Design Patterns 1

Week 11

A design that doesn't take change into account risks major redesign in the future

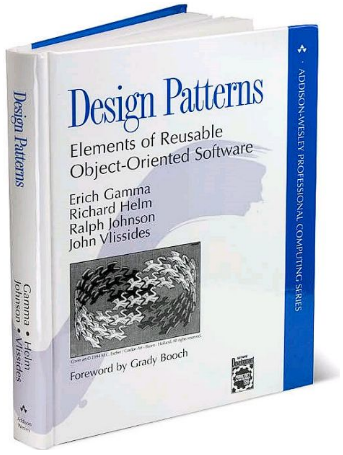
– Erich Gamma

Objectives

- Learn about the concept of design patterns
- Understand the various design patterns for object-oriented programming

Contents

- The basis of design pattern
- SOLID design principle for object-oriented program
- Gangs of Four (GoF) design patterns
  - Creational patterns
  - Structural patterns
  - Behavioral patterns



What is the Design Pattern?

- Christopher Alexander, an architect, studied ways to improve the process of designing buildings
  - “A pattern describes a problem that occurs often, along with a tried solution to the problem”
  - Later, people realized that the patterns are applicable to software development, too!

- Experts often recall
  - similar problems they have already solved
  - reuse the essence of its solution

Why Use Design Patterns?

- Designing object-oriented software is hard,
  Designing reusable object-oriented software is even harder
  - Erich Gamma
- Experienced designers reuse solutions which were proved to work in the past
- Experience = toolbox of reusable solutions
  - Classify problems and apply solution templates

Design Patterns in Object-Oriented Program

- **SOLID** is a widely-used OOP design principles proposed by Robert C. Martin (in his 2000 paper *Design Principles and Design Patterns*)
    - **S**ingle responsibility principle
    - **O**pen-closed principle
    - **L**iskov substitution principle
    - **I**nterface segregation principle
    - **D**ependency inversion principle

Single Responsibility Principle (SRP)

- A class should have **only one reason to exist**, meaning that a class should handle only one job
- What happens if a class does more than one job?
    - Other jobs may be accidently affected when you change the code
    - Hard to know where to change when working on a large codebase
- The same applies to design packages, methods, etc.

Design with SRP

```java
public class UserSettingService {
  public void changeEmail(User user) {
    if (checkAccess(user)) {
       // Change user's email.
    }
  }

  public boolean checkAccess(User user) {
    // Verify if the user is valid.
  }
}
```

Without SRP

```java
public class UserSettingService {

  public void changeEmail(User user) {

       // Change user's email

  }

}

public class SecurityService {

  public boolean checkAccess(User
user) {

     // Verify if the user is valid.

  }
}
```

With SRP

Open-Closed Principle (OCP)

- Objects or entities should be open for extension, but closed for modification
- When not followed,
  - you need to directly change the existing code, which may cause many unexpected side effects
  - you often need to write many if-else statements with frequent type checking and down casting

Design without OCP

```java
public class AreaCalculator {
  public double totalArea(Shape[] shapes) {
    double area = 0;

    for (Shape shape : shapes) {
      if (shape instanceof Rectangle) {
        Rectangle rectangle = (Rectangle) shape;
        area += rectangle.width * rectangle.height;
      } else {
        Circle circle = (Circle) shape;
        area += circle.radius * circle.radius * Math.PI;
      }
    }
    return area;
  }
}
```

What if you want to add another shape?

Design with OCP (1/2)

```java
public interface Shape {
    double getArea();
}

public class Rectangle implements Shape {
    private double width;
    private double height;

    @Override
    public double getArea() {
        return width * height;
    }
}
```

Design with OCP (2/2)

```java
public class Circle implements Shape {
    private double radius;

    @Override
    public double getArea() {
        return radius*radius*Math.PI;
    }
}
```

```java
// used in AreaCalculator class
public double totalArea(Shape[]
shapes) {
    double area = 0;
    for (Shape shape : shapes) {
        area += shape.getArea();
    }
    return area;
}
```

