LECTURE 03

Abstract Factory and **Builder**

Software Development Principle 37~56

Course: Soft Design

Course: Soft Design

Abstract Factory (抽象工厂)

Intent

Provide an interface for creating families of related or dependent objects without
specifying their concrete classes.

Also Known As

Kit

Motivation

Consider a user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager.

Applicability

Use the Abstract Factory pattern when

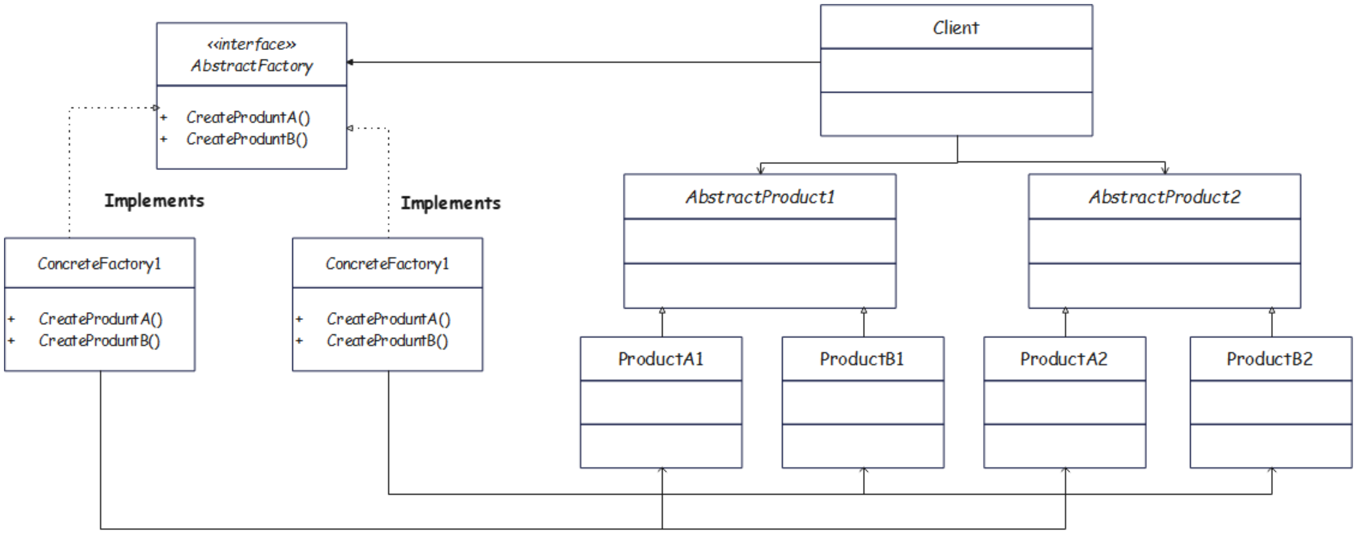
- a system should be independent of how its products are created, composed, and

represented.

- a system should be configured with one of multiple families of products.
- a family of related product objects is designed to be used together, and you need to enforce this constraint.
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Course: Soft Design

*



Participants

AbstractFactory

declares an interface for operations that create abstract product objects.

ConcreteFactory

implements the operations to create concrete product objects.

AbstractProduct

declares an interface for a type of product object.

ConcreteFactory

defines a product object to be created by the corresponding concrete factory.
implements the AbstractProduct interface.

Client

uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Collaborations

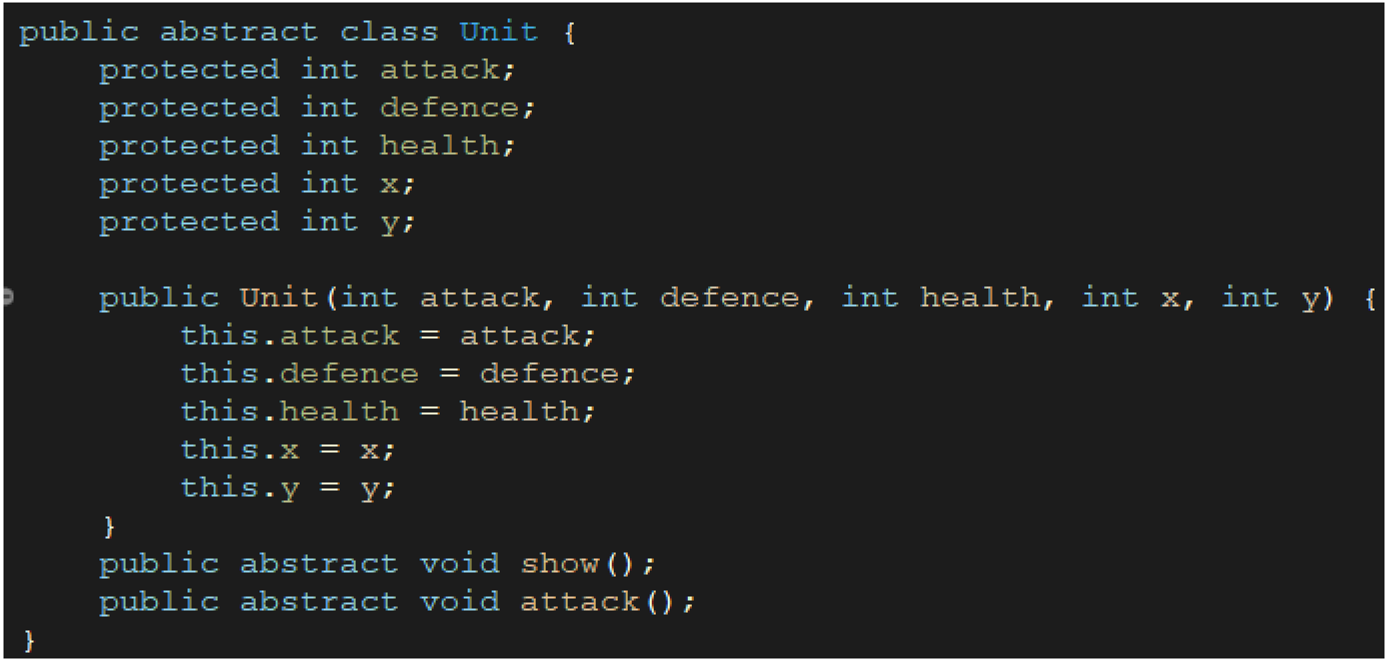
- Normally a single instance of a ConcreteFactory class is created at run-time.
- AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

Consequences

The Abstract Factory pattern has the following benefits and liabilities:

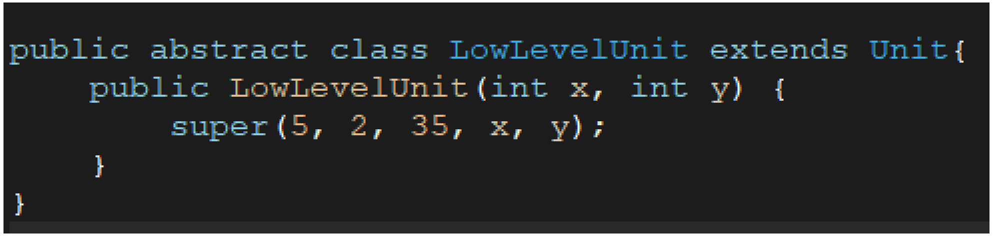
1. *It isolates concrete classes.*
2. *It makes exchanging product families easy.*
3. *It promotes consistency among products.*
4. *Supporting new kinds of products is difficult.*

Course: Soft Design

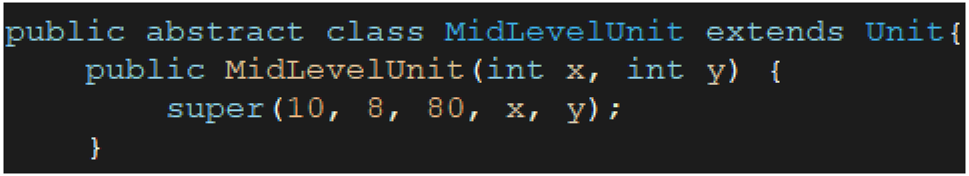


```java
public abstract class Unit {
    protected int attack;
    protected int defence;
    protected int health;
    protected int x;
    protected int y;

    public Unit(int attack, int defence, int health, int x, int y) {
        this.attack = attack;
        this.defence = defence;
        this.health = health;
        this.x = x;
        this.y = y;
    }
    public abstract void show();
    public abstract void attack();
}
```

