CSE 223: Programming -2
05-Creational Design Patterns (I)

Prof. Dr. Khaled Nagi

Department of Computer and Systems Engineering,

Faculty of Engineering, Alexandria University, Egypt.

Agenda

- Factory method
- Abstract Factory
- Builder

Factory method

Factory method

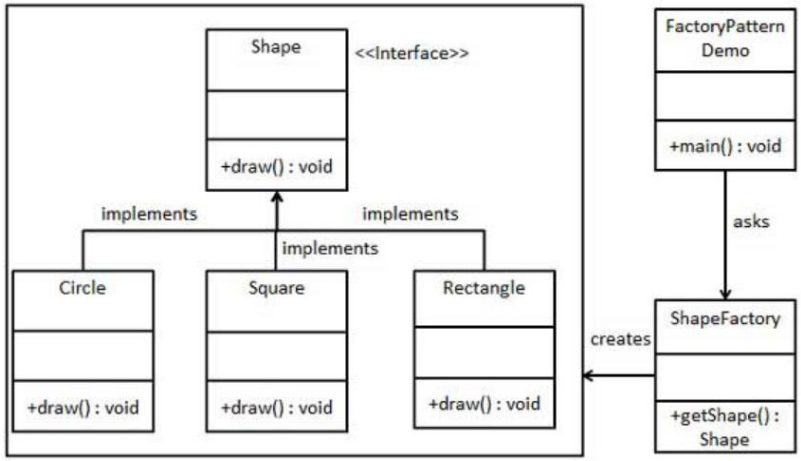
- Intention
  - create object without exposing the creation logic to the client and refer to newly created object using a common interface.
- Implementation
  - create a *Shape* interface and concrete classes implementing the Shape interface. A factory class **ShapeFactory** is defined as a next step.
  - **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.

Factory method

- *Step 1*: Create an interface.

Shape.java

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



- *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.");
   }
}
```

Square.java

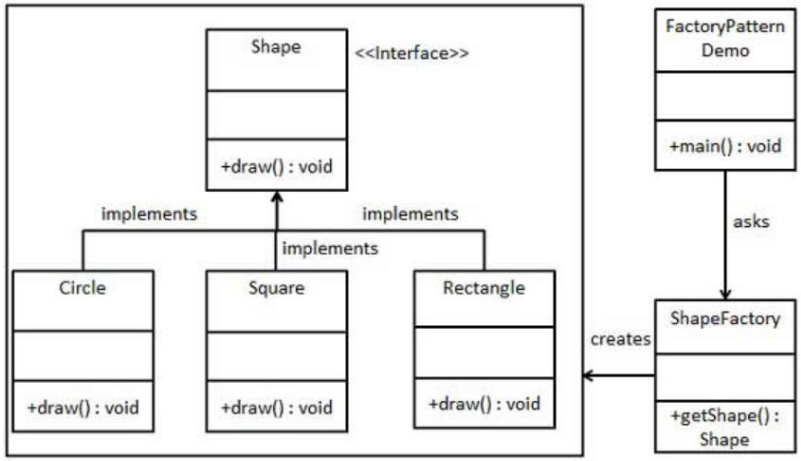
```
public class Square implements Shape {

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

Factory method

- Step 3: Create a Factory to generate object of concrete class based on given information.

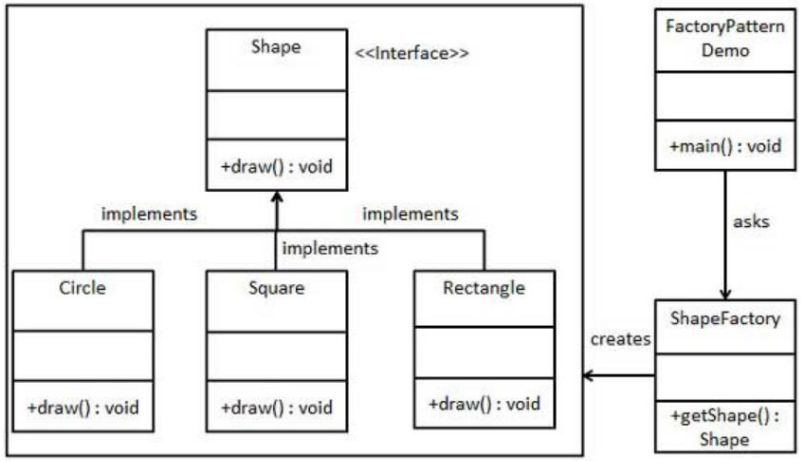
ShapeFactory.java

```
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;
   }
}
```



Factory method

- *Step 4*: Use the Factory to get object of concrete class by passing an information such as type.