We can easily add a new shape by implements *Shape* interface

Liskov Substitution Principle (LSP)

- Let q(x) be a property provable about an object x of type T. q(y) should be provable for an object y of type S where S is a subtype of T
- **Subclass** should **extend** the capabilities of its superclass, not reduce
- Every subclass should be substitutable for their superclass

LSP Example

- Consider `abstract class Bird`, the base class for specific kinds of birds such as Duck, Eagle, Pelican, Pigeons, etc.
- Subclasses of `Bird` contain the methods `setLocation`, `setAltitude`, and `draw` to show flying patterns on the map

```
abstract class Bird {
  abstract void setLocation(double lon, double lat);
  abstract void setAltitude(double alt);
  abstract void draw();
}
```

Design without LSP

- Now let's say we want to add a `Penguin` class, a new subclass of `Bird`
- A penguin *is-a* bird, but it cannot fly
- `setAltitude` of `Penguin` cannot be properly defined, and calling it raises a bug

```
class Penguin extends Bird {
  // Cannot set altitude because penguins cannot fly.
  @Override
  public void setAltitude(double alt) { }
}
```

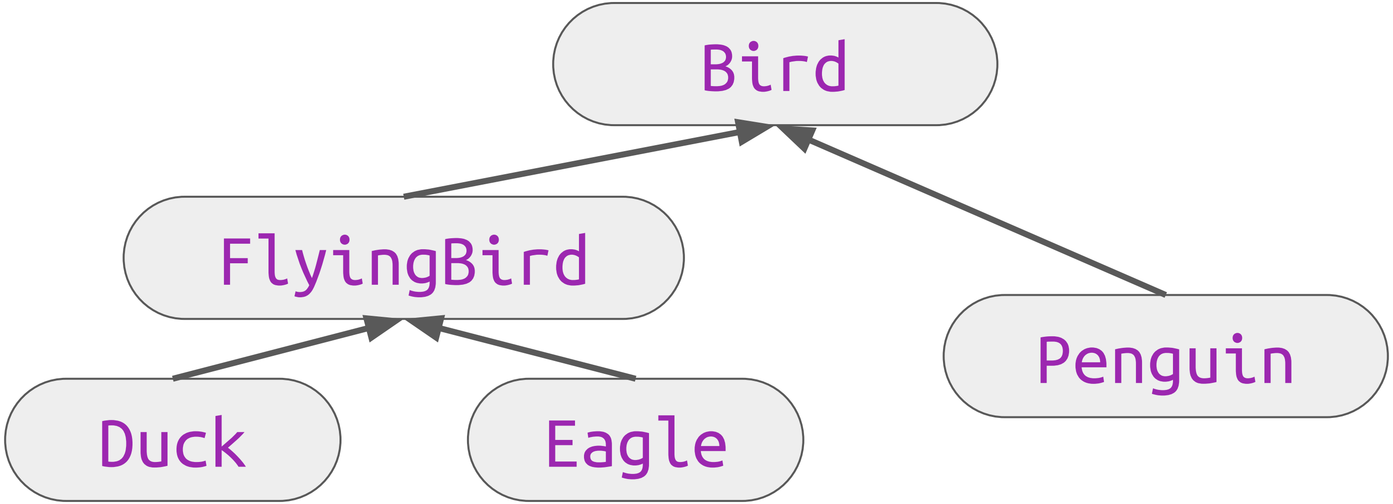
Design with LSP

- A solution is to add an intermediate class `FlyingBird`, and move `setAltitude` from `Bird` to `FlyingBird`

```
abstract class Bird {
  abstract void setLocation(double lon, double lat);
  abstract void setAltitude(double alt);
  abstract void draw();
}
abstract class FlyingBird extends Bird {
  abstract void setAltitude(double alt);
}
```

Design with LSP

- Make classes for flying birds by extending `FlyingBird` class
- Define `setAltitude` only in subclasses of `FlyingBird` class



Interface Segregation Principle (ISP)