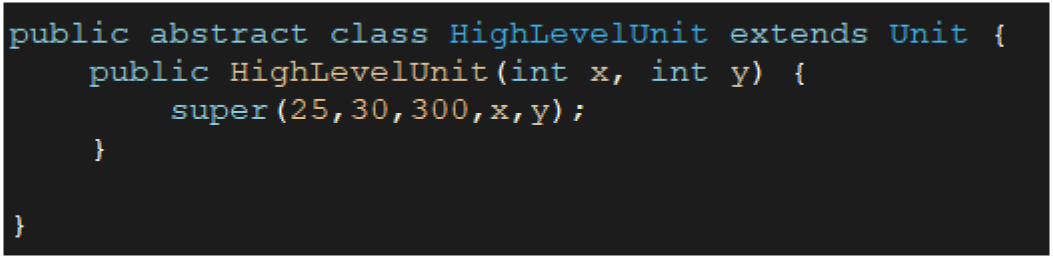
Course: Soft Design



```
public abstract class LowLevelUnit extends Unit{
    public LowLevelUnit(int x, int y) {
        super(5, 2, 35, x, y);
    }
}
```

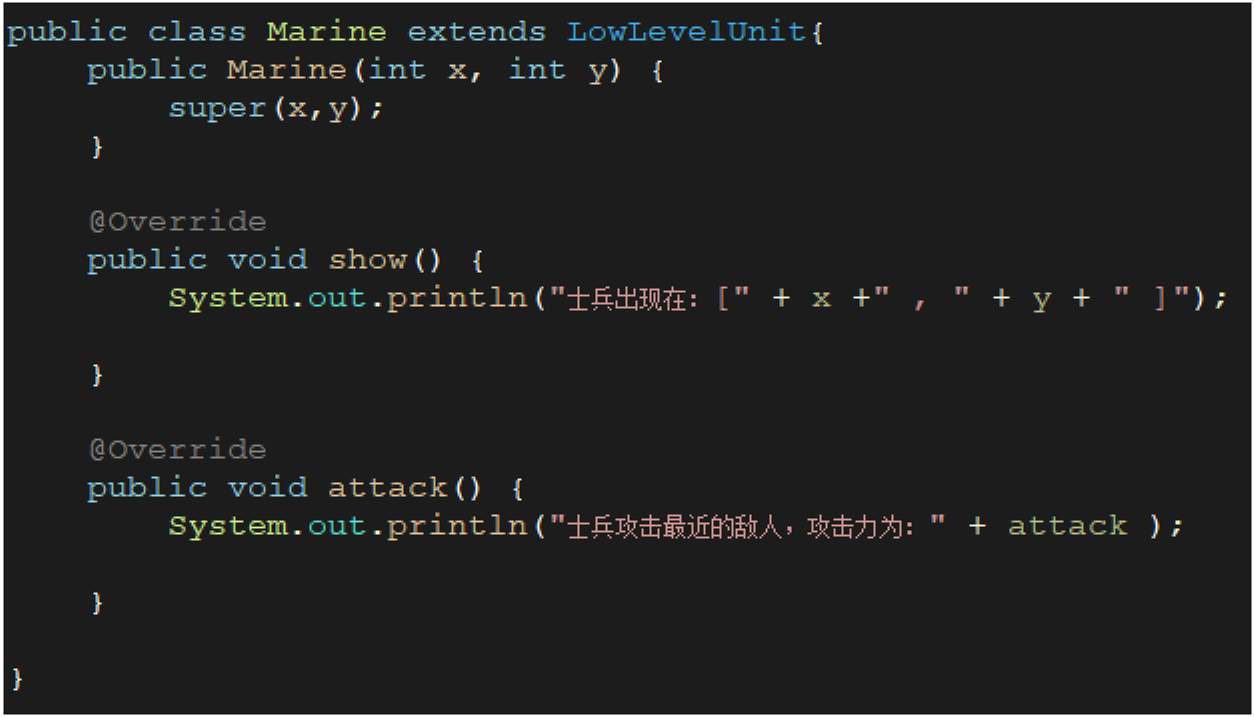


```
public abstract class MidLevelUnit extends Unit{
    public MidLevelUnit(int x, int y) {
        super(10, 8, 80, x, y);
    }
```



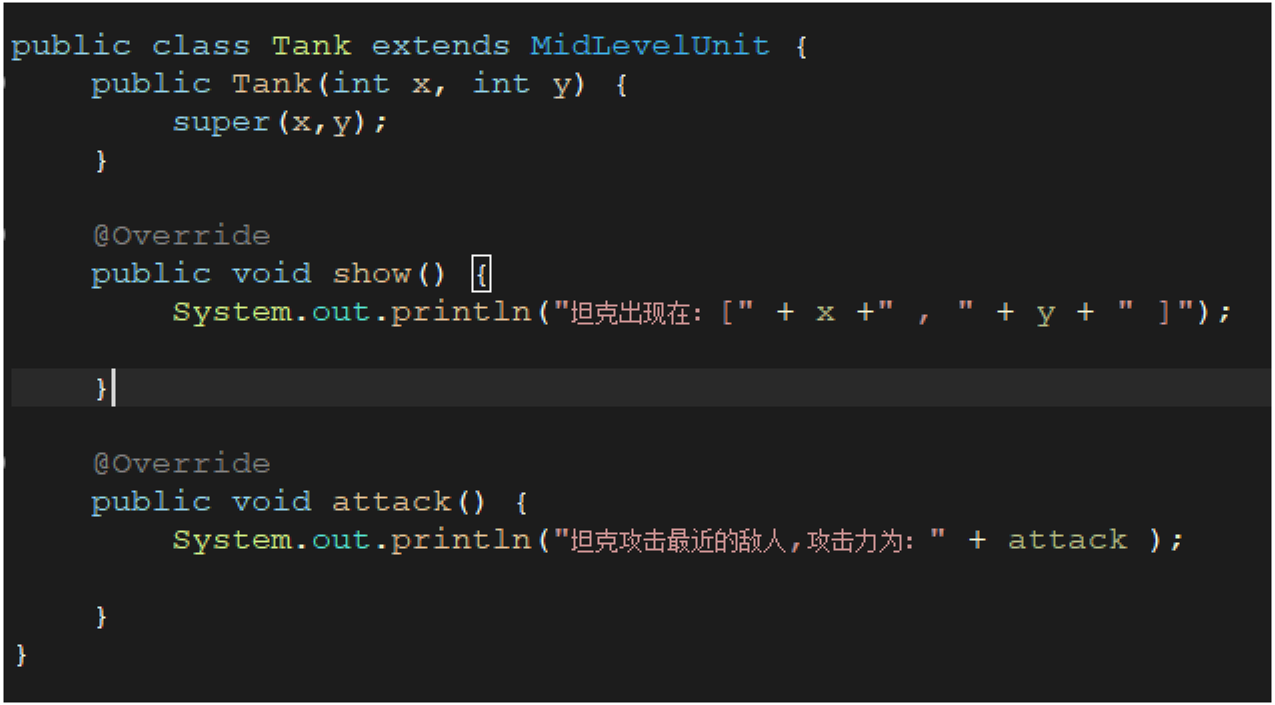
```
public abstract class HighLevelUnit extends Unit {
    public HighLevelUnit(int x, int y) {
        super(25,30,300,x,y);
    }

}
```

Course: Soft Design



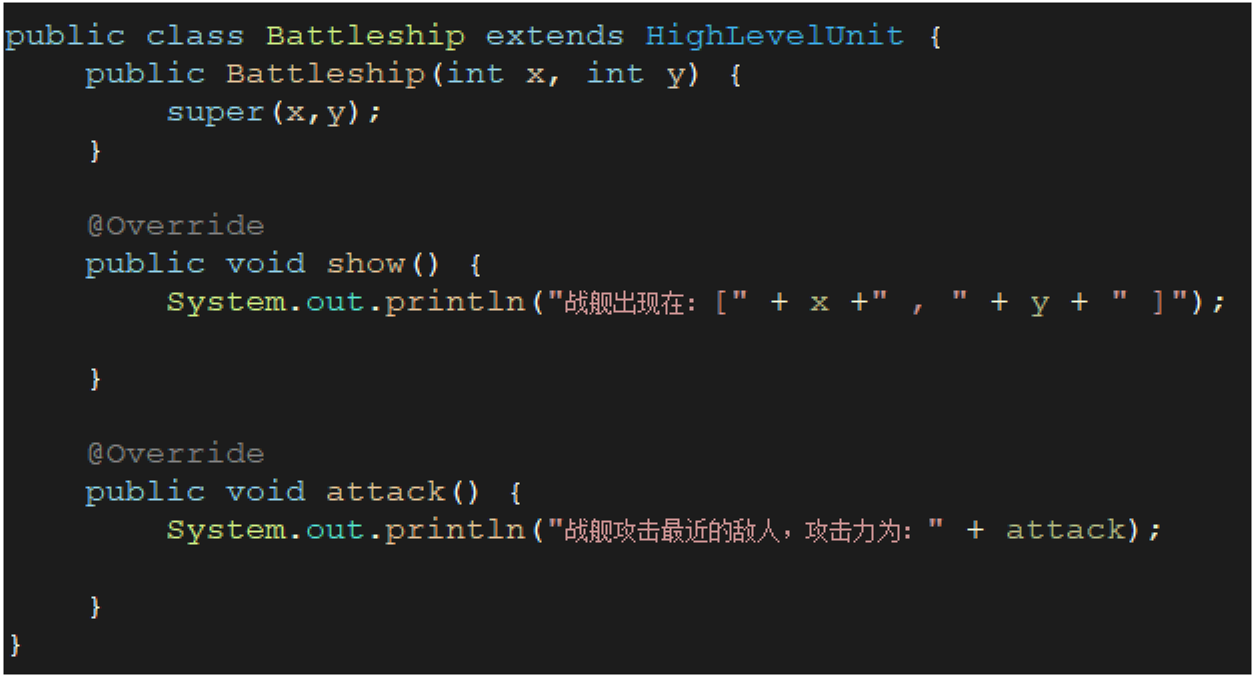
```java
public class Marine extends LowLevelUnit{
    public Marine(int x, int y) {
        super(x,y);
    }

    @Override
    public void show() {
        System.out.println("士兵出现在: [" + x +" , " + y + " ]");

    }

    @Override
    public void attack() {
        System.out.println("士兵攻击最近的敌人，攻击力为: " + attack );

    }

}
```

