shape2 = shapeFactory.getShape("RECTANGLE");
    shape2.draw();
    Shape shape3 = shapeFactory.getShape("SQUARE");
    shape3.draw();
  }
}
```

Step 5

Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.

That is all