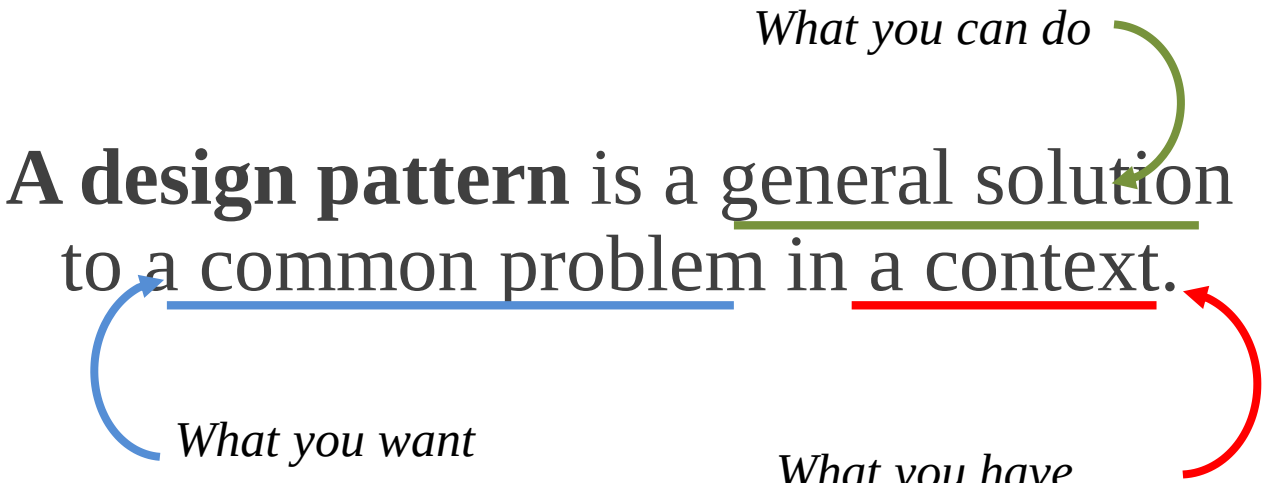


Design Patterns

Lecture 17

What is a design pattern (DP)?

"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*

Design patterns are repeatable / reusable solutions to commonly occurring problems in a certain context in software design.

What is a design pattern (DP)? (cont…)

Four essential elements of a pattern :

- Pattern name: provides a way to describe the design problem, its solution and consequences in a word or two. A pattern name provides vocabulary using which we can communicate with other people, document them and reference them in software designs. Finding a good name for a design pattern is a major issue.
- Problem: describes when to apply the pattern and in which context to apply the pattern. The problem may describe the class and object structures in a bad design context. Sometimes, the problem might also list out the conditions that must be pre satisfied to apply the pattern.
- Solution: describes the elements (classes and objects) that make up the design, their relationships, responsibilities and collaborations. The solution does not describe a particular concrete design or implementation (no code). A pattern is a template that can be applied in many different situations.
- Consequences These are the costs and benefits of applying the patterns. These consequences involve space and time trade-offs, language and implementation issues as well as the effect of pattern on system's flexibility, extensibility or portability. Listing these consequences explicitly enables us to understand and evaluate them.

Need for design patterns

- Designing reusable object-oriented software (API's) is hard.
- Experienced designers Vs novice designers.
- Patterns make object-oriented software flexible, elegant and reusable.
- Solved a problem previously but don't remember when or how?

Use of design patterns

- Make it easier to reuse successful designs and architectures.
- Make it easy for the new developers to design software.
- Allows to choose different design alternatives to make the software reusable.
- Helps in documenting and maintaining the software.

To get a design "right", faster.

Why should we use DP's?

These are already tested and proven solutions used by many experienced designers.

How to select a design pattern

- Consider how design patterns solve design problems
- Scan intent sections
- Study how patterns interrelate
- Study patterns of like purpose
- Examine a cause of redesign
- Consider what should be variable in your design

How to use a design pattern

1. Read the pattern once through for a overview.
2. Study the structure, participants and collaborations sections.
3. Look at the sample code section to see a concrete example of the pattern in action.
4. Choose names for pattern participants that are meaningful in the application context.
5. Define the classes.
6. Define application-specific names for the operations in the pattern.
7. Implement the operations to carry out the responsibilities and collaborations in the pattern.

