**体系结构期末复习**

1. **选择题**

**（一）**

1. 设计模式的基本原理是( **C** )

A. 面向实现编程 B. 面向对象编程 C. 面向接口编程 D. 面向组合编程

2. 设计模式的两大主题是( **D** )

A. 系统的维护与开发 B. 对象组合与类的继承

C. 系统架构与系统开发 D. 系统复用与系统扩展

3. 依据设计模式思想，程序开发中应优先使用的是( **A** )关系实现复用。

A. 组合聚合 B. 继承 C. 创建 D. .以上都不对

4. 关于继承表述错误的是( **D** )

A. 继承是一种通过扩展一个已有对象的实现，从而获得新功能的复用方法。

B. 泛化类（超类）可以显式地捕获那些公共的属性和方法。特殊类（子类）则通过附加属性和方法来进行实现的扩展。

C. 破坏了封装性，因为这会将父类的实现细节暴露给子类。

D. 继承本质上是“白盒复用”，对父类的修改，不会影响到子类。

5. 常用的设计模式可分为( **A** )

A. 创建型、结构型和行为型 B. 对象型、结构型和行为型

C. 过程型、创建型和结构型 D. 抽象型、接口型和实现型

6. “不要和陌生人说话” 是对( **D** )设计原则的通俗表述。

A. 接口隔离 B. 里氏代换 C. 依赖倒转 D. .迪米特法则

7. 在适配器模式中，对象适配器模式是对( **A** )设计原则的典型应用

A. 合成聚合 B. 里氏代换 C. 依赖倒转 D. .迪米特法则

8. 将一个类的接口转换成客户希望的另一个接口，这句话是对（**C**）设计模式的描述

A. 策略模式 B. 桥接模式 C. 适配器模式 D. 单例模式

9. 以下设计模式中属于结构模式的是( **D** )

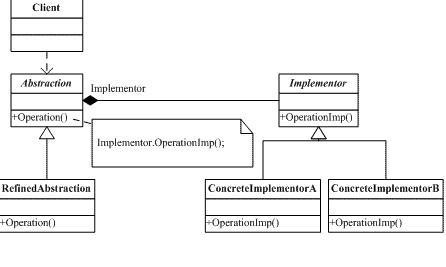
A. 观察者模式 B. 单例模式 C. 策略模式 D. 外观模式

10. 以下不属于对象行为型模式是( **D** )

A. 命令模式 B. 策略模式 C. 访问者模式 D. 桥接模式

11. 下面的类图表示的是哪个设计模式( **D** )

A. 抽象工厂模式 B. 观察者模式 C. 策略模式 D. 桥接模式

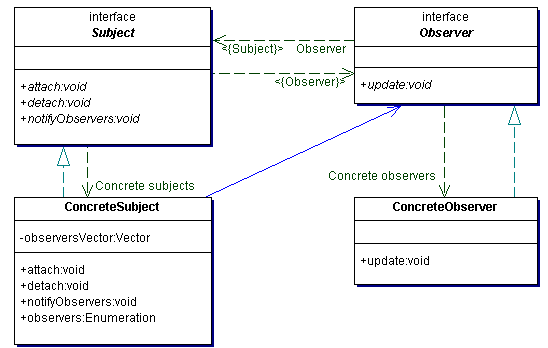


12. Open-Close开闭原则的含义是一个软件实体( **A** )

A. 应当对扩展开放，对修改关闭。 B. 应当对修改开放，对扩展关闭。

C. 应当对继承开放，对修改关闭。 D. 以上都不对。

13. 下面的类图表示的是哪个设计模式( **D** )



A. 策略模式 B. 装饰模式 C. 桥接模式 D. 观察者模式

14. 保证一个类仅有一个实例，并提供一个访问它的全局访问点。这句话是对( **D** )设计模式的描述。

A. 外观模式 B. 策略模式 C. 适配器模式 D. 单例模式

15. 以下意图哪个是用来描述组合模式？( **C** )

A. 为其他对象提供一种代理以控制对这个对象的访问。

B. 运用共享技术有效地支持大量细粒度的对象。

C. 将对象组合成树形结构以表示“部分-整体”的层次结构。

D. 将一个复杂对象的构建与它的表示分离。

16. 以下意图哪个是用来描述命令模式？( **A** )

A. 将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化。

B. 定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。

C. 为其他对象提供一种代理以控制对这个对象的访问。

D. 保证一个类仅有一个实例，并提供一个访问它的全局访问点。

17. 以下哪种情况不适合使用适配器模式？( **D** )

A. 你想使用一个已经存在的类，而它的接口不符合你的需求。

B. 你想创建一个类，该类可以复用另外一个类的功能。

C. 你想创建一个类，该类可以复用另外多个类的功能。

D. 你想在类中用相对较少的对象引用取代很多对象的引用。

18. 以下意图哪个是用来描述观察者模式？( **B** )

A. 将抽象部分与它的实现部分分离，使它们都可以独立地变化。

B. 定义对象间的一种一对多的依赖关系,当一个对象的状态发生改变时, 所有依赖于它的对象都得到通知并被自动更新。

C. 用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

D. 使多个对象都有机会处理请求，避免请求的发送者和接收者之间的耦合关系。

19. 以下意图哪个是用来描述状态模式？( **C** )

A. 使多个对象都有机会处理请求，避免请求的发送者和接收者之间的耦合关系。

B. 顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。

C. 允许一个对象在其内部状态改变时改变它的行为。看起来似乎修改了它的类。

D. 捕获一个对象的内部状态，并在该对象之外保存这个状态。

20. 以下意图哪个是用来描述策略模式？( **D** )

A. 将抽象部分与它的实现部分分离，使它们都可以独立地变化。

B. 将一个复杂对象的构建与它的表示分离。

C. 将抽象部分与它的实现部分分离，使它们都可以独立地变化。

D. 定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | |
| C | D | A | D | A | D | A | C | D | D | |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| D | A | D | D | C | A | D | B | C | D |

**（二）**

1. 要依赖于抽象，不要依赖于具体。即针对接口编程，不要针对实现编程,是( **D** )

A. 开闭原则 B. 接口隔离原则 C. 里氏代换原则 D. 依赖倒转原则

2. 以下对"开-闭"原则的一些描述错误的是( **A** )

A. "开-闭"原则与"对可变性的封装原则"没有相似性。

B. 找到一个系统的可变元素,将它封装起来,叫"开-闭"原则。

C. 对修改关闭: 是其原则之一。

D. 从抽象层导出一个或多个新的具体类可以改变系统的行为,是其原则之一。

3. 依据设计模式思想，程序开发中应优先使用的是( **B** )关系实现复用。

A. 继承 B. 组合聚合 C. 创建 D. .以上都不对

4. 设计模式的两大主题是( **C** )

A. 系统的维护与开发 B. 对象组合与类的继承

C. 系统复用与系统扩展 D. 系统架构与系统开发

5. 常用的设计模式可分为( **C** )

A. 过程型、创建型和结构型 B. 对象型、结构型和行为型

C. 创建型、结构型和行为型 D. 抽象型、接口型和实现型

6. “知道的越少越好” 是对( **D** )设计原则的通俗表述。

A. 接口隔离 B. 里氏代换 C. 依赖倒转 D. .迪米特法则

7. 在适配器模式中，对象适配器模式是对( **A** )设计原则的典型应用

A. 合成聚合 B. 里氏代换 C. 依赖倒转 D. .迪米特法则

8. 观察者模式定义了一种( **A** )的依赖关系

A. 一对多 B. 多对多 C. 一对一 D. 以上都不对

9. 以下设计模式中不属于创建型模式的是( **B** )

A. 工厂模式 B. 外观模式 C. 生成器模式 D. 单例模式

10. 以下不属于结构型模式是( **C** )

A. 组合模式 B. 适配器模式 C. 访问者模式 D. 桥接模式

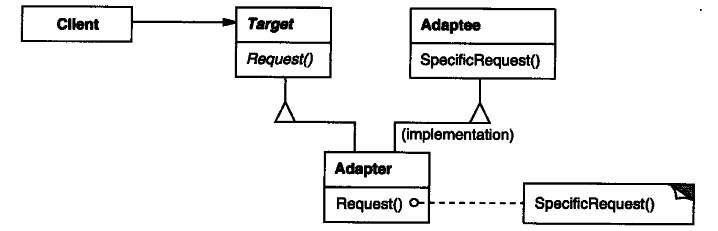
11. 以下不属于行为型模式是( **B** )

A. 迭代器模式 B. 外观模式 C. 状态模式 D. 策略模式

12. 将一个类的接口转换成客户希望的另一个接口，这句话是对（**C**）设计模式的描述

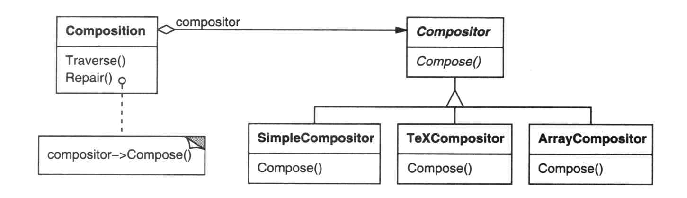
A. 策略模式 B. 桥接模式 C. 适配器模式 D. 单例模式

13. 下面的类图表示的是哪个设计模式( **C** )



A. 策略模式 B. 装饰模式 C. 适配器模式 D. 观察者模式

14. 下面的类图表示的是哪个设计模式( **B** )

 A. 桥接模式 B. 组合模式 C. 命令模式 D. 观察者模式

15. 保证一个类仅有一个实例，并提供一个访问它的全局访问点。这句话是对( **D** )设计模式的描述。

A. 外观模式 B. 策略模式 C. 适配器模式 D. 单例模式

16. 以下哪项不是桥接模式的优点？( **C** )

A. 分离接口及其实现部分。 B. 提高可扩充性。

C. 改变值以指定新对象。 D. 实现细节对客户透明。

17. 在观察者模式中，表述错误的是？( **C** )

A. 观察者角色的更新是被动的。

B. 被观察者可以通知观察者进行更新。

C. 观察者可以改变被观察者的状态，再由被观察者通知所有观察者依据被观察者的状态进行。

D. 以上表述全部错误。

18. 当我们想创建一个具体的对象而又不希望指定具体的类时，可以使用( **A** )模式

A. 创建型 B. 结构型 C. 行为型 D. .以上都不对

19. 以下意图哪个是用来描述状态模式？( **C** )

A. 使多个对象都有机会处理请求，避免请求的发送者和接收者之间的耦合关系。

B. 顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。

C. 允许一个对象在其内部状态改变时改变它的行为。看起来似乎修改了它的类。

D. 捕获一个对象的内部状态，并在该对象之外保存这个状态。

20. 以下意图哪个是用来描述组合模式？( **C** )

A. 为其他对象提供一种代理以控制对这个对象的访问。

B. 运用共享技术有效地支持大量细粒度的对象。

C. 将对象组合成树形结构以表示“部分-整体”的层次结构。

D. 将一个复杂对象的构建与它的表示分离。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | |
| D | A | B | C | C | D | A | A | B | C | |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| B | C | C | B | D | C | C | A | C | C |

（三）

1. 常用的设计模式可分为( **A** )

A. 创建型、结构型和行为型 B. 对象型、结构型和行为型

C. 过程型、创建型和结构型 D. 抽象型、接口型和实现型

2. “不要和陌生人说话” 是对( **D** )设计原则的通俗表述。

A. 接口隔离 B. 里氏代换 C. 依赖倒转 D. .迪米特法则

3. 在适配器模式中，对象适配器模式是对( **A** )设计原则的典型应用

A. 合成聚合 B. 里氏代换 C. 依赖倒转 D. .迪米特法则

4. 将一个类的接口转换成客户希望的另一个接口，这句话是对（**C**）设计模式的描述

A. 策略模式 B. 桥接模式 C. 适配器模式 D. 单例模式

5. 以下设计模式中属于结构模式的是( **D** )

A. 观察者模式 B. 单例模式 C. 迭代器模式 D. 适配器模式

6. 以下意图哪个是用来描述命令模式？( **A** )

A. 将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化。

B. 定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。

C. 为其他对象提供一种代理以控制对这个对象的访问。

D. 保证一个类仅有一个实例，并提供一个访问它的全局访问点。

7. 下面的类图表示的是哪个设计模式( **D** )

A. 抽象工厂模式 B. 桥接模式 C. 状态模式 D. 适配器模式

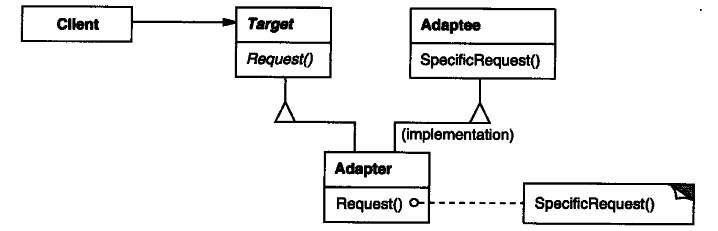


图1

8. Open-Close开闭原则的含义是一个软件实体( **A** )

A. 应当对扩展开放，对修改关闭。

B. 应当对修改开放，对扩展关闭。

C. 应当对继承开放，对修改关闭。

D. 以上都不对。

9. 以下意图哪个是用来描述组合模式？( **C** )

A. 为其他对象提供一种代理以控制对这个对象的访问。

B. 运用共享技术有效地支持大量细粒度的对象。

C. 将对象组合成树形结构以表示“部分-整体”的层次结构。

D. 将一个复杂对象的构建与它的表示分离。

10. 以下意图哪个是用来描述状态模式？( **C** )

A. 使多个对象都有机会处理请求，避免请求的发送者和接收者之间的耦合关系。

B. 顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。

C. 允许一个对象在其内部状态改变时改变它的行为。看起来似乎修改了它的类。

D. 捕获一个对象的内部状态，并在该对象之外保存这个状态。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| A | D | A | C | D | A | D | A | C | C |

**（四）**

1. 当我们想创建一个具体的对象而又不希望指定具体的类时，可以使用( **A** )模式

A. 创建型 B. 结构型 C. 行为型 D. .以上都不对

2. 以下对"开-闭"原则的一些描述错误的是( **A** )

A. "开-闭"原则与"对可变性的封装原则"没有相似性。

B. 找到一个系统的可变元素,将它封装起来,叫"开-闭"原则。

C. 对修改关闭: 是其原则之一。

D. 从抽象层导出一个或多个新的具体类可以改变系统的行为,是其原则之一。

3. 依据设计模式思想，程序开发中应优先使用的是( **B** )关系实现复用。

A. 继承 B. 组合聚合 C. 创建 D. .以上都不对

4. “知道的越少越好” 是对( **D** )设计原则的通俗表述。

A. 接口隔离 B. 里氏代换 C. 依赖倒转 D. .迪米特法则

5. 在适配器模式中，对象适配器模式是对( **A** )设计原则的典型应用

A. 合成聚合 B. 里氏代换 C. 依赖倒转 D. .迪米特法则

6. 保证一个类仅有一个实例，并提供一个访问它的全局访问点。这句话是对( **D** )设计模式的描述。

A. 外观模式 B. 策略模式

C. 适配器模式 D. 单例模式

7. 以下意图哪个是用来描述策略模式？( **D** )

A. 将抽象部分与它的实现部分分离，使它们都可以独立地变化。

B. 将一个复杂对象的构建与它的表示分离。

C. 将抽象部分与它的实现部分分离，使它们都可以独立地变化。

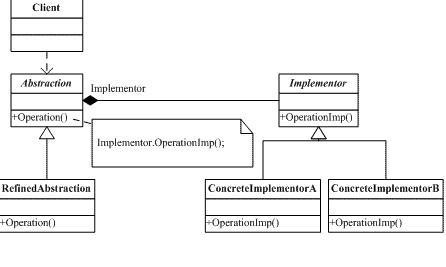
D. 定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。

8. 以下不属于结构型模式是( **C** )

A. 组合模式 B. 适配器模式

C. 访问者模式 D. 桥接模式

9. 下面的类图表示的是哪个设计模式( **D** )



