Design Patterns

M.HEMASRI

What a Design pattern ?

Each pattern describe a problem which occurs over and over in environment and describe the core solution to that problem.

pattern=problem+solution

Essential Elements of pattern

- There are 4 Essential elements

1. Pattern Name

- Act as a handle to describe a design problem,its solution and consequences.
- Increases our design vocabulary.
- Helps to talk about them with collegues,in documentation,and even to ourselves.
- It easier to think about designs and to communicate them and their trade offs to others.
- Finding name is hardest part of developing a catlog.

2. Problem

- Describes when to apply the pattern
- It explains about the problem and its context
- Some specific problems are how to represent algorithm as objects.

- It might describe class or object structures that are sympotomatic of an inflexible design
- Sometimes the problem may include a set of conditions to be met before applying a pattern.

3. Solution:

- A solution describes elements that make up the design, their relationship, responsibilities and collaborations.
- Solution does not describe a particular concrete design or implementation because a pattern is like a template that can applied in many situations.
- Instead the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solve it

- 4.Consequences:
  - Are the results and tradeoffs of applying the pattern
  - They are critical for evaluating design alternatives
  - Understanding costs (space and time)and benefits of applying the pattern

Describing Design Patterns

- How do we describe design patterns?
- Graphical notations.
- The end product of the design process as relationship between classes and objects.
- To reuse the design, we must record the decisions, alternatives, and trade offs that led to it.
- Each pattern is divided into sections according to the following template.
- The template lens a uniform structure to the information, making design patters easier to learn, compare and use

1. Pattern Name and Classification
   - The name conveys essence of the pattern
2. Intent
   - What does the design pattern do?
   - What particular design issue or problem does it address?
3. Also Known As
   - other well known name for the pattern,if any
4. Motivation
   - A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem

5. Applicability

- What are the situations in which the design pattern can be applied?

How can you recognize these situations

6. Structure

- A graphical representation of the classes in the pattern using a notation based on QMT(Object Modeling Technique)
- We also use interaction diagram to illustrate sequence of requests and collaborations between objects .

7. Participants

- The classes and objects participating in the design pattern and their responsibilities.

8. Collaborations

- How the participants collaborate to carry out their responsibilities

9. Consequences

- How does the pattern support its objectives?
- What are the trade offs and results of using the pattern?
- What aspect of system structure does it let you vary independently?

10. Implementation

- What pitfalls, hints, or techniques should you be aware of when implementing the pattern?
- Are there language specific issues?

11. Sample code
    - Code fragments that illustrate how you might implement the pattern in C++,java
12. Known Uses
    - Examples of the pattern found in real systems.
13. Related Patterns
    - What design patterns are closely related to this one ? What are the important differences ? with which other patterns should this one be used?

The catalog of Design Patterns

Design Pattern Space

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory method	Adapter (class)	Interpreter Template method
	Object	**Abstract factory** Builder Prototype **Singleton**	**Adapter (object)** Bridge Composite Decorator Façade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento **Observer** State Strategy Visitor

Creational Patterns

- Creational Design Pattern responsibility is to create objects/controls the creation of objects.
- Types of Creational Pattern
  1. Prototype Pattern
  2. Singleton Pattern
  3. Factory Pattern
  4. Abstract Factory Pattern
  5. Builder Pattern

Prototype Design Pattern

- It is used when we have to make Copy/Clone from Existing Object.

Eg: Expensive Robot ,cloning many robots by copying 98% properties or instance variables and do minor modification by changing some features different .

```
public class Student{
    int age;
    private int rollNumber;
    String name;
    Student(){

    }
    Student(int age,int rollNumber,String name){
        this.age=age;
        this.rollNumber=rollNumber;
        this.name=name;
    }
}
```

```
public class StudentMain {
    public static void main(String args[]){
        Student obj=new Student(20,76,"Rani");
        //creating clone of obj
        Student cloneobj= new Student();
        cloneobj.age=obj.age;
        cloneobj.rollNumber=obj.rollNumber;
        cloneobj.name=obj.name;
        System.out.println(cloneobj.age);
        System.out.println(cloneobj.name);
        System.out.println(cloneobj.rollNumber);
    }
}
```