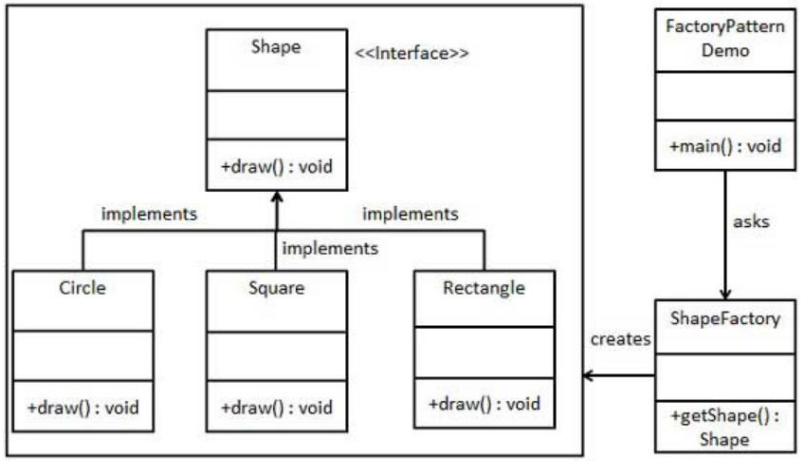
FactoryPatternDemo.java

```
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 square
      shape3.draw();
   }
}
```

Factory method

- *Step 5*: Verify the output.

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



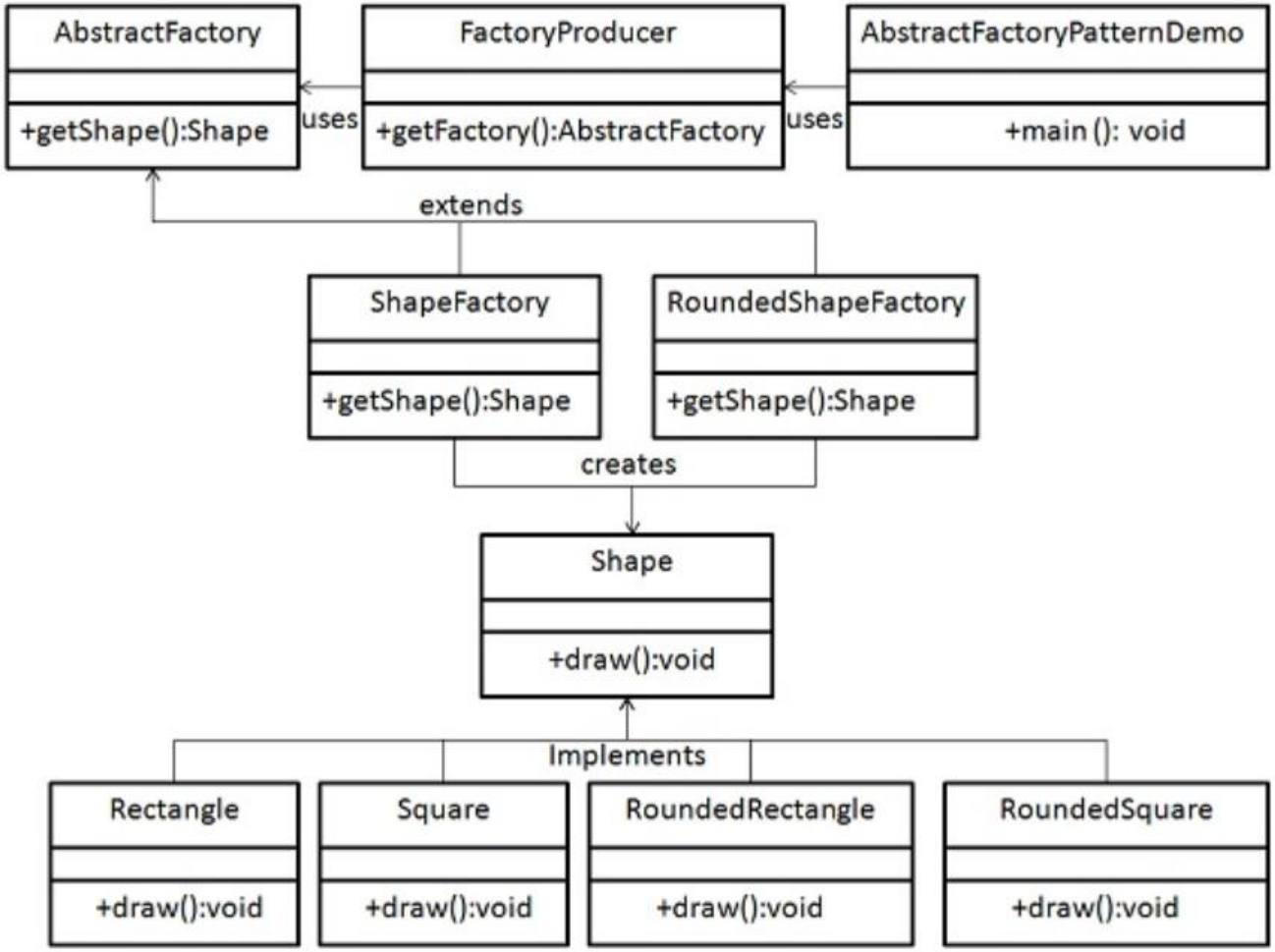
- Consequences
  - Encapsulate what varies
    - Only one class is change when we add further shapes
  - Program through an interface not to an implementation
    - All calls are done to *Shape*