A. 策略模式 B. 状态模式

C. 适配器模式 D. 桥接模式

10. 以下意图哪个是用来描述状态模式？( **C** )

A. 使多个对象都有机会处理请求，避免请求的发送者和接收者之间的耦合关系。

B. 顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。

C. 允许一个对象在其内部状态改变时改变它的行为。看起来似乎修改了它的类。

D. 捕获一个对象的内部状态，并在该对象之外保存这个状态。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| A | A | B | D | A | D | D | C | D | C |

**（五）**

1. 常用的设计模式可分为( **A** )

A. 创建型、结构型和行为型 B. 对象型、结构型和行为型

C. 过程型、创建型和结构型 D. 抽象型、接口型和实现型

2. “不要和陌生人说话” 是对( **D** )设计原则的通俗表述。

A. 接口隔离 B. 里氏代换 C. 依赖倒转 D. .迪米特法则

3. 在适配器模式中，对象适配器模式是对( **A** )设计原则的典型应用

A. 合成聚合 B. 里氏代换 C. 依赖倒转 D. .迪米特法则

4. 将一个类的接口转换成客户希望的另一个接口，这句话是对（**C**）设计模式的描述

A. 策略模式 B. 桥接模式 C. 适配器模式 D. 单例模式

5. 以下设计模式中属于结构模式的是( **D** )

A. 观察者模式 B. 单例模式 C. 迭代器模式 D. 适配器模式

6. 以下意图哪个是用来描述命令模式？( **A** )

A. 将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化。

B. 定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。

C. 为其他对象提供一种代理以控制对这个对象的访问。

D. 保证一个类仅有一个实例，并提供一个访问它的全局访问点。

7. 下面的类图表示的是哪个设计模式( **D** )

A. 抽象工厂模式 B. 桥接模式 C. 状态模式 D. 适配器模式

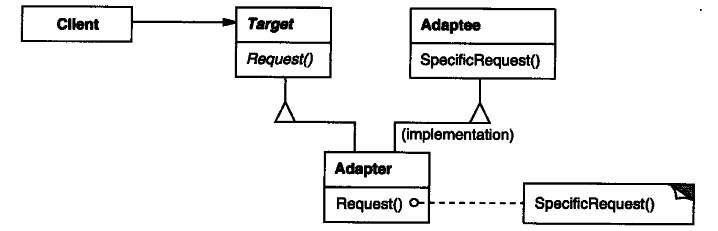


图1

8. Open-Close开闭原则的含义是一个软件实体( **A** )

A. 应当对扩展开放，对修改关闭。

B. 应当对修改开放，对扩展关闭。

C. 应当对继承开放，对修改关闭。

D. 以上都不对。

9. 以下意图哪个是用来描述组合模式？( **C** )

A. 为其他对象提供一种代理以控制对这个对象的访问。

B. 运用共享技术有效地支持大量细粒度的对象。

C. 将对象组合成树形结构以表示“部分-整体”的层次结构。

D. 将一个复杂对象的构建与它的表示分离。

10. 以下意图哪个是用来描述状态模式？( **C** )

A. 使多个对象都有机会处理请求，避免请求的发送者和接收者之间的耦合关系。

B. 顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。

C. 允许一个对象在其内部状态改变时改变它的行为。看起来似乎修改了它的类。

D. 捕获一个对象的内部状态，并在该对象之外保存这个状态。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| A | D | A | C | D | A | D | A | C | C |

1. **填空题**

**（一）**

1. 面向对象的七条设计原则包括：单一职责原则，\_\_\_\_\_\_\_\_\_\_，\_\_\_\_\_\_\_\_\_\_\_，\_\_\_\_\_\_\_\_\_，合成聚合原则，接口隔离原则以及\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 。其中外观模式是\_\_\_\_\_\_\_\_\_原则的典型应用。

2. 工厂模式中分为\_\_\_\_\_\_\_\_\_\_\_,工厂方法,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_三种模式。其中，可以应用平行等级结构完成创建工作的模式是\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_模式。

3. 适配器模式，可以细分为\_\_\_\_\_\_\_\_\_\_\_适配器和\_\_\_\_\_\_\_\_\_\_\_适配器两种实现。其中\_\_\_\_\_\_\_\_\_适配器采用的是继承复用，而\_\_\_\_\_\_\_\_\_\_\_适配器采用的是合成聚合复用。

4. Java API中，有两个与观察者模式相关的类和接口，分别是Observable和Observer，Observer的Update函数中用到的两个参数的参数类型是\_\_\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_\_\_\_\_。

5. 事件体系结构中的三个基本角色包括事件源、\_\_\_\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。其中在编程时一定要将\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_注册添加到事件源中。

6. 单例模式有两种方式实现，分别为\_\_\_\_\_\_\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。它们共同的特征是构造函数的访问属性必须是\_\_\_\_\_\_\_\_\_。

**答案：**

1、开闭原则 里氏代换原则 依赖倒转原则 迪米特法则 迪米特法则

2、简单工厂 抽象工厂 工厂方法

3、类 对象 类 对象

4、Observable Object

5、事件 事件监听者 事件监听者

6、饿汉式 懒汉式 私有(private)

**（二）**

1. 面向对象的七条设计原则包括：\_\_\_\_\_\_\_\_\_\_\_\_\_\_，开闭原则，\_\_\_\_\_\_\_\_\_\_\_，依赖倒转原则，\_\_\_\_\_\_\_\_\_\_\_\_\_，接口隔离原则以及\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 。其中外观模式是\_\_\_\_\_\_\_\_\_原则的典型应用。

2. 工厂模式中分为简单工厂,\_\_\_\_\_\_\_\_\_\_\_\_\_\_,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_三种模式。其中，可以应用平行等级结构完成创建工作的模式是\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_模式。

3. 适配器模式，可以细分为\_\_\_\_\_\_\_\_\_\_\_适配器和\_\_\_\_\_\_\_\_\_\_\_适配器两种实现。其中\_\_\_\_\_\_\_\_\_适配器采用的是继承复用，而\_\_\_\_\_\_\_\_\_\_\_适配器采用的是合成聚合复用。

4. Java API中，有两个与观察者模式相关的类和接口，分别是Observable和Observer，Observer的Update函数中用到的两个参数的参数类型是\_\_\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_\_\_\_\_。

5. 事件体系结构中的三个基本角色包括\_\_\_\_\_\_\_\_\_\_\_、\_\_\_\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。

6. 单例模式有两种方式实现，分别为\_\_\_\_\_\_\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。其中\_\_\_\_\_\_\_能够实现类被加载时就同时生成类的实例。

**答案：**

1、单一职责原则 里氏代换原则 合成聚合原则 迪米特法则 迪米特法则

2、工厂方法 抽象工厂 工厂方法

3、类 对象 类 对象

4、Observable Object

5、事件源 事件 事件监听者

6、饿汉式 懒汉式 饿汉式

**（三）**

1. 面向对象的七条设计原则包括：单一职责原则，\_\_\_\_\_\_\_\_\_\_\_\_\_，里氏代换原则，\_\_\_\_\_\_\_\_\_\_\_\_，合成聚合原则，接口隔离原则以及\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 。

2. 工厂模式中分为简单工厂,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_三种模式。

3. Java API中，有两个与观察者模式相关的类和接口，分别是\_\_\_\_\_\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_\_\_\_\_。

4. 单例模式有两种方式实现，分别称为\_\_\_\_\_\_\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。它们共同的特征是构造函数的访问修饰符必须是\_\_\_\_\_\_\_\_\_。

**答案：**

1. 开闭原则 、 依赖倒转原则 、 迪米特法则
2. 工厂方法 、 抽象工厂（方法）

3、Observable 、 Observer

4、懒汉式 、 饿汉式 、 private(私有的）

**（四）**

1. 面向对象的七条设计原则包括：\_\_\_\_\_\_\_\_\_\_\_\_\_\_，开闭原则，\_\_\_\_\_\_\_\_\_\_\_，依赖倒转原则，\_\_\_\_\_\_\_\_\_\_\_\_\_，接口隔离原则以及\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 。

2. 适配器模式，可以细分为\_\_\_\_\_\_\_\_\_\_\_适配器和\_\_\_\_\_\_\_\_\_\_\_适配器两种实现。其中\_\_\_\_\_\_\_\_\_适配器采用的是继承复用。

3. 能够定义对象间的一种“一对多”的依赖关系，当一个对象的状态改变，所有依赖于它的对象都能得到通知并自动更新的设计模式的名称是\_\_\_\_\_\_\_\_\_\_\_\_\_。

4. 单例模式中能够实现延时加载的方式，称为\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_,实现即时加载的方式称为\_\_\_\_\_\_\_\_\_\_\_\_\_\_。

**答案：**

1. 单一职责原则 、 里氏代换原则 、 合成聚合原则、迪米特法则
2. 类、对象、类

3. 观察者模式

4. 懒汉式 、 饿汉式

**（五）**

1. 面向对象的七条设计原则包括：单一职责原则，\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_，里氏代换原则，\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_，合成聚合原则，接口隔离原则以及\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 。

2. 工厂模式中分为简单工厂,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_三种模式。

3. 适配器模式，可以细分为\_\_\_\_\_\_\_\_\_\_\_适配器和\_\_\_\_\_\_\_\_\_\_\_适配器两种实现。

4. 单例模式有两种方式实现，分别称为\_\_\_\_\_\_\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。它们共同的特征是构造函数的访问修饰符必须是\_\_\_\_\_\_\_\_\_\_\_\_\_。

**答案：**

1. 开闭原则 、 依赖倒转原则 、 迪米特法则
2. 工厂方法 、 抽象工厂（方法）

3、类、对象

4、懒汉式 、 饿汉式 、 private(私有的）

1. **判断题**

**（一）**

1. 开闭原则的关键是抽象化。

2. 在软件开发中，如果要实现复用，应尽量较多使用继承，较少使用合成聚合的方式。

3. 当一个对象的行为取决于它所处的状态时，这时我们应该使用桥接模式

4. 适配器模式是一种创建型设计模式

5. MVC结构中模型和视图之间交互的实现可以基于观察者模式，其中模型是被观察者。

**答案：**1. √ 2. × 3. × 4. × 5. √

**（二）**

1. 设计模式的基本原理是面向实现编程。

2. 在软件开发中，如果要实现复用，应尽量较多使用继承，较少使用合成聚合的方式。

3. 当一个对象的行为取决于它所处的状态时，这时我们应该使用状态模式

4. 适配器模式是一种创建型设计模式

5. MVC结构中模型和视图之间交互的实现可以基于观察者模式，其中模型是被观察者。

**答案：**1. × 2. × 3. √ 4. × 5. √

**（三）**

1. 开闭原则的关键是抽象化。

2. 在软件开发中，如果要实现复用，应尽量较多使用继承，较少使用合成聚合的方式。

3. 当一个对象的行为取决于它所处的状态时，这时我们应该使用桥接模式。

4. 适配器模式是一种创建型设计模式。

5. 命令模式标准类图中的Receiver不是必须存在的。

**答案：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| **√** | **×** | **×** | **×** | **√** |

**（四）**

1. 设计模式的基本原理是面向实现编程。

2. 在软件开发中，如果要实现复用，应尽量较多使用继承，较少使用合成聚合的方式。

3. 当一个软件需要实现操作可撤销的功能时，我们应该使用命令模式。

4. 迭代器模式是一种结构型设计模式。

5. 抽象工厂方法完全符合开闭模式。

**答案：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| **×** | **×** | **√** | **×** | **×** |

**（五）**

1. 开闭原则的关键是抽象化。

2. 在软件开发中，如果要实现复用，应尽量较多使用继承，较少使用合成聚合的方式。

3. 当一个对象的行为取决于它所处的状态时，这时我们应该使用桥接模式。

4. 适配器模式是一种创建型设计模式。

5. 命令模式标准类图中的Receiver不是必须存在的。

**答案：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| **√** | **×** | **×** | **×** | **√** |

1. **简答题**

**(一)**

1. 假设系统中有三个类，分别为类A、类B和类C。在现有的设计中，让类A直接依赖类B，如果要将类A改为依赖类C，必须通过修改类A的代码来达成，请问这样的设计符合开闭原则吗（2分）？如果符合，请依据开闭原则进行解释，如果不符合请给出重构的方法（3分）。

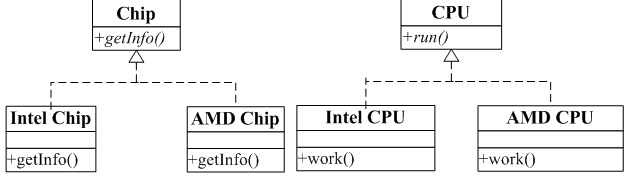
2. 假设某一软件系统中存在类A,B,C,D,E。请分别画出使用这些类的中介者模式类图及外观模式类图。

3. 假如系统中存在一组具有相同结构的产品类，如图所示，如果要创建具体产品对象，

应该使用哪一种创建模式？（2分）

根据给出的类图，绘制出产品等级和产品族的图示。（3分）

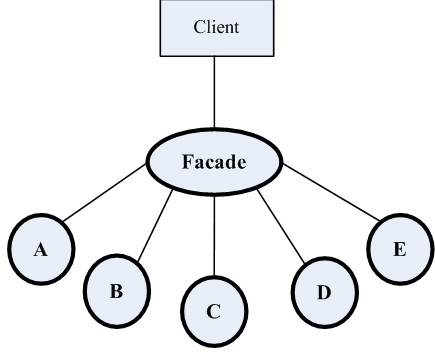
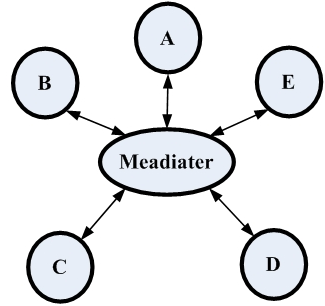
根据产品族图示，绘制工厂类的层次结构图（必须写清类的方法）。（4分）

答案：

目前的设计不符合开闭原则（2分）

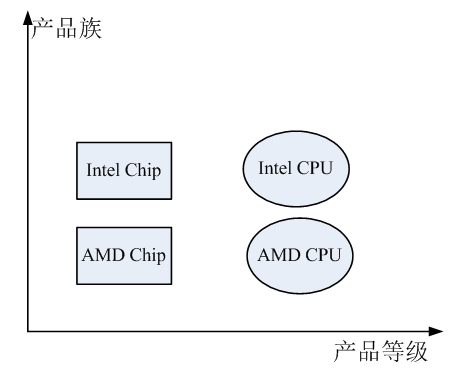
重构问题：新建接口I，使得类A依赖于接口I ,而类B和C实现接口I。（或相同概念的表述）（3分）

中介者模式类图 （4分） 外观模式类图（4分）

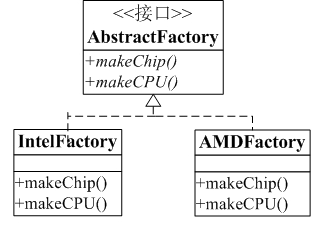


1）应该使用抽象工厂模式（2分）

2）产品等级和产品族的图示(3分）



工厂类图（4分,不对类名和方法名做限制，能够正确表达含义即可）



**（二）**

1. 简述开闭原则、依赖倒转原则的定义。

2. 简述策略模式的应用场景（3分），并绘制策略模式的类图(3分）。

3. 假如要用软件实现自动茶水销售机的功能，茶水的价格取决于茶的品种和杯子的大小。请回答以下问题，

1. 应该使用哪一种模式？（2分）
2. 请绘制该模式的类图。（4分）

**答案：**

1. 开闭原则：一个软件实体，应该对扩展开放，对修改关闭（3分）

依赖倒转原则：高层模块不应该依赖于低层模块，它们都应该依赖抽象。（3分）

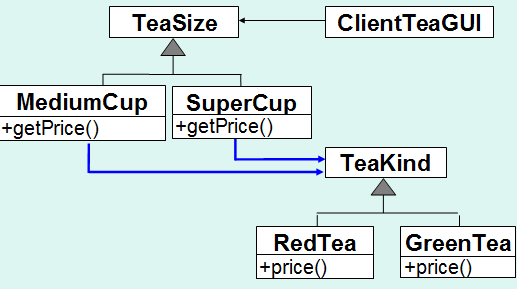
1. 1）策略模式的应用场景：当解决一个问题，有很多种方法或解决方案可以使用时，可以将每一种方法或解决方案封装成一个类，可以相互替换（或相同意义表达，3分).