Types of Design Patterns

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

Structural Patterns

- **Structural design patterns** are concerned with how classes and objects can be composed, to form larger structures.
- The structural design patterns **simplifies the structure by identifying the relationships**.
- These patterns focus on, how the classes inherit from each other and how they are composed from other classes.

Behavioral patterns

- Behavioral design patterns are concerned with **the interaction and responsibility of objects.**
- In these design patterns, **the interaction between the objects should be in such a way that they can easily talk to each other and still should be loosely coupled.**

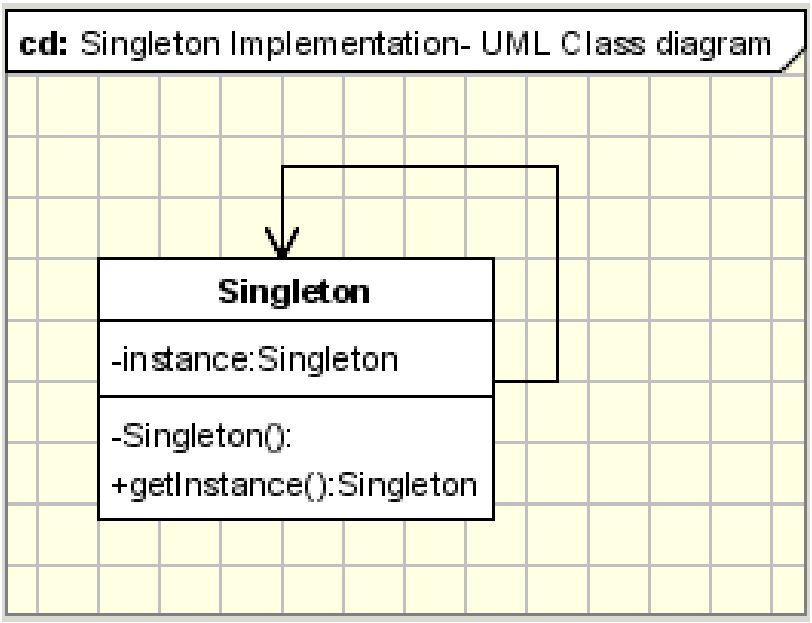
Creational Pattern

- Deals with how objects are created.
- Increase the system's flexibility in terms of what, who, how and when the object is created.
- Further classified into two types:
  1. Class-creational patterns
  2. Object-creational patterns

Singleton Pattern (INTENT)

- To create exactly one object of a class and to provide a global point of access to it.