Abstract Factory

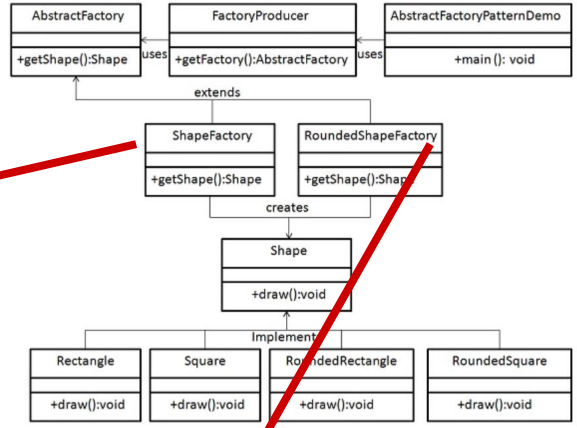
Abstract Factory

- Intention
  - Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories.
  - In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.
- Implementation
  - Create a *Shape* interface and a **concrete class** implementing it.
  - Create an abstract factory class **AbstractFactory** as next step.
    - Factory class **ShapeFactory** is defined, which extends **AbstractFactory**.
    - A factory creator/generator class **FactoryProducer** is created.
  - **AbstractFactoryPatternDemo** uses **FactoryProducer** to get a **AbstractFactory** object. It will pass information (CIRCLE / RECTANGLE / SQUARE for Shape) to **AbstractFactory** to get the type of object it needs.

Abstract Factory



Abstract Factory



ShapeFactory.java

```java
public class ShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}
```

RoundedShapeFactory.java

```java
public class RoundedShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new RoundedRectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new RoundedSquare();
      }
      return null;
   }
}
```

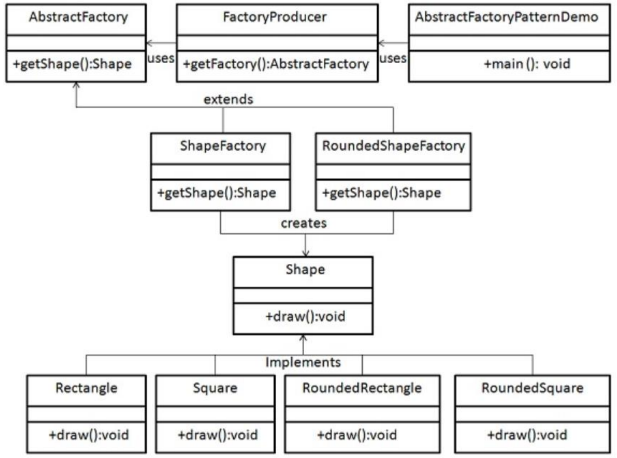
FactoryProducer.java

```java
public class FactoryProducer {
   public static AbstractFactory getFactory(boolean rounded){
      if(rounded){
         return new RoundedShapeFactory();
      }else{
         return new ShapeFactory();
      }
   }
}
```