Problems with traditional cloning

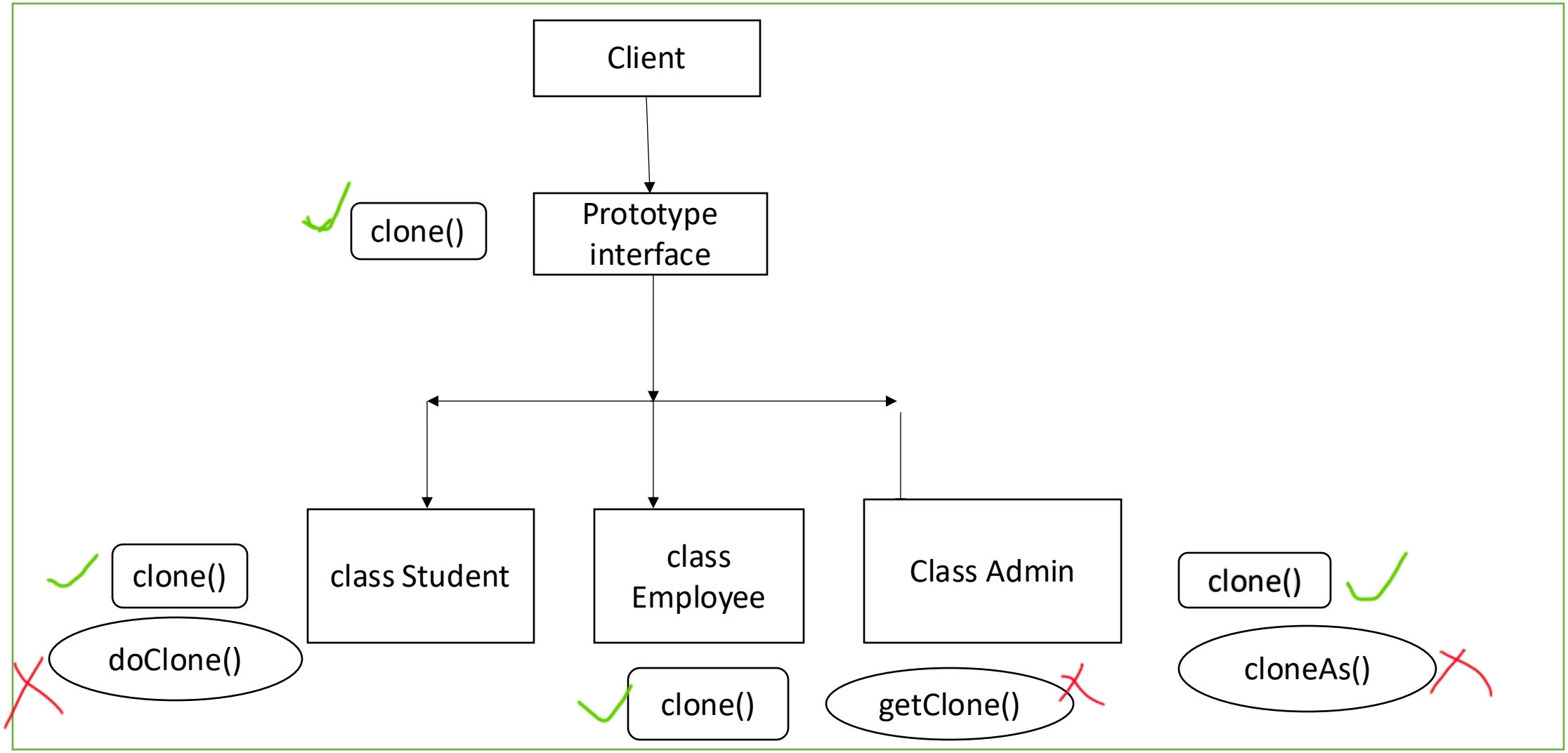
1st problem

Here client (main) has to know about all instances variables of the object to clone or copy and tells say out of 100, what to copy ? ,what to leave?

