LECTURE 02

Prototype and **Factory**

Software Development Principle 20~39

Prototype(原型模式)

Intent

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Motivation

In order to save costs, we usually use printers to print the contents of electronic documents onto A4 paper, and then use copiers to copy multiple copies of this paper document.

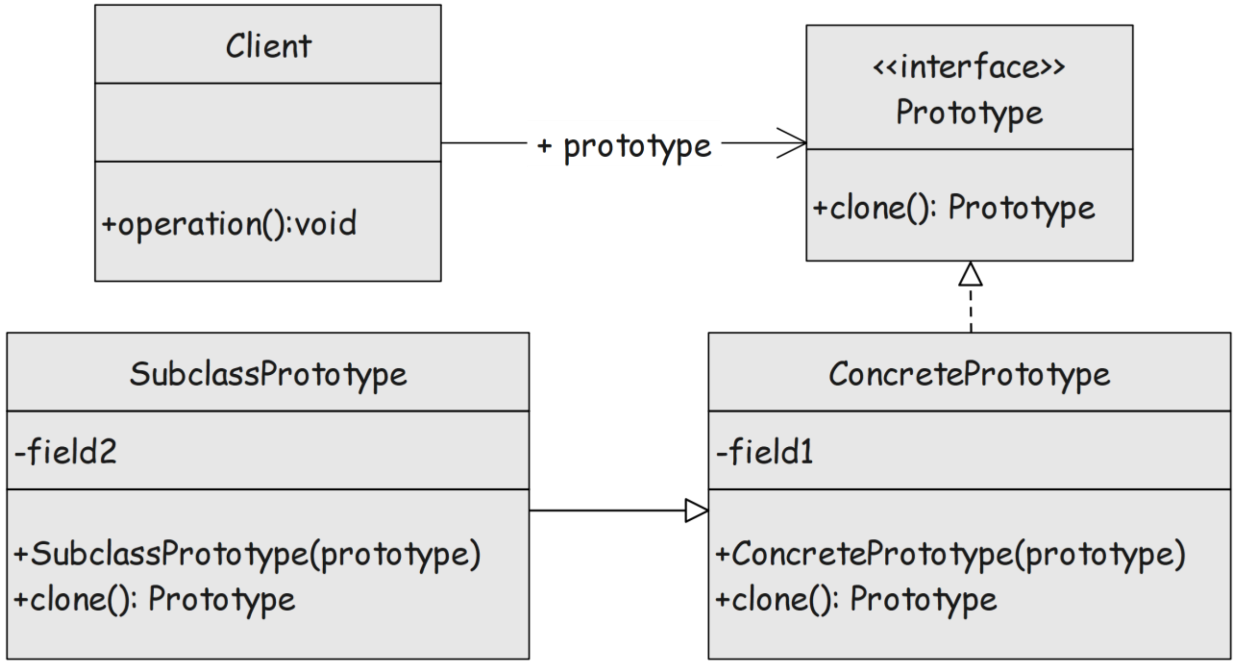
Applicability

- Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented;

- When the classes to instantiate are specified at run-time, for example, by dynamic loading;

- To avoid building a class hierarchy of factories that parallels the class hierarchy of products;

- When instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.


Client
+operation():void
+ prototype
<<interface>>
Prototype
+clone(): Prototype
SubclassPrototype
-field2
+SubclassPrototype(prototype)
+clone(): Prototype
ConcretePrototype
-field1
+ConcretePrototype(prototype)
+clone(): Prototype

Participants

Prototype (Graphic)

- declares an interface for cloning itself.

ConcretePrototype (Staff, WholeNote, HalfNote)

- implements an operation for cloning itself.

Client (GraphicTool)

- creates a new object by asking a prototype to clone itself.

Collaborations

A client asks a prototype to clone itself.

Consequences

It hides the concrete product classes from the client, thereby reducing the number of names clients know about. Moreover, these patterns let a client work with application-specific classes without modification.

Additional benefits：

① Adding and removing products at run-time.

② Specifying new objects by varying values.

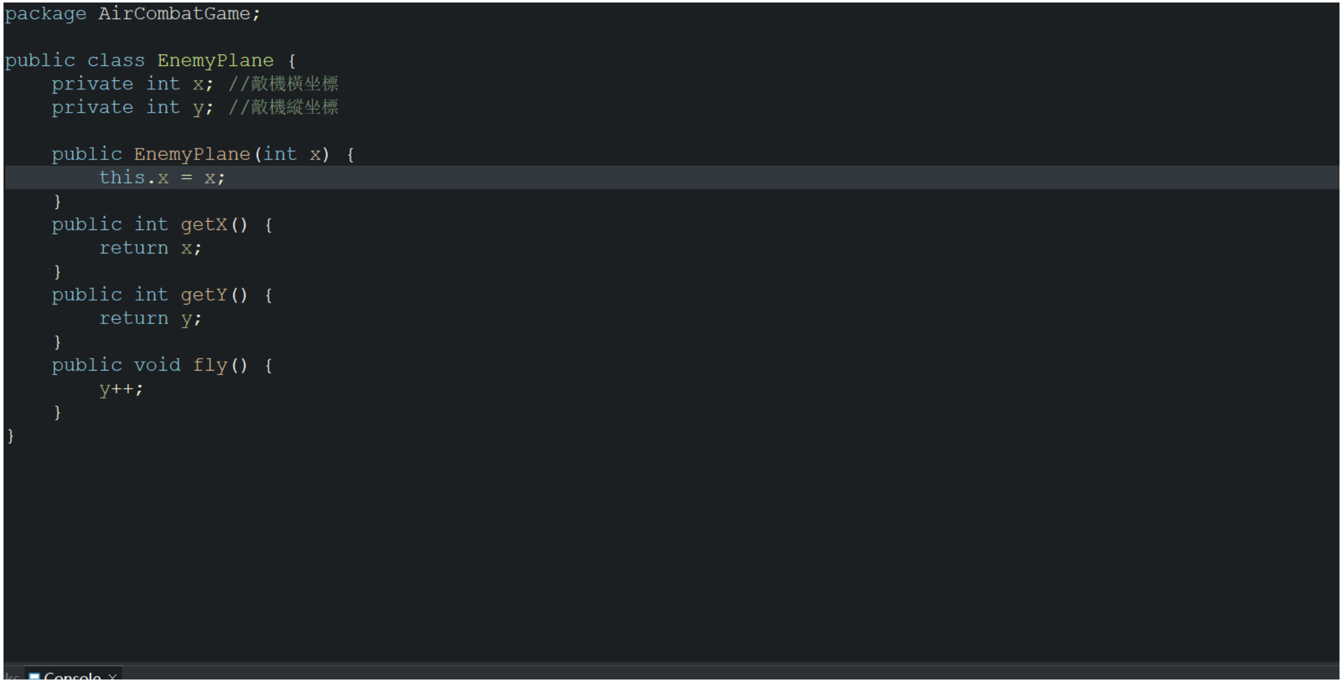
③ Specifying new objects by varying structure.

④ Reduced subclassing.

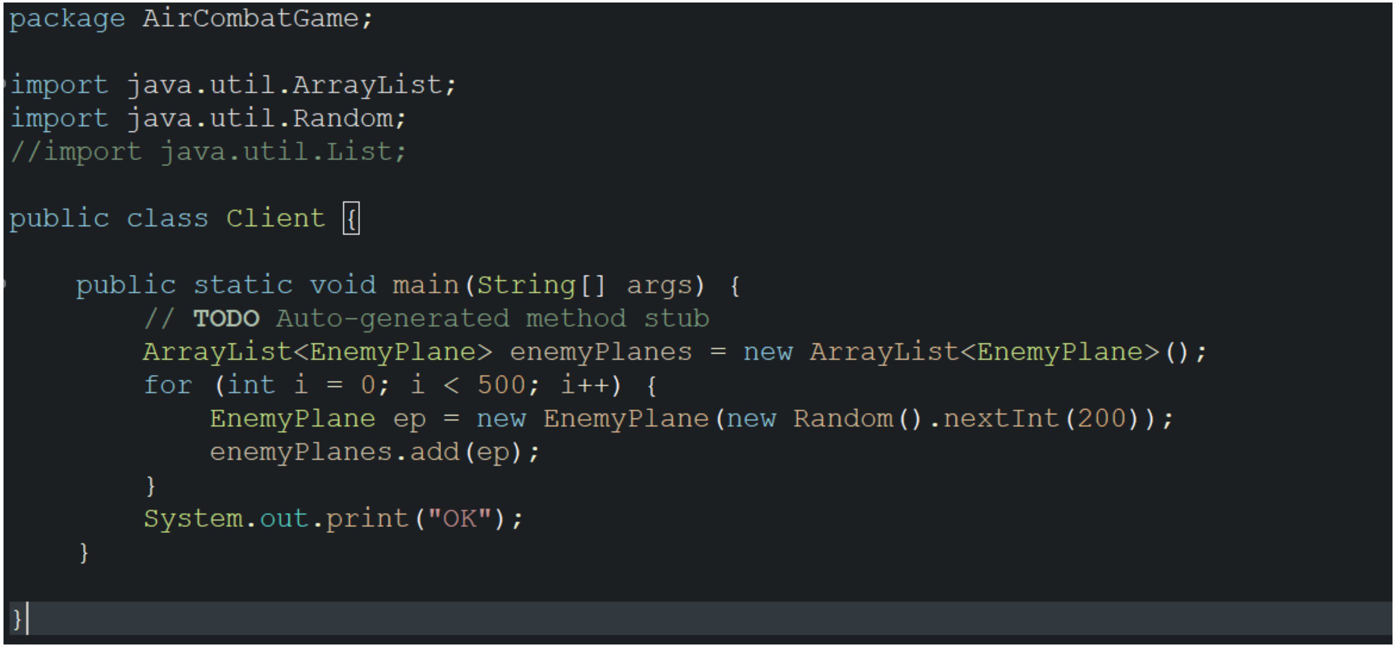
⑤ Configuring an application with classes dynamically.