Abstract Factory

AbstractFactoryPatternDemo.java

```
public class AbstractFactoryPatternDemo {
   public static void main(String[] args) {
      //get shape factory
      AbstractFactory shapeFactory = FactoryProducer.getFactory(false);
      //get an object of Shape Rectangle
      Shape shape1 = shapeFactory.getShape("RECTANGLE");
      //call draw method of Shape Rectangle
      shape1.draw();
      //get an object of Shape Square
      Shape shape2 = shapeFactory.getShape("SQUARE");
      //call draw method of Shape Square
      shape2.draw();
      //get shape factory
      AbstractFactory shapeFactory1 = FactoryProducer.getFactory(true);
      //get an object of Shape Rectangle
      Shape shape3 = shapeFactory1.getShape("RECTANGLE");
      //call draw method of Shape Rectangle
      shape3.draw();
      //get an object of Shape Square
      Shape shape4 = shapeFactory1.getShape("SQUARE");
      //call draw method of Shape Square
      shape4.draw();

   }
}
```



Factory method vs. Abstract Factory

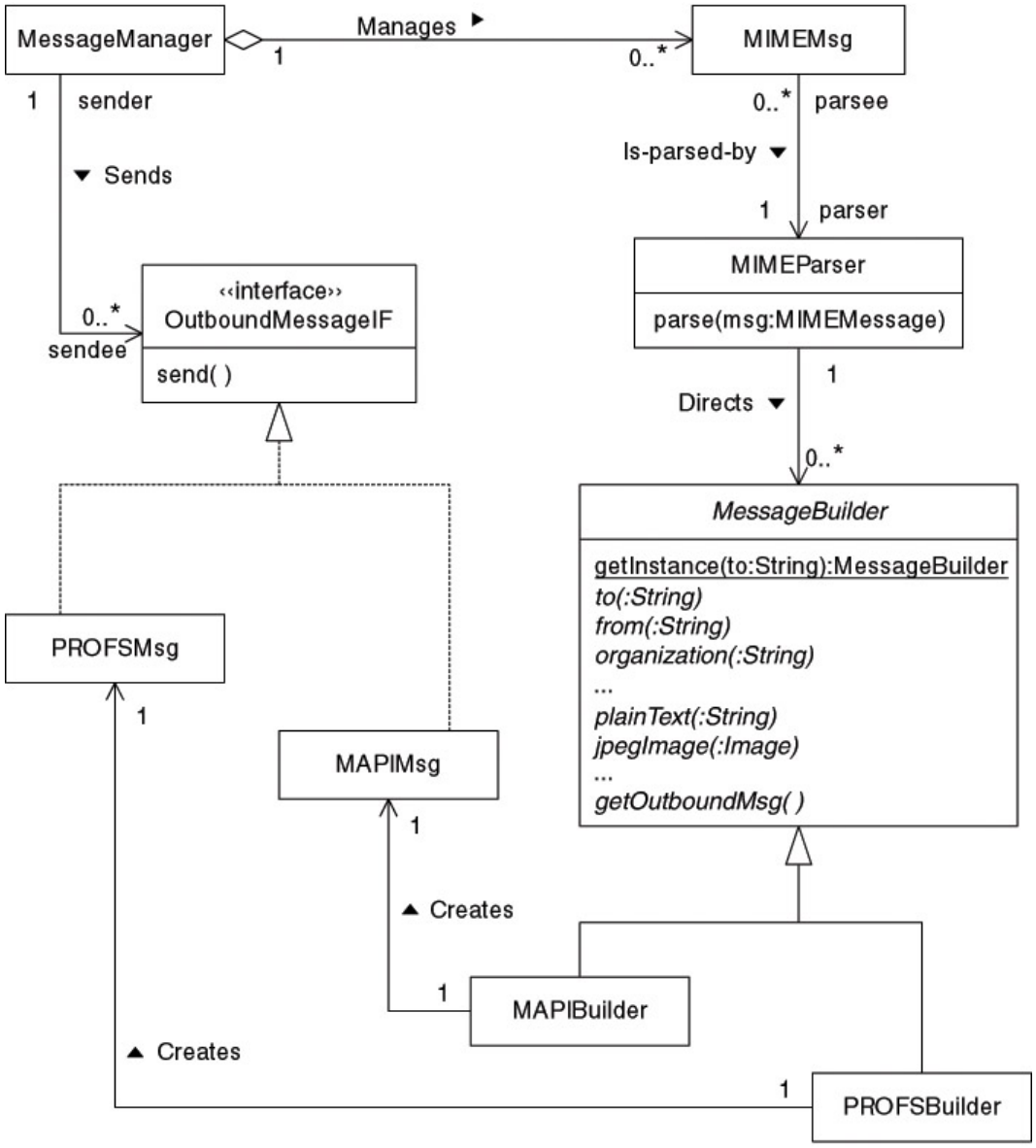
- Factory Method is used to create one family of products.
- Abstract Factory is about creating families of related products.
- Inheritance vs Composition
  - Factory Method and Abstract Factory *both* use **Inheritance**
  - They use it differently
- Difference to the Dependency Injection (DI)
  - By using Factory, your code is still actually responsible for creating objects.
  - Factory pattern is just one way to separate the responsibility of creating objects of other classes to another entity.
  - Factory pattern can be called as a tool to implement DI.
  - By DI you outsource that responsibility to another class or a framework, which is separate from your code. → Spring
  - DI is more of an architectural pattern for loosely coupling software components.
  - DI can be implemented in many ways like constructors, using mapping xml files etc.