2)类图（3分）



1. 1）应该使用桥接模式（2分）

2） 类图(4分,不对类名和方法名做限制，能够正确表达含义即可）



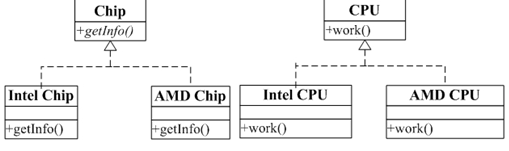
**（三）**

1. 简述依赖倒转原则的定义。

2. 假如系统中存在一组具有相同结构的产品类，如图所示，如果要创建具体产品对象，

1. 应该使用哪一种创建模式？（2分）
2. 根据给出的类图，绘制出产品等级和产品族的图示。（4分）

3） 根据产品族图示，绘制工厂类的层次结构图（必须写清类的方法）。（4分）

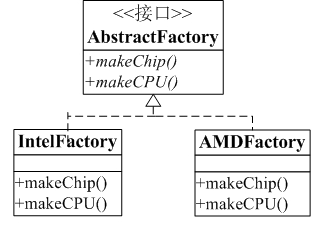


**答案：**1. 高层模块不应该依赖低层模块，它们都应该依赖抽象(3分）。抽象不应该依赖于细节，细节应该依赖于抽象（3分）。

2.1）应该使用抽象工厂模式（2分）

2）产品等级和产品族的图示(4分）

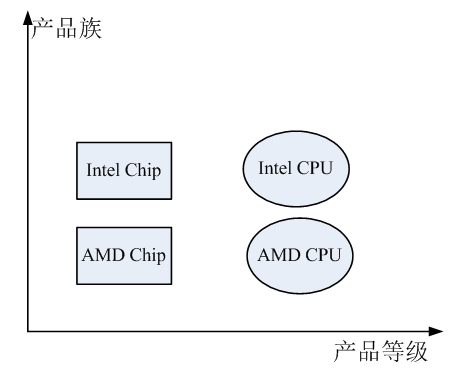
3) 类图（4分,不对类名和方法名做限制，能够正确表达含义即可）



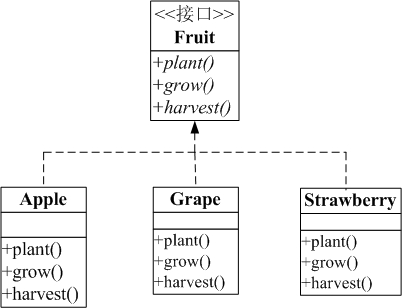
1. **程序设计题**

**（一）**

1. 现在需要开发一款游戏软件，请以单例模式来设计其中的Boss角色。角色的属性和动作可以任意设计。 要求：该Boss类可以在多线程中使用。（8分）



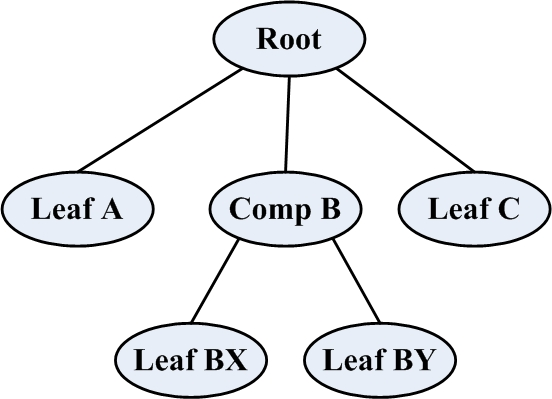
2. 一个农场公司，专门负责培育各种水果，有葡萄，草莓和苹果，请使用工厂方法，编写工厂类和主程序，并在主程序中来完成草莓生长状态的描述。（8分）



3. 给定如图所示的树形结构，请应用组合模式，在客户端完成数据的展示。具体要求如下：

绘制组合模式的类图。（4分）

编写简单元素和复杂元素的代码。（4分）

1. 在客户端构造出树形数据并输出。（4分）

**提示：程序运行后，输出信息应为**

**Root**

**Leaf A**

**Comp B**

**Leaf BX**

**Leaf BY**

**Leaf C**

**答案：**

1. Public class Boss{

Private static Boss instance; //(2分）

Private Boss(){ }//(2分）

Public static Boss getInstance(){ //（2分）

If(instance == null){

Synchronized(Boss.Class){ //（synchronized关键字，2分）

If(instance == null)

Instance = new Boss();

}

}

return instance;

}

}或者

Public class Boss{

Private static Boss instance = new Boss();//(4分）

Private Boss(){}//（2分）

Public static Boss getInstance(){//2分

Return instance;

}

}

2.

Public interface Factory{

Fruit build();

}

Public class AppleFactory implements Factory{

Public Fruit build(){

Return new Apple();

}

}

Public class GrapeFactory implements Factory{

Public Fruit build(){

Return new Grape();

}

}

Public class StrawberryFactory implements Factory{

Public Fruit build(){

Return new Strawberry();

}

}

Public class MainUI{

Public static void main(string[] str){

Factory fac = new StrawberryFactory();

Fruit ft = fac.build();

ft.plant();

Ft.grow();

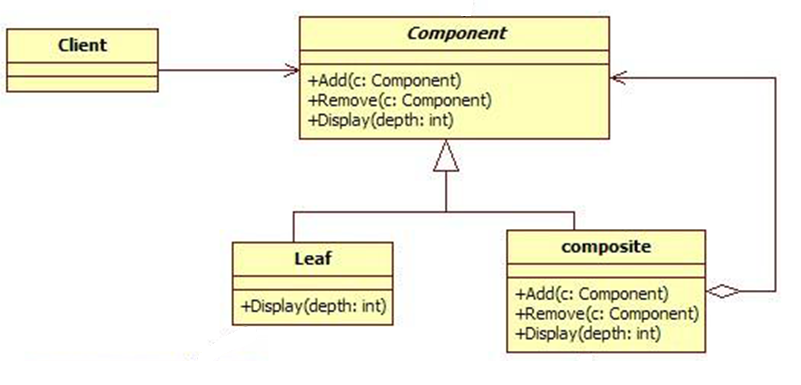
Ft.harvest();

}

}

3.

1)类图，类名不限，但必须将抽象的概念，以及Composite和Component之间的关系用正确的连线表示。（4分）



1. 简单元素、复杂元素（4分）：

class Leaf implements Component{

String name;

public Leaf(String name){this.name = name; }

public void display(){

System.out.println( name);

}

public void add(Component c){}

public void remove(Component c){}

}

class Composite implements Component{

String name;

ArrayList<Component> list = new ArrayList<Component>();

public Composite(String name){

this.name = name;

}

public void display(){

System.out.println(name);

for(int i = 0 ; i < list.size() ;i ++)

{

list.get(i).display();

}

}

public void add(Component c){list.add(c); }

public void remove(Component c){list.remove(c);}

}

客户端（4分）：

public class Test {

public static void main(String[] args){

Component root = new Composite("Root");

root.add(new Leaf("Leaf A"));

Component comp = new Composite("Comp B");

root.add(comp);

comp.add(new Leaf("Leaf BX"));

comp.add(new Leaf("Leaf BY"));

root.add(new Leaf("Leaf C"));

root.display();

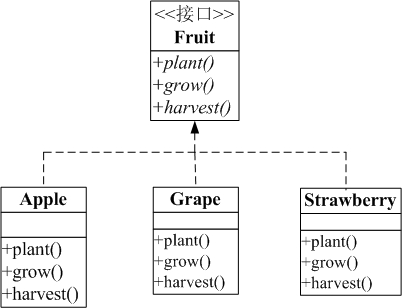
}

　　}

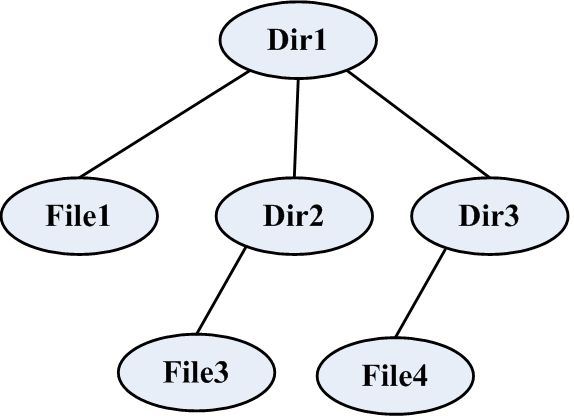
**（二）**

1. 现在需要开发一款打印机管理软件，请以单例模式来设计其中的打印池PrintSpooler。要求：该PrintSpooler类可以在多线程中使用。（10分）

2. 一个农场公司，专门负责培育各种水果，有葡萄，草莓和苹果，请使用简单工厂模式，编写**简单工厂类和主程序**，并在主程序中来完成苹果生长状态的描述。（10分）



3. 给定如图所示的树形结构，请应用组合模式，在客户端完成数据的展示。具体要求如下：

1. 绘制组合模式的类图。（4分）
2. 编写简单元素和复杂元素的代码。（4分）
3. 在客户端构造出树形数据并输出。（4分）

**提示：程序运行后，输出信息应为**

**Dir1**

**File1**

**Dir2**

**File3**

**Dir3**

**File4**

**答案：**

1. Public class PrintSpooler{

Private static PrintSpooler instance; //(2分）

Private PrintSpooler(){} //(2分）

Public static PrintSpooler getInstance(){ //（2分）

If(instance == null){

Synchronized(PrintSpooler.Class){ //（synchronized关键字，2分）

If(instance == null)

Instance = new PrintSpooler();//2分

}

}

return instance;

}

}或者

1. Public class PrintSpooler{

Private static PrintSpooler instance = new PrintSpooler(); //(4分）

Private PrintSpooler(){ //(2分）

}

Public static PrintSpooler getInstance(){ //（2分）

return instance; //(2分）

}

}

2.

Public class FruitFactory{ //6分，要有静态方法，返回fruit

Public static Fruit creatFruit(String type){

Fruit ft = null ;

If(type.equals(“Apple”)

Ft = new Apple();

Else if(type.equals(“Strawberry”)

Ft = new Strawberry();

Else if(type.equals(“Grape”)

Ft = new Grape();

Return ft;

}

}

Public class MainUI{ //(4分）

Public static void main(string[] str){

Fruit ft = FruitFactory.creatFruit(“Apple”);

ft.plant();

Ft.grow();

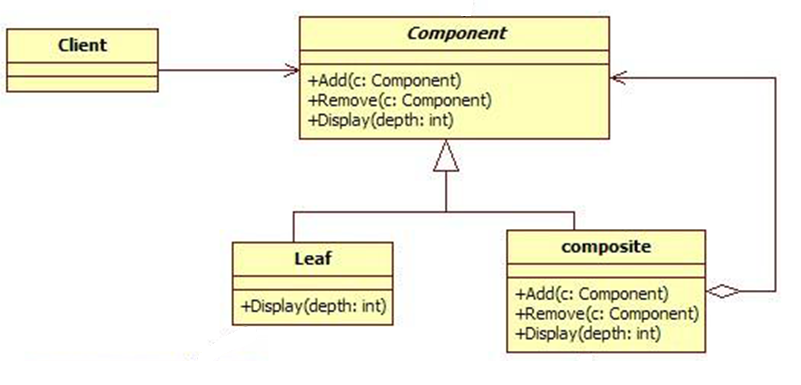
Ft.harvest();

}

}

3.

1)类图，类名不限，但必须将抽象的概念，以及Composite和Component之间的关系用正确的连线表示。



1. 简单元素、复杂元素（4分）：

class Leaf implements Component{

String name;

public Leaf(String name){

this.name = name;

}

public void display(){

System.out.println(name);

}

public void add(Component c){}

public void remove(Component c){}

}

class Composite implements Component{

String name;

ArrayList<Component> list = new ArrayList<Component>();

public Composite(String name){

this.name = name;

}

public void display(){

System.out.println(name);

for(int i = 0 ; i < list.size() ;i ++)

{

list.get(i).display();

}

}

public void add(Component c){

list.add(c);

}

public void remove(Component c){

list.remove(c);

}

}

1. 客户端（4分）：

public class Test {

public static void main(String[] args){

Component root = new Composite("Dir1");

root.add(new Leaf("File1"));

Component comp = new Composite("Dir2");

root.add(comp);

comp.add(new Leaf("File3"));

Component comp = new Composite("Dir3");

root.add(comp);

comp.add(new Leaf("File4"));

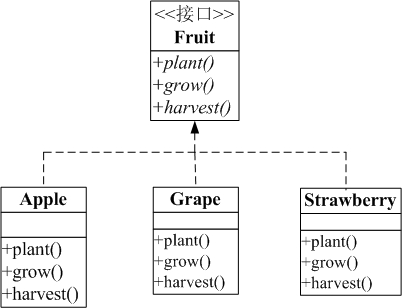
root.display();

}

　　}

**（三）**

1. 一个农场公司，专门负责培育各种水果，有葡萄，草莓和苹果，请使用简单工厂模式，编写简单工厂类和主程序，并在主程序中来完成苹果生长状态的描述。（8分）

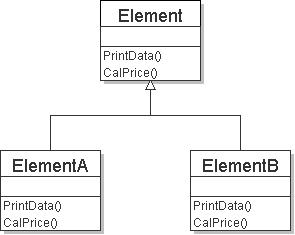


2.下图是某系统的数据部分的类图。因为该层次结构中的操作需要经常变化，所以需要用访问者模式对其进行重构，请按以下要求完成题目：

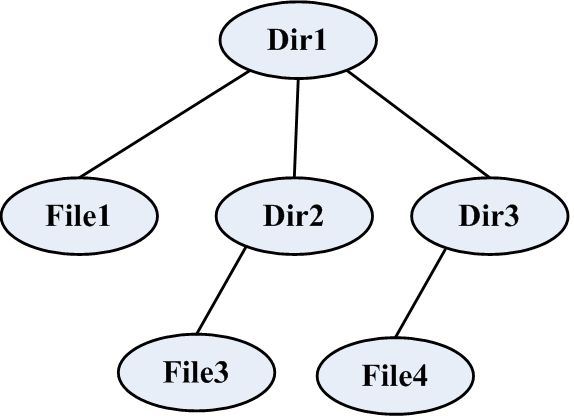
（1）绘制重构后系统完整类图。（4分）

（2）给出重构后ElementA类的代码。（4分）

（3）在客户端运用访问者模式，对ElementA的实例，完成CalPrice操作。（2分）



3. 给定如图所示的树形结构，请应用组合模式，在客户端完成数据的展示。具体要求如下：

绘制组合模式的类图。（4分）

1. 编写简单元素的代码。（2分）
2. 编写复杂元素的代码。（6分）
3. 编写客户端的代码（4分）

**提示：程序运行后，输出信息应为**

**Dir1**

**File1**

**Dir2**

**File3**

**Dir3**

**File4**

**答案：**

1. Public class FruitFactory{ //4分，要有静态方法，返回fruit

Public static Fruit creatFruit(String type){

Fruit ft = null ;

If(type.equals(“Apple”)

Ft = new Apple();

Else if(type.equals(“Strawberry”)

Ft = new Strawberry();

Else if(type.equals(“Grape”)

Ft = new Grape();

Return ft;

}

}

Public class MainUI{ //(4分）

Public static void main(string[] str){

Fruit ft = FruitFactory.creatFruit(“Apple”);

ft.plant();

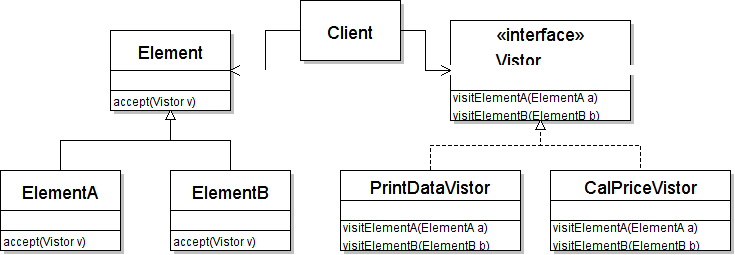
Ft.grow();

Ft.harvest();

}

}

1)类图为(4分）



1. class ElementA extends Element{//4分

Public void accept(Visitor v){

V.accept(this)

}

}

1. 客户端 2分

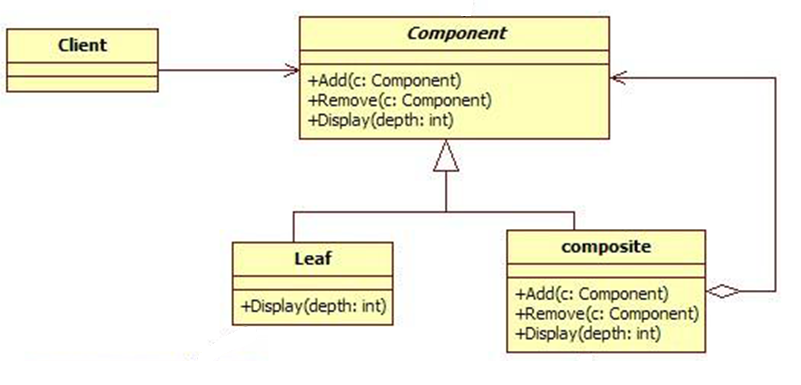
Visitor v = new CalPriceVistor();

ElementA a = new ElementA();

A.accept(v);

3.