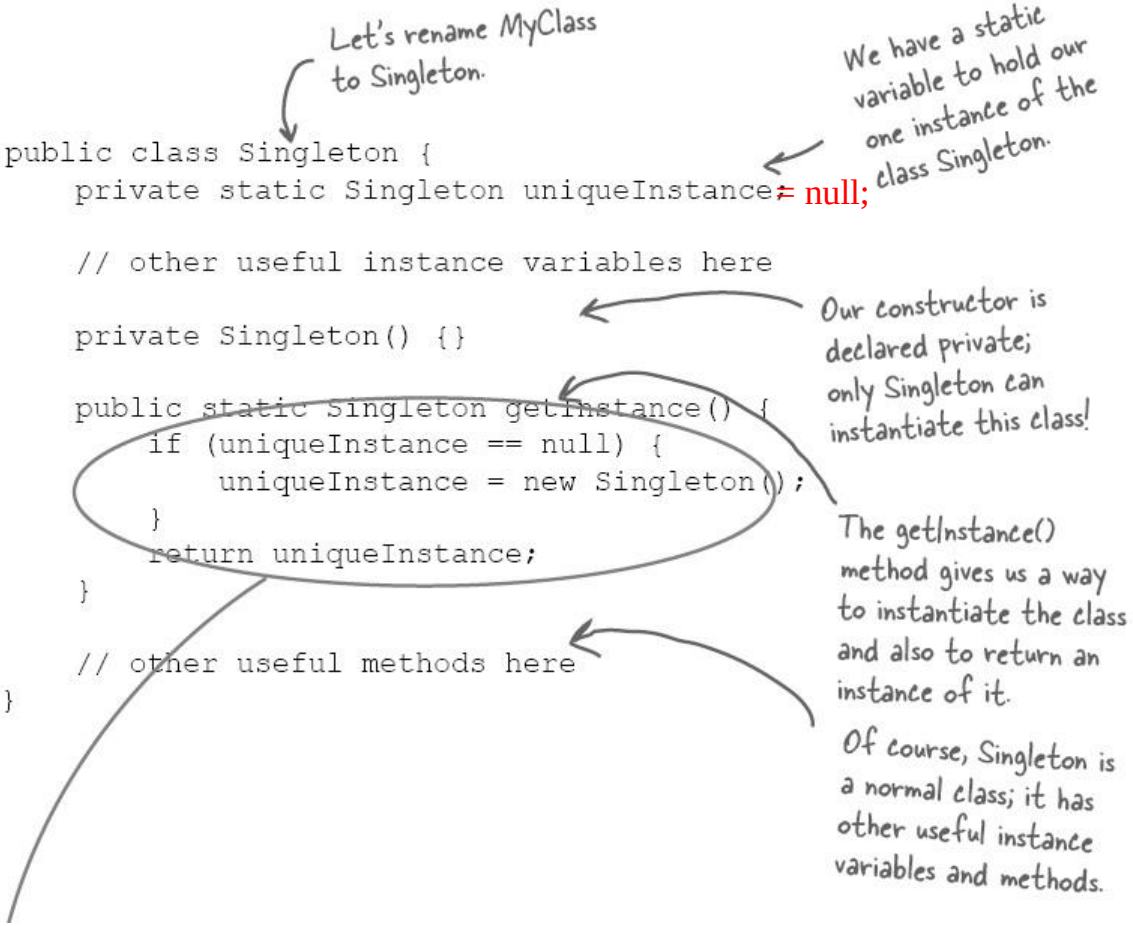
Structure

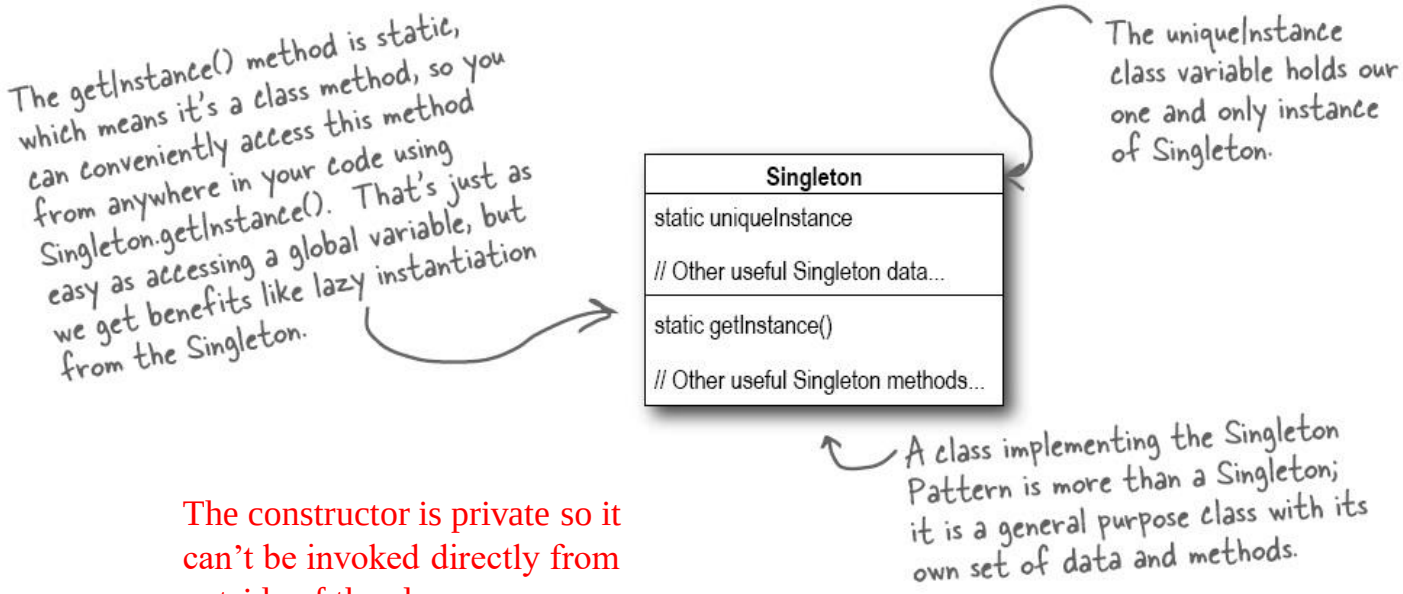


Singleton Pattern (cont…)

Motivation:

1) Need to create exactly one user interface object.
2) Need to create only one print spooler.