Course: Soft Design

Air Combat Game



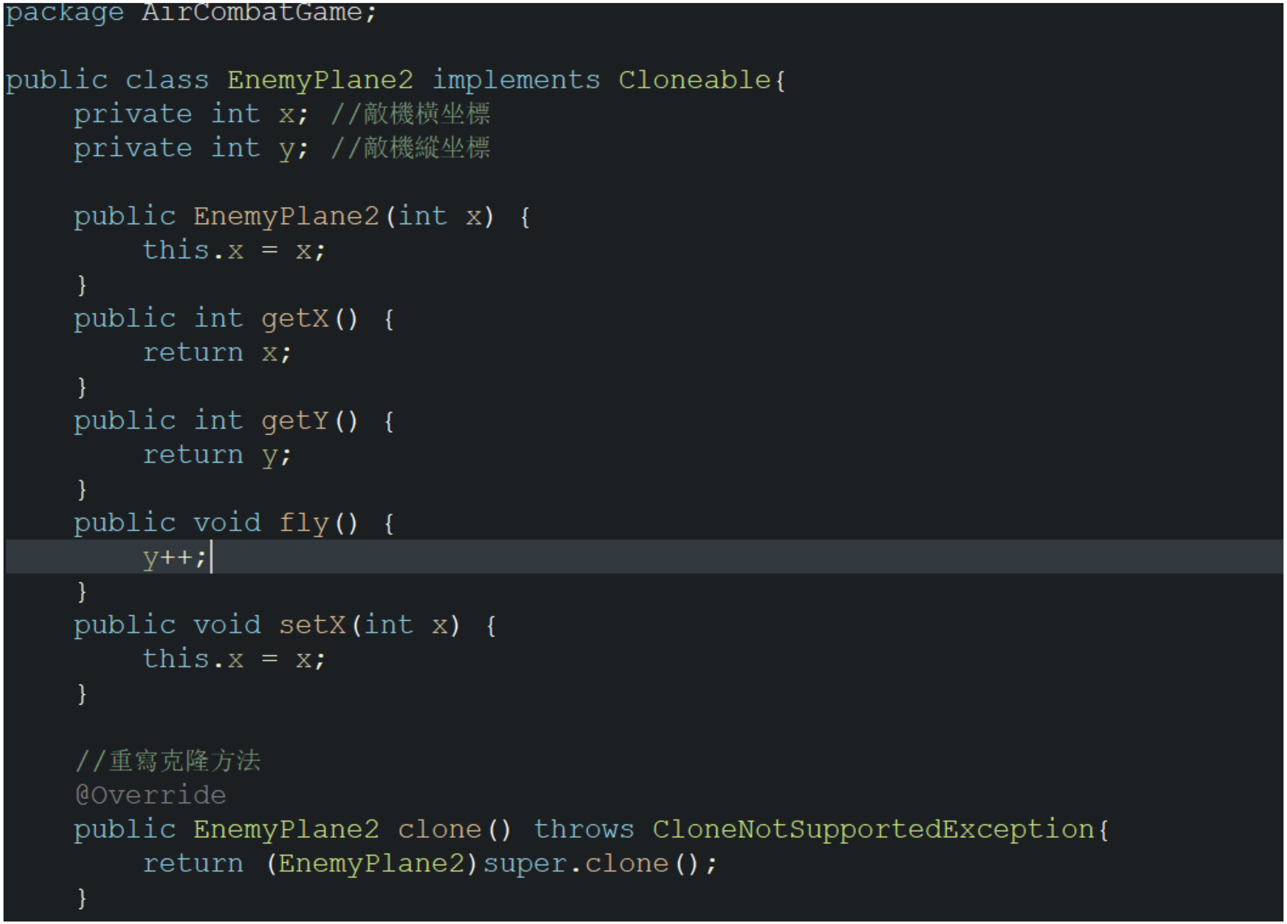
```
package AirCombatGame;

public class EnemyPlane {
    private int x; //敵機橫坐標
    private int y; //敵機縱坐標

    public EnemyPlane(int x) {
        this.x = x;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
    public void fly() {
        y++;
    }
}
```



```
package AirCombatGame;

import java.util.ArrayList;
import java.util.Random;
//import java.util.List;

public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ArrayList<EnemyPlane> enemyPlanes = new ArrayList<EnemyPlane>();
        for (int i = 0; i < 500; i++) {
            EnemyPlane ep = new EnemyPlane(new Random().nextInt(200));
            enemyPlanes.add(ep);
        }
        System.out.print("OK");
    }

}
```

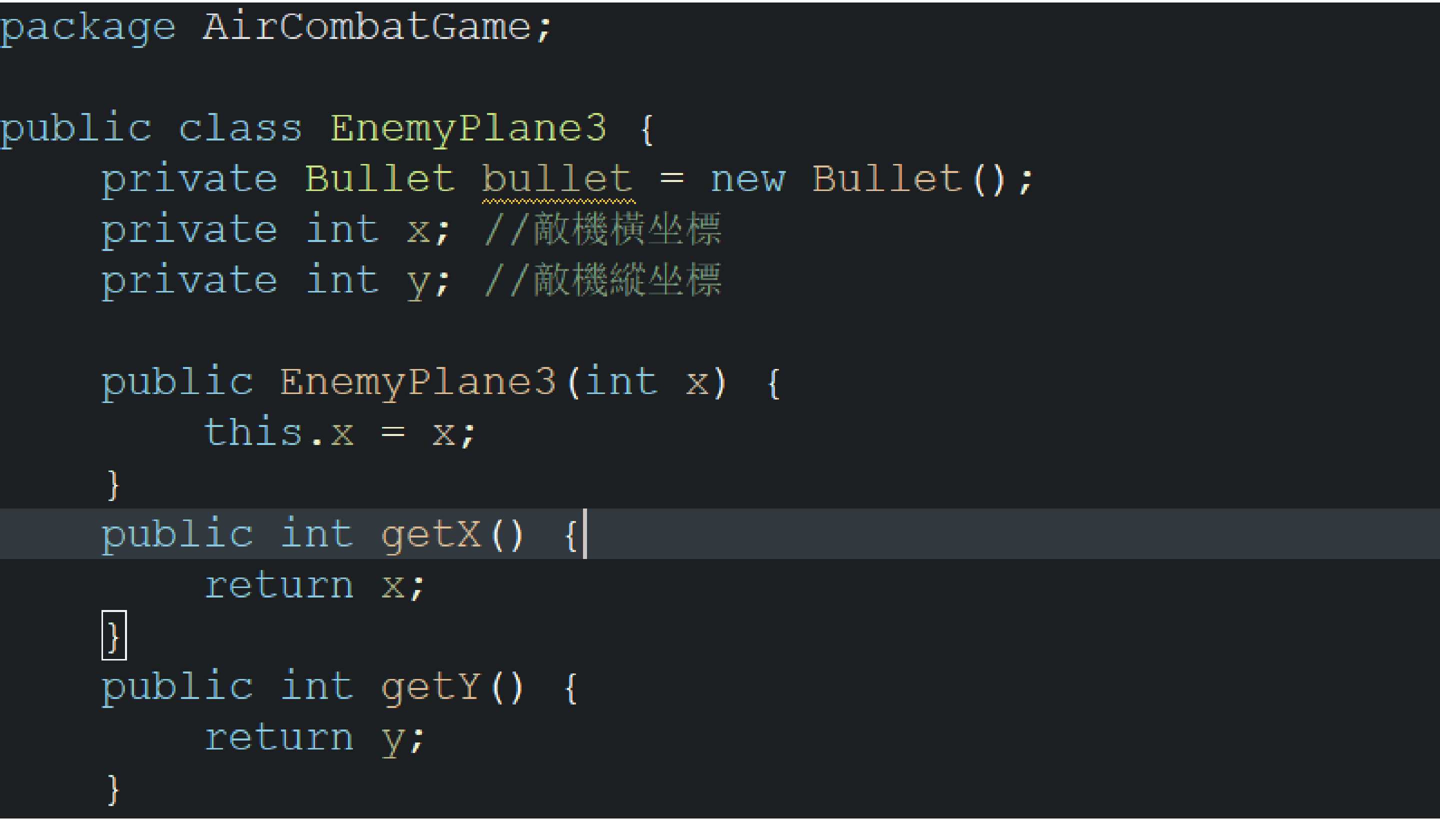
There seems to be no problem, but in fact the efficiency is very low.