1)类图，类名不限，但必须将抽象的概念，以及Composite和Component之间的关系用正确的连线表示。



2)简单元素（2分）：

class Leaf implements Component{

String name;

public Leaf(String name){

this.name = name;

}

public void display(){

System.out.println(name);

}

}

1. 复杂元素（6分）

class Composite implements Component{

String name;

ArrayList<Component> list = new ArrayList<Component>();

public Composite(String name){

this.name = name;

}

public void display(){

System.out.println(name);

for(int i = 0 ; i < list.size() ;i ++)

{

list.get(i).display();

}

}

public void add(Component c){

list.add(c);

}

public void remove(Component c){

list.remove(c);

}

}

4）客户端（4分）：

public class Test {

public static void main(String[] args){

Component root = new Composite("Dir1");

root.add(new Leaf("File1"));

Component comp = new Composite("Dir2");

root.add(comp);

comp.add(new Leaf("File3"));

Component comp = new Composite("Dir3");

root.add(comp);

comp.add(new Leaf("File4"));

root.display();

}

　　}

1. **详情知识点**

**面向对象设计原则：**

1. **单一职责原则:** 不要存在多于一个导致类变更的原因。通俗的说，即一个类只负责一项职责。
2. **开闭原则（重构）（抽象）:** 一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。
3. **里氏代换原则:**

**定义1：**如果对每一个类型为 T1的对象 o1，都有类型为 T2 的对象o2，使得以 T1定义的所有程序 P 在所有的对象 o1 都代换成 o2 时，程序 P 的行为没有发生变化，那么类型 T2 是类型 T1 的子类型。

**定义2：**所有引用基类的地方必须能透明地使用其子类的对象。

1. **依赖倒转原则:** 高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。
2. **迪米特法则（重构、转发、调用）:** 迪米特法则（Law of Demeter）又叫作最少知识原则（Least Knowledge Principle 简写LKP），就是说一个对象应当对其他对象有尽可能少的了解,不和陌生人说话。
3. **接口隔离原则:** 客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。
4. **合成、聚合、复用原则:**

**1、继承复用**

继承复用通过扩展一个已有对象的实现来得到新的功能，基类明显地捕获共同的属性和方法，而子类通过增加新的属性和方法来扩展超类的实现。继承是类型的复用。

**继承复用的优点：**

新的实现较为容易，因为超类的大部分功能可通过继承关系自动进入子类；

修改或扩展继承而来的实现较为容易。

继承复用的缺点：

继承复用破坏封装，因为继承将超类的实现细节暴露给子类。“白箱”复用；

如果超类的实现发生改变，那么子类的实现也不得不发生改变。

从超类继承而来的实现是静态的，不可能再运行时间内发生改变，因此没有足够的灵活性。

**2、合成/聚合复用**

由于合成/聚合可以将已有的对象纳入到新对象中，使之成为新对象的一部分，因此新的对象可以调用已有对象的功能，

**其优点在于：**

新对象存取成分对象的唯一方法是通过成分对象的接口；

成分对象的内部细节对新对象不可见。 “黑箱”复用；

该复用支持封装。

该复用所需的依赖较少。

每一个新的类可将焦点集中在一个任务上。

该复用可在运行时间内动态进行，新对象可动态引用于成分对象类型相同的对象。

**缺点：**

通过这种复用建造的系统会有较多的对象需要管理。

为了能将多个不同的对象作为组合块（composition block）来使用，必须仔细地对接口进行定义。

1. **设计模式分类：**

**创建型模式：**

工厂模式

简单工厂

工厂方法

抽象工厂（等级结构和产品族）

单例模式

原型模式（不要求代码）Cloneable（继承）🡪继承Clone（）方法

序列化（不要求代码）🡪Serializable锁

结构化设计模式

组合模式

适配器模式

装饰者模式

桥接模式

代理模式（不写代码）

**行为性设计模式**

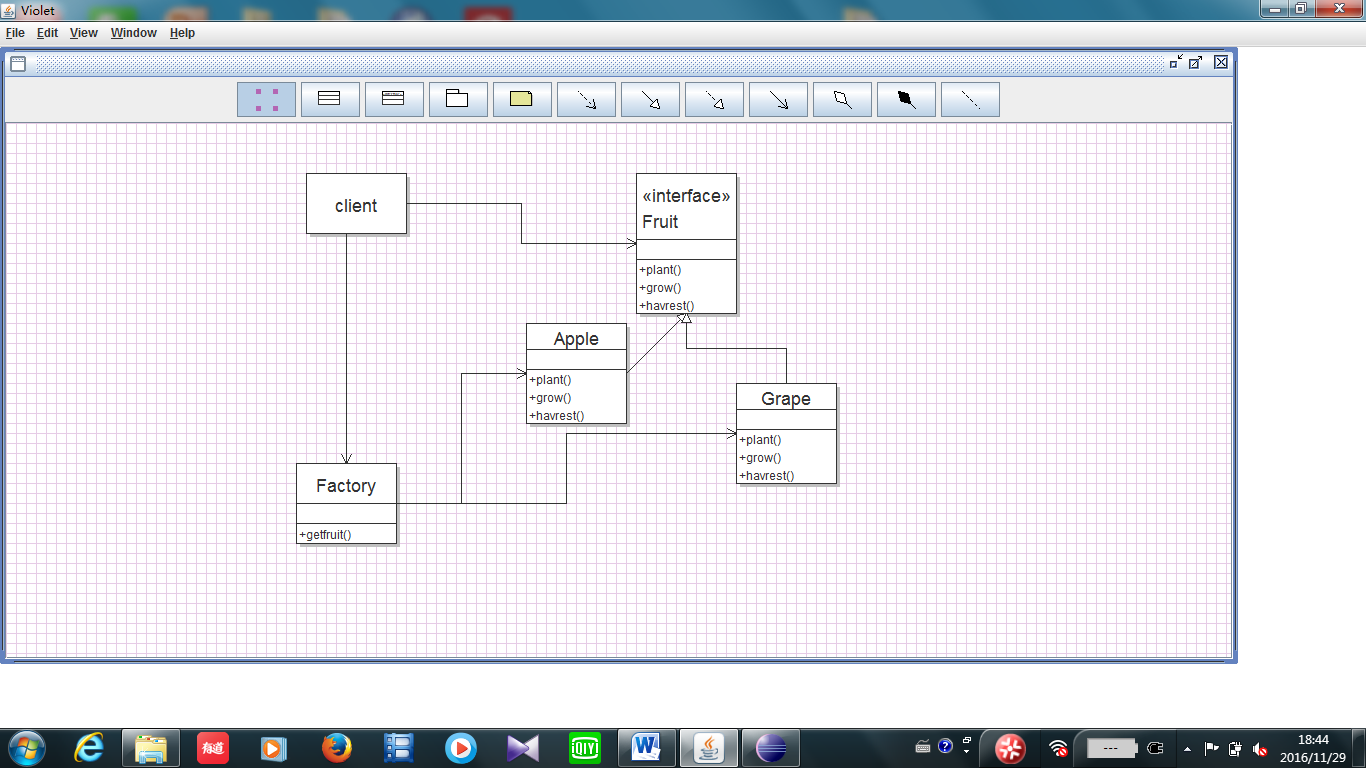
迭代器模式

访问者模式（重构代码）

状态模式 定义（电灯开关）

观察者模式 定义

**简单工厂**



Fruit接口

**public** **interface** Fruit {

**public** **void** plant();

**public** **void** grow();

**public** **void** havrest();}

Apple类

**public** **class** Apple **implements** Fruit {

**public** **void** plant() { System.*out*.println("苹果正在发芽"); }

**public** **void** grow() { System.*out*.println("苹果正在生长"); }

**public** **void** havrest() { System.*out*.println("苹果已经成熟"); }}

Grape类

**public** **class** Grape **implements** Fruit {

**public** **void** plant() { System.*out*.println("葡萄正在发芽"); }

**public** **void** grow() { System.*out*.println("葡萄正在生长"); }

**public** **void** havrest() { System.*out*.println("葡萄成熟"); }}

FruitFactory方法

**public** **class** FruitFactory {

**public** **static** Fruit getFruit(String str){

Fruit fruit=**null**;

**if**(str.equals("apple")){ **return** **new** Apple();

}**else** **if**( str.equals("grape")){

**return** **new** Grape(); }**else**

{ System.*out*.print("没有你要的水果。"); }

**return** fruit; }}

主函数

**public** **class** Cliet {

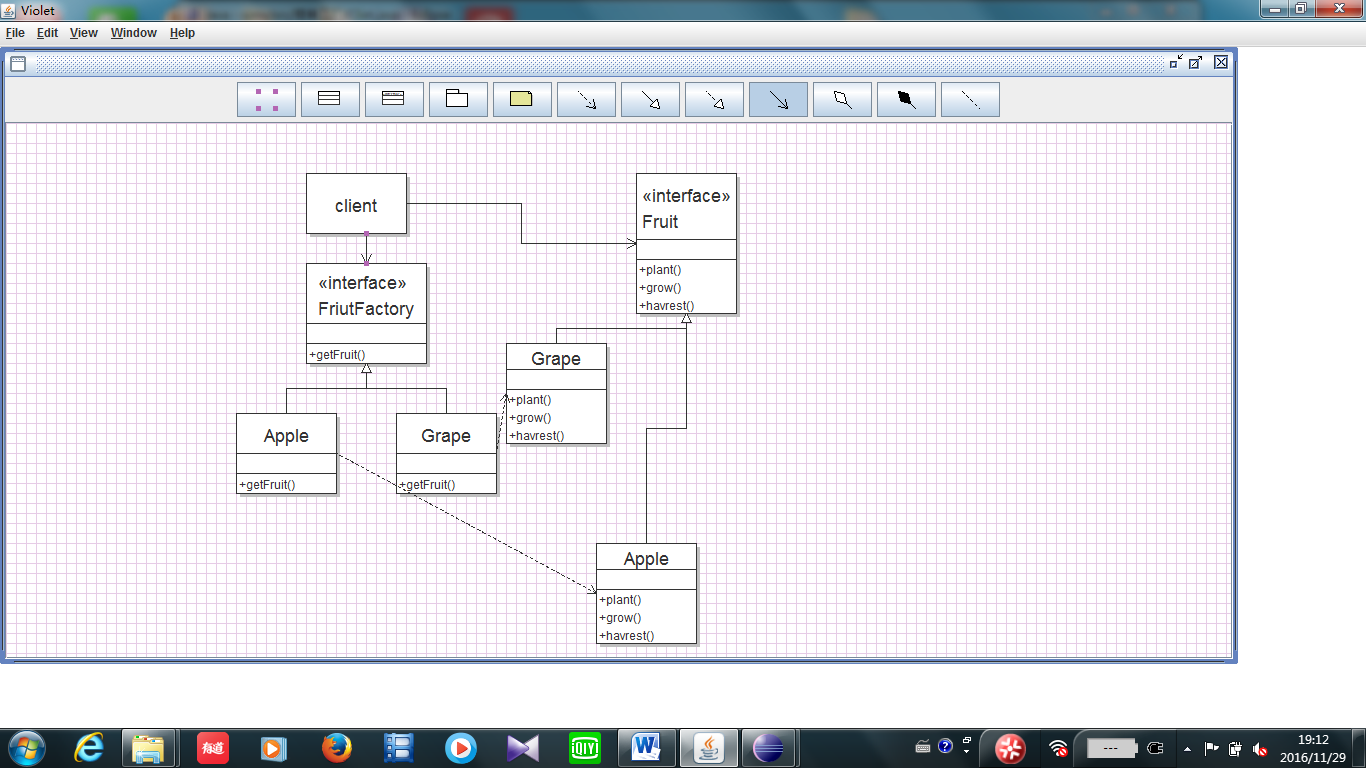
**public** **static** **void** main(String[] args) {

Fruit f=**null**;

f=FruitFactory.*getFruit*("apple");

f.grow(); f.plant(); f.havrest(); }}

**工厂方法**



Fruit接口

**public** **interface** Fruit {

**public** **void** plant();

**public** **void** grow();

**public** **void** havrest();}

Apple类

**public** **class** Apple **implements** Fruit {

**public** **void** plant() { System.*out*.println("苹果正在发芽"); }

**public** **void** grow() { System.*out*.println("苹果正在生长"); }

**public** **void** havrest() { System.*out*.println("苹果已经成熟"); }}

Grape类

**public** **class** Grape **implements** Fruit {

**public** **void** plant() { System.*out*.println("葡萄正在发芽"); }

**public** **void** grow() { System.*out*.println("葡萄正在生长"); }

**public** **void** havrest() { System.*out*.println("葡萄成熟"); }}

FruitFactory接口

**public** **interface** FruitFactory { **public** Fruit getFruit();}

AppleFactory类

**public** **class** AppleFactory **implements** FruitFactory{

**public** Fruit getFruit() { **return** **new** Apple(); }}

GrapeFactory类

**public** **class** GrapeFactory **implements** FruitFactory {

**public** Fruit getFruit() { **return** **new** Grape(); }}

主函数

**public** **class** Cliet {

**public** **static** **void** main(String[] args) {

FruitFactory fruitfactory = **null**;