Builder Design Pattern

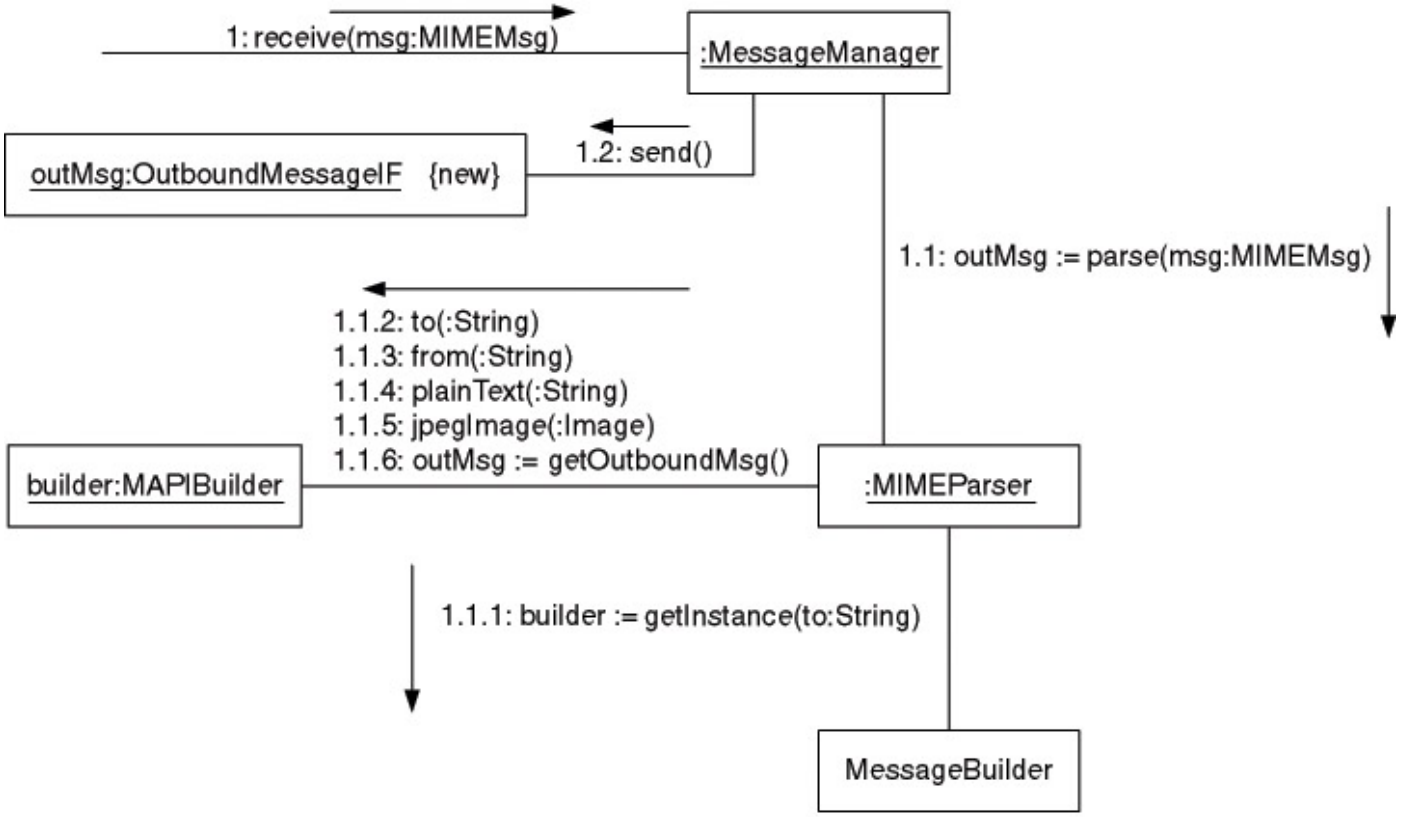
Builder Pattern

- The Builder pattern allows a client object to construct a complex object by specifying only its type and content. The client is shielded from the details of the object's construction.
- Example
  - Consider the problem of writing an email gateway program. The program receives e-mail messages that are in MIME format.
  - It forwards them in a different format for different kinds of e-mail systems.
  - This situation is a good fit for the Builder pattern. It is straightforward to organize this program with an object that parses MIME messages.
  - Each message to parse is paired with a builder object that the parser uses to build a message in the required format. As the parser recognizes each header field and message body part, it calls the corresponding method of the builder object it is working with.