Let's rename MyClass to Singleton.
We have a static variable to hold our one instance of the class Singleton.
public class Singleton {
private static Singleton uniqueInstance = null;
// other useful instance variables here
private Singleton() {}
public static Singleton getInstance() {
if (uniqueInstance == null) {
uniqueInstance = new Singleton();
}
return uniqueInstance;
}
// other useful methods here
}
Our constructor is declared private; only Singleton can instantiate this class!
The getInstance() method gives us a way to instantiate the class and also to return an instance of it.
Of course, Singleton is a normal class; it has other useful instance variables and methods.



The constructor is private so it can't be invoked directly from outside of the class

Singleton Pattern (cont…)

Applicability:

Singleton Pattern can be used when:

1)There must be exactly one instance of a class and to provide a global point of access to it.

2)When the sole instance should be extensible by sub classing, and clients should be able to use the extended instance without modifying its code.

Singleton Pattern (cont…)

Participants:

Consists of a “Singleton” class with the following members:

1)A static data member

2)A private constructor

3)A static public method that returns a reference to the static data member.

Singleton Pattern (cont…)

Collaborations:

Clients access the singleton instance solely through the singleton's class operation.

Singleton Pattern (cont…)

Consequences:

- Controlled access to sole instance.
- Permits refinement of operations and representation.
- Permits a variable number of instances.

Singleton Pattern (cont…)

Implementation:

```
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!");
         }
}
```

```
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();
        }
}
```

Check list

1. Define a private static attribute in the "single instance" class.
2. Define a public static accessor function in the class.
3. Define all constructors to be protected or private.
4. Clients may only use the accessor function to manipulate the Singleton.

Factory Design Patterns

Factory Pattern

- Word of the Day = Factory

Simple Factory

- Separate what changes from what does not
- Encapsulate what varies behind an interface
- Design should be open for extension but closed to modification

Intent

- “Define an interface for creating an object, but let subclasses decide which class to instantiate”
- It lets a class accept instantiation to subclasses at run time.
- It refers to the newly created object through a common interface.

FACTORY METHOD

- Motivation:
  - Framework use abstract classes to define and maintain relationships between objects
  - Framework has to create objects as well - must instantiate classes but only knows about abstract classes - which it cannot instantiate
  - Factory method encapsulates knowledge of which subclass to create - moves this knowledge out of the framework

