第二次实验参考资料

1 继承、多态和抽象

1.1 继承中构造方法的执行

1. 子类的构造过程中必须调用其父类的构造方法。
2. 子类可以在自己的构造方法中使用 `super(argument_list)` 调用父类的构造方法，也可以用 `this(argument_list)` 调用本类的另外构造方法。如果调用 `super`，必须写在子类构造方法的第一行。
3. 如果子类的构造方法中没有显式调用父类的构造方法，则系统默认调用基类的无参数构造方法。
4. 如果子类构造方法中既没有显式调用父类的构造方法，而父类又没有无参数的构造方法，则编译出错。

1.2 方法的重写

1. 类的继承关系可以产生一个子类，子类继承父类，它具备了父类所有的特征，继承了父类所有的方法和变量。
2. 子类可以定义新的特征，当子类需要修改父类的一些方法进行扩展，增加功能，称为重写（override），也叫称为覆写或覆盖。
3. 重写是建立在继承关系上的功能扩展机制。
4. 方法重写的规则：
    - 重写方法不能使用比被重写方法更严格的可见性
    - 参数列表必须与被重写方法的相同
    - 返回类型必须与被重写方法的相同

1.3 类型转换

1. 向上转型是允许的，但无法访问子类独有的方法和重写的方法。
2. 下转型可能存在不安全。
3. 类型转换不改变对象本身的内容，只是改变对象引用的类型。

1.4 接口

1. 接口使抽象的概念更深入一层，定义类之间的共同行为。接口规定操作的基本形式，包括名称、参数列表以及返回类型，但不规定操作的实现，接口中也可以包含常量，但如果包含一定声明为 `static` 和 `final`
2. 类可以实现接口中定义的操作，其中所有的操作都必须实现，实现接口的方法必须声明为 `public`

1.5 多态

多态是同一个行为具有多个不同表现形式或形态的能力。可以使用重写和实现接口两种方法实现。

- 通过重写实现多态。必要条件是继承、重写、父类引用指向子类对象。当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误；如果有，再去检查子类的是否有同名方法。如果子类有同名方法，则调用子类的方法，否则调用父类的方法。
- 通过实现接口实现多态。直接调用实现该接口的类的对应方法。

2 工厂模式

开始介绍工厂模式之前，首先我们需要知道工厂模式的适用场景。

首先，作为一种创建类模式，在任何需要生成**复杂对象**的地方，都可以使用工厂模式。有一点需要注意的地方就是复杂对象适合使用工厂模式，而简单对象，特别是只需要通过new就可以完成创建的对象，无需使用工厂模式。如果使用工厂模式，就需要引入一个工厂类，会增加系统的复杂度。

其次，工厂模式是一种典型的解耦模式。如果调用者在组装产品时会造成用于组装产品的类对于调用者的依赖时，可以考虑使用工厂模式，将会大大降低对象之间的耦合度。

再次，由于工厂模式依赖于接口，把具体产品的实例化工作交由实现类完成，扩展性比较好。也就是说，当需要系统有比较好的扩展性时，可以考虑工厂模式，不同的产品用不同的实现工厂来组装。

2.1 简单工厂模式

2.1.1 描述

简单工厂模式是由一个工厂对象根据收到的消息决定要创建的类的对象实例。在一个简单工厂中一般包含下面几个部分：

- 工厂类：当我们想要获得由此工厂生产的产品时，只要调用工厂类里的生产产品的方法，传入我们想要生产的产品名即可。
- 产品接口：一个接口，工厂所生产的产品需要实现这个接口。可以理解为该工厂所生产的产品的一个共同的模具。
- 产品类：具体生产的产品，产品需要实现产品接口所要求的方法。

2.1.2 使用举例

我们使用电脑生产工厂举例。为了简化场景，我们认为一台电脑由`基本部分`、`处理器`、`内存`、`显示屏`四个部分组成，即：

```
interface BasicPart {...}
public class WeakBasicPart implements BasicPart {...}
public class StrongBasicPart implements BasicPart {...}

interface Cpu {...}
public class IntelCpu implements Cpu {...}
public class AmdCpu implements Cpu {...}

interface Ram {...}
public class SmallCapacityRam implements Ram {...}
public class LargeCapacityRam implements Ram {...}

interface Display {...}
public class LowQualityDisplay implements Display {...}
public class HighQualityDisplay implements Display {...}
```