Course: Soft Design

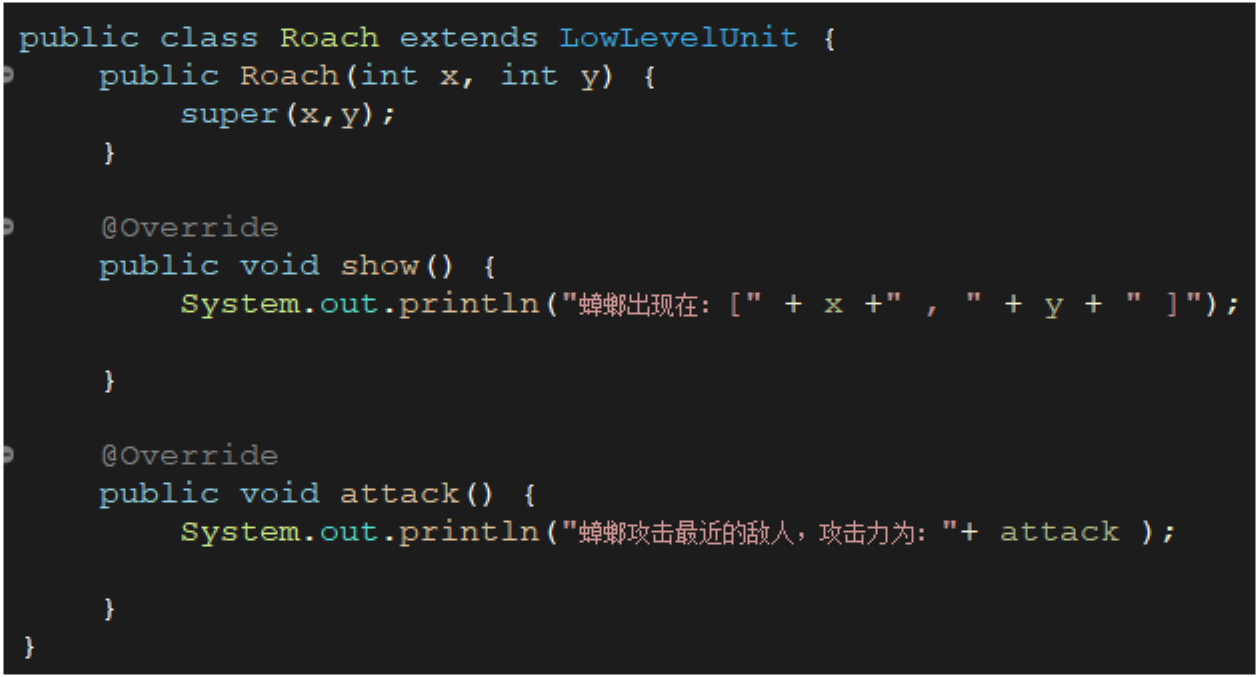


```
public class Tank extends MidLevelUnit {
    public Tank(int x, int y) {
        super(x,y);
    }

    @Override
    public void show() {
        System.out.println("坦克出现在: [" + x +" , " + y + " ]");

    }

    @Override
    public void attack() {
        System.out.println("坦克攻击最近的敌人,攻击力为: " + attack );

    }
}
```

Course: Soft Design

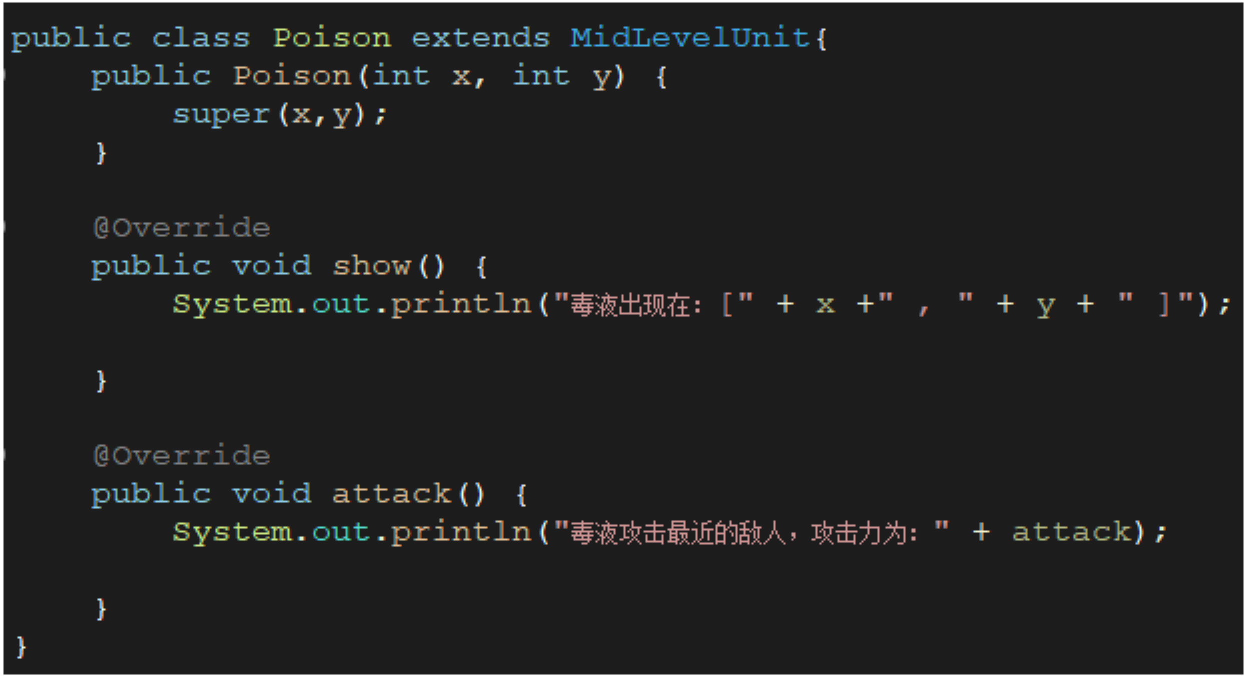


```
public class Battleship extends HighLevelUnit {
    public Battleship(int x, int y) {
        super(x,y);
    }

    @Override
    public void show() {
        System.out.println("战舰出现在: [" + x +" , " + y + " ]");

    }

    @Override
    public void attack() {
        System.out.println("战舰攻击最近的敌人，攻击力为: " + attack);

    }
}
```



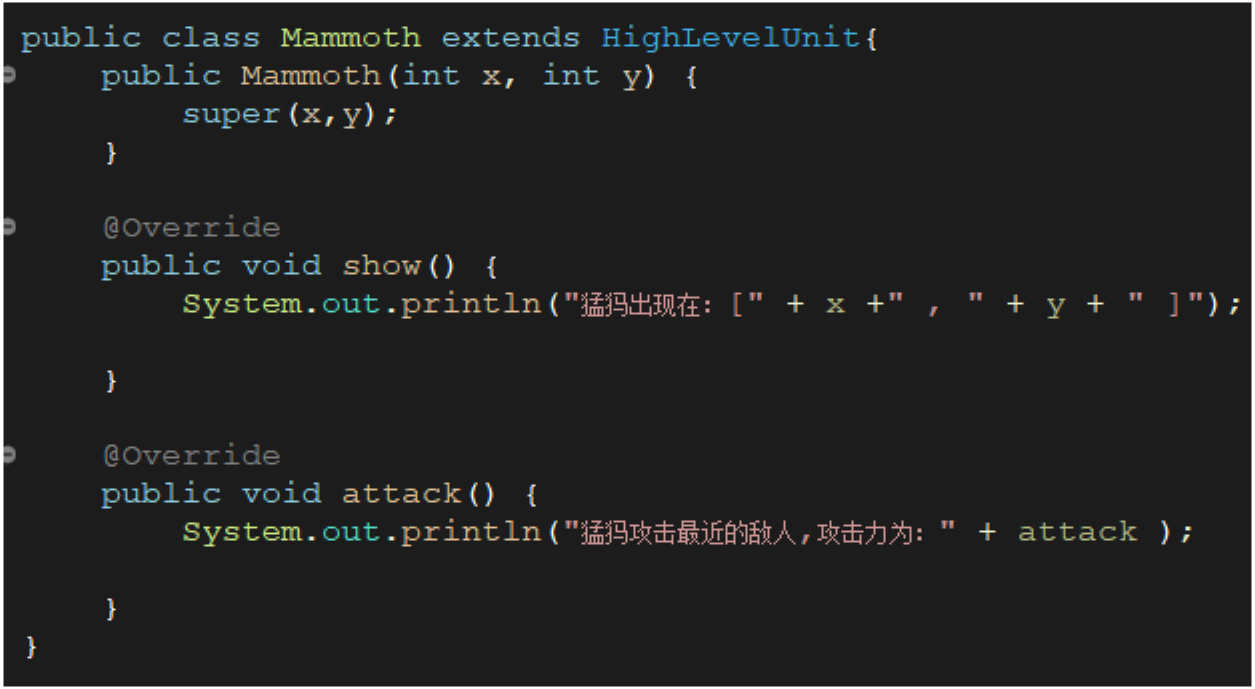
```java
public class Roach extends LowLevelUnit {
    public Roach(int x, int y) {
        super(x,y);
    }

    @Override
    public void show() {
        System.out.println("蟑螂出现在: [" + x +" , " + y + " ]");

    }

    @Override
    public void attack() {
        System.out.println("蟑螂攻击最近的敌人，攻击力为: "+ attack );

    }
}
```