Course: Soft Design



```
package AirCombatGame;

public class EnemyPlane2 implements Cloneable{
    private int x; //敵機橫坐標
    private int y; //敵機縱坐標

    public EnemyPlane2(int x) {
        this.x = x;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
    public void fly() {
        y++;
    }
    public void setX(int x) {
        this.x = x;
    }

    //重寫克隆方法
    @Override
    public EnemyPlane2 clone() throws CloneNotSupportedException{
        return (EnemyPlane2)super.clone();
    }
```

Course: Soft Design



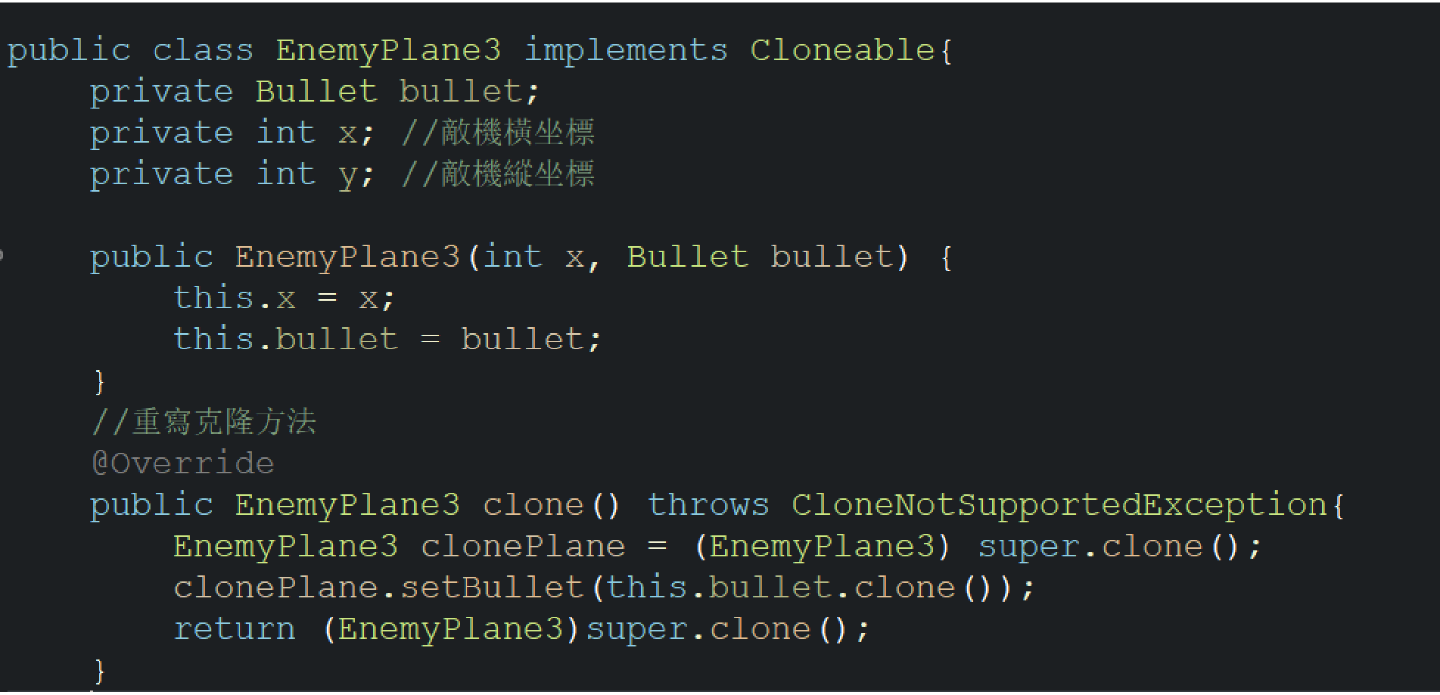
```
package AirCombatGame;

public class EnemyPlane3 {
    private Bullet bullet = new Bullet();
    private int x; //敵機横坐標
    private int y; //敵機縱坐標

    public EnemyPlane3(int x) {
        this.x = x;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
```

*

Shallow copy and Deep copy

Can we copy bullet objects together by copying? The answer is No. variables in Java are divided into original types and reference types. The original type such as int type will be copied to the new object, and the reference type will also be copied, but only its address reference will be copied.

Course: Soft Design



```
public class EnemyPlane3 implements Cloneable{
    private Bullet bullet;
    private int x; //敵機橫坐標
    private int y; //敵機縱坐標

    public EnemyPlane3(int x, Bullet bullet) {
        this.x = x;
        this.bullet = bullet;
    }
    //重寫克隆方法
    @Override
    public EnemyPlane3 clone() throws CloneNotSupportedException{
        EnemyPlane3 clonePlane = (EnemyPlane3) super.clone();
        clonePlane.setBullet(this.bullet.clone());
        return (EnemyPlane3)super.clone();
    }
```

Factory Method

Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

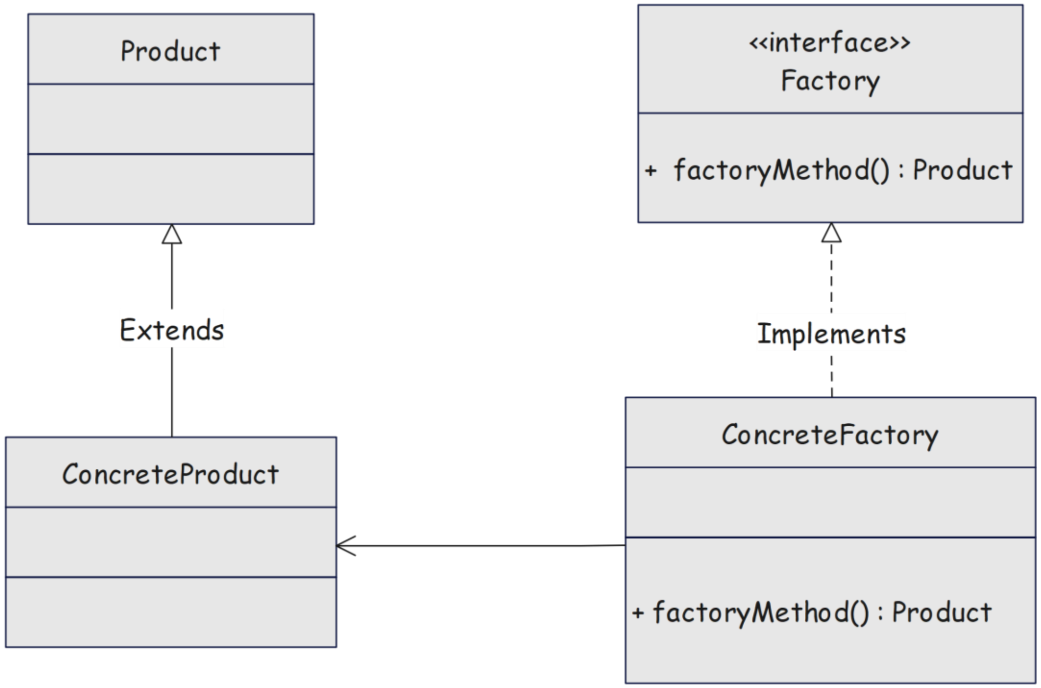
Also Known As

Virtual Constructor

Motivation

The external class does not need to care about the details inside the factory, and the external class only needs to call.
The code of object instantiation is firmly hard coded in the client class, and the client is strongly coupled with the instantiation process.
In fact, this task should be entrusted to the corresponding factory. The factory can finally deliver the product to us for use, which realizes the complete decoupling of the manufacturing process.