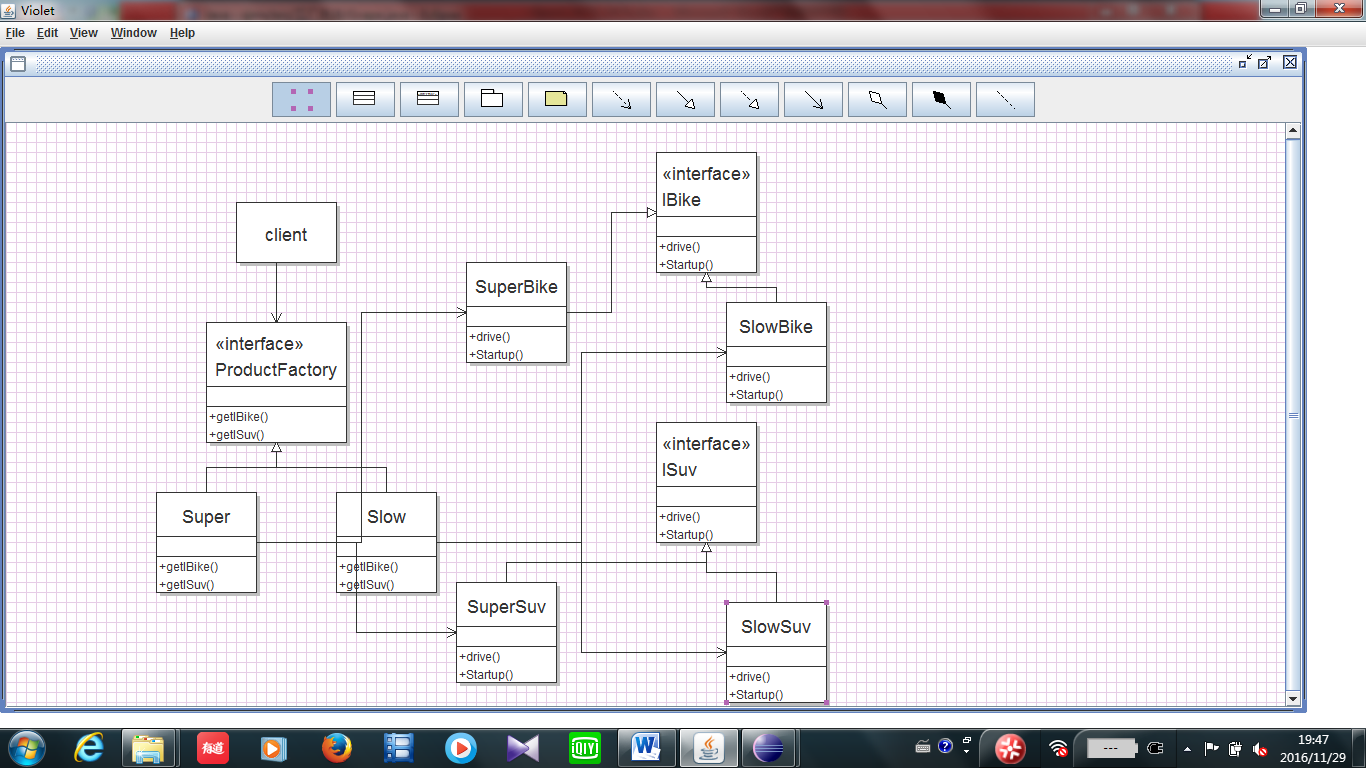
fruitfactory=**new** AppleFactory();

fruitfactory.getFruit().plant();

fruitfactory.getFruit().grow();

fruitfactory.getFruit().havrest(); }}

**抽象工厂**



IBike接口

**public** **interface** IBicylce {

**public** **void** ignition();

**public** **void** startup();}

ISuv接口

**public** **interface** ISUV {

**void** ignition();

**void** startup();}

SuperBike类

**public** **class** SuperBicycle **implements** IBicylce {

**public** **void** ignition() {

System.*out*.println("SuperBicycle is on ignition"); }

**public** **void** startup() {

System.*out*.println("SuperBicycle is startup"); }}

SuperSuv类

**public** **class** SuperSUV **implements** ISUV {

**public** **void** ignition() {

System.*out*.println("SuperSUV is on ignition"); }

**public** **void** startup() {

System.*out*.println("SuperSUV is startup"); }}

SlowBike类

**public** **class** SlowBicycle **implements** IBicylce {

**public** **void** ignition() {

System.*out*.println("SlowBicycle is on ignition"); }

**public** **void** startup() {

System.*out*.println("SlowBicycle is startup"); }}

SlowSuv类

**public** **class** SlowSUV **implements** ISUV {

**public** **void** ignition() {

System.*out*.println("SlowSUV is on ignition"); }

**public** **void** startup() {

System.*out*.println("SlowSUV is startup"); }}

ProductFactory抽象类

**public** **abstract** **class** ProductFactory {

**public** **abstract** ISUV getIsuv();

**public** **abstract** IBicylce getIBicylce();

**public** **static** ProductFactory getProductFactory(String type){

ProductFactory pf=**null**;

**if**(type.equals("super")){ pf=**new** SuperFactory();

}**else** **if** ( type.equals("slow")) {

pf=**new** SlowFactory(); }

**return** pf;}}

Super类

**public** **class** SuperFactory **extends** ProductFactory{

**public** ISUV getIsuv() {

**return** **new** SuperSUV();

} **public** IBicylce getIBicylce() {

**return** **new** SuperBicycle(); }}

Slow类

**public** **class** SlowFactory **extends** ProductFactory{

**public** ISUV getIsuv() {

**return** **new** SlowSUV(); }

**public** IBicylce getIBicylce() {

**return** **new** SlowBicycle(); }}

Client类

**public** **class** Client {

**public** **static** **void** main(String[] args) {

ProductFactory pf=**null**;

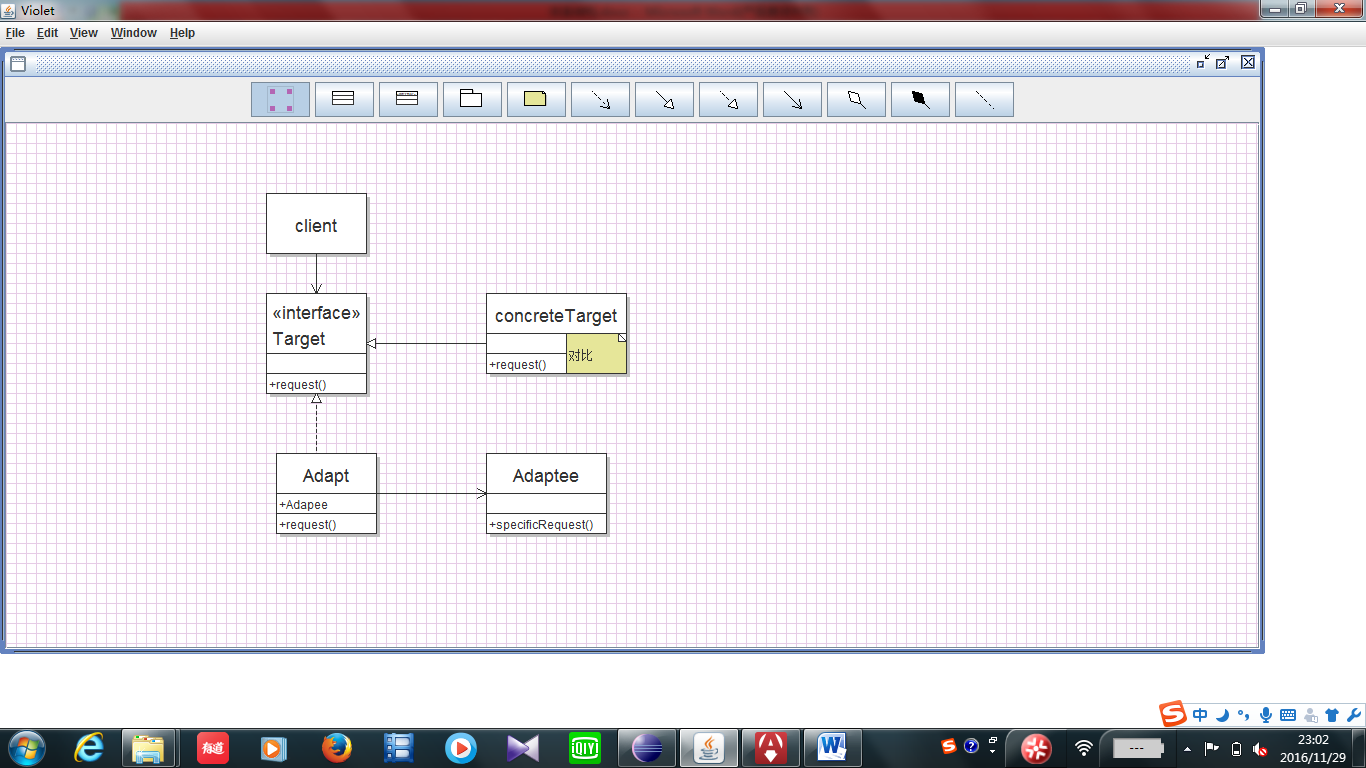
pf=**new** SuperFactory();

pf.*getProductFactory*("slow").getIsuv().ignition();

pf.*getProductFactory*("super").getIsuv().startup(); }}

**适配器模式：**将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的可以一起工作。

**对象适配器：**聚合、合成、复用



Target接口

**public** **interface** Target {

**public** **void** request();}

concreteTarget类

**public** **class** ConcreteTarget **implements** Target{

**public** **void** request() {

System.*out*.println("普通类 具有 普通功能..."); }}

Adaptee类

**public** **class** Adaptee {

**public** **void** specificRequest(){

System.*out*.println("被适配类具有 特殊功能...");}}

Adapter类

**public** **class** Adapter **implements** Target{

**private** Adaptee adaptee;

**public** Adapter(Adaptee adaptee){

**this**.adaptee=adaptee;}

**public** **void** request() {

**this**.adaptee.specificRequest(); }}

Client主类

**public** **class** Client {

**public** **static** **void** main(String arg[]){

Target target=**null**;

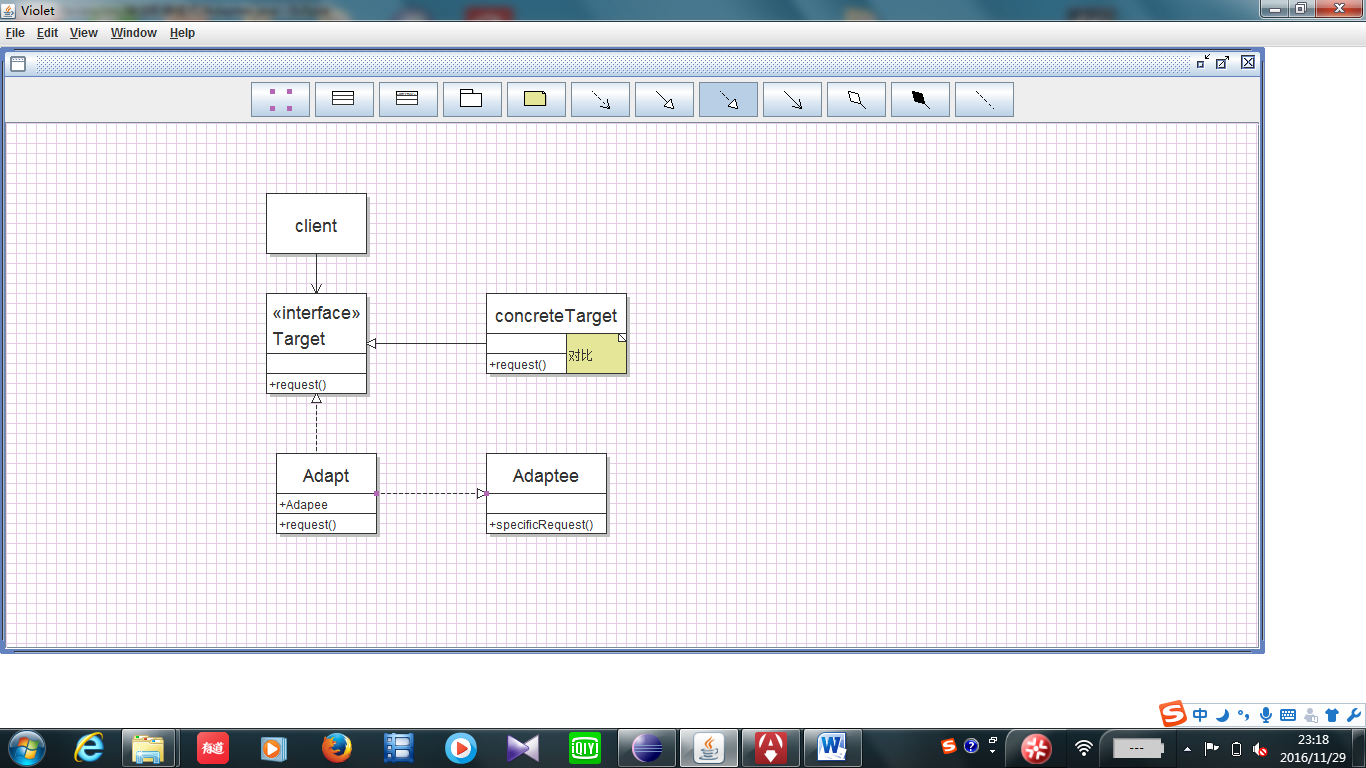
target=**new** ConcreteTarget();

target.request();

target=**new** Adapter(**new** Adaptee());

target.request();}}

**类适配器：**继承、复用



Target接口

**public** **interface** Target {

**public** **void** request();}

ConcreteTarget类

**public** **class** ConcreteTarget **implements** Target{

**public** **void** request() {

System.*out*.println("普通类 具有 普通功能..."); }}

Adaptee类

**public** **class** Adaptee {

**public** **void** specificRequest(){

System.*out*.println("被适配类具有 特殊功能...");}}

Adapt类

**public** **class** Adapter **extends** Adaptee **implements** Target{

**public** **void** request() {

**super**.specificRequest(); }}

主类

**public** **class** Client {

**public** **static** **void** main(String[] args) {

Target t=**null**;

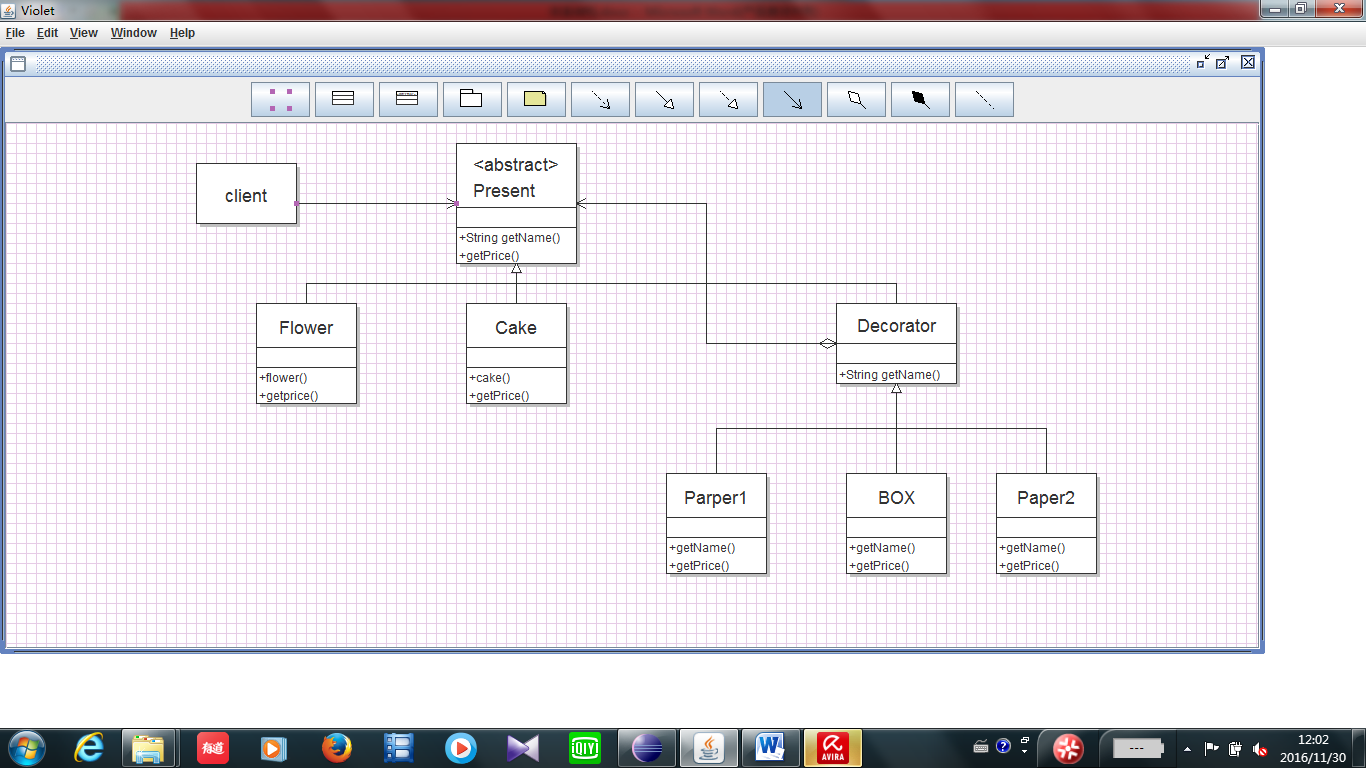
t=**new** ConcreteTarget();

t.request();

t=**new** Adapter();

t.request(); }}

**装饰者模式：**动态地给一个对象添加一些额外的职责，就增加功能来说；装饰者模式比继承更灵活；主要特征：对象层层包裹。



Present 抽象类

**public** **abstract** **class** Present {

**protected** String name;

**public** String getName(){

**return** name;}

**public** **abstract** **double** getPrice();}

Flower类

**public** **class** Flower **extends** Present {

**public** Flower(){ name="鲜花";}

**public** **double** getPrice() {

**return** 5; }}

Cake类

**public** **class** Cake **extends** Present {

**public** Cake(){ name="蛋糕";}

**public** **double** getPrice() { **return** 10; }}

Decorator包装类

**public** **abstract** **class** Decorator **extends** Present {

**public** **abstract** String getName();}

Paper1包装类

**public** **class** Paper1 **extends** Decorator {

Present present=**null**;