- A client should not implement unnecessary interfaces or depend on methods they do not use
- Interfaces should be segregated based on usage

```
Interface Human {
  void run();
  void scubaDive();
  void study();
  void createContents();
  void deliverPizza();
  ...
}
```

```
// Student class only needs study method, but all methods of Human interface should be implemented.

class Student implements Human {
  @Override
  void study() { }
}
```

Design without ISP

```java
public interface ArticleService {
    void list();
    void write();
    void delete();
}

public class UIList implements ArticleService {
    @Override
    public void list() {}

    @Override
    public void write() {}

    @Override
    public void delete() {}
}
```

```java
public class UIWrite implements
ArticleService {
    @Override
    public void list() {}

    @Override
    public void write() {}

    @Override
    public void delete() {}
}
```

Design with ISP

```java
public interface ArticleListService { void list(); }

public interface ArticleWriteService { void write(); }

public interface ArticleDeleteService { void delete(); }
```

```java
public class UIList implements ArticleListService {
    @Override
    public void list() { }
}

public class UIWrite implements ArticleWriteService {
    @Override
    public void write() { }
}
```

Dependency Inversion Principle (DIP)

- Entities must depend on abstractions (e.g., interface) not on concreations (e.g., class)
- The high-level module must not depend on the low-level module, but they should depend on abstractions
- For example, develop a program in this order:
    - Design packages
    - Design class inheritance tree
    - Implement public methods
    - Implement private methods

Design without DIP

```java
// "Low level Module" equivalent
public class Logger {
    public void logInformation(String logInfo) {
        System.out.println(logInfo);
    }
}

// "High level module" equivalent.
public class Foo {
    // direct dependency to a low level module.
    private Logger logger = new Logger();

    public void doStuff() {
        logger.logInformation("Something important.");
    }
}
```

Design with DIP (1/2)

```
public interface ILogger {
    void logInformation(String logInfo);
}

public class GoodLogger implements ILogger {
    @Override
    public void logInformation(string logInfo) {
        System.out.println(logInfo);
    }
}
```

Design with DIP (2/2)

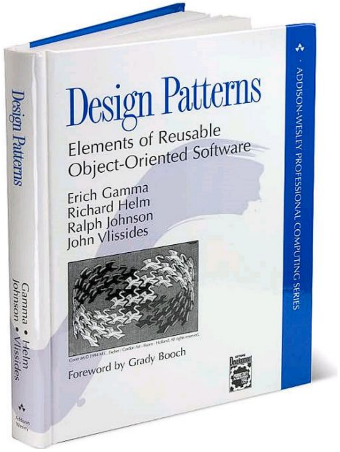
```
public class Foo {
    private ILogger logger;
    public void setLoggerImpl(ILogger loggerImpl) {
        this.logger = loggerImpl;
    }
    public void doStuff() {
        logger.logInformation("Something important.");
    }
}
```

Main method

```
Foo foo = new Foo();
// Any class implementing ILogger is allowed.
ILogger logger = new GoodLogger(); foo.setLoggerImpl(logger);
foo.doStuff();
```

The Gang of Four (GoF)

- 23 design patterns
- Description of communicating objects and classes
  - Captures common solution to a category of related problems
  - Can be customized to solve a specific problem in that category
- Pattern is not
  - Individual classes of libraries (list, hash, …)
  - Full design



GoF Classification of Design Patterns

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

Creational Patterns

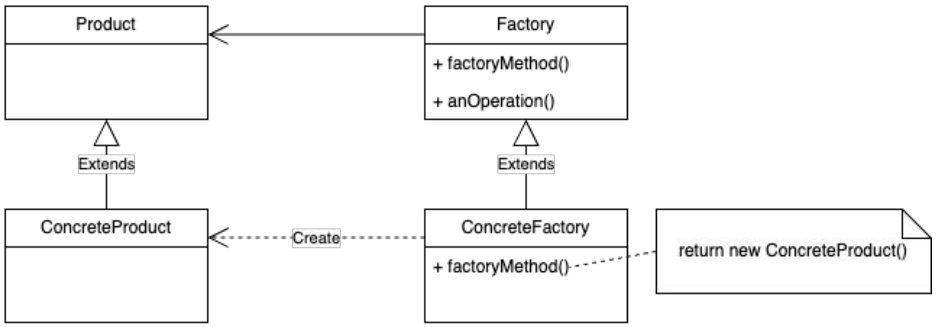
- Creational patterns involve object instantiation
- They encapsulate which specific class objects are instantiated
- List of creational patterns
  - Factory patterns (Factory Method and Abstract Factory)
  - Builder
  - Prototype
  - Singleton