*


Product
<<interface>>
Factory
+ factoryMethod() : Product
Extends
Implements
ConcreteProduct
ConcreteFactory
+ factoryMethod() : Product

Participants

Product (Document)

- defines the interface of objects the factory method creates.

ConcreteProduct (MyDocument)

- implements the Product interface.

Creator (Application)

- declares the factory method, which returns an object of type Product.
- Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
- may call the factory method to create a Product object.

ConcreteCreator (MyApplication)

- overrides the factory method to return an instance of a ConcreteProduct.

Collaborations

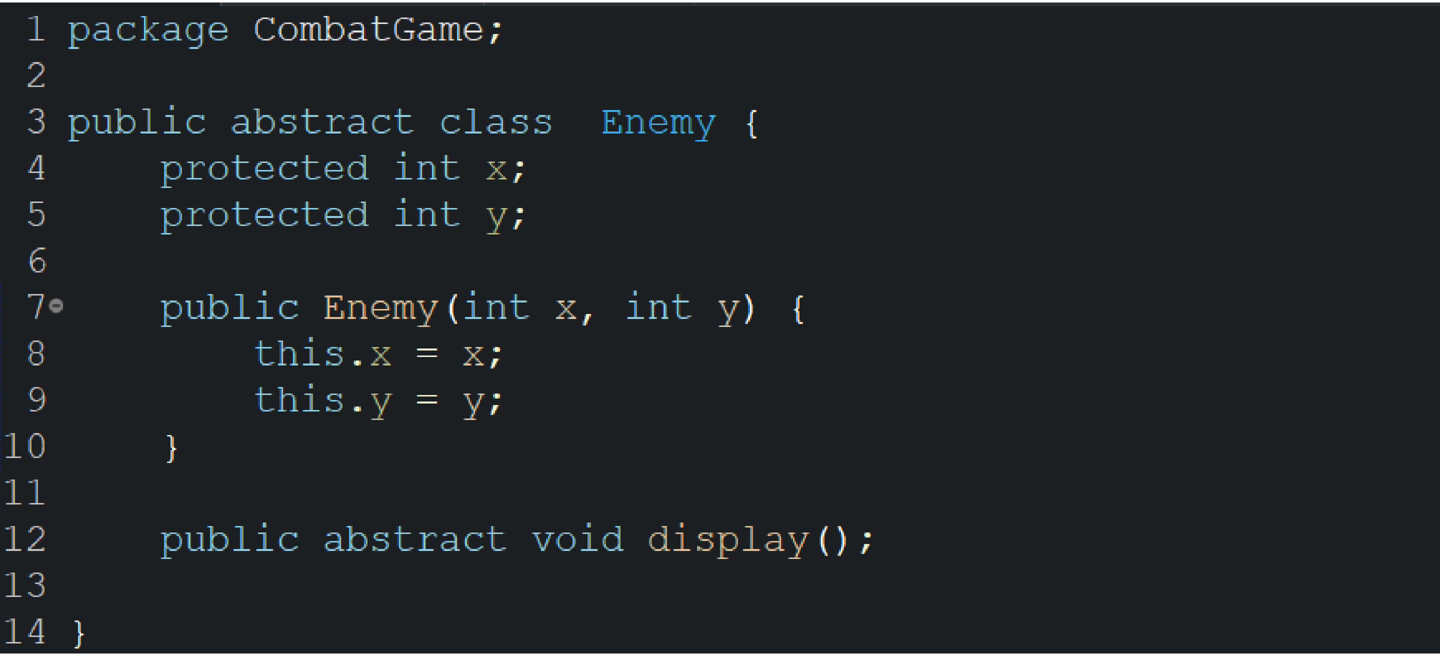
Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct.

Consequences

Provides hooks for subclasses.

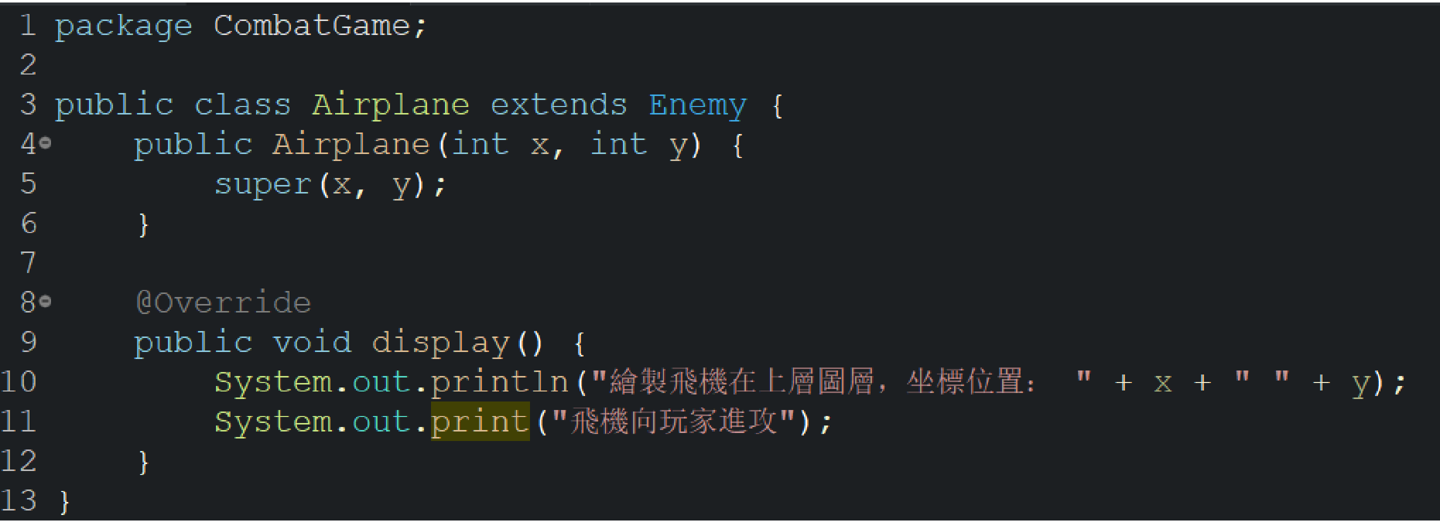
Connects parallel class hierarchies.

Combat Game



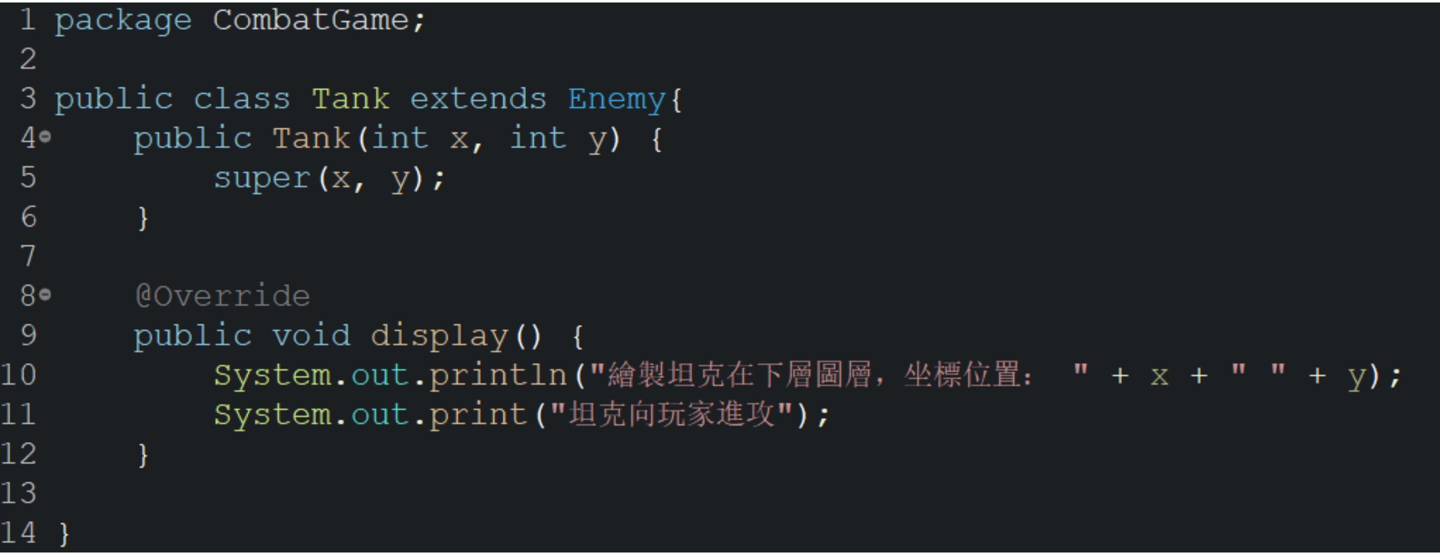
```java
package CombatGame;

public abstract class  Enemy {
    protected int x;
    protected int y;

    public Enemy(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public abstract void display();

}
```