**public** Paper1(Present present){

**this**.present=present;}

**public** String getName() {

**return** present.getName()+"加上Paper1包装"; }

**public** **double** getPrice() {

**return** present.getPrice()+4; }}

Paper2包装类

**public** **class** Paper2 **extends** Decorator {

Present present=**null**;

**public** Paper2(Present present){

**this**.present=present;

}

**public** String getName() {

**return** present.getName()+"用Paper2包装";

}

**public** **double** getPrice() {

**return** present.getPrice()+8;

}

}

Box包装类

**public** **class** Box **extends** Decorator {

Present present=**null**;

**public** Box(Present present){

**this**.present=present;}

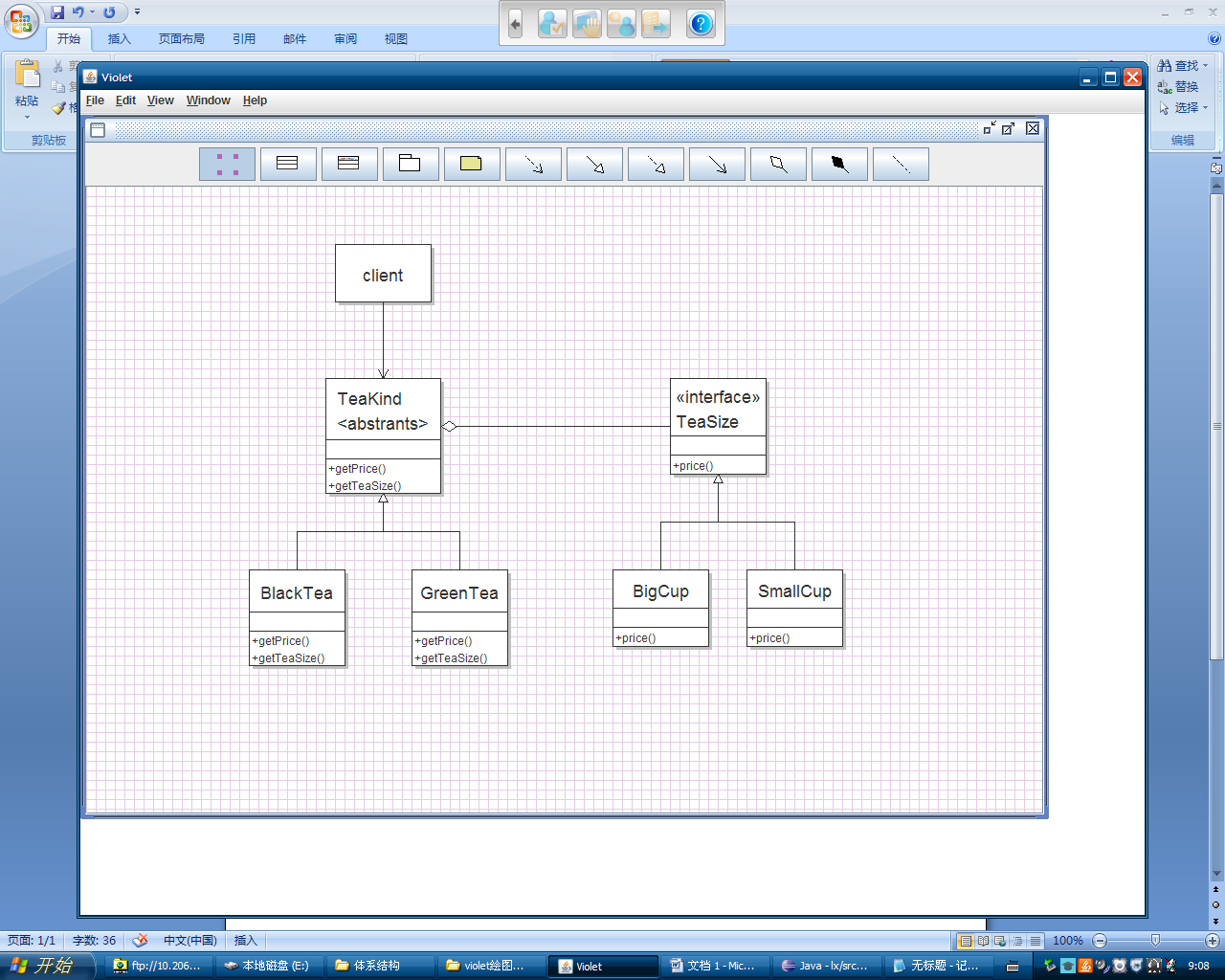
**public** String getName() {

**return** present.getName()+"用BOX包装"; }

**public** **double** getPrice() {

**return** present.getPrice()+10; }}

**桥接模式：**将抽象部分与它的实现部分分离，使它们都可以变化。



TeaSize接口

**public** **abstract** **class** Teakind {

TeaSize ts;

**public** **abstract** **double** getprice();

**public** **void** setTeaSize(TeaSize ts){

**this**.ts=ts; }}

TeaKind类

**public** **interface** TeaSize {

**public** **double** price();}

BlackTea类

**public** **class** BlackTea **extends** Teakind {

**public** **double** getprice() {

**return** 3\*ts.price(); }}

GreenTea类

**public** **class** GreenTea **extends** Teakind {

**public** **double** getprice() {

**return** 4\*ts.price(); }}

BigCup类

**public** **class** BigCup **implements** TeaSize {

**public** **double** price() { **return** 4; }}

SmallCup类

**public** **class** SmallCup **implements** TeaSize {

**public** **double** price() { **return** 2; }}

Client主类

**public** **class** Client {

**public** **static** **void** main(String[] args) {

Teakind tk=**null**;

TeaSize ts=**null**;

tk=**new** BlackTea();

ts=**new** BigCup();

tk.setTeaSize(ts);

**double** price=tk.getprice();

System.*out*.println("中杯红茶的价格是"+price);

tk=**new** GreenTea();

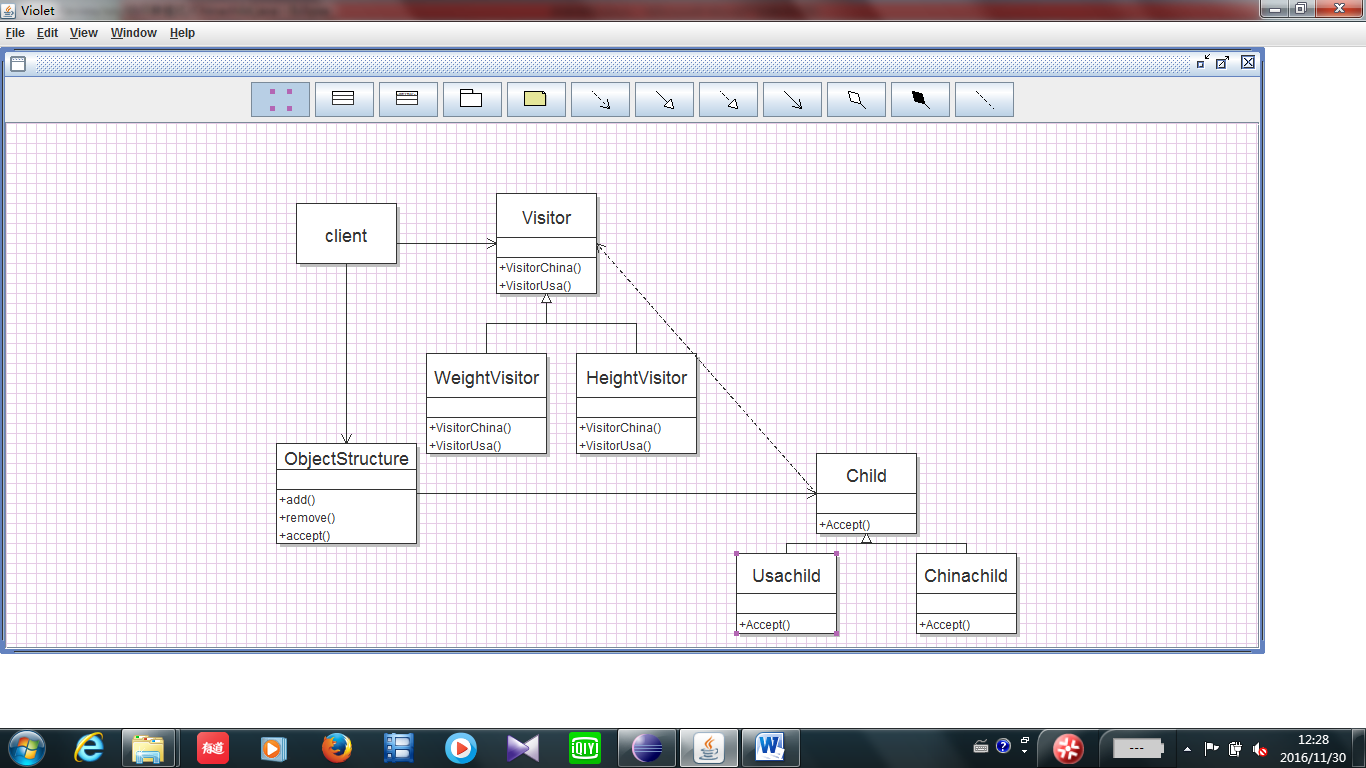
ts=**new** BigCup();

tk.setTeaSize(ts);

**double** price1=tk.getprice();

System.*out*.println("大杯绿茶的价格是"+price1); }}

**访问者模式：**作用于某对象结构中的各元素的操作，它使我们可以在不变各元素的类的前提下定义，这些元素的新操作



Child类

**public** **abstract** **class** Child {

**public** **abstract** **void** Accept(Visitor visitor);

}

Usachild类

**public** **class** USAchild **extends** Child{

**public** **void** Accept(Visitor visitor) {

visitor.visitUSA(**this**); }}

Chinachild类

**public** **class** Chinachild **extends** Child{

**public** **void** Accept(Visitor visitor) {

visitor.visitChina(**this**); }}

Vistor类

**public** **abstract** **class** Visitor {

**public** **abstract** **void** visitChina(Chinachild c);

**public** **abstract** **void** visitUSA(USAchild u);}

Weightvistor类

**public** **class** WeightVisitor **extends** Visitor{

**public** **void** visitChina(Chinachild c) {

System.*out*.println("中国孩子体重标准是60kg"); }

**public** **void** visitUSA(USAchild u) {

System.*out*.println("美国孩子体重标准是70kg"); }}

Heightvistor类

**public** **class** HeightVisitor **extends** Visitor{

**public** **void** visitChina(Chinachild c) {

System.*out*.println("中国孩子身高标准是175cm"); }

**public** **void** visitUSA(USAchild u) {

System.*out*.println("美国孩子身高标准是180cm"); }}

ObjectStructure类

**public** **class** ObjectStructure {

**private** ArrayList<Child> list=**new** ArrayList<Child>();

**public** **void** add(Child c){

list.add(c);}

**public** **void** remove(Child c){

list.remove(c);}

**public** **void** accept(Visitor v){

**for**(Child c:list){

c.Accept(v); }}}

主类

**public** **class** Client {

**public** **static** **void** main(String[] args) {

ObjectStructure o=**new** ObjectStructure();

o.add(**new** Chinachild());

o.add(**new** USAchild());

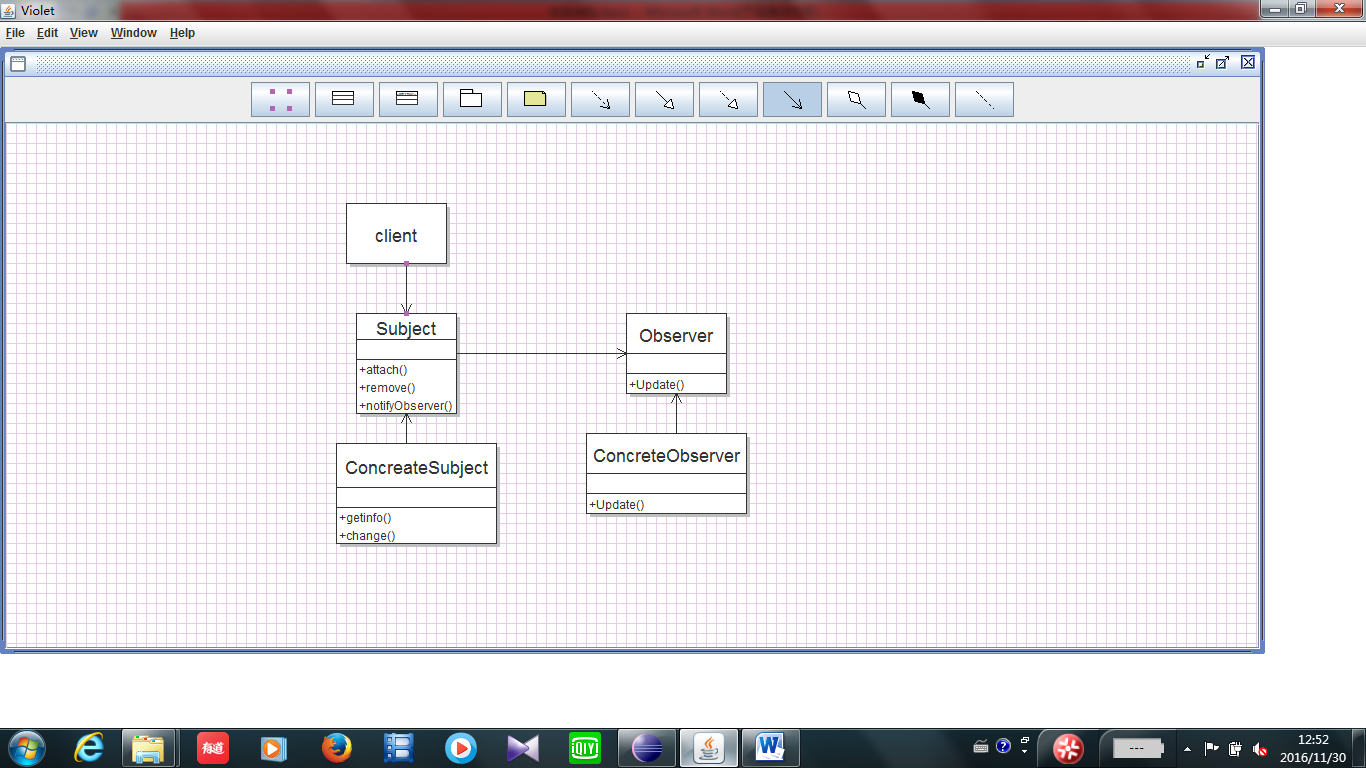
HeightVisitor h=**new** HeightVisitor();

WeightVisitor w=**new** WeightVisitor();

o.accept(h);

o.accept(w); }}

**观察者模式：**对象间的一种1对n的依赖关系，当一个对象的状态发生变化时，所有依赖于它的随想得到通知被更新。



Observer类（观察者）

**public** **interface** Observer {

**public** **void** update(Subject obj);}

Subject类（被观察者）

**public** **abstract** **class** Subject {

**private** ArrayList<Observer> list=**new** ArrayList<Observer>();

**public** **void** attach(Observer o){

list.add(o);