Factory Patterns

- We create objects (with “new” keyword) in many parts
- It is hard to change every object creation code when classes are modified or added
- **Factory patterns** encapsulate object creation logic and provide a common interface to access new objects
- Two mechanisms
  - **Factory Method** with **inheritance**
  - **Abstract Factory** with **composition** and **delegation**

Factory Method

- Factory Method defines an interface for creating an object, but lets subclasses decide which class to instantiate
- It lets a class defer instantiation to subclasses
- Use when
  - a class cannot anticipate the class of objects it must create
  - the creation of objects have complex logic or patterns



Code without Factory Method (1/2)

Product (Interface)

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

ConcreteProduct (Class)

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

ConcreteProduct (Class)

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

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

Code without Factory Method (2/2)

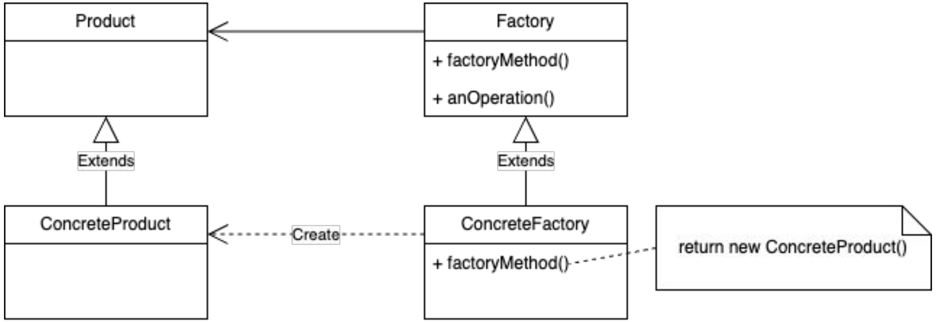
```java
public class Client {
    public static void main(String args[]){
        Shape shape1;
        String type = "Circle";

        // We want to initialize circle object
        if (type.equals("Circle")){
            shape1 = new Circle();
        } else if(type.equals("Rectangle")){
            shape1 = new Rectangle();
        }else{
            shape1 = new Square();
        }
```

```java
        shape1.draw();

        Shape shape2;
        type = "Rectangle";
        if ...
        // We have to write many if-else statement
    }
}
```

You need to change client’s code every time when you add more types

Code with Factory Method (1/2)

```
public abstract class ShapeFactory {
    public void info(){
        System.out.println("This is factory");
    }
    public abstract Shape createShape(String type);
}
```




```
public class TypedShapeFactory extends ShapeFactory{
    @Override
    public Shape createShape(String type) {
        if (type.equals("Circle")){
            return new Circle();
        } else if(type.equals("Rectangle")){
            return new Rectangle();
        }
        return new Square();
    }
}
```

Similarly, RandomShapeFactory may exist to return a new random shape

Code with Factory Method (2/2)