现在我们构建一个简单工厂，假设现在此工厂只使用上面的零件生产两种型号的笔记本电脑。

- 工厂类

```
public class ComputerFactory {
    public Computer getComputer(String type) {
        if ("LaptopA".equals(type)) {
            BasicPart bp = new StrongBasicPart();
```

```
            Cpu c = new AmdCpu();
            Ram r = new SmallCapacityRam();
            Display d = new LowQualityDisplay();
            return new Laptop(bp, c, r, d);
        } else if ("LaptopB".equals(type)) {
            BasicPart bp = new WeakBasicPart();
            Cpu c = new IntelCpu();
            Ram r = new LargeCapacityRam();
            Display d = new HighQualityDisplay();
            return new Laptop(bp, c, r, d);
        } else {
            return null;
        }
    }
}
```

- 产品接口

```
public interface Computer {
    public void boot();
}
```

- 产品类

```
public class Laptop implements Computer {
    private BasicPart basicPart;
    private Cpu cpu;
    private Ram ram;
    private Display display;

    public Laptop(BasicPart bp, Cpu c, Ram r, Display d){
        ... ...
    }

    @Override
    public void boot() {
        System.out.println("Starting to boot Laptop");
    }
}
```

- 使用

```
public class Main {
    public static void main(String[] args) {
        ComputerFactory computerFactory = new ComputerFactory();
        Computer c0 = computerFactory.getComputer("LaptopA");
        Computer c1 = computerFactory.getComputer("LaptopB");
        c0.boot();
        c1.boot();
        // do something
    }
}
```

2.1.3 分析

上述例子由于一台笔记本电脑需要由多个部件组装，所以如果不使用简单工厂模式，这些组装改为在`Main`中进行，但是对于笔记本电脑对象的方法的调用者`Main`而言，其不应当关心笔记本电脑是如何组装的，因此这种做法会造成耦合度增加，不利于扩展。

如果除了笔记本电脑，还想要生产其他电子设备，比如台式电脑、平板电脑等，由于它们都有着共同的功能属性，所以只要对应增加产品类，实现工厂的产品接口，同时对应修改工厂类中的生产方法`getComputer`即可。

2.2 工厂方法模式

2.2.1 描述

上面介绍的简单工厂模式中，我们每需要生产新的产品*系列*时，都需要重新去写新的工厂。所以对上面的简单工厂模式进一步抽象后就得到了工厂方法模式。相比于简单工厂模式，工厂方法模式增加定义了一个创建工厂的*工厂接口*，提高了工厂实现的扩展性。

我们可以看到，上面的简单工厂模式中的例子仍存在一些不必要的耦合。我们在添加产品时需要修改工厂类的生产方法，例如增加一个笔记本型号`LaptopC`，需要在`getComputer`中的条件判断进行增加，并且对其进行组装。

因此，我们引入了工厂方法模式，将上面一个工厂生产多个型号的产品改为一个工厂生产一个型号的产品，然后有多个工厂，这多个工厂共同实现一个工厂接口。

2.2.2 使用举例

我们仍继续使用上述笔记本电脑组装工厂的例子。

- 工厂接口

```
interface ComputerFactory {
    public Computer getComputer();
}
```

- 工厂类

```
public class LaptopAFactory implements ComputerFactory {
    @Override
    public Computer getComputer() {
        BasicPart bp = new StrongBasicPart();
        Cpu c = new AmdCpu();
        Ram r = new SmallCapacityRam();
        Display d = new LowQualityDisplay();
        return new Laptop(bp, c, r, d);
    }
}

public class LaptopBFactory implements ComputerFactory {
    @Override
    public Computer getComputer() {
        BasicPart bp = new WeakBasicPart();
        Cpu c = new IntelCpu();
        Ram r = new LargeCapacityRam();
        Display d = new HighQualityDisplay();
        return new Laptop(bp, c, r, d);
    }
}
```

- 产品接口与产品类与2.1.2中相同
- 使用

```
public class Main {
    public static void main(String[] args) {
        ComputerFactory laptopAFactory = new LaptopAFactory();
        ComputerFactory laptopBFactory = new LaptopBFactory();
        Computer c0 = laptopAFactory.getComputer();
        Computer c1 = laptopBFactory.getComputer();
        c0.boot();
        c1.boot();
        // do something
    }
}
```