2nd problem

- private members are only available within a class outside class not available ,their getter method can also be private ,
- how will you set private instance variables in current way of cloning ?

Prototype Resloved



- Cloning logic should not be responsible of client(main) ,it should be responsible of class (Student) itself.
- By this clone () can access all private instance variables and also know about all instance variable of object ,what to copy and all .
- With prototype interface consistency in method name clone() comes among all class who wants to support cloning by implementing prototype interface by exposing method clone().

```
public class Main{
Public static void main()
{
        Student obj1= new Student(20,75,”ram”);
        Student cloneObj = (Student) obj.clone();
}
}

Public interface Prototype {
        Prototype clone();
}
```

```
Public class Student implements Prototype{
        int age;
        private int rollNumber;
        String name;
        Student(){
}
        Student(int age,int rollNumber,String name){
                this.age=age;
                this.rollNmuber=rollNumber;
                this.name=name;
        }
        public Prototype clone() {
                return new Student(age,rollNumber,name);
        }
}
```

Singleton Pattern

- It is used when we have to create only 1 instance of the class

Usecases: Db connection, ATM

4 ways to achieve this:

1.Eager

2. Lasy

3. Synchronized method

4.Double Locking(used in industry)

Eager Initialization

- First, we have to restrict creation of object who creates object ? constructor.
- Make constructor private ,so outside of the class no one can call constructor.
- Expose one getinstance() method ,make sure that it will return only one object.

```
public class DBConnection {
        private static DBConnection conObject = new DBConnection();
        private DBConnection(){
        }
        public static DBConnection getInstance(){
                return conobject;
        }
}
```

```
public class Main{
    public static void main(String args[]) {
        DBConnection conObject=DBConnection.getInstance();
        }

    }
```

Lazy Initialization

When ever requirement of object , then only create an object.

```
public class DBConnection {
    private static DBConnection conObject;
    Private DBConnection() {
    }
    public static DBConnection getInstance() {
                if(conObject ==null) {
                        conObject = new DBConnection();
                }
                return conObject;
                }
    }
```

Problem with lazy

- When more than one thread call getInstance() method simultaneously as conObject ==null is true,then more than one object will get created in memory.
- This can be handled through Synchronized method.

Synchronized Method

```
public class DBConnection {
    private static DBConnection conObject;
    private DBConnection() {
    }
    synchronized public static DBConnection getInstance() {
        If(conObject == null) {
            conObject = new DBConnection();
            }
            return conObject;
            }
    }
```