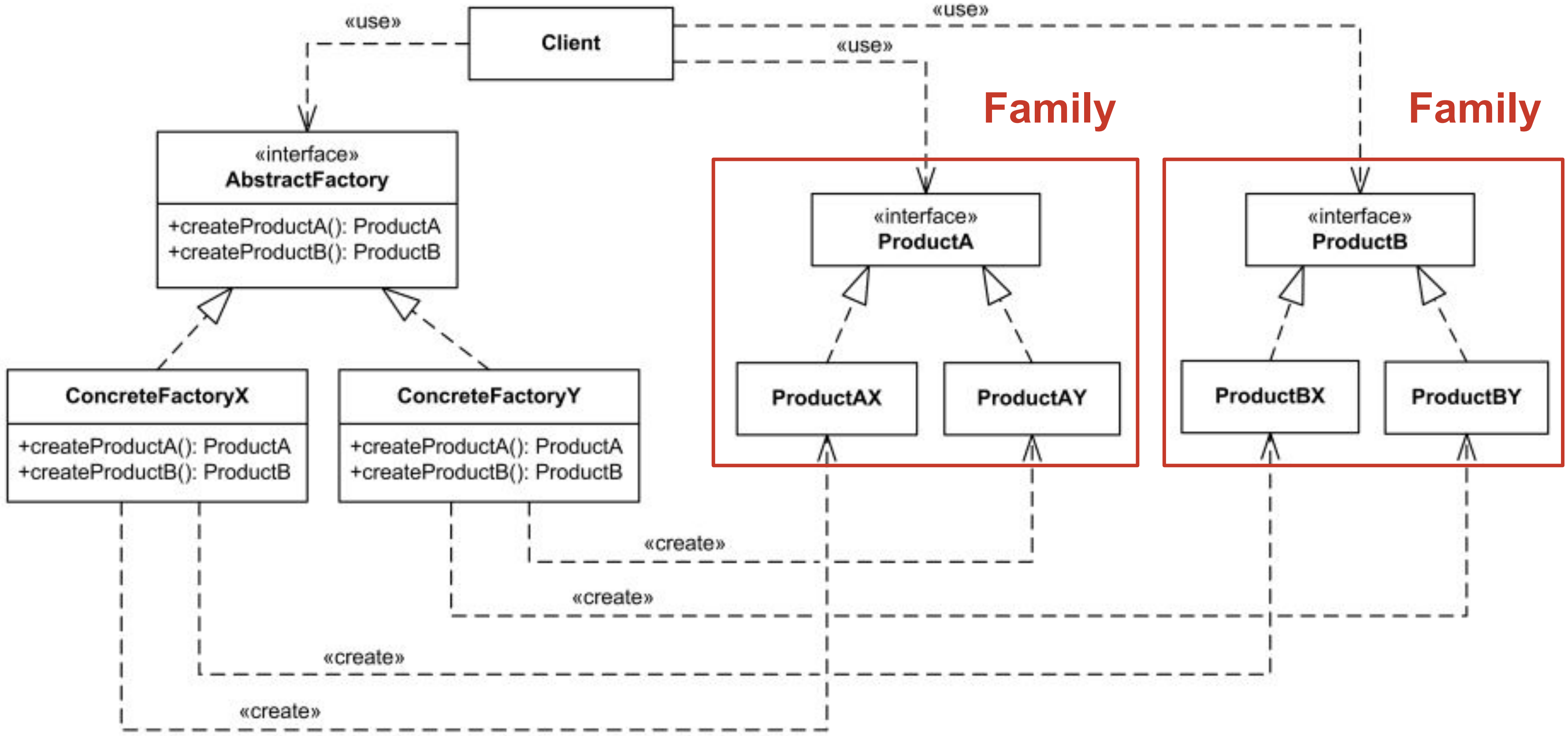
- Client doesn't need to write many if-else statements
- When we need to add another shape, we can modify only `ShapeFactory` or create more concrete factories

```
public class Client {

    public static void main(String args[]){

        Factory shapeFactory = new TypedShapeFactory();

        Shape shape1 = shapeFactory.createShape("Circle");

        shape1.draw();


        Shape shape2 = shapeFactory.createShape("Rectangle");

        shape2.draw();

    }

}
```

Abstract Factory

- Abstract Factory method provides an interface for creating **families** of related or dependent objects without specifying their concrete classes
- Use when
  - Families of objects must be used together
  - The creation of **objects** should be independent of the system utilizing them

Abstract Factory



Code without Abstract Factory (1/2)

ProductA, ProductB Interfaces

```java
public interface Mouse
{
    void click();
}
public interface Button
{
    void push();
}
```