Course: Soft Design



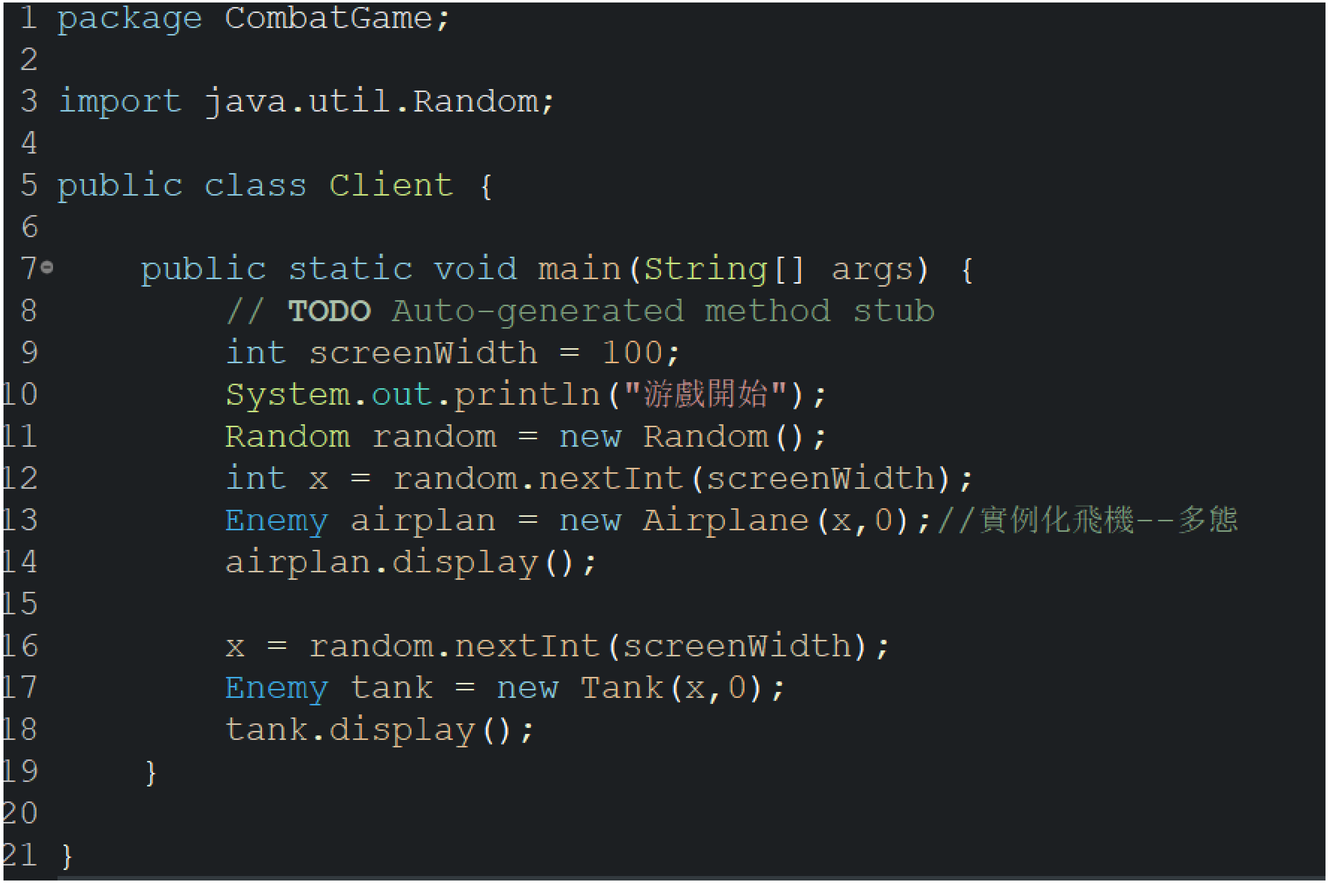
```java
package CombatGame;

public class Airplane extends Enemy {
    public Airplane(int x, int y) {
        super(x, y);
    }

    @Override
    public void display() {
        System.out.println("繪製飛機在上層圖層，坐標位置： " + x + " " + y);
        System.out.print("飛機向玩家進攻");
    }
}
```

Course: Soft Design



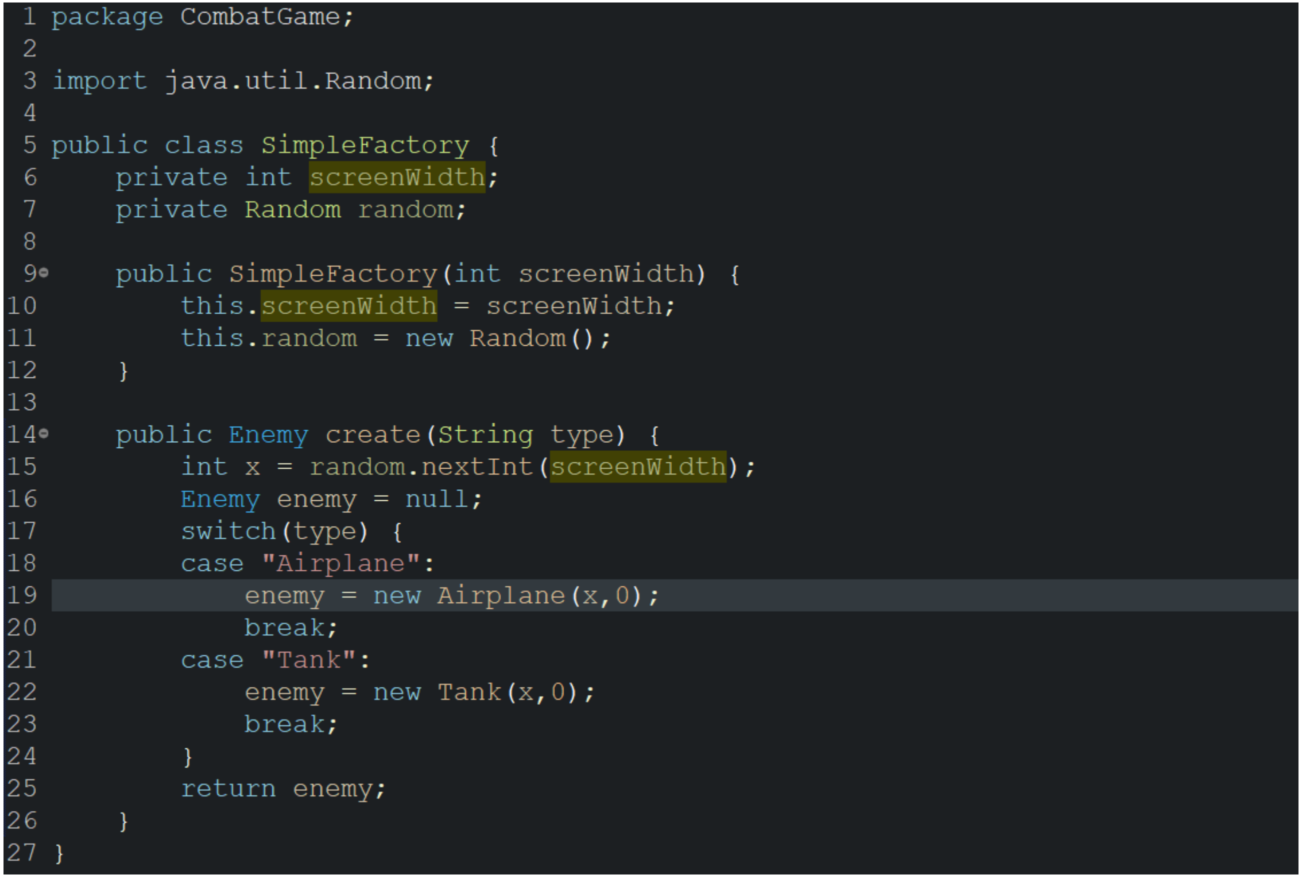
```java
package CombatGame;

public class Tank extends Enemy{
    public Tank(int x, int y) {
        super(x, y);
    }

    @Override
    public void display() {
        System.out.println("繪製坦克在下層圖層，坐標位置： " + x + " " + y);
        System.out.print("坦克向玩家進攻");
    }

}
```