2.2.3 分析

由上面的举例可以看出，在使用工厂方法模式后，每当我们增加其他类型的电脑时，只要再增加一个对应的工厂类的实现即可。

2.3 抽象工厂模式

2.3.1 描述

相比于最初的简单工厂模式，工厂方法模式已经从一定程度上进行了解耦，而且就我们在2.2.2部分的举例而言此模式比较合理。在该工厂中，生产的产品只有一种**类别**，那就是笔记本电脑，尽管这一个类别中有两个**型号**（即`LaptopA`和`LaptopB`）。

但是考虑一个新的需求，我们还需要生产台式电脑（Desktop）,台式电脑也分A和B两种型号。这时我们可以得到这样的一个树状的层级关系，即`LaptopA`与`LaptopB`为`Laptop`的子节点，`DesktopA`与`DesktopB`为`Desktop`的子节点，`Laptop`与`Desktop`属于同一级，且均为`Computer`的子节点。这时，我们引入一个**产品族**的概念：所谓的产品族，是指位于不同产品等级结构中功能相关联的产品组成的家族。抽象工厂模式所提供的一系列产品就组成一个**产品族**；而工厂方法提供的一系列产品称为一个**等级结构**。例如，上述示例中`Laptop`与`Desktop`下属的不同型号的产品构成两个等级结构，每个等级结构代表一类产品，这两类产品组成的一个产品系列`Computer`称为产品族。

如2.2.2中的示例，如果只有一个等级结构（即`Laptop`），那么使用工厂方法模式就足够，但是对于多个等级结构的工厂而言，例如使生产`LaptopA`和`DesktopB`的工厂同级并列，那么显然是不合适的。

基于以上考虑，我们引入抽象工厂模式。总体而言与工厂方法模式相同，仍然包括工厂接口、工厂类、产品接口、产品类。不同之处在于，在工厂类中，我们需要实现一个**等级结构**下的不同产品的创建方法。

2.3.2 使用举例

- 工厂接口

```
interface ComputerFactory {
    public Computer getComputerA();
    public Computer getComputerB();
}
```

- 工厂类

```
public class LaptopFactory implements ComputerFactory {
```

```
    @Override
    public Computer getComputerA() {
        BasicPart bp = new StrongBasicPart();
        Cpu c = new AmdCpu();
        Ram r = new SmallCapacityRam();
        Display d = new LowQualityDisplay();
        return new Laptop(bp, c, r, d);
    }
    @Override
    public Computer getComputerB() {
        BasicPart bp = new WeakBasicPart();
        Cpu c = new IntelCpu();
        Ram r = new LargeCapacityRam();
        Display d = new HighQualityDisplay();
        return new Laptop(bp, c, r, d);
    }
}

public class DesktopFactory implements ComputerFactory {
    @Override
    public Computer getComputerA() {
        ... ...
        return new Desktop(... ...);
    }
    @Override
    public Computer getComputerB() {
        ... ...
        return new Desktop(... ...);
    }
}
```

- 产品接口，由于台式电脑与笔记本电脑功能类似，所以仍使用之前举例使用的`Computer`接口。
- 产品类，笔记本电脑`Laptop`类与之前相同，`Desktop`类类似`Laptop`类，只要实现`Computer`接口并提供合适的构造方法供`DesktopFactory`使用即可。
- 使用

```
public class Main {
    public static void main(String[] args) {
        ComputerFactory laptopFactory = new LaptopFactory();
        ComputerFactory desktopFactory = new DesktopFactory();
        Computer la = laptopFactory.getComputerA();
        Computer lb = laptopFactory.getComputerB();
        Computer da = desktopFactory.getComputerA();
        Computer db = desktopFactory.getComputerB();
        la.boot();
        lb.boot();
        da.boot();
        db.boot();
        // do something
    }
}
```

2.3.3 分析

抽象工厂模式是工厂方法模式的一个更加一般的形式。总结上述三种工厂模式可知，简单工厂是只有一个工厂（而没有可用于扩展的工厂接口）的工厂方法模式，而工厂方法是只包含一个等级结构（即只有一个类别的产品）的抽象工厂。

3 参考资料

[1] 工厂方法模式

[2] 抽象工厂模式

[3] 工厂模式、工厂方法模式、抽象工厂模式详解