ProductAX, ProductAY

```java
public class MacOSMouse implements Mouse {
    @Override
    public void click() {
        System.out.println("MacOS click");
    }
}
public class WindowsMouse implements Mouse{
    @Override
    public void click() {
        System.out.println("Windows click");
    }
}
```

ProductBX, ProductBY

```java
public class MacOSButton implements Button {
    @Override
    public void push() {
        System.out.println("MacOS push");
    }
}
public class WindowsButton implements Button{
    @Override
    public void push() {
        System.out.println("Windows push");
    }
}
```

Code without Abstract Factory (2/2)

```
public static void main(String args[]) {
    Mouse mouse;
    Button button;
    String os = "MacOS";

    if(os.equals("MacOS")){
        mouse = new MacOSMouse();
        button = new MacOSButton();
    } else{
        mouse = new WindowsMouse();
        button = new WindowsButton();
    }
    mouse.click();
    button.push();
```

```
    os = "Windows";
    if ...
    // We have to write an if-else statement every time
}
```

The problem is when there is a new GUI component, you need to add many if-else statements

Code with Abstract Factory (1/2)

AbstractFactory

```
public interface GUIFactory {
    Mouse createMouse();
    Button createButton();
}
```

ConcreteFactory

```
public class MacOSFactory implements GUIFactory{
    @Override
    public Mouse createMouse() {
        return new MacOSMouse();
    }
    @Override
    public Button createButton() {
        return new MacOSButton();
    }
}
```

```
public class WindowsFactory implements GUIFactory{
    @Override
    public Mouse createMouse() {
        return new WindowsMouse();
    }
    @Override
    public Button createButton() {
        return new WindowsButton();
    }
}
```

Code with Abstract Factory (2/2)

When there is a new GUI component, client doesn't need to add new lines in each if-else statements

```
public class Client {
    public static void main(String args[]){
        GUIFactory guiFactory;
        String os = "MacOS";

        if (os.equals("MacOS")){
            guiFactory = new MacOSFactory();
        } else{
            guiFactory = new WindowsFactory();
        }
```

```
            Mouse mouse = guiFactory.createMouse();
            Button button = guiFactory.createButton();

            mouse.click();
            button.push();
        }
    }
```

Factory Method vs. Abstract Factory

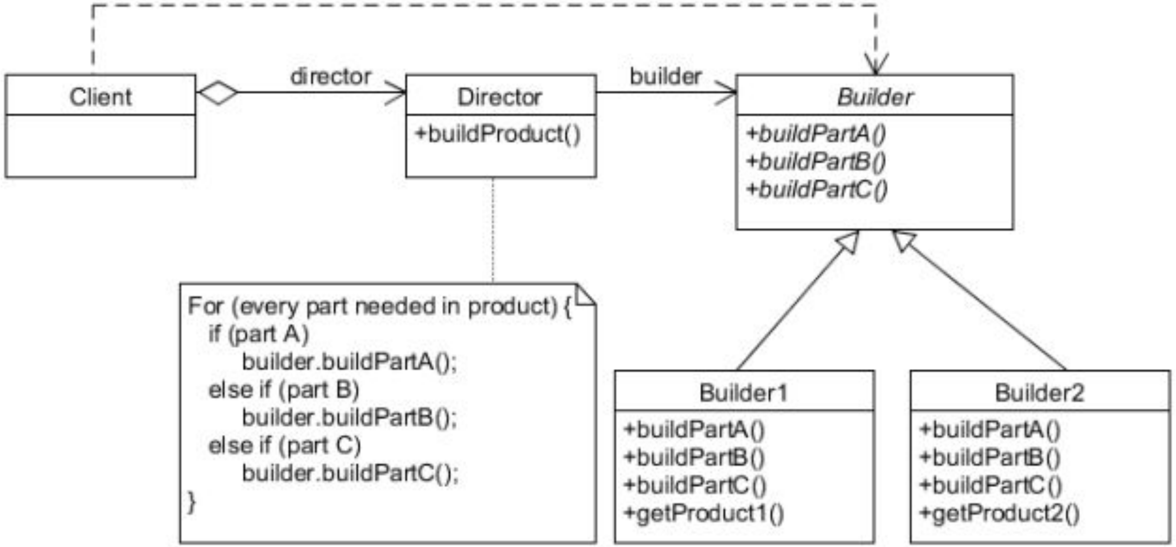
- The job of an Abstract Factory is to define an interface for creating a **set of products**
- Factory Methods are a natural way to implement your product methods in your Abstract Factories