Problem with synchronized method

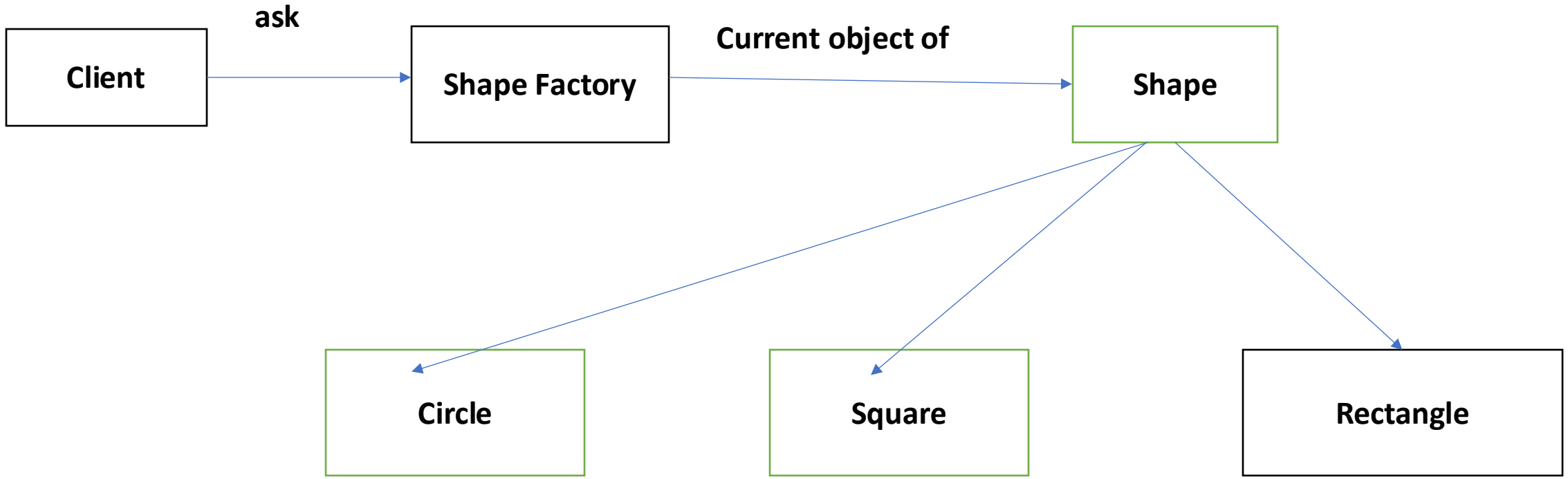
- If many threads calling getInstance method , synchronized will allow only one object to go into method and lock method till completion of execution.
- Many locks will be taken place if more than one thread is called.so it is expansive. So double locking is used.

Double locking method

```
public class DBConnection {
        private static DBConnection conObject;
        private DBConnection() {
        }
        public static DBConnection getInstance() {
                If(conObject == null) {
                        synchronized  (DBConnection.class) {
                                If(conObject == null) {
                                        conObject = new DBConnection();
                                                                }
                                   }
                                }
                                return conObject;
                                }
                        }
```

Factory method

- It is used when all the object creation and its business logic we need to keep at one place .



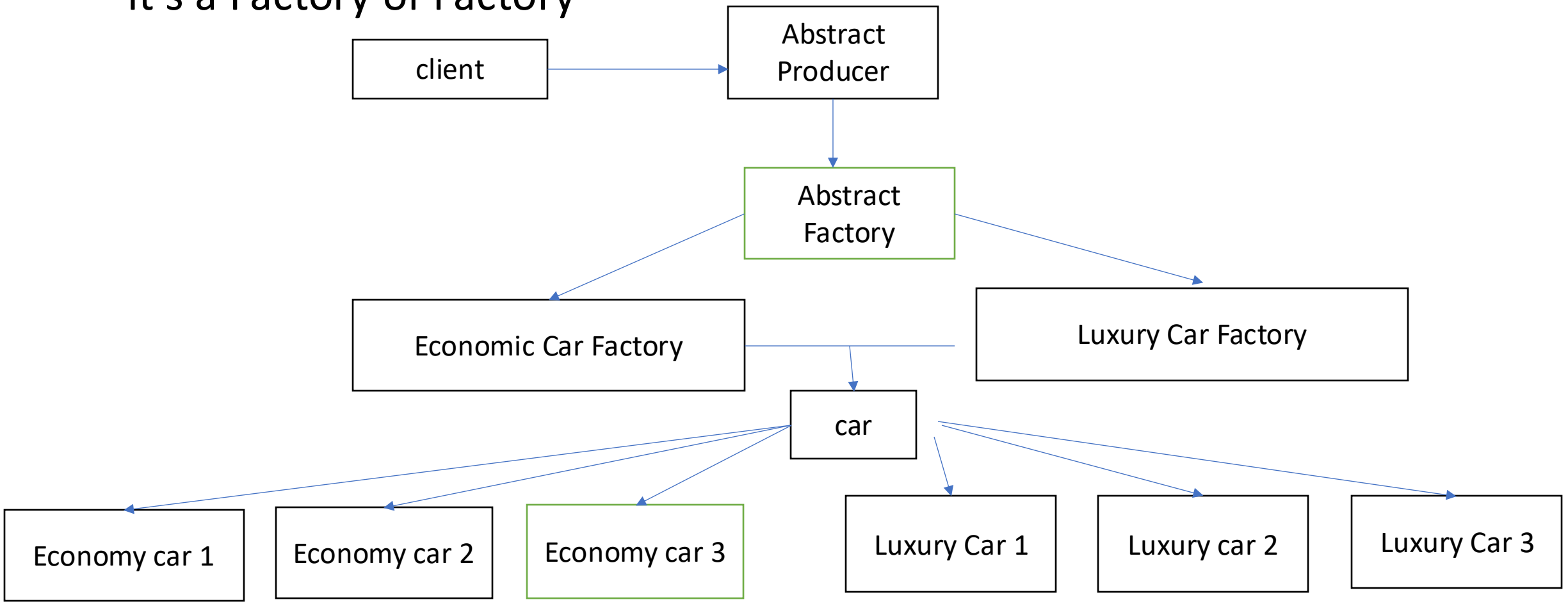
```
public interface Shape {
        public void computeArea();
 }
public class Square implements Shape {
        public void computeArea() {
                System.out.println(“compute Suare Area”);
        }
}
```

```
public class Circle implements Shape {
    public void computeArea() {
        System.out.println("Comput Circle Area");
        }
        }
        public class  ShapeInstanceFactory {
                public Shape getShapeInstance(String value) {
                        if(value.equals("Circle")) {
                                return new Circle();
                        }
                        else if(value.equals("Square")) {
                                return new Square();
        }
```

```
                else if (value.equals("Rectangle")) {
                        return new Rectangle();
                        }
                return null;
        }
}
public class Main() {
        public static void main(String args[]) {
                ShapeInstanceFactory factoryObj =new ShapeInstanceFactory();
                Shape circleObj= factoryObj.getShapeInstance("Circle");
                circleObj.computeArea();
                }
}
```