Course: Soft Design



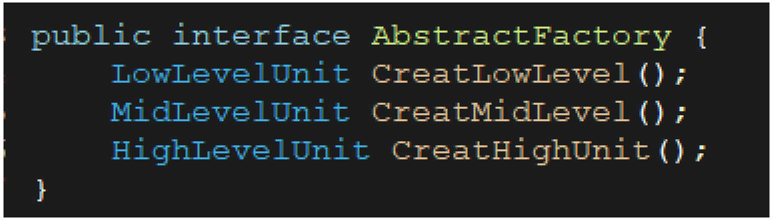
```
public class Poison extends MidLevelUnit{
    public Poison(int x, int y) {
        super(x,y);
    }

    @Override
    public void show() {
        System.out.println("毒液出现在: [" + x +" , " + y + " ]");

    }

    @Override
    public void attack() {
        System.out.println("毒液攻击最近的敌人, 攻击力为: " + attack);

    }
}
```

Course: Soft Design

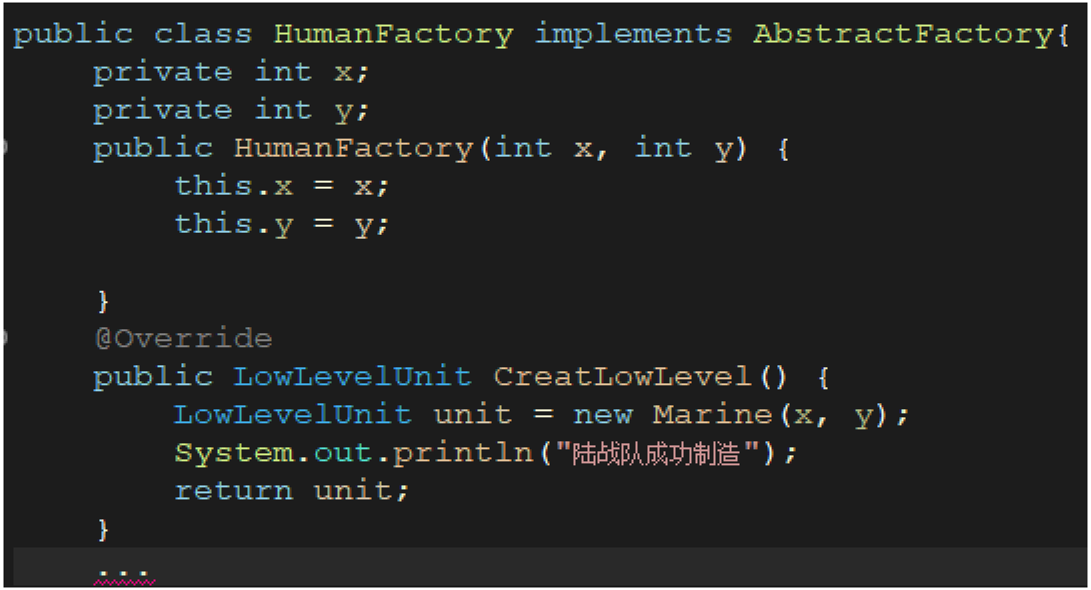


```
public class Mammoth extends HighLevelUnit{
    public Mammoth(int x, int y) {
        super(x,y);
    }

    @Override
    public void show() {
        System.out.println("猛犸出现在: [" + x +" , " + y + " ]");

    }

    @Override
    public void attack() {
        System.out.println("猛犸攻击最近的敌人,攻击力为: " + attack );

    }
}
```

Course: Soft Design

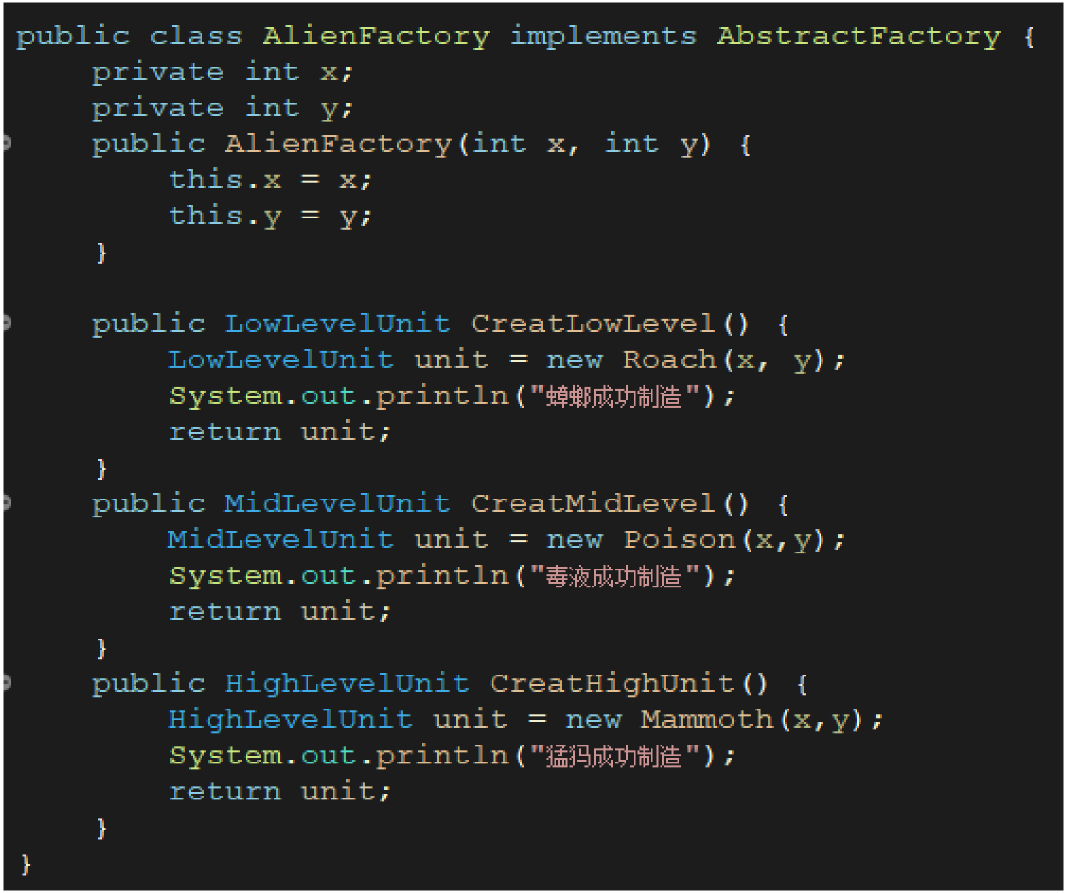


```java
public interface AbstractFactory {
    LowLevelUnit CreatLowLevel();
    MidLevelUnit CreatMidLevel();
    HighLevelUnit CreatHighUnit();
}
```