Builder Pattern



Builder Pattern
Collaboration Diagram

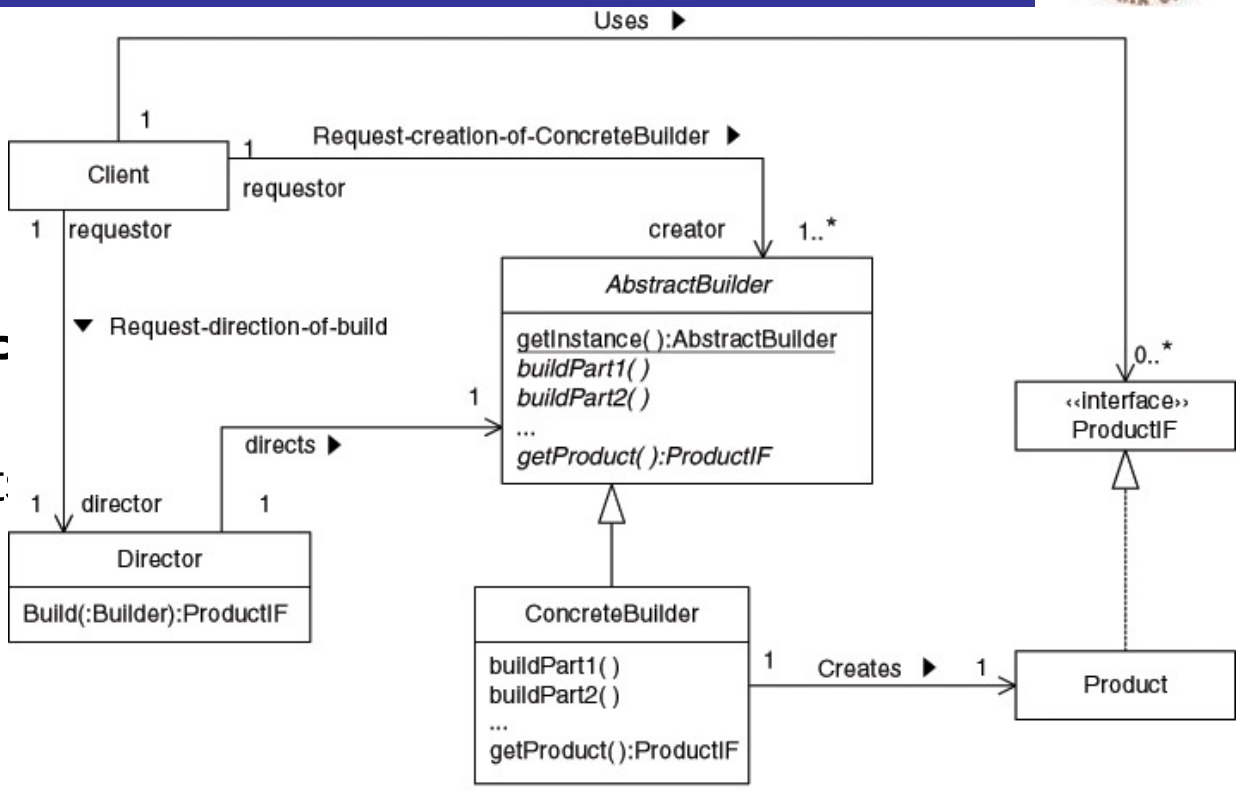


Builder Pattern

- Forces
  - A program must to be able to produce multiple external representations of the same data.
  - The classes responsible for providing content should be independent of any external data representation and the classes that build them.
