第一单元第二次实验指导

面向对象设计与构造课程组

实验时间安排

实验时间	预计实验内容
第三周 周三 11 ~12节 2020/3/12 19:00-20:35	第一单元第二次实验
第五周 周三 11 ~12节 2020/3/26 19:00-20:35	第二单元第一次实验
第七周 周三 11 ~12节 2020/4/9 19:00-20:35	第二单元第二次实验
第九周 周三 11 ~12节 2020/4/23 19:00-20:35	第三单元第一次实验
第十二周 周三 11 ~12节 2020/5/13 19:00-20:35	第三单元第二次实验
第十四周 周三 11 ~12节 2020/5/27 19:00-20:35	第四单元第一次实验
第十六周 周三 11 ~12节 2020/6/10 19:00-20:35	第四单元第二次实验

- 每次实验需按照课表时间上课，如无特殊情况准时开始与结束实验，请同学们把握好时间。
- 为了保证实验高质量和高效完成，请同学们课前做好预习，观看实验指导视频。
- 考虑到网络的问题，实验作业的开放时间为19:00-21:00。

上节实验回顾

- 学习基础的git操作命令
  - 如何关联远程仓库 (git remote add ...)
  - 如何编写 .gitignore 忽略不必要的文件
  - 对于已经提交过的需要忽略的文件需要先 git rm --cached files
  - 如何删除和远程仓库的关联 (git remote remove ...)
  - 整体完成情况效果较好，存在少部分同学 git push 的时候出现问题
  - 最后的辅助资料：git - 简明指南（对Git还不是特别理解的同学一定要看，也可以从gitlab中获取该指南）

上节实验回顾

- 学习接口的基本知识并实现一个接口
  - 整体完成情况较好，有179名同学获得满分
- 参考答案：(来自一位满分同学的代码)

Term.java toString

```
@Override
public String toString() {
    if (coef == 0) {
        return "0";
    } else if (index == 0) {
        return String.valueOf(coef);
    } else if (coef == 1 && index != 1) {
        return "x^" + index;
    } else if (coef == 1) {
        return "x";
    } else if (coef == -1 && index != 1) {
        return "-x^" + index;
    } else if (coef == -1) {
        return "-x";
    } else if (index == 1) {
        return coef + "*x";
    } else {
        return coef + "*x^" + index;
    }
}
```

Term.java compareTo

```
@Override
public int compareTo(Term o) {
    if (this.index != o.index) {
        return o.index - this.index;
    } else {
        return this.coef - o.coef;
    }
}
```