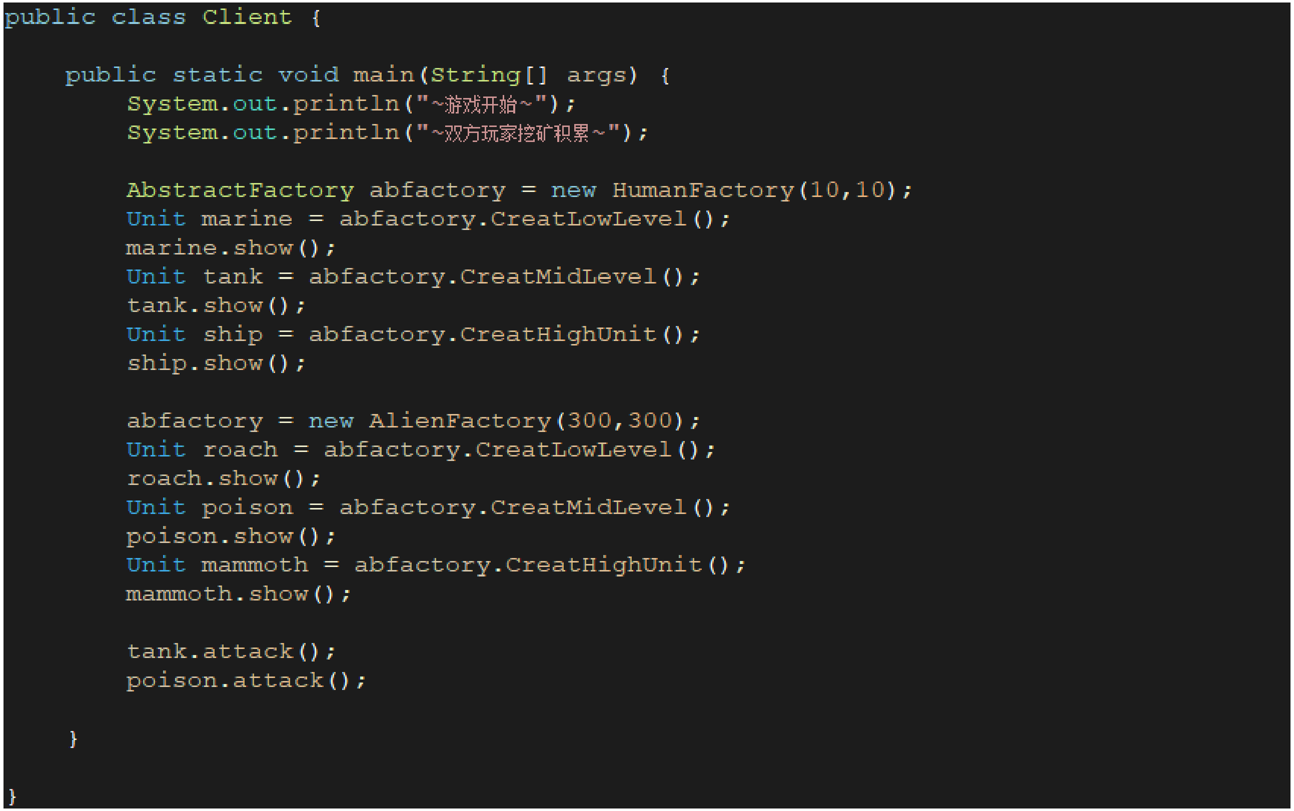
```java
public class HumanFactory implements AbstractFactory{
    private int x;
    private int y;
    public HumanFactory(int x, int y) {
        this.x = x;
        this.y = y;

    }
    @Override
    public LowLevelUnit CreatLowLevel() {
        LowLevelUnit unit = new Marine(x, y);
        System.out.println("陆战队成功制造");
        return unit;
    }
    ...
```

Course: Soft Design



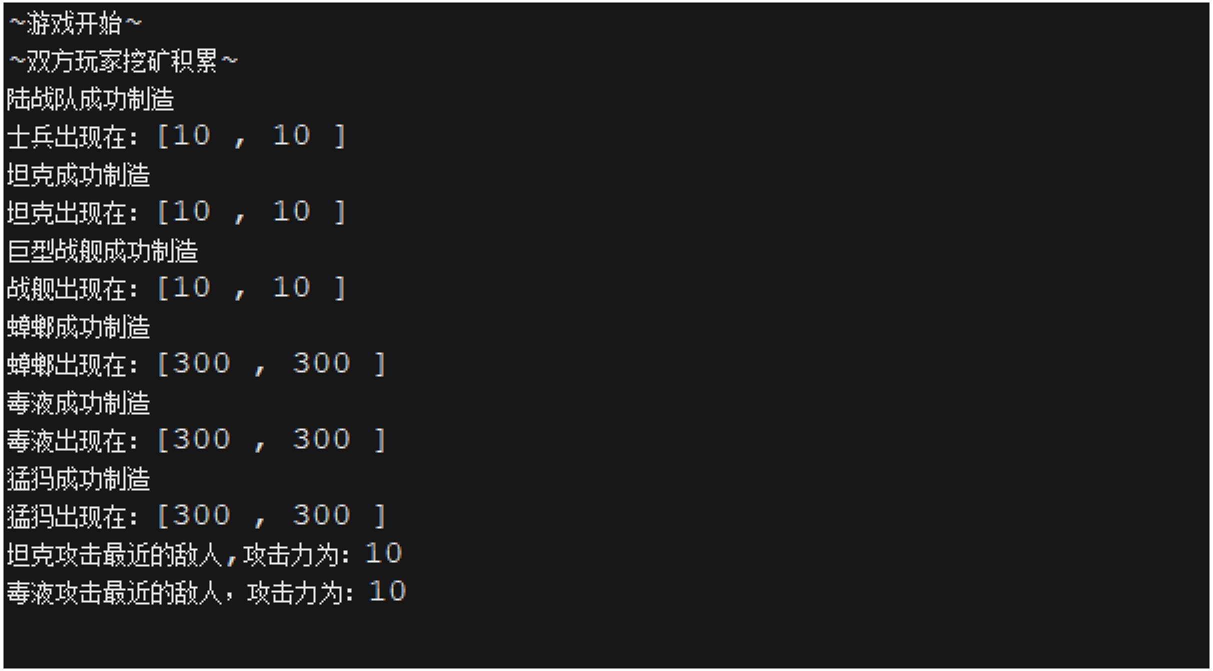
```
public class AlienFactory implements AbstractFactory {
    private int x;
    private int y;
    public AlienFactory(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public LowLevelUnit CreatLowLevel() {
        LowLevelUnit unit = new Roach(x, y);
        System.out.println("蟑螂成功制造");
        return unit;
    }
    public MidLevelUnit CreatMidLevel() {
        MidLevelUnit unit = new Poison(x,y);
        System.out.println("毒液成功制造");
        return unit;
    }
    public HighLevelUnit CreatHighUnit() {
        HighLevelUnit unit = new Mammoth(x,y);
        System.out.println("猛犸成功制造");
        return unit;
    }
}
```

Course: Soft Design



```
public class Client {

    public static void main(String[] args) {
        System.out.println("~游戏开始~");
        System.out.println("~双方玩家挖矿积累~");

        AbstractFactory abfactory = new HumanFactory(10,10);
        Unit marine = abfactory.CreatLowLevel();
        marine.show();
        Unit tank = abfactory.CreatMidLevel();
        tank.show();
        Unit ship = abfactory.CreatHighUnit();
        ship.show();

        abfactory = new AlienFactory(300,300);
        Unit roach = abfactory.CreatLowLevel();
        roach.show();
        Unit poison = abfactory.CreatMidLevel();
        poison.show();
        Unit mammoth = abfactory.CreatHighUnit();
        mammoth.show();

        tank.attack();
        poison.attack();

    }

}
```

Course: Soft Design



```
~游戏开始~
~双方玩家挖矿积累~
陆战队成功制造
士兵出现在: [10 , 10 ]
坦克成功制造
坦克出现在: [10 , 10 ]
巨型战舰成功制造
战舰出现在: [10 , 10 ]
蟑螂成功制造
蟑螂出现在: [300 , 300 ]
毒液成功制造
毒液出现在: [300 , 300 ]
猛犸成功制造
猛犸出现在: [300 , 300 ]
坦克攻击最近的敌人,攻击力为: 10
毒液攻击最近的敌人, 攻击力为: 10
```

Course: Soft Design

Builder (生成器)

Intent

Separate the construction of a complex object from its representation so that the
same construction process can create different representations.