  - The classes responsible for building external data representations are independent of the classes that provide the content.

Builder Pattern
Structure Elements

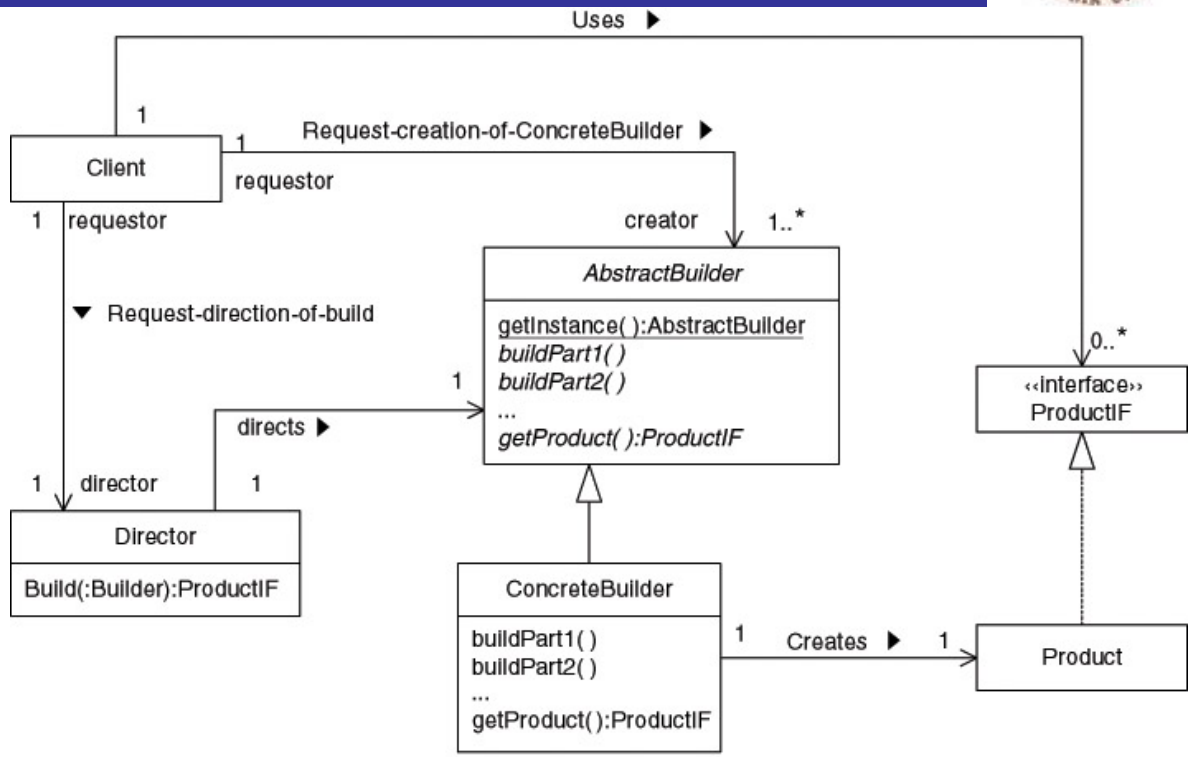
- **Product.** A class in this role defines a type of data representation. All Product classes should implement the **P** that other classes can refer to **Product** object through the interface.