Input.java

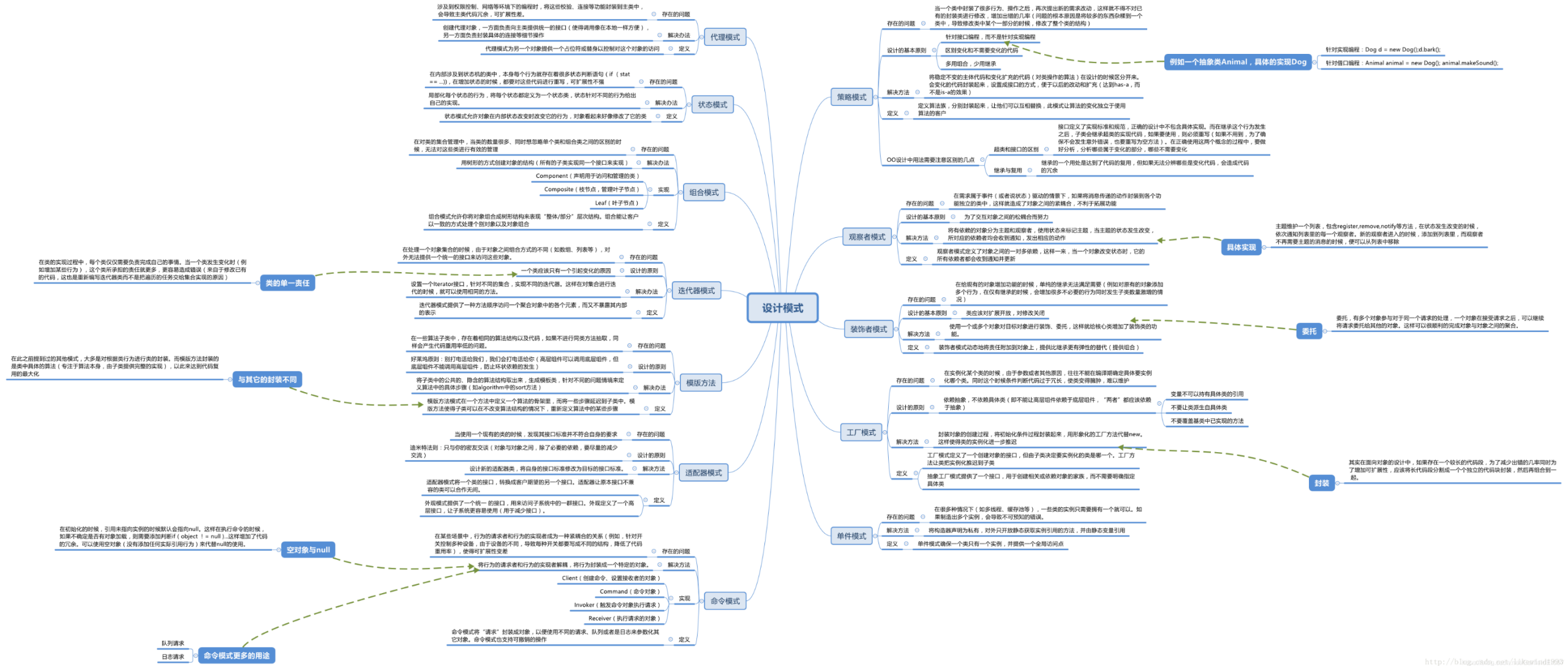
```
    // TODO
    // code for sort
    Collections.sort(poly);

    // code for output
    for (Term t : poly) {
        System.out.println(t.toString());
    }
```

第一单元第二次实验训练要点

- 了解Java设计模式
- 熟悉工厂模式中的简单工厂模式、工厂方法模式和抽象工厂模式的使用范围
- 通过使用工厂模式完成课上任务

Java 设计模式思维导图(辅助学习)