Motivation

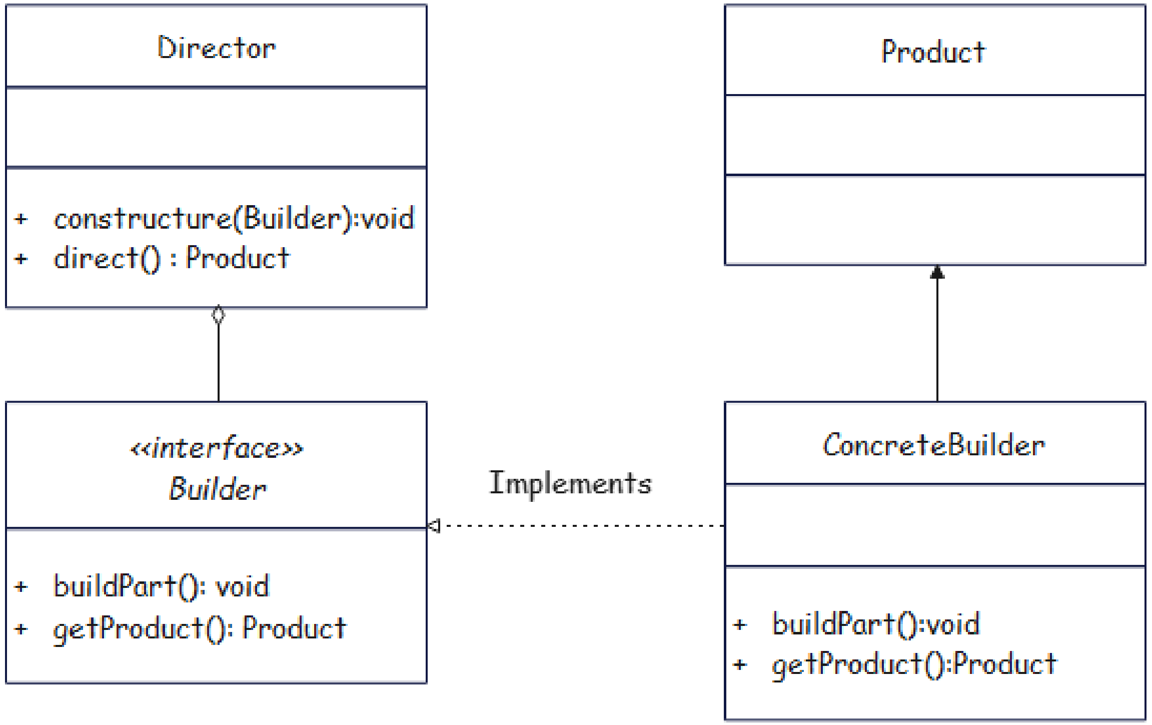
A reader for the RTF (Rich Text Format) document exchange format should be able to
convert RTF to many text formats.

Applicability

Use the Builder pattern when

- the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- the construction process must allow different representations for the object that's constructed.

*


Director
+ constructure(Builder):void
+ direct() : Product
Product
«interface»
Builder
+ buildPart(): void
+ getProduct(): Product
Implements
ConcreteBuilder
+ buildPart():void
+ getProduct():Product

Participants

Builder

- ➢ specifies an abstract interface for creating parts of a Product object.

ConcreteBuilder

- ➢ constructs and assembles parts of the product by implementing the Builder interface.
- ➢ defines and keeps track of the representation it creates. provides an interface for retrieving the product.

Director

- ➢ constructs an object using the Builder interface.

Product

- ➢ represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
- ➢ includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

Collaborations

- The client creates the Director object and configures it with the desired Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

Consequences

Here are key consequences of the Builder pattern:

1. *It lets you vary a product's internal representation.*
2. *It isolates code for construction and representation.*
3. *It gives you finer control over the construction process.*

Additional benefits：

① Adding and removing products at run-time.

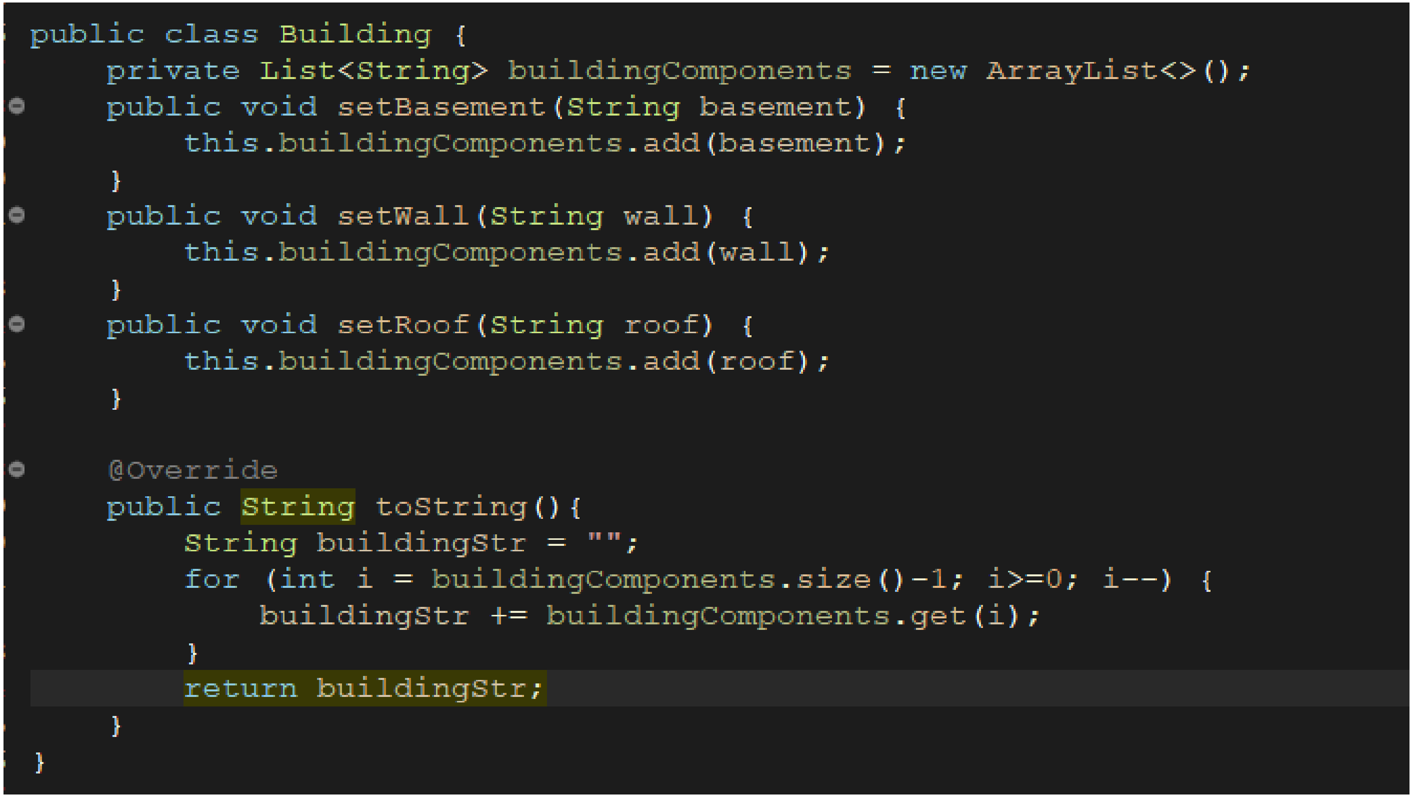
② Specifying new objects by varying values.

③ Specifying new objects by varying structure.

④ Reduced subclassing.

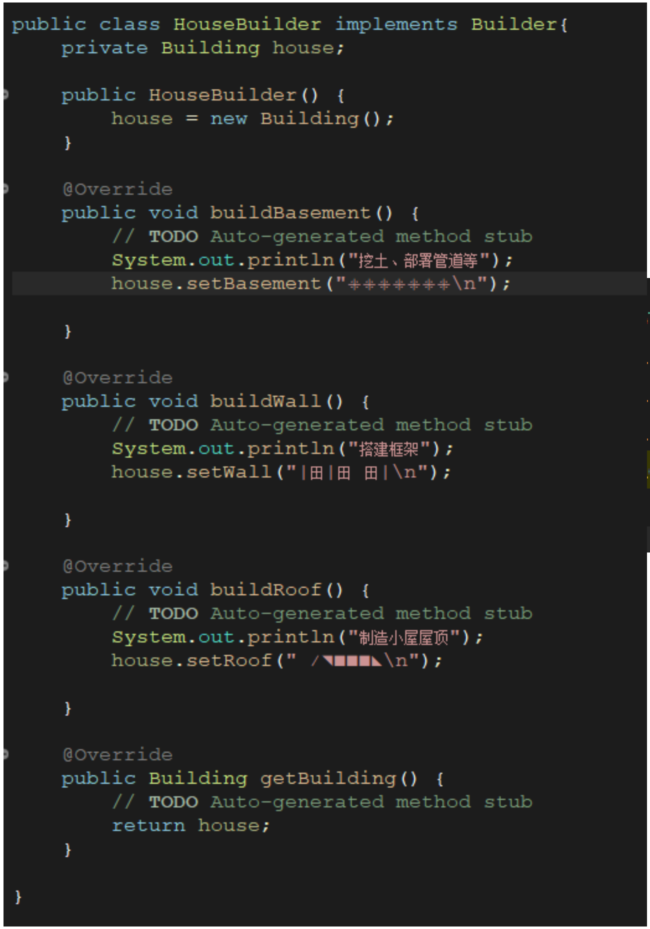
⑤ Configuring an application with classes dynamically.