System.*out*.println("添加新的观察者");}

**public** **void** remove(Observer o){

list.remove(o); }

**public** **void** notifyObserver(){

**for**(Observer observer:list){

observer.update(**this**); }}}

ConcreteObserver类

**public** **class** Concretorobersver **implements** Observer {

String info;

**public** **void** update(Subject obj) {

ConcretorSubject cs=**new** ConcretorSubject();

info=cs.getinfo();

System.*out*.println("状态为:"+info); }}

ConcreteSubject类

**public** **class** ConcretorSubject **extends** Subject {

String info;

**public** String getinfo(){

**return** info;}

**public** **void** change(String state){

info=state;

System.*out*.println("改变状态为:"+info);

**this**.notifyObserver();}}

主类

**public** **class** Client {

**public** **static** **void** main(String[] args) {

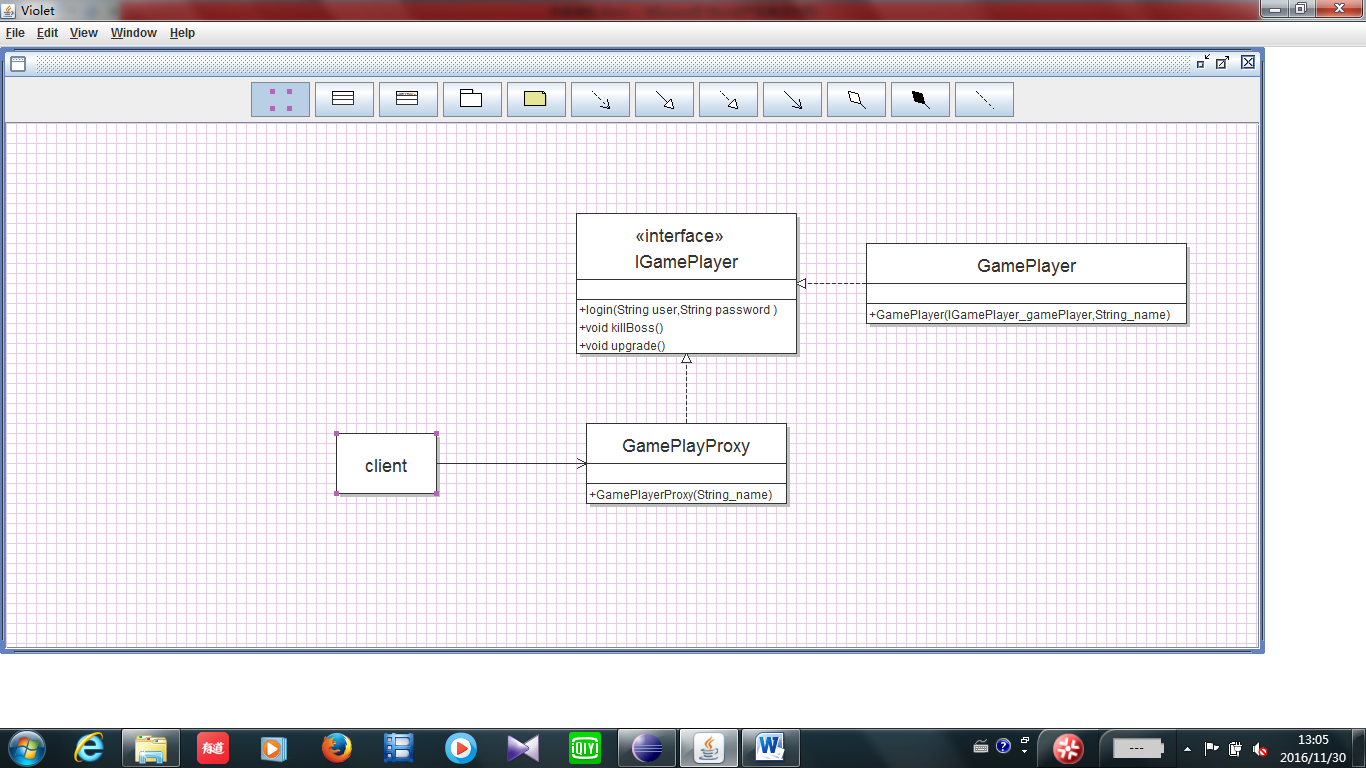
ConcretorSubject con=**new** ConcretorSubject();

Observer c=**new** Concretorobersver();

con.attach(c);

con.change("ABC"); }}

**代理模式：**



IGamePlayer接口

**public** **interface** IGamePlayer {

**public** **void** login(String user,String password);

**public** **void** killBoss();

**public** **void** upgrade();}

GamePlayProxy类

**public** **class** GamePlayerProxy **implements** IGamePlayer {

**private** IGamePlayer gamePlayer=**null**;

**public** GamePlayerProxy(String name){

**try**{ gamePlayer =**new** GamePlayer(gamePlayer, name);

}**catch** (Exception e){

} }

**public** **void** login(String user, String password) {

**this**.gamePlayer.killBoss(); }

**public** **void** killBoss() {

**this**.gamePlayer.login("user","password"); }

**public** **void** upgrade() {

**this**.gamePlayer.upgrade(); }}

GamePlayer类

**public** **class** GamePlayer **implements** IGamePlayer {

**private** String name=" ";

**public** GamePlayer(IGamePlayer \_gamePlayer,String \_name)**throws** Exception{

**if**(\_gamePlayer==**null**){

**throw** **new** Exception("不能创建真实角色!");

}**else**{

**this**.name=name; } }

**public** **void** login(String user, String password) {

System.*out*.println("登录名为"+user+"的用户"+**this**.name+"登陆成功！”); }

**public** **void** killBoss() {

System.*out*.println(**this**.name+"在打怪!"); }

**public** **void** upgrade() { System.*out*.println(**this**.name+"又升了一级！”);}}

主类

**public** **class** client {

**public** **static** **void** main(String[] args) {

IGamePlayer proxy=**new** GamePlayerProxy("张三");

System.*out*.println("开始时间是：2010-10-10 10：10");

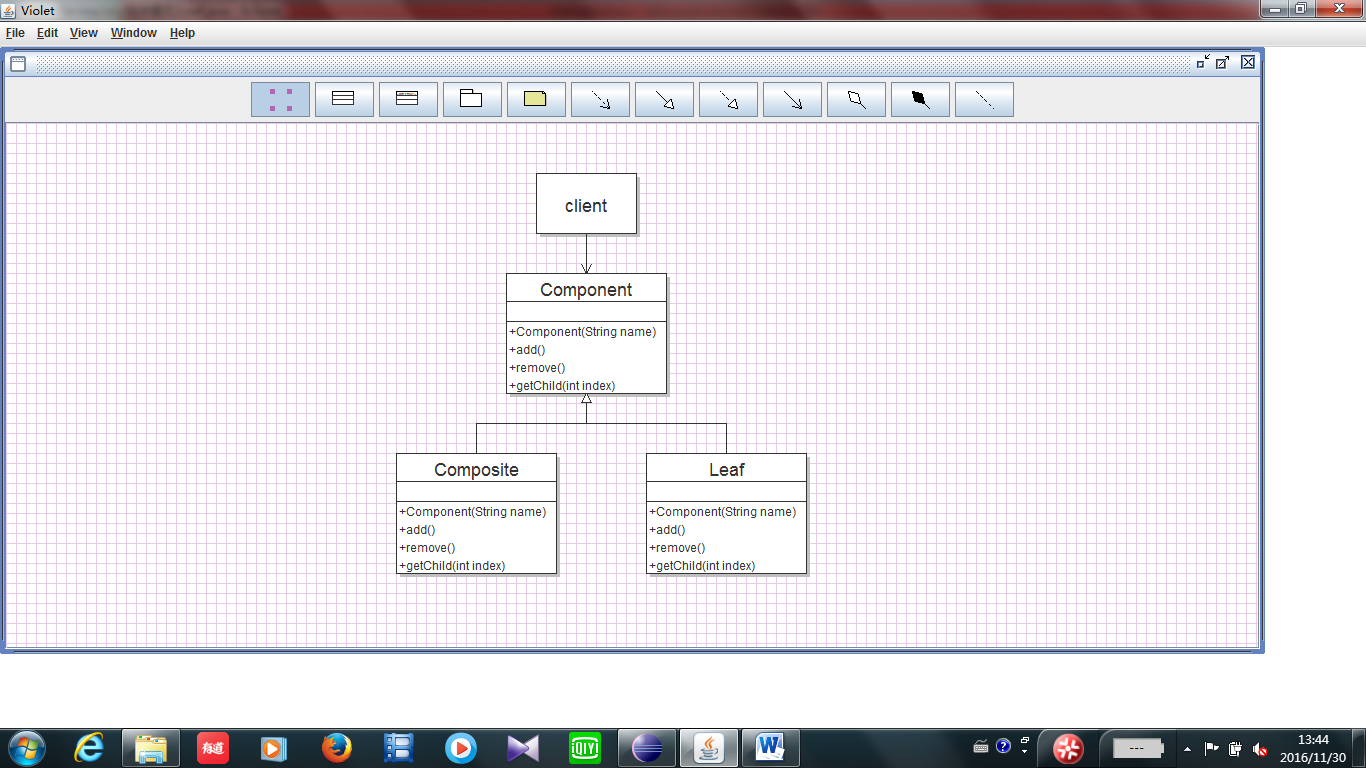
proxy.login("zhangsan", "mimazhangsan");

proxy.killBoss();

proxy.upgrade();

System.*out*.println("结束时间是：2010-10-19 15：10"); }}

**组织模式：**将对象组合成树形成以表现“部分整体”的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。



Component类

**public** **abstract** **class** Component {

**protected** String name;

Component(String name){

**this**.name=name; }

**public** **abstract** **void** add(Component c);

**public** **abstract** **void** remove(Component c);

**public** **abstract** **void** getChild(**int** index);}

Composite类

**public** **class** Composite **extends** Component {

**int** index;

Composite(String name) {

**super**(name); }

**public** ArrayList<Component> composite=**new** ArrayList<Component>();

**public** **void** add(Component c) {

**if**(c!=**null**){

composite.add(c); } }

**public** **void** remove(Component c) {

**if**(c!=**null**){ composite.remove(c); } }

**public** **void** getChild(**int** index) {

**this**.index=index;

StringBuilder sb=**new** StringBuilder("");

**for**(**int** i=0;i<index;i++){

sb.append("-"); }

System.*out*.println(**new** String(sb)+name);

**for**(Component c:composite){ c.getChild(index+2); } }}

Leaf类

**public** **class** Leaf **extends** Component {

Leaf(String name) {

**super**(name); }

**public** **void** add(Component c) {

System.*out*.println("add"); }

**public** **void** remove(Component c) {

System.*out*.println("remove"); }

**public** **void** getChild(**int** index) {

StringBuilder sb=**new** StringBuilder("");

**for**(**int** i=0;i<index;i++){

sb.append('-'); }

System.*out*.println(**new** String(sb)+name); }}

主类

**public** **class** Client {

**public** **static** **void** main(String[] args) {

Component c=**new** Composite("root");

c.add(**new** Leaf("leafA"));

Component comp=**new** Composite("compB");

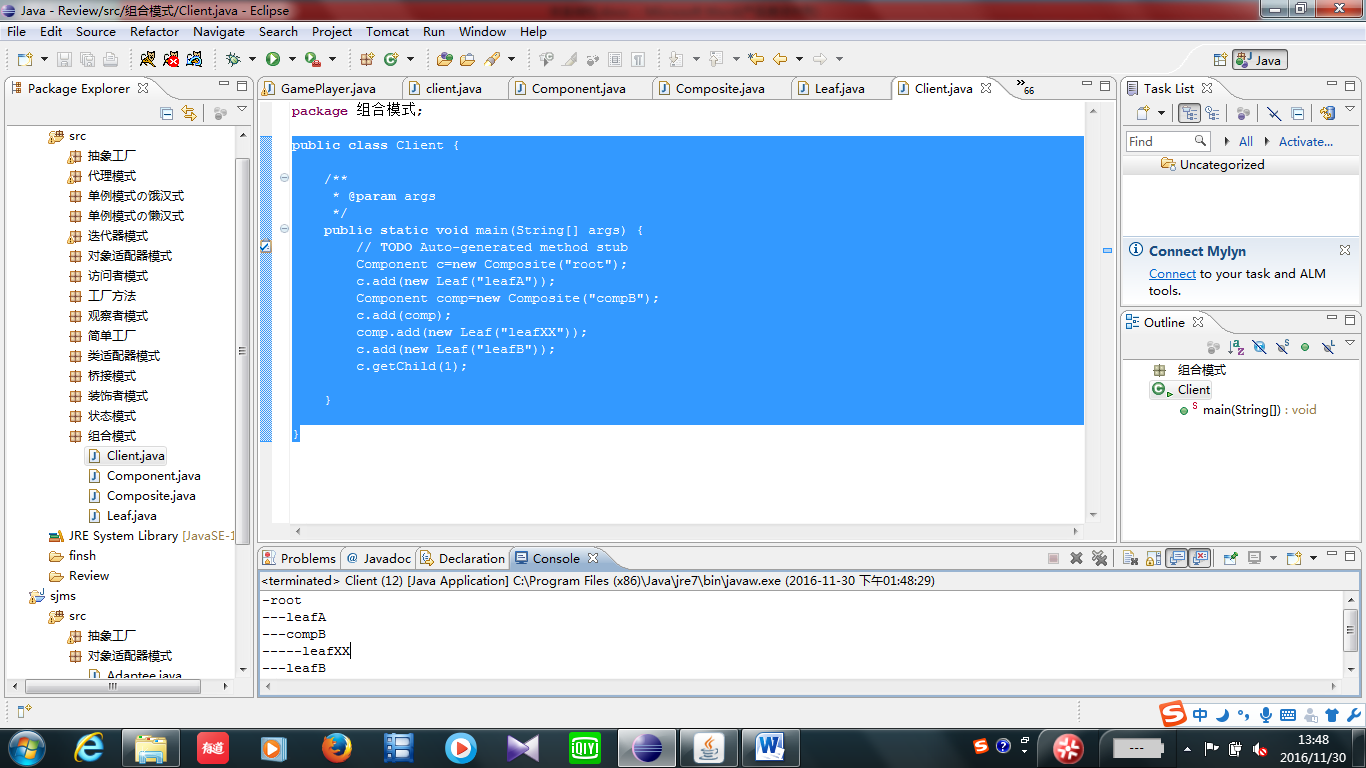
c.add(comp);

comp.add(**new** Leaf("leafXX"));