Applicability

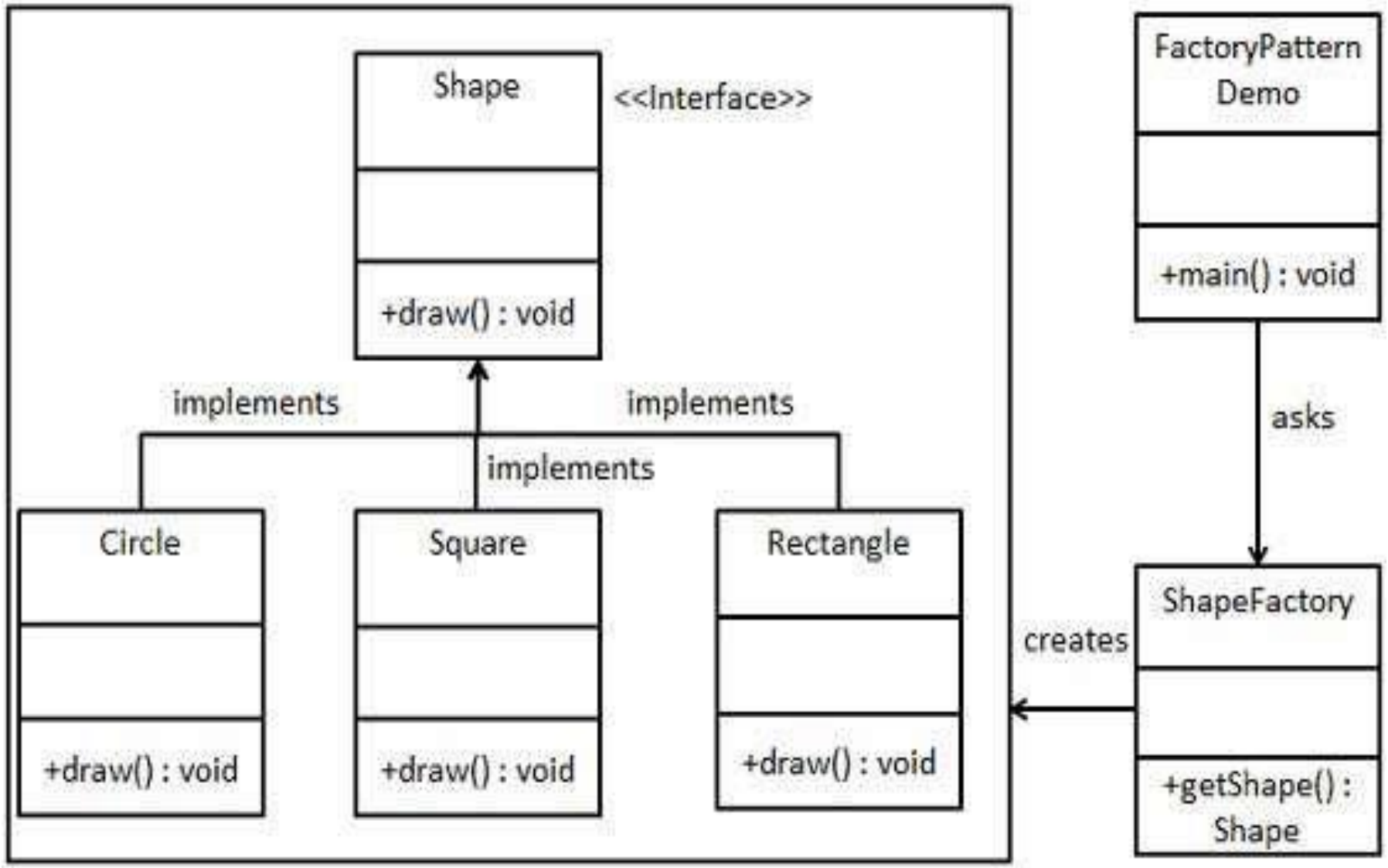
- Use the Factory Method pattern when
  - a class can´t expect the class of objects it must create.
  - a class wants its subclasses to specify the objects it creates.
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

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
creates
ShapeFactory
+getShape() : Shape