Course: Soft Design

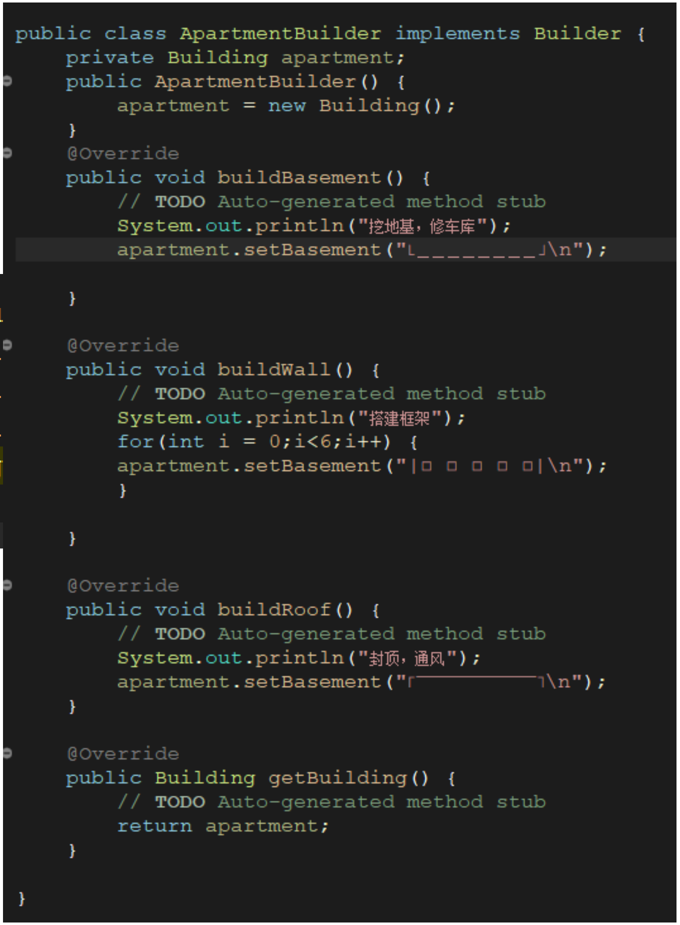


```java
public class Building {
    private List<String> buildingComponents = new ArrayList<>();
    public void setBasement(String basement) {
        this.buildingComponents.add(basement);
    }
    public void setWall(String wall) {
        this.buildingComponents.add(wall);
    }
    public void setRoof(String roof) {
        this.buildingComponents.add(roof);
    }

    @Override
    public String toString(){
        String buildingStr = "";
        for (int i = buildingComponents.size()-1; i>=0; i--) {
            buildingStr += buildingComponents.get(i);
        }
        return buildingStr;
    }
}
```

Course: Soft Design

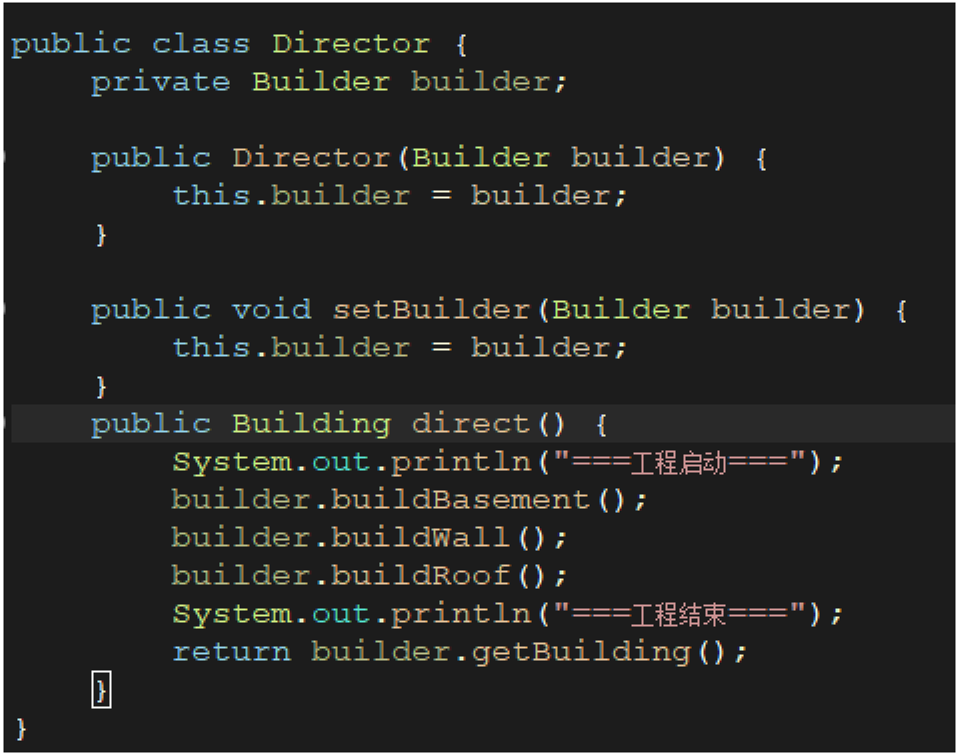


```java
public class HouseBuilder implements Builder{
    private Building house;

    public HouseBuilder() {
        house = new Building();
    }

    @Override
    public void buildBasement() {
        // TODO Auto-generated method stub
        System.out.println("挖土、部署管道等");
        house.setBasement("╪╪╪╪╪╪╪\n");

    }

    @Override
    public void buildWall() {
        // TODO Auto-generated method stub
        System.out.println("搭建框架");
        house.setWall("|田|田 田|\n");

    }

    @Override
    public void buildRoof() {
        // TODO Auto-generated method stub
        System.out.println("制造小屋屋顶");
        house.setRoof(" /◥■■■◣\n");

    }

    @Override
    public Building getBuilding() {
        // TODO Auto-generated method stub
        return house;
    }

}
```



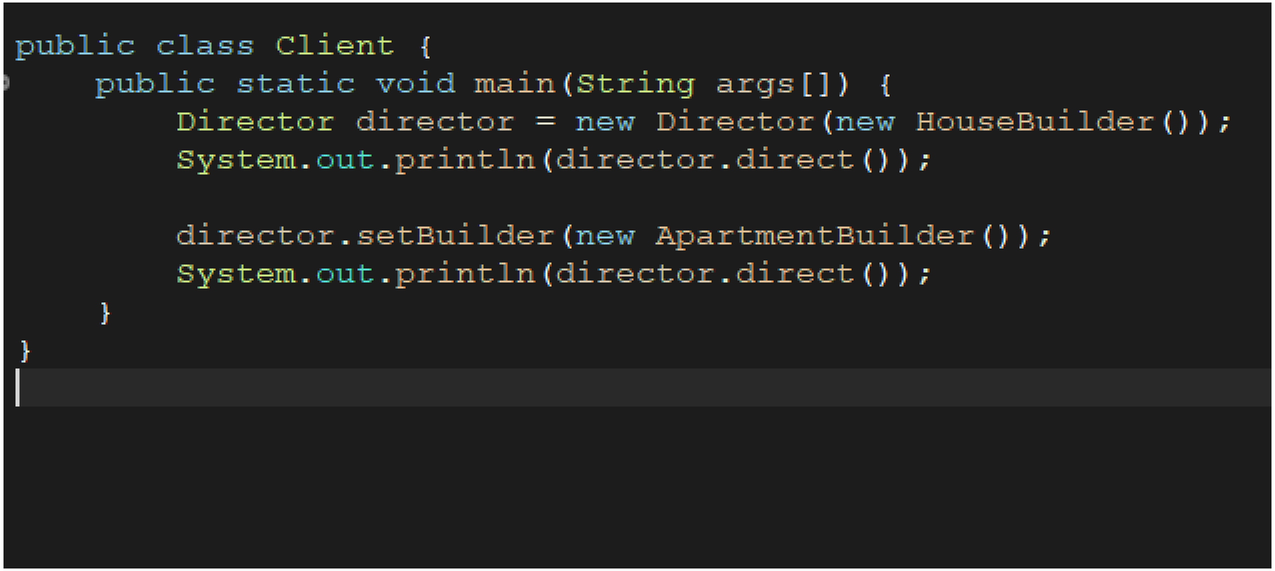
```java
public class ApartmentBuilder implements Builder {
    private Building apartment;
    public ApartmentBuilder() {
        apartment = new Building();
    }
    @Override
    public void buildBasement() {
        // TODO Auto-generated method stub
        System.out.println("挖地基，修车库");
        apartment.setBasement("└＿＿＿＿＿＿＿＿＿┘\n");

    }

    @Override
    public void buildWall() {
        // TODO Auto-generated method stub
        System.out.println("搭建框架");
        for(int i = 0;i<6;i++) {
        apartment.setBasement("|□ □ □ □ □|\n");
        }

    }

    @Override
    public void buildRoof() {
        // TODO Auto-generated method stub
        System.out.println("封顶，通风");
        apartment.setBasement("┌──────────┐\n");
    }

    @Override
    public Building getBuilding() {
        // TODO Auto-generated method stub
        return apartment;
    }

}
```