Course: Soft Design

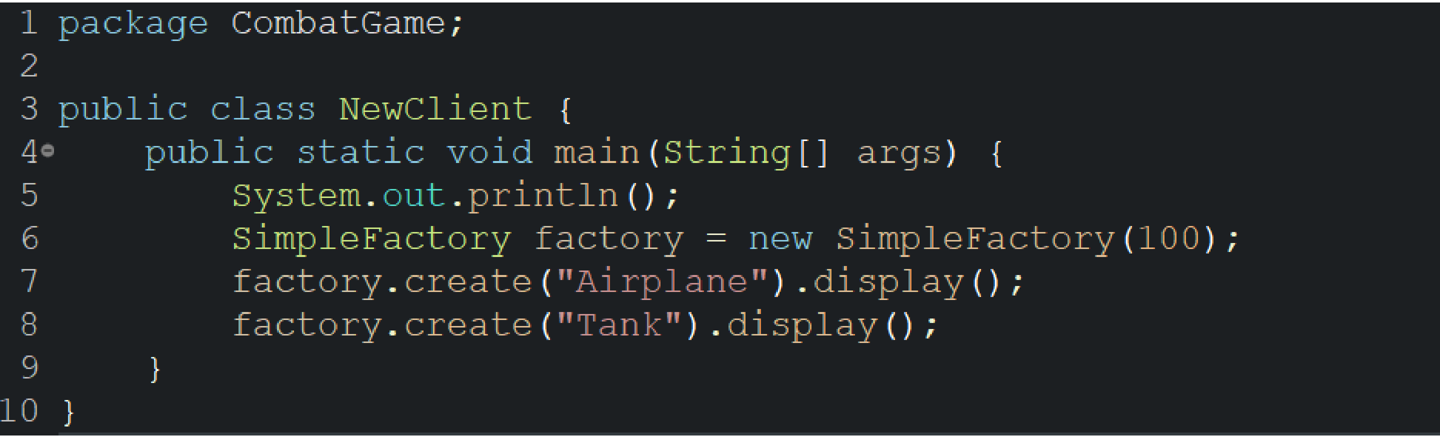


```java
package CombatGame;

import java.util.Random;

public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int screenWidth = 100;
        System.out.println("游戲開始");
        Random random = new Random();
        int x = random.nextInt(screenWidth);
        Enemy airplan = new Airplane(x,0);//實例化飛機--多態
        airplan.display();

        x = random.nextInt(screenWidth);
        Enemy tank = new Tank(x,0);
        tank.display();
    }

}
```

Course: Soft Design



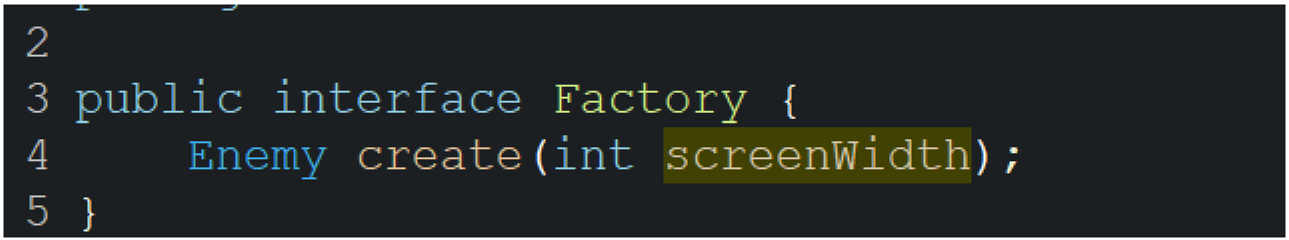
```java
package CombatGame;

import java.util.Random;

public class SimpleFactory {
    private int screenWidth;
    private Random random;

    public SimpleFactory(int screenWidth) {
        this.screenWidth = screenWidth;
        this.random = new Random();
    }

    public Enemy create(String type) {
        int x = random.nextInt(screenWidth);
        Enemy enemy = null;
        switch(type) {
        case "Airplane":
            enemy = new Airplane(x,0);
            break;
        case "Tank":
            enemy = new Tank(x,0);
            break;
        }
        return enemy;
    }
}
```



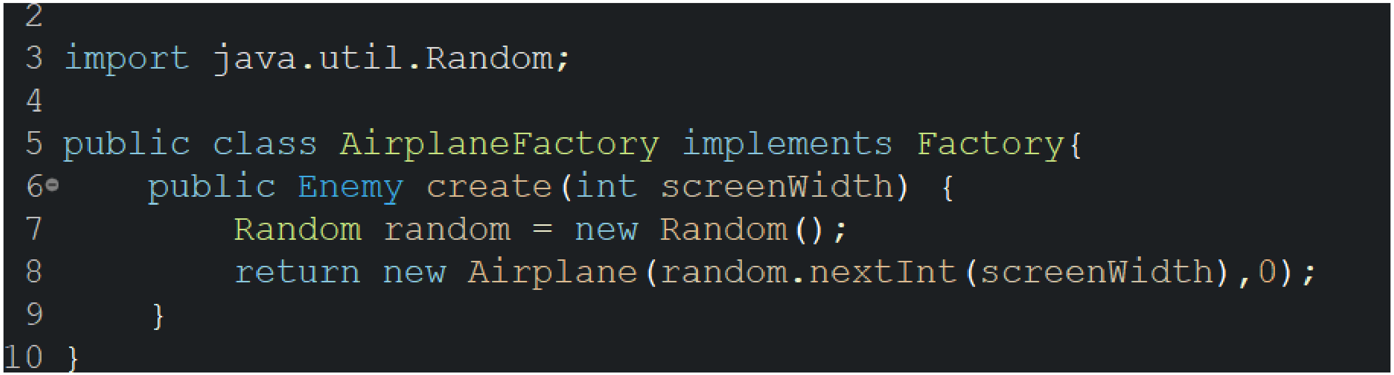
```
package CombatGame;

public class NewClient {
    public static void main(String[] args) {
        System.out.println();
        SimpleFactory factory = new SimpleFactory(100);
        factory.create("Airplane").display();
        factory.create("Tank").display();
    }
}
```

We have completed very good encapsulation, but this is not enough. Although the code for instantiating the product no longer appears directly in the client, we have just changed places, which is just another coupling method.