Builder

- Builder encapsulates the construction of a product and allow it to be constructed in steps
- It allows for the dynamic creation of objects based upon easily interchangeable algorithms
- Use when
  - Runtime control over the creation process is required
  - Multiple representations of creation algorithms are required

Builder

- Director knows what parts are needed for the final product
- Concrete builder knows how to produce the part and add it to the final product



Code without Builder

```
public class Car{
    private int id;
    Private String brand;
    private String model;
    ....
    public Car(int id, String brand, String model, ...){
        this.id = id;
        this.brand = brand;
        ...
    }
    public Car(...) {...} // For other arguments
}
```

```
Car bugatti = new Car(brand,
color, engine);
Car lambo = new Car(brand, model,
color);
// Incorrect constructor
arguments order
Car bugatti = new Car(color,
brand, engine);
```

The construction process requires to allow different representations for the object constructed

Code with Builder: Builder

```
public class CarBuilder {
    private Car car;
    public CarBuilder() {
        this.car = new Car();
    }
    public CarBuilder id(int id){
        car.setId(id);
        return this;
    }
    ...
```

```
    ...
    public CarBuilder brand(String brand) {
        car.setBrand(brand);
        return this;
    } ... // Other methods

    public Car getCar(){
        return car;
    }
}
```

Code with Builder: Director & Client

```java
public class Director{
    public void buildBugatti(CarBuilder builder){
        builder.brand("Bugatti")
               .color("Blue")
               .engine("8L");
    }
    public void buildLambo(CarBuilder builder){
        builder.brand("Lamborghini")
               .model("Aventador")
               .color("Yellow");
    }
}
```

```java
Director director = new Director();
CarBuilder builder = new CarBuilder();
director.buildBugatti(builder);
Car car = builder.getCar();
```

Prototype

- Prototype creates objects based upon a template of an existing objects through **cloning**
- Use when
  - Objects or object structures are required that are identical or closely resemble other existing objects or object structures
  - The creation of an individual object is an expensive operation

Code without Prototype

```java
Car carA = new Car();
// Expensive initialization of carA


Car carB = new Car();
carB.setBrand(carA.getBrand());
carB.setModel(carA.getModel());
carB.setColor(carA.getColor());
carB.setTopSpeed(carA.getTopSpeed());
```

This cloning may not be possible for private fields

Code with Prototype

```
public interface Prototype{
    Car clone();
}
public class Car implements Prototype{
    private String brand;
    private String model
    private String color;
    private int topSpeed;
    public Car(){}
    .
    .
    .
```

```
    public Car(Car car){
        this.brand = car.brand;
        this.model = car.model;
        this.color = car.color;
        this.topSpeed = car.topSpeed;
    }
    @Override
    public Car clone(){
        return new Car(this);
    }
}
```

Delegates the object duplication or cloning process to the actual objects that are being cloned

Singleton

- Singleton ensures a class has **only one instance**, and provides a global point of access to it
- Use when
  - Exactly one instance of a class is required
  - Controlled access to a single object is necessary
- Difficulty
  - Global variables
  - Multi-threading issue

Singleton (Basic)

```java
public class Singleton{
    private static Singleton uniqueInstance;
    // other useful instance variables
    private Singleton() {}
    public static Singleton getInstance(){
        if (uniqueInstance == null){
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
    // other useful method
}
```

```java
public class Client {
    public void operation1(){
        Singleton singletonObject =
Singleton.getInstance();
        // Do operation 1
    }
    public void operation2(){
        Singleton singletonObject =
Singleton.getInstance();
        // Do operation 2
    }
}
```