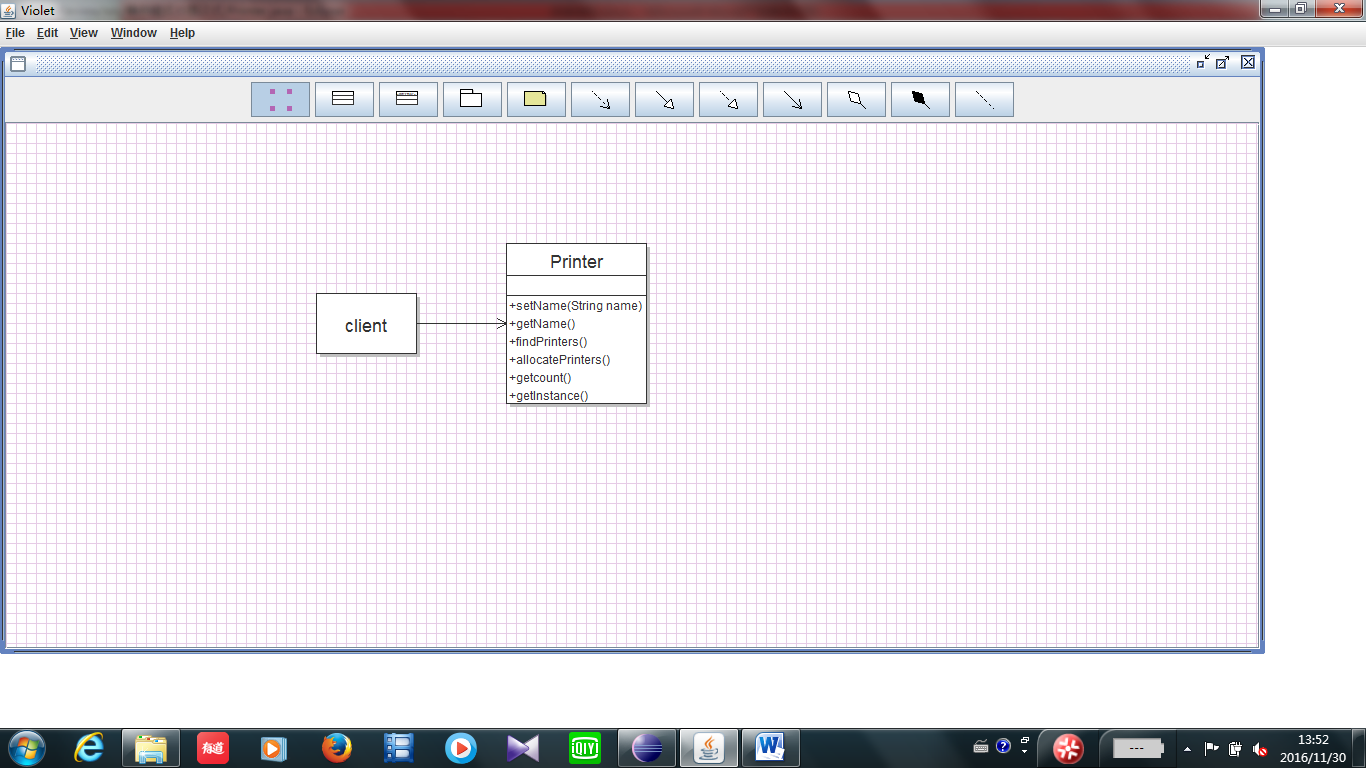
c.add(**new** Leaf("leafB"));

c.getChild(1); }}

输出内容



**单例模式：**



**饿汉式**

Printer类

**public** **class** Printer {

**private** **static** Printer *instance*=**new** Printer();

**public** **static** **int** *count*;

**private** String name;

**private** Printer() {}

**public** **void** setName(String name){

**this**.name=name;}

**public** String getName(){

**return** name;}

**public** **static** Printer getInstance(){

*count*++;

**return** *instance*;}

**public** **void** findPrinters(){

System.*out*.println(name+" is finding printer");}

**public** **void** allocatePrinters(){

System.*out*.println(name+" is allocating Printer");}

**public** **static** **int** getcount(){

**return** *count*;}}

主类

**public** **class** Client {

**public** **static** **void** main(String agr[]){

Printer p=Printer.*getInstance*();

p.setName("HP");

p.findPrinters();

p.allocatePrinters();

System.*out*.println(Printer.*getcount*());

Printer p1=Printer.*getInstance*();

p1.setName("canon");

p1.findPrinters();

p1.allocatePrinters();

System.*out*.println(Printer.*getcount*());}}

安全Printer类

**public** **class** SafePrinter {

**private** **static** SafePrinter *instance*=**new** SafePrinter();

**public** **static** **int** *count*;

**private** String name;

**private** SafePrinter() {

}

**public** **void** setName(String name){

**this**.name=name; }

**public** String getName(){

**return** name; }

**public** **synchronized** **static** SafePrinter getInstance(){

*count*++;

**return** *instance*; }

**public** **void** findPrinters(){

System.*out*.println(name+" is finding printer"); }

**public** **void** allocatePrinters(){

System.*out*.println(name+" is allocating Printer"); }

**public** **static** **int** getcount(){

**return** *count*; } }

安全主类

**public** **class** SafeClient {

**public** **static** **void** main(String[] args) {

Printer p=Printer.*getInstance*();

p.setName("HP");

p.findPrinters();

p.allocatePrinters();

System.*out*.println(Printer.*getcount*());

Printer p1=Printer.*getInstance*();

p1.setName("canon");

p1.findPrinters();

p1.allocatePrinters();

System.*out*.println(Printer.*getcount*()); }}

**懒汉式**

Printer类

**public** **class** Printer {

**private** **static** Printer *instance*=**null**;

**public** **static** **int** *count*;

**private** String name;

**private** Printer(String name) {

**this**.name=name;}

**public** **static** Printer getInstance(String name){

**if**(*instance*==**null**){ *instance*=**new** Printer(name); }

*count*++;

**return** *instance*;}

**public** **void** findPrinters(){

System.*out*.println(name+" is finding printer");}

**public** **void** allocatePrinters(){

System.*out*.println(name+" is allocating Printer");}

**public** **static** **int** getcount(){

**return** *count*;}}

主类

**public** **class** Client {

**public** **static** **void** main(String agr[]){

Printer p=Printer.*getInstance*("HP");

p.findPrinters();

p.allocatePrinters();

System.*out*.println(Printer.*getcount*());

Printer p1=Printer.*getInstance*("Canon");

p1.findPrinters();

p1.allocatePrinters();

System.*out*.println(Printer.*getcount*());}}

安全Printer类

**public** **class** SafePrinter {

**private** **static** SafePrinter *instance*=**null**;

**public** **static** **int** *count*;

**private** String name;

**private** SafePrinter(String name) {

**this**.name=name; }

**public** **synchronized** **static** SafePrinter getInstance(String name){

**if**(*instance*==**null**){ *instance*=**new** SafePrinter(name); }

*count*++;

**return** *instance*; }

**public** **void** findPrinters(){

System.*out*.println(name+" is finding printer"); }

**public** **void** allocatePrinters(){

System.*out*.println(name+" is allocating Printer"); }

**public** **static** **int** getcount(){ **return** *count*; } }

安全主类

**public** **class** SafeClient {

**public** **static** **void** main(String[] args) {

SafePrinter sp=**null**;

sp=SafePrinter.*getInstance*("HP");

sp.findPrinters();

sp.allocatePrinters();

System.*out*.println(SafePrinter.*getcount*());

SafePrinter sp1=**null**;

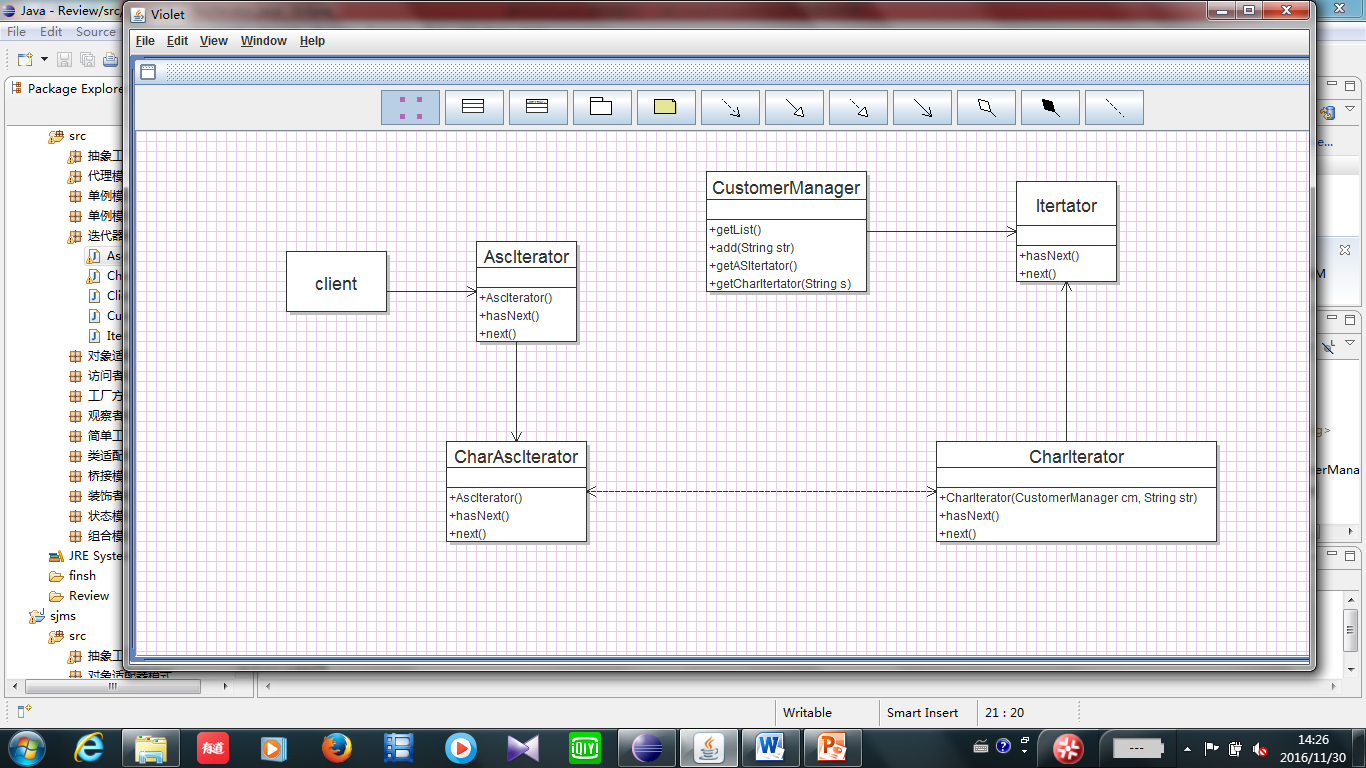
sp1=SafePrinter.*getInstance*("canon");

sp1.findPrinters();

sp1.allocatePrinters();

System.*out*.println(SafePrinter.*getcount*()); }}

**迭代器模式：**提供一种方法来访问聚合对象，而不用暴露这个对象的内部表示，其别名为游标(Cursor)。



CustomerManager类

**public** **class** CustomerManager {

**private** ArrayList<String> list = **new** ArrayList<String>();

**public** **void** add(String str) {

list.add(str); }

**protected** ArrayList<String> getList() {

**return** list; }

**public** Itertator getASItertator() {

**return** **new** AscIterator(**this**); }

**public** Itertator getCharItertator(String s) {

**return** **new** CharIterator(**this**, s); }}

Itertator类

**public** **abstract** **class** Itertator

{ **public** **abstract** **boolean** hasNext();

**public** **abstract** String next();}

AscIterator类

**public** **class** AscIterator **extends** Itertator {

ArrayList<String> list = **new** ArrayList<String>();

Iterator iter;

**public** AscIterator(CustomerManager cm) {

ArrayList<String> ls = cm.getList();

**for** (String s : ls) {

list.add(s);

Collections.*sort*(list);

iter = list.iterator(); } }

**public** **boolean** hasNext() {

**return** iter.hasNext(); }

**public** String next() {

**return** (String) iter.next(); }}

CharIterator类

**public** **class** CharIterator **extends** Itertator {

ArrayList<String> list = **new** ArrayList<String>();

Iterator iter;

**public** CharIterator(CustomerManager cm, String str) {

ArrayList<String> ls = cm.getList();

**for** (String s : ls) {

**if** (s.startsWith(str)) {

list.add(s); }

iter = list.iterator(); } }

**public** **boolean** hasNext() {

**return** iter.hasNext();}

**public** String next() {

**return** (String) iter.next(); }}

主类

**public** **class** Client {

**public** **static** **void** main(String[] args) {

CustomerManager cm=**new** CustomerManager();

cm.add("zhangsan");

cm.add("lisi");

cm.add("brown");

cm.add("alien");

Itertator iter;

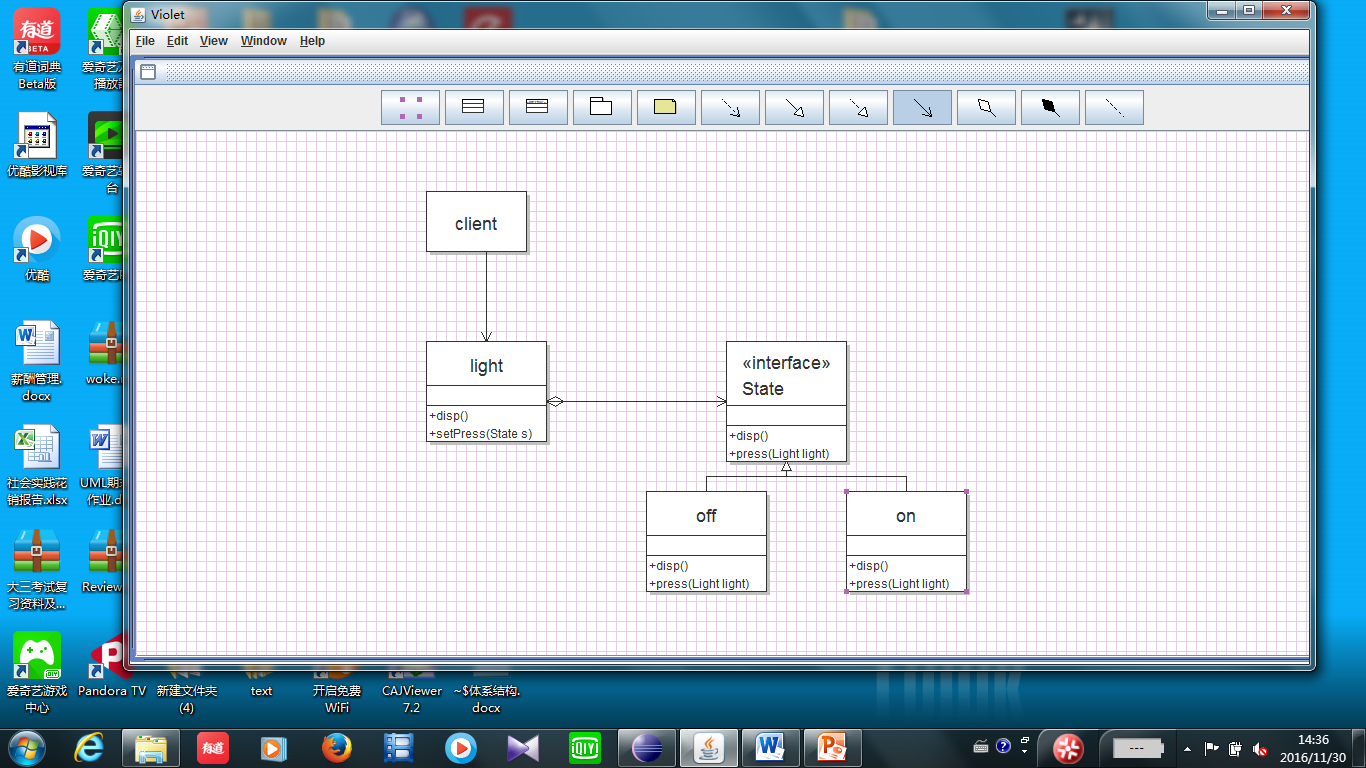
iter=cm.getCharItertator("a");

**while**(iter.hasNext()){

String str=iter.next();

System.*out*.println(str); } }}

**状态模式：**允许一个对象在其内部状态改变时改变它的行为，对象看起来似乎修改了它的类。



Light类

**public** **class** Light {

State state;

**public** **void** disp(){

state.disp(); }

**public** **void** setPress(State s){

**this**.state=s; } }

State接口

**public** **interface** State {

**public** **void** disp();

**public** **void** press(Light light);}

ON类

**public** **class** ON **implements** State {

**public** **void** disp() {

System.*out*.println("开灯状态"); }

**public** **void** press(Light light) {

light.setPress(**new** OFF()); }}

Off类

**public** **class** OFF **implements** State{

**public** **void** disp() {

System.*out*.println("关灯状态"); }

**public** **void** press(Light light) {

light.setPress(**new** ON()); }}

主类

**public** **class** Client {

**public** **static** **void** main(String[] args) {

Light light=**new** Light();

State state=**new** ON();

light.setPress(state);

System.*out*.print("现在是:");

light.disp();

System.*out*.print("按下开关后是:");

state.press(light);

light.disp(); }}

**原型模式及序列化：（不明确）**

原型模式

Cloneable

继承 Clone()方法

序列化

Serializable