Abstract Factory Pattern

- It’s a Factory of Factory



```
public interface AbstractFactory {
	public Car getInstance(int price);
}
public class EConomicFactory implements AbstractFactory {
	public Car getinstance(int price) {
		if(price<=300000) {
			return new EconomicCar1();
		}else if (price>300000) {
			return new EconomicCar2();
		}
}
```

```
public class LuxuryFactory implements AbstractFactory {
        public Car getInstance (int price) {
                if(price >= 1000000 && price<=2000000) {
                        return new LuxuryCar1();
                }
                if(price>2000000) {
                        return new LuxuryCar2();
                }
                return null; }
```

```
public class AbstractFactoryProducer {
    public AbstractFactory  getFactoryInstance(String value) {
                if(value.equals(“Economic”)) {
                        return new EconomicCarFactory();
                }
                else if(value.equals(“Luxury”) \\ value.equals(“premium”)) {
                        return new LuxuryCarFactory();
                }
                return null;
    }
    }
```

```
public interface Car {
        public int getTopSpeed();
        }
public class EconomicCar1 implements Car {
        public int getTopSpeed() {
                return 100;
        }
public class EconomicCar2 implements Car {
        public int getTopSpeed() {
                return 150;
        }
```

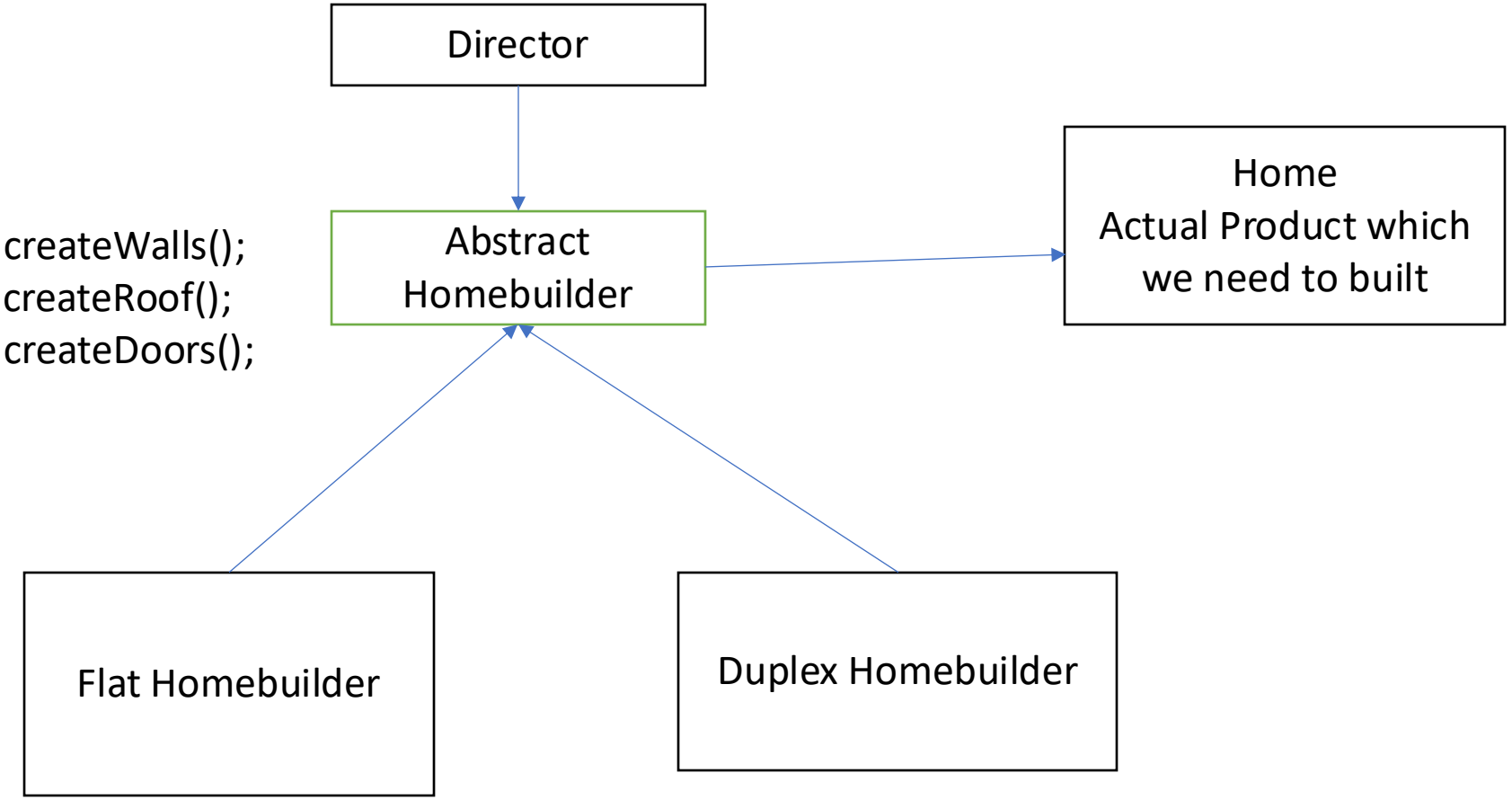
```
public class LuxruryCar1 implements Car {
        public int getTopSpeed() {
                return 300;
        }
public class LuxruruyCar2 implements Car {
         public int getTopSpeed() {
                return 250;
        }
}
```

```
public class Main{
        public static void main(String args[])  {
                AbstractFactoryProducer abstractFactoryProducerOb=new
AbstractFactoryProducer();
                AbstractFactory abstractFactoryObj
=abstractFactoryProducerOb.getFactoryInstance(“Premiun”);
        Car carObj= abstractFactoryObj.getInstance(5000000);
        System.out.println(carObj.getTopSpeed());
        }
}
```

Builder Pattern

- When you want to create Object Step by Step.