- **ProductIF**. To avoid the need for Client objects to know the actual class of **Product** objects built for them, all **Product** classes implement the **ProductIFinterface**.

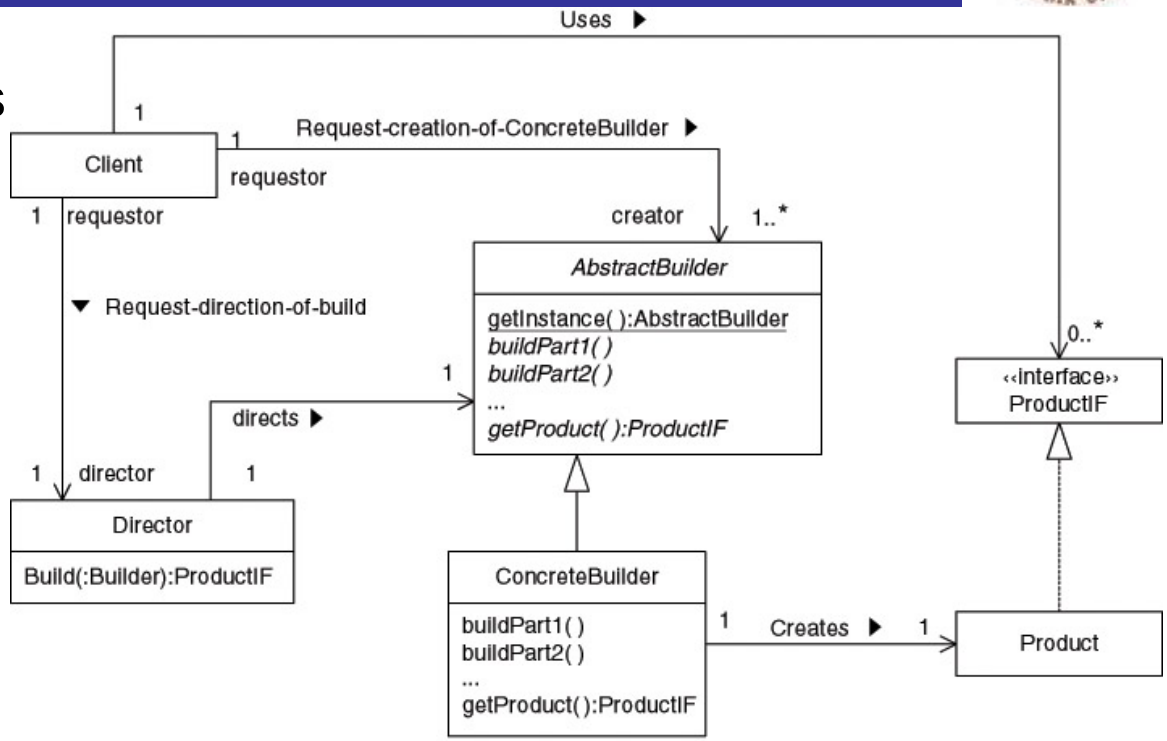
Builder Pattern
Structure Elements

- **Client**. An instance of a client class initiates the actions of the Builder pattern.
  It calls the **AbstractBuilderclass's** *getInstance* method.
  It passes information to *getInstance* telling it what sort of product it wants to have built.



The *getInstance* method determines the subclass of **AbstractBuilder** to create and returns it to the **Client** object. **The** Client object then passes the object it got from *getInstance* to a **Director** object's build method, which builds the desired object.

Builder Pattern
Structure Elements



- **ConcreteBuilder**. A class in this role is a concrete subclass of the **AbstractBuilder** class that is used to build a specific kind of data representation of a **Director** object.
- **AbstractBuilder**. A class in this role is the abstract superclass of **ConcreteBuilder** classes.
- **Director**. A Director object calls the methods of a concrete builder object to provide the concrete builder with the content for the product object that it builds.