Course: Soft Design



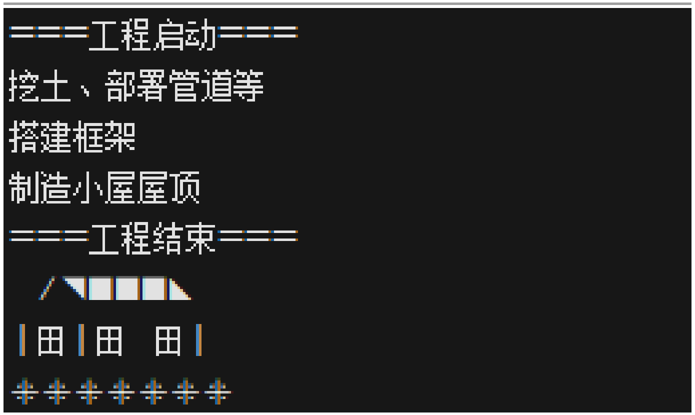
```
public class Director {
    private Builder builder;

    public Director(Builder builder) {
        this.builder = builder;
    }

    public void setBuilder(Builder builder) {
        this.builder = builder;
    }
    public Building direct() {
        System.out.println("===工程启动===");
        builder.buildBasement();
        builder.buildWall();
        builder.buildRoof();
        System.out.println("===工程结束===");
        return builder.getBuilding();
    }
}
```

Course: Soft Design

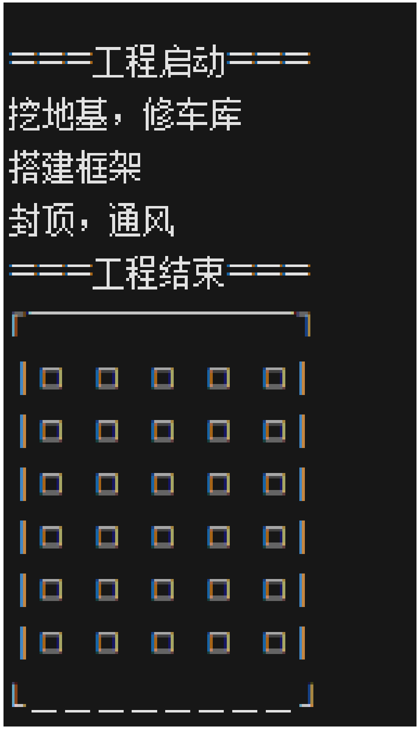


```
public class Client {
    public static void main(String args[]) {
        Director director = new Director(new HouseBuilder());
        System.out.println(director.direct());

        director.setBuilder(new ApartmentBuilder());
        System.out.println(director.direct());
    }
}
```

Course: Soft Design



```
===工程启动===
挖土、部署管道等
搭建框架
制造小屋屋顶
===工程结束===
  /◥■■■◣
 |田|田  田|
 ╪╪╪╪╪╪╪
```



```
===工程启动===
挖地基，修车库
搭建框架
封顶，通风
===工程结束===
┌─────────┐
|□ □ □ □ □|
|□ □ □ □ □|
|□ □ □ □ □|
|□ □ □ □ □|
|□ □ □ □ □|
|□ □ □ □ □|
└_ _ _ _ _ _ _ _┘
```

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.

Principle 40

Determine the requirements now!

The right solution is to do whatever it takes to learn as many of the requirements as possible now. Do prototyping. Talk with more customers. Work for a month with a customer to get to know his or her job firsthand. Collect data. Do whatever it takes. Now document the requirements that you understand and plan to build a system to meet those requirements. If you expect requirements to change significantly, that's okay; plan to build incrementally (Principle 14), but that is no excuse for doing a poor job of requirements specification on any one increment.

Principle 41

Fix requirements specification errors now

If you have errors in the requirements specification, they will cost you:

- Five times more to find and fix if they remain until design.
- Ten times more if they remain until coding.
- Twenty times more if they remain until unit testing. Two hundred times more if they remain until delivery.

That is more than convincing evidence to fix them during the requirements phase!

Principle 42

Prototypes reduce risk in selecting user interfaces

There is nothing as useful as a prototype for taking a low-risk, high-payoff approach for reaching agreement on a user interface prior to full-scale development. There are myriad tools to assist in creating screen displays quickly. These so-called"storyboards" give the users the impression of a real system. Not only do they help nail down requirements, they also win the hearts of the customers and users.

Principle 43

Record why requirements were included

When a requirements decision is made(such as a two-second response time), record a pointer to its origin. For example, if the decision was made during an interview with a customer, record the day and time, as well as the participants in the interview.Ideally,refer explicitly to a tran-script, tape recording, or video recording. It is only with such documenta-tion that one can (1) evolve requirements later or(2) respond to situations where the as-built system fails to satisfy the requirements.

Principle 44

Identify subsets

When writing a requirements specification, clearly identify the minimal subset of requirements that might be useful. Such identification provides software designers with insight into optimal software design. For example, it will enable designers to:

1. More easily embed just one function per component.
2. 2. Select architectures that are more contractible and extendible.
3. 3. And understand how to reduce functionality in the case of a schedule or budget crunch.

Principle 45

Review the requirements

Many parties have a stake in the success of a product development: users, customers, marketing personnel, developers, testers, quality assurance personnel, and so on. All of them also have a stake in the correctness and completeness of the requirements specification. A formal review of the SRS should be conducted prior to a major investment in the design or code.

Principle 46

Avoid design in requrements

The purpose of the requirements phase is to specify external behavior of the solution system. This behavior should be specific enough to ensure that all designers will reach the same conclusion about intended behavior when they use the specification as an oracle. It should not, however, specify a software architecture or algorithm, for this is the realm of the designer. Designers will later select architectures and algorithms for optimal satis-faction of requirements.

Principle 47

Use the right techniques

No requirements technique works for all applications. The requirements for complex applications can be understood only when multiple techniques are used. Use a technique or set of techniques most appropriate for your application.

Principle 48

Use multiple views of requirements

Any one"view" of requirements is insufficient to understand or describe the desired external behavior of a complex system. Instead of using structured analysis, or object-oriented analysis, or statecharts, select a combination that make sense—and use them.

Principle 49

Organize requirements sensibly

We usually organize requirements hierarchically. This helps readers under-stand the system's functions and helps requirements writers locate sections when needs change. There are many ways to organize requirements; selection of the most appropriate way is dependent on the specific product.

Principle 50

Prioritize requirements

One way to prioritize requirements is to suffix every requirement in the specification with an M, D, or O to connote mandatory, desirable, and optional requirements. Although this creates the oxymoronic concept of an optional requirement, it expresses clearly and precisely the relative priori-ties. An even better way is to rate the importance of every requirement on a scale from 0 to 10.

Principle 51

Write concisely

I often see requirements specifications with sentences like:

- *The tarcet tracking function shall provide the capability to display the current tracking coordinates of all active tarcets.*

Contrast this with:

- *When tracking,the system shall disilay the current positions of all active targets.*

Principle 52

Separately number every requirement

It is essential that every requirement in the requirements specification be easily referenceable. This is necessary to enable later tracing to the requirements from design (Principle 62) and from test(Principle 107). The easiest way to do this is to tag every requirement with a unique identifier(such as"[Requirement R27]").An alternative is to number every paragraph and then refer to a requirement in sentence k of paragraph i.j as "requirement i.j-sk."

Principle 53

Reduce ambiguity in requirements

Three effective techniques at reducing ambiguity are:

1. Performing Fagan-type inspections on the SRS.
2. 2. Trying to construct more formal models of the requirements and rewriting the natural language as problems are found (Principle 28).
3. 3. Organizing the SRS so that facing pages contain natural language and more formal models,respectively.

Principle 54

Augment, never replace, natural language

To alleviate this problem when using a formal notation, retain the natural language specification. In fact, one good idea is to keep the natural language and more formal specification side-by-side on opposing pages. Do a manual check between the two to verify conformity. The results will be that all readers can understand something and that some nonmathematical readers may learn something useful.

Principle 55

Write natural language before a more formal model

Note that, in the latter example, the text does not help the reader at all. The best approach is to (1) write the natural language,(2) write the formal model, and (3) adapt the natural language to reduce ambiguities that become apparent when writing the formal model.

Principle 56

Poor requirements yield poor cost estimates

A requirements specification must be read and understood by a wide range of individuals and organizations: users, customers, marketing persormel, requirements writers, designers, testers, managers, and others. The docu-ment must be written in a manner that enables all these people to fully appreciate the system needed and being built so that there are no surprises.

Principle 57

Keep the requirements specification readable

When writing reliability requirements, differentiate between:

1. Failure on demand. What is the likelihood, measured as a percentage of requests, that the system will fail to respond correctly? For example, "THE SYSTEM SHALL CORRECTLY REPORT 99.999 PERCENT OF PATIENT VITAL SIGN ANOMALIES."
2. 2. Rate of failure. This is the same as"failure on demand" but it is measured as a percentage of time. For example,"THE SYSTEM MAY FAIL TO REPORT A PATIENT VITAL SIGN ANOMALY NO MORE OFTEN THAN ONCE PER YEAR."
3. 3.Availability. What percentage of time may the system be unavailable for use? For example,"THE TELEPHONE SYSTEM SHALL BE AVAILABLE 999 PER-CENT OF THE TIME IN ANY GIVEN CALENDAR YEAR."