Same object

Singleton (Example)

```java
public class Printer{
    private static Printer uniqueInstance;
    // other useful instance variables
    private Printer() {}
    public static Printer getInstance(){
        if (uniqueInstance == null){
            uniqueInstance = new Printer();
        }
        return uniqueInstance;
    }
    // other useful method
}
```

```java
public class Employee {
   public void usePrinter(String text){
       Printer printer = Printer.getInstance();
       printer.print(text);
   }
}
```

```java
public static void main(String[] args){
    Employee employee1 = new Employee();
    Employee employee2 = new Employee();
    employee1.usePrinter("Hi");
    // Use the same printer as employee1
    employee2.usePrinter("Hello");
}
```

Singleton (Basic) in Multi-threading

Thread 1	Thread 2	uniqueInstance
`public static Singleton getInstance(){`		null
	`public static Singleton getInstance(){`	null
`if (uniqueInstance == null){`		null
	`if (uniqueInstance == null){`	null
`uniqueInstance = new Singleton();`		<object 1>
`return uniqueInstance;`		<object 1>
	`uniqueInstance = new Singleton();`	<object 2>
	`return uniqueInstance;`	<object 2>

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

Singleton (Option 1) in Multi-threading

```
public class Singleton{
    private static Singleton instance;
    ...
    public static Singleton getInstance(){
        if (uniqueInstance == null){
            synchronized (Singleton.class){
                if (instance == null){
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

Solution option 1: Double-check

Sometimes fail due to the semantics of some programming language

1. Thread A obtains the lock and begin to initialize
2. Thread B notices the shared variable has been initialized (although A has not finished performing initialization)
3. Since Thread B uses the value before A finishes initialization, the program will likely crash

Singleton (Solution) in Multi-threading

```
public class Singleton{
    private static volatile Singleton instance;
    ...
    public static Singleton getInstance(){
        if (uniqueInstance == null){
            synchronized (Singleton.class){
                if (instance == null){
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

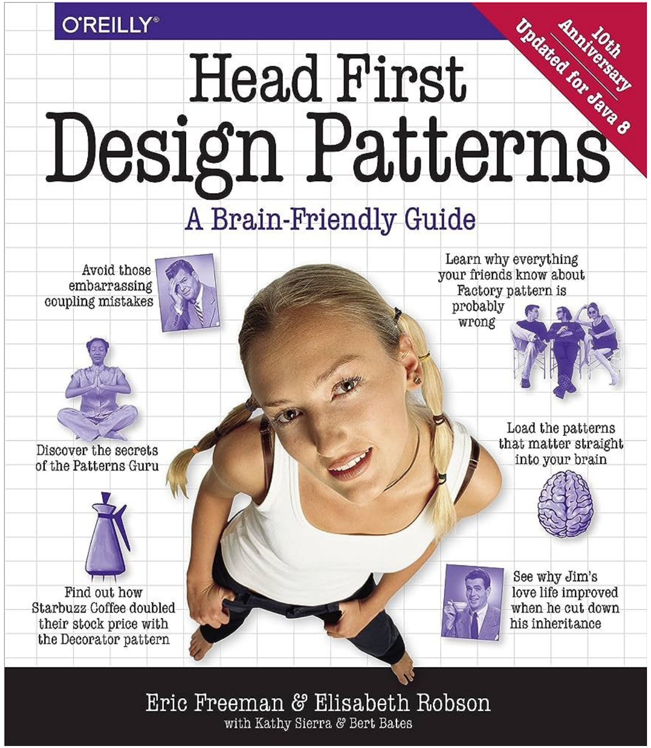
- **Solution**
  - **Using *volatile* keyword**
- Compiler does not optimize for volatile variables
- The volatile keyword specifies that the variable is 'stored in main memory’
- Thread B waits until Thread A finishes declaring the variable

Summary

- Design patterns provide reusable solutions to previously solved problems
- Creational patterns concern the process of object creation
- List of creational patterns
  - Factory Method
  - Abstract Factory
  - Builder
  - Prototype
  - Singleton

References

- Freeman Eric et al. *Head First Design Patterns*. 1st ed. O'Reilly 2004.



Thank You.
Any Questions?