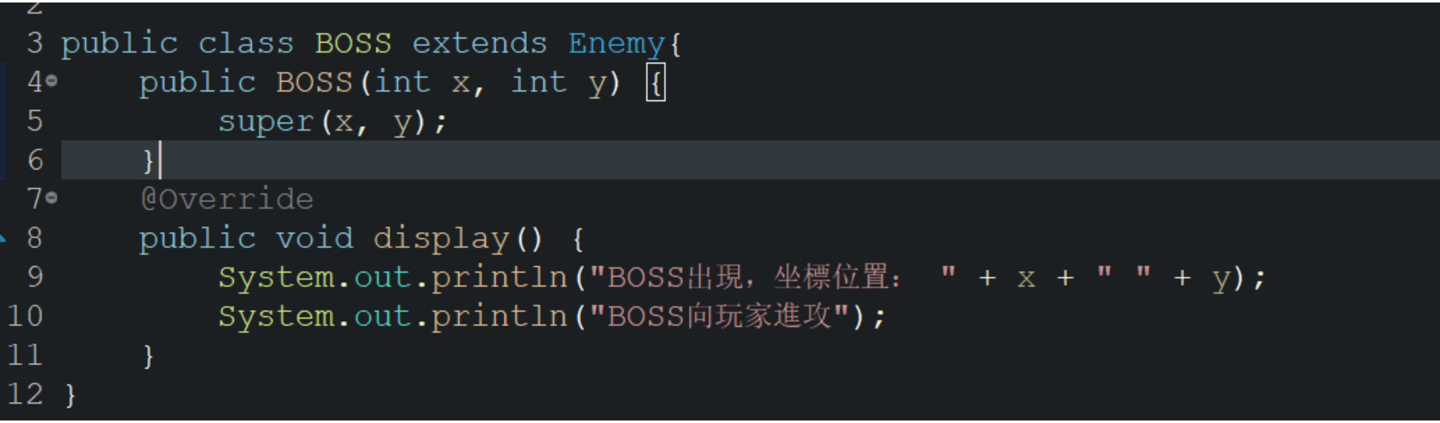
Course: Soft Design



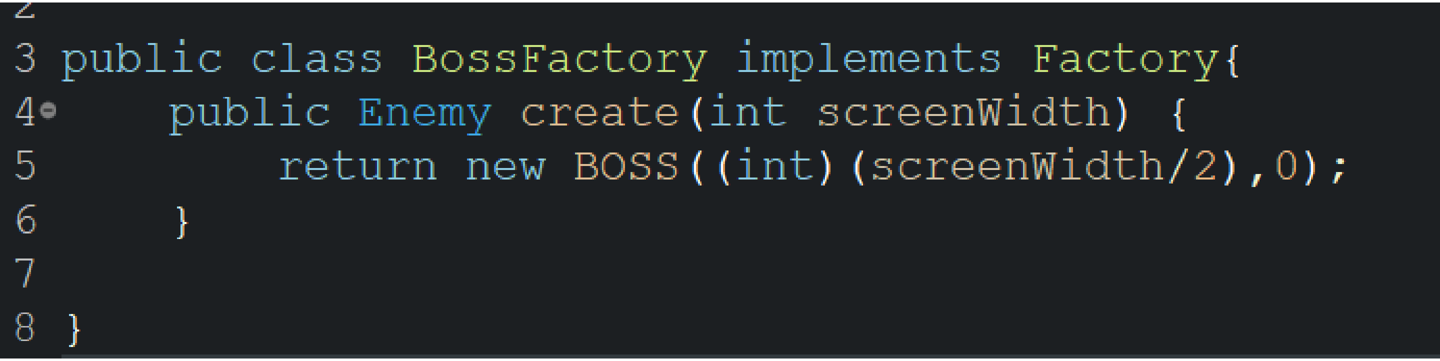
```java

public interface Factory {
        Enemy create(int screenWidth);
}
```



```java

import java.util.Random;

public class AirplaneFactory implements Factory{
    public Enemy create(int screenWidth) {
        Random random = new Random();
        return new Airplane(random.nextInt(screenWidth),0);
    }
}
```

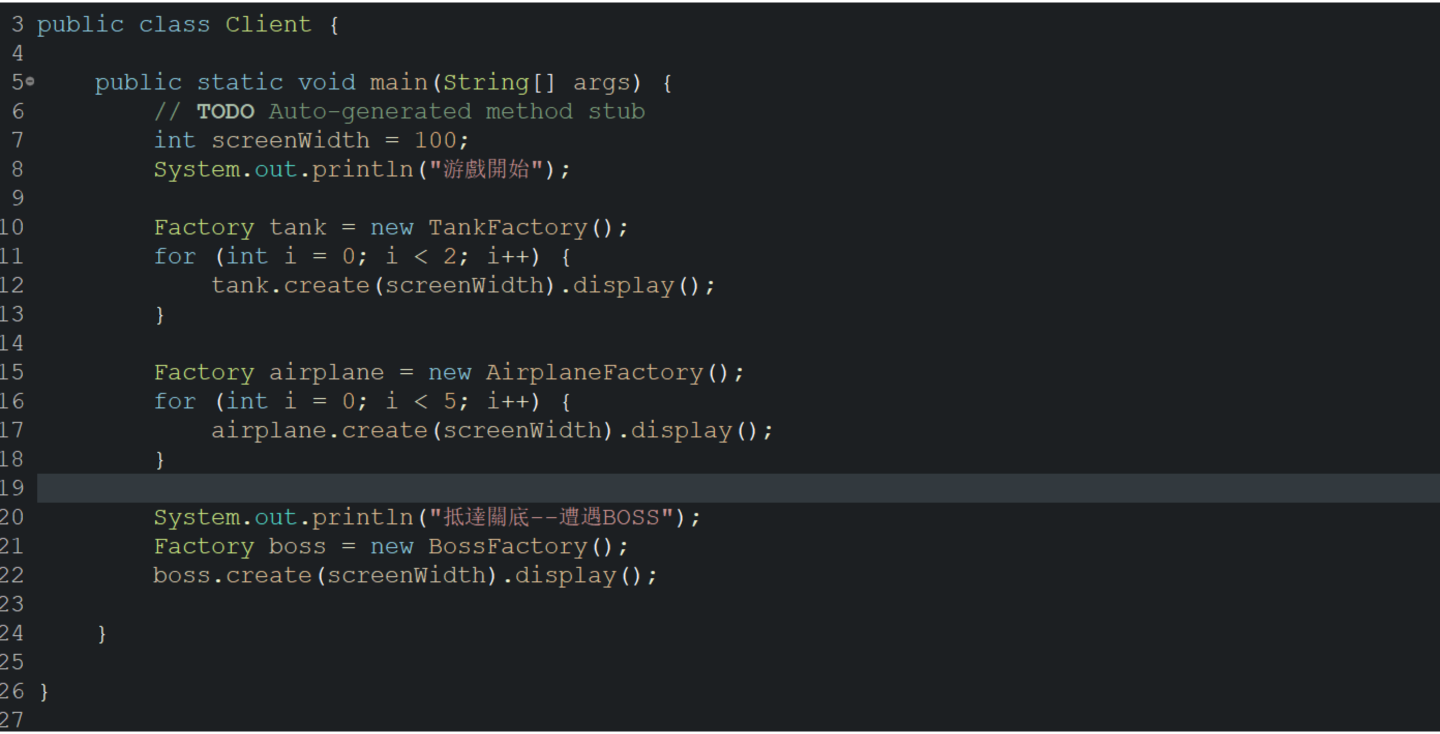


```java
public class BOSS extends Enemy{
    public BOSS(int x, int y) {
        super(x, y);
    }
    @Override
    public void display() {
        System.out.println("BOSS出現，坐標位置： " + x + " " + y);
        System.out.println("BOSS向玩家進攻");
    }
}
```



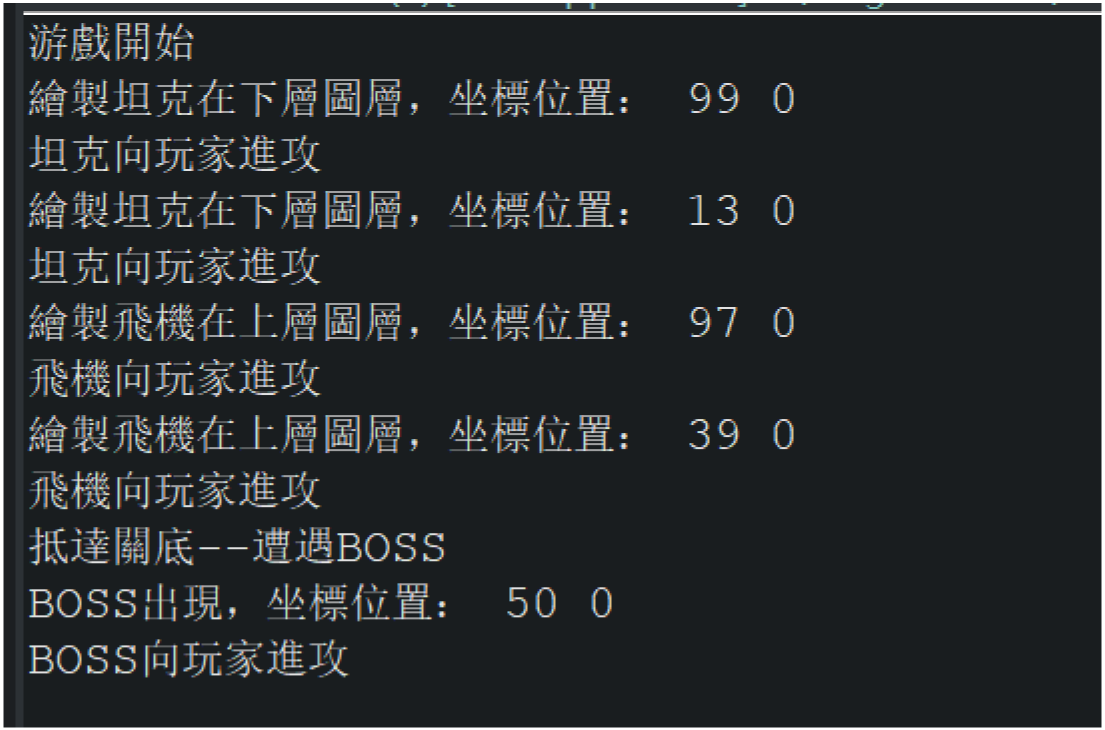
```java
public class BossFactory implements Factory{
    public Enemy create(int screenWidth) {
        return new BOSS((int)(screenWidth/2),0);
    }

}
```

Course: Soft Design



```
public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int screenWidth = 100;
        System.out.println("游戲開始");

        Factory tank = new TankFactory();
        for (int i = 0; i < 2; i++) {
            tank.create(screenWidth).display();
        }

        Factory airplane = new AirplaneFactory();
        for (int i = 0; i < 5; i++) {
            airplane.create(screenWidth).display();
        }

        System.out.println("抵達關底--遭遇BOSS");
        Factory boss = new BossFactory();
        boss.create(screenWidth).display();

    }

}

```

Course: Soft Design



```
遊戲開始
繪製坦克在下層圖層，坐標位置:   99  0
坦克向玩家進攻
繪製坦克在下層圖層，坐標位置:   13  0
坦克向玩家進攻
繪製飛機在上層圖層，坐標位置:   97  0
飛機向玩家進攻
繪製飛機在上層圖層，坐標位置:   39  0
飛機向玩家進攻
抵達關底--遭遇BOSS
BOSS出現，坐標位置:   50  0
BOSS向玩家進攻
```

GENERAL PRINCIPLES

Principle 19~37

Principle 19

Every complex problem has a solution

Wlad Turski said, 'To every complex problem, there is a simple solution ... and it is wrong!" Be highly suspicious of anybody who offers you some-thing like, "Just follow these 10 simple steps and your software quality problems will disappear.'