Java 设计模式分类

工厂模式介绍

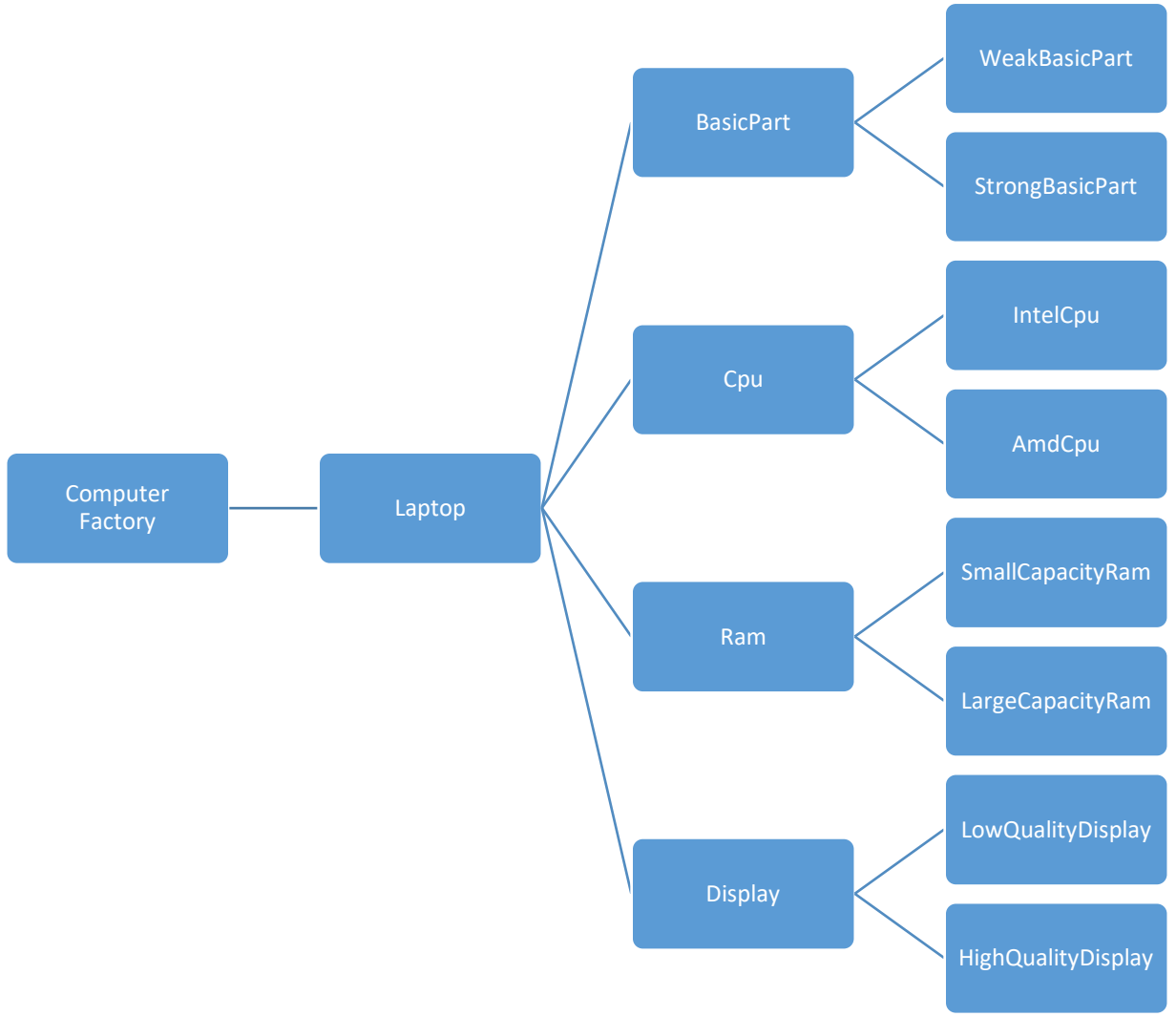
- 工厂模式是创建类模式，在任何需要生成复杂对象的地方，都可以使用工厂模式。
- 工厂模式是一种典型的解耦模式。如果调用者在组装产品时会造成用于组装产品的类对于调用者的依赖时，可以考虑使用工厂模式，将会大大降低对象之间的耦合度。
- 工厂模式依赖于接口，把具体产品的实例化工作交由实现类完成，扩展性好。
- 注：下面提到的例子在实验二的参考资料中都有代码的实现，具体细节查阅参考资料

工厂模式介绍（cont.）

- 简单工厂模式
- 工厂方法模式
- 抽象工厂模式

简单工厂模式

- 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。



简单工厂模式（cont.）

- 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");
    }
}
```

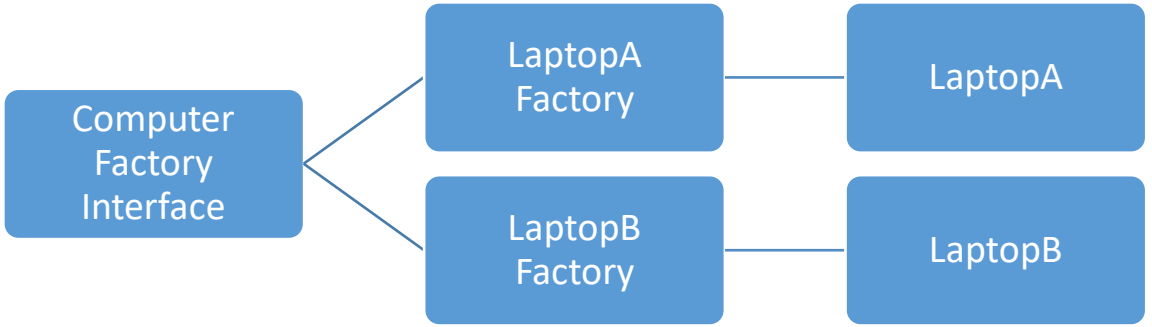
- 工厂类