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();
    }
}
```

Output

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

Summary

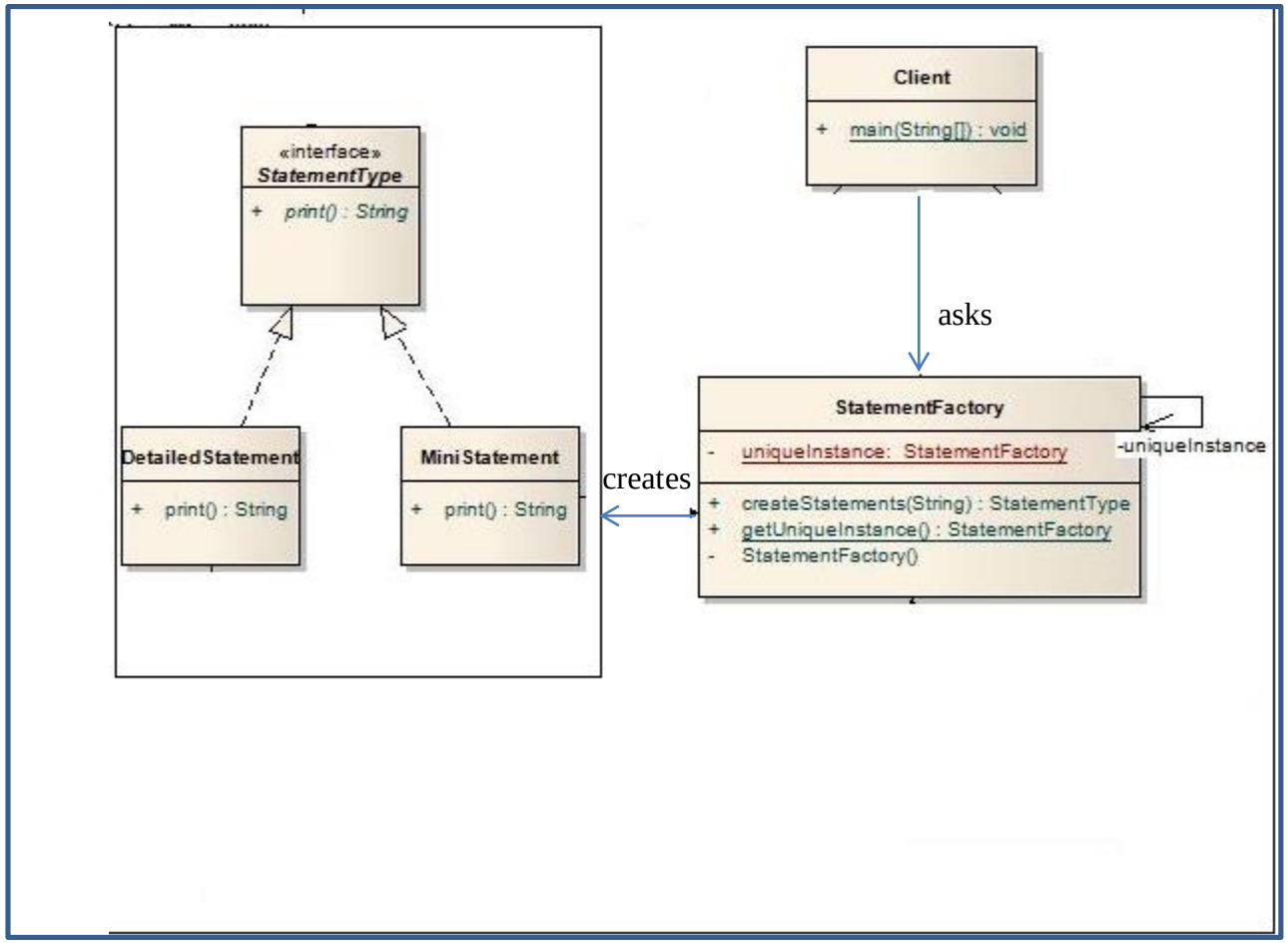
- ***Pattern Name*** – Factory Method
- ***Problem*** – Do not want our framework to be tied to a specific implementation of an object.
- ***Solution***
  - Let subclass decide which implementation to use (via use of an abstract method)
  - Tie the Creator with a specific Product Implementation
- ***Consequences***
  - Simple Factory Method
  - Simple Factory Method with Default Behavior
  - Parameterized Factory Method
  - Factory and Template Together

Factory

- Advantage: We have one place to go to add a new instance.

- Disadvantage: Whenever there is a change, we need to break into this code and add a new line. (but at least it is in one place!!)

Design a small ATM printing application which can generate multiple types of statements of the transaction including Mini Statement, Detailed statement. However the customer should be aware of the creation of these statements. Ensure that the memory consumption is minimized.


«interface»
StatementType
+ print() : String
Client
+ main(String[]) : void
asks
StatementFactory
- uniqueInstance: StatementFactory
+ createStatements(String) : StatementType
+ getUniqueInstance() : StatementFactory
- StatementFactory()
-uniqueInstance
creates
DetailedStatement
+ print() : String
MiniStatement
+ print() : String

Statement Type

```
interface StatementType {
    String print();
}
```

Mini Statement

```
class MiniStatement implements StatementType {
    @Override
    public String print()
{
        System.out.println("Mini Statement Created");
        return "miniStmt";
    }
}
```

Detailed Statement

```
class DetailedStatement implements StatementType
{
    @Override
    public String print() {
        System.out.println("Detailed Statement
Created");
        return "detailedStmt";
    }
}
```

Statement Factory

```
class StatementFactory  {
    private static StatementFactory uniqueInstance;
    private StatementFactory() { }
    public static StatementFactory getUniqueInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new StatementFactory();
            System.out.println("Creating a new StatementFactory instance"); }
        return uniqueInstance;
    }
 public StatementType createStatements(String selection) {
        if (selection.equalsIgnoreCase("detailedStmt")) {
            return new DetailedStatement();
        } else if (selection.equalsIgnoreCase("miniStmt")) {
            return new MiniStatement();
        }
        throw new IllegalArgumentException("Selection doesnot exist");
    }
}
```

Client

```
Class Client
{
  public static void main(String[] args) {
   StatementFactory factory =
StatementFactory.getUniqueInstance();
    StatementType objStmtType =
factory.createStatements("miniStmt");
        System.out.println(objStmtType.print());
  }
}
```