Principle 20*

Record your assumptions

It is impossible to be conscious of all the assumptions you make during requirements engineering, design, coding, and testing. Nonetheless, I recommend you maintain a diary of assumptions that you make consciously. Do this even if the assumption seems obvious or if the alternatives seem preposterous. Also record their implications, that is, where in the product does the assumption manifest itself? Ideally, you would like to isolate such implications by encapsulating each assumption (Principle 65).

Principle 21

Different Languages for different phases

For requirements engineering, select a set of optimal techniques and languages (Principles 47 and 48). For design, select a set of optimal techniques and languages (Principles 63 and 81). For coding, select an optimal language (Principles 102 and 103).Transitions between phases are difficult. Using the same language doesn't help. On the other hand, if a language is optimal for certain aspects of two phases, by all means use it.

Principle 22*

Technique before tools

An undisciplined carpenter with a power tool becomes a dangerous undisciplined carpenter. An undisciplined software engineer with a tool becomes a dangerous undisciplined software engineer. Before you use a tool, you should have discipline (that is, understand and be able to follow an appropriate software technique).

Principle 23

Use tools, but be realistic

Software tools (such as CASE)make their users more efficient. By all means, use them. Use CASE but be realistic concerning its effect on productivity. Be aware that 70 percent of all CASE tools purchased are never used. I believe the primary reason for this is overoptimism and the resulting disappointment, rather than the ineffectiveness of the tools

Principle 24

Give software tools to good engineers

Users of software tools (such as CASE) become more productive just as writers become more productive using word processors (Principle 23). CASE tool cannot convert a poor software engineer (one that produces software that is unreliable, fails to satisfy user needs, and so on) into a good one. Thus, you want to give CASE tools only to the good engineers. The last thing you want to do is to provide CASE tools to the poor engineers: You want them to produce less, not more, poor-quality software.

Principle 25

CASE tools are expensive

CASE tools are essential for software development. They should be considered part of the cost of being in the business. When doing a payback analysis, take into consideration the high costs of buying the tools, but also take into account the higher costs of not buying the tools (lower productivity, higher probability of customer dissatisfaction, delayed product release, increased rework, poorer product quality, increased employee turnover).

Principle 26

"Know-When" is as important as "Know-How"

All too often in our industry, a software engineer learns a new technique and decides that it is the be-all and end-all of techniques. Meanwhile, another software engineer on the same team learns a different new technique and an emotional battle ensues. The fact is that neither engineer is right. Knowing how to use a technique well does not make it a good technique, nor does it make you a good engineer. Knowing how to use a wood lathe well does not make you a good carpenter. The good engineer knows dozens of diverse techniques well and knows when each is appropriate for a project or a segment of a project. The good carpenter knows how to use dozens of tools, knows lots of diverse techniques, and, most importantly, knows when to employ each.

Principle 27

Stop when you achieve your goal

Don't be guilty of goals displacement. If, for example, you understand your problem after doing only half the steps of a method, stop. On the other hand, you need to have a good view of the entire software process because a later step of a method that appears discardable by this principle may generate something critical for later use.

Principle 28

Know formal methods

Many people think that the only way to use formal methods is to specify a system completely using them. This is not true. In fact, one of the most effective methods is to write a natural language specification first. Then attempt to write parts using formal methods. Just trying to write things more formally will help you find problems in the natural language. Fix the natural language and you now have a better document. Discard the formalism if desired after it has helped you.

Principle 29

Align reputation with organization

In general, when anybody finds an error in a software engineer's product, that engineer should be thankful, not defensive. To err is human. To accept, divine! When an engineering error is found, the person causing it should broadcast it, not hide it. The broadcasting has two effects: (1) It helps other engineers avoid the same error, and (2) it sets the stage for future nondefensive error repair.

Principle 30

Follow the lemmings with care

Just because everybody is doing something does not make it right for you. It may be right, but you need to carefully assess its applicability to your environment. When you learn about a"new" technology, don't readily accept the inevitable hype associated with it (Principle 129).Read carefully. Be realistic with respect to payoffs and risks. Run experiments before making major commitments. But by no means can you afford to ignore"new" technologies(see related Principle 31).

Principle 31*

Don't ignore technology

Software engineering technology is evolving rapidly. You cannot afford to sit around for a few years without keeping abreast of new developments. Software engineering appears to grow by waves. Each wave brings with it a large collection of "fads" and buzzwords. Although each wave appears to last just five to seven years, the wave does not simply disappear. Instead each subsequent wave stands upon the best features of all previous waves. (Hopefully "best" means "most effective," but unfortunately it often means "most popular.")

Principle 32

Use documentation standards

If your project, organization, or customer demands that a documentation standard be followed, then, of course, follow it. However, never blame a standard for doing a bad job. All the standards I'm familiar with, whether government or commercial, provide organizational and content guidance.

Principle 33

Every document needs a glossary

The definitions of all terms should be written in a manner that minimizes the need to look up in the glossary any of the words used in the definitions. One technique is first to explain the term in common, everyday terminology, and then add a second definition that uses other glossary terms. Terms used within definitions that are themselves defined else-where should be *italicized*.

Principle 34

Every software document needs an index

This principle is self-evident to all readers of software documents. It is surprising that authors do not realize this (considering the fact that every author wears the hat of a reader on occasion). An index is a list of all terms and concepts used in the document, together with one or more page numbers where the term or concept is defined, used, or referenced. This is true for requirements, design, code, test, users, and maintenance documents. **The index is used when a reader wants to find information quickly, and it is essential during later maintenance or enhancement of the document.**

Principle 35

Use the same name for the same concept

Unlike writing fiction where maintaining the readers' interest is the number one goal, technical documentation must always use the same words to refer to the same concept and the same sentence structure for similar messages. To do otherwise would confuse the reader, causing the reader to spend time trying to determine if there was a technical message in the rewording itself. Apply this principle to all technical writing: requirements specifications, users' manuals, design documentation, in-line comments, and so on.

Principle 36

Research-Then-Transfer doesn't work

The most successful transfers of ideas from the research laboratory to the development facility have resulted from close ties between the two facilities—from the beginning. They have used the industrial environment as the laboratory in which the ideas germinate and are demonstrated to be effective, rather than trying to do technology transfer after idea formulation.

Principle 37*

Take responsibility

The fact is that the best methods can be utilized in any engineering discipline to produce awful designs. And the most antiquated methods can be utilized in any engineering discipline to produce elegant designs. There are no excuses.If you are the developer of a system, it is your responsibility to do it right. Take that responsibility. Do it right, or don't do it at all.

REQUIREMENTS ENGINEERING PRINCIPLES

Principle 37~40

Requirements engineering is the set of activities including (1) eliciting or learning about a problem that needs a solution, and (2)specifying the external (black box) behavior of a system that can solve that problem. The final product of requirements engineering is a requirements specification.

Principle 38

Poor requirements yield poor cost estimates

A customer will not tolerate a product with poor quality, regardless of the definition of quality. Quality must be quantified and mechanisms put into place to motivate and reward its achievement. There is no trade-off to be made. The first requirement must be quality.

Principle 39*

Determine the problem before writing requirements

When faced with what they believe is a problem, most engineers rush into offering solutions. If the engineer's perception of the problem is accurate, the solution may work. However, problems are often elusive. Before trying to solve a problem, be sure to explore all alternative options for who really has the problem and what the problem really is. When solving the problem, don't be blinded by the potential excitement of the first solution. Procedural changes are always less expensive than system construction.