AbstractHomeBuilder createWalls();
AbstractHomebuilder createRoof();
AbstractHomeBuilder createDoors();
AbstractHomeBuilder
createWindows();
Home build();



```
public class Student {
        int rollNumber;
        int age;
        String fatherName;
        String motherName;
        List<String> subjects;
        public Student(int rollNumber,int age, String fatherName,String motherName)
         {
                this.rollNumber = rollNumber;
                this.age = age;
                this. fatherName= fatherName;
                this. motherName= motherName;
        }
```

```
public Student(int rollNumber,int age, String fatherName)
                {
                        this.rollNumber = rollNumber;
                        this.age = age;
                        this. fatherName= fatherName;
                }
public Student(int rollNumber,int age, String motherName)
                {
                        this.rollNumber = rollNumber;
                        this.age = age;
                        this. fatherName= fatherName;
                        this. motherName= motherName;
                }
}
```

```
public class Client {
        public static void main(String args[]) {
Director directorObj1= new Director(new EngineeringStudentBuilder());
Director directorObj2 = new Director(new MBAStudentBuilder());
Student engineerStudent = directorObj1.createStudent();
Student mbaStudent=directorObj2.createStudent();
System.out.println(engineerStudent.toString());
System.out.println(mbaStudent.toString());
}   }
```

```
public class Director {
        StudentBuilder studentBuilder;
        Director(StudentBuilder studentBuilder) { this.studentBuilder =
studentBuilder; }
public Student createStudent() {
        if(studentBuilder instanceof EngineeringStudentBuilder) {
                return createEngineeringStudent();
        }else if(studentBuilder instanceof MBAStudentBuilder) {
                return createMBAStudent();  }
        return null;      }
```

```
private Student createEngineeringStudent() {


        return studentBuilder.setRollNumber
1).setAge(22).setName(sj).setSubject().build();}
}
private Student createMBAStudents() {
        return
studentBuilder.setRollNumber(2),setAge(74),setName(“sj”),setFatherName(“
My Father name”),setMotherName(“My MotherName”);
}
}
```

```
public abstract class StudentBuilder {
        int RollNumber;
        int age;
        String name;
        String FatherName;
        String MotherName;
        List<String> subjects;
```

```
public StudentBuilder setRollNumber(int rollNumber) {
        this.rollNumber=rollNumber;
        return this;
        }
public StudentBuilder setAge(int age) {
                this.age=age;
                return this;
        }
```

```
public StudentBuilder setName(String name) {
        this.ame=name;
        return this;
        }
public StudentBuilder setFatherName(String fatherName) {
        this.FatherName= fatherName;
        return this;
        }
```

```
public StudentBuilder setMotherName(String MotherName) {
        this.MotherName= MotherName;
        return this;
        }
abstract public StudentBuilder setSubjects();
public Student build() {
                return new Student(this);
        }
```

```
public class MBAStudentBuilder extends StudentBuilder {
        public StudentBuilder setSubjects() {
                List<String> subs =new ArrayList<>();
                subs.add(“Micro Economics”);
                subs.add(“Business Studies”);
                subs.add(“operations management”);
                this.subjects =subs;
                return this;
                }
        }
```

```
public class EngineeringStudentBuilder extends Studentbuilder {
        public StudentBuilder setSubjects() {
                List<String> subs = new ArrayList<>();
                subs.add(“DSA”);
                subs.add(“OS”);
                subs.add(“Computer Architecture”);
                this.subjects = subs;
                return this;
                }
        }
```

```
public class Student {
        int rollNumber;
        int age;
        String fatherName;
        String motherName;
        List<String> subjects;
        public Student(StudentBuilder builder) {
                this.rollNumber = builder.rollNumber;
                this.age = builder.age;
```

```
        this.name=builder.name;
        this.fatherName=builder.fatherName;
        this.motherName=builder.motherName;
        this.subjects= builder.subjects;
        }
public String toString() {
        return  “”+ “ roll number: “ + this.rollNumber + “age:” + this.age
+ “name:” + this .name + “father name:” + this.fatherName + “mother
name:” + this.motherName + “subjects:” + subjects.get(0) + “,” +
subjects.get(1);     }       }
```