```
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;
        }
    }
}
```

```
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
    }
}
```

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
- 举个例子：
  - 现在我们要生产两种Laptop，分别是LaptopA和LaptopB
  - 我们可以通过构建两个分别的工厂LaptopA Factory和LaptopB Factory，他们分别根据需要生产的Laptop的不同,具体实现了 Computer Factory Interface 中的方法



工厂方法模式(cont.)

- 工厂接口

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

- 工厂类

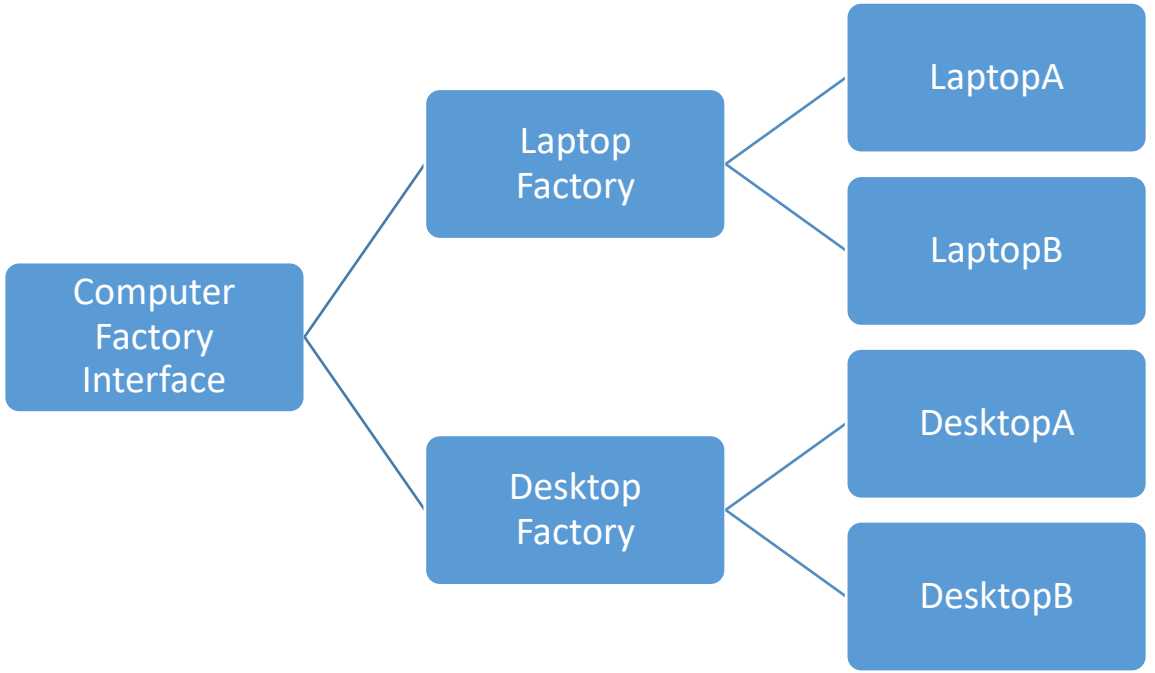
```
public class LaptopAFactory implements ComputerFactory {
    @Override
    public Computer getComputer() {
        BasicPart bp = new StrongBasicPart();
        Cpu c = new AmdCpu();
        Ram r = new SmallCapacityRam();
        Display d = new LowQualityDisplay();
        return new Laptop(bp,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

```
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
    }
}
```

抽象工厂模式

- 在上面的工厂方法模式中，对比简单工厂模式已经有了一定的解耦——对于新加的Laptop的种类可以通过构建新的工厂去生产，而不是在唯一一个工厂中添加生产方式。
- 现在我们考虑一个新的需求：刚才的LaptopA 和 LaptopB 都是 Laptop 系列的两种产品，现在我们想要生产 Desktop 系列的产品，并且也有两种 DesktopA 和 DesktopB



抽象工厂模式（cont.）

- 工厂接口

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

```
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
    }
}
```

- 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(... ...);
    }
}
```

Q&A