软件工程

Software Engineering

极夜酱

目录

Chapter 1 需求工程

1.1 软件开发

1.1.1 软件开发

什么是好的软件？一个好的软件需要满足三点要求：

1. 一般要求（general requirements）
2. 法律要求（legal requirements）
3. 具体要求（specific requirements）

一般要求包括软件系统需要有详细的文档、可读性高的代码、易于修改和维护的代码、能够开展测试、容易移植、使用方便等方面。

法律要求表示系统必须要遵守相关的法律法规。例如加拿大安大略省制定了残疾人无障碍法案，要求软件必须考虑照顾到残障人士的使用；加拿大反垃圾邮件立法规制了垃圾邮件、间谍软件、恶意软件和僵尸网络等问题。

具体要求即来自客户和用户具体的需求，软件系统应该要能够实现客户和用户提出的功能。

想要开发出一款好的软件，必须要拥有良好的编程能力，这可以通过大量练习提高。同时也需要具备良好的沟通和规划能力，例如在处理问题时，应该与客户或者熟悉问题的人沟通，确保理解了客户/用户的需求。编程前也可以先查找别人已经做过的相关内容，不要重复造轮子。

1.2 需求

1.2.1 用户故事（User Story）

用户故事是用几句话描述用户会做的一件事，以及他们为什么要这样做。用户故事是关于用户将会做什么，而不是怎么做。

用户故事一般采用这样的格式："As a [user type], I want [some action] so that [some reason]."。例如，"作为一个活跃用户，我想不用每次登录都输入账号密码，以便更方便快速地登录。"

1.2.2 需求（Requirements）

需求描述了软件做的某一件事，它可以是功能性（functional）或非功能性（non-functional）的。同样一个需求只需要描述做什么，而不是如何去做。

需求的作用是为了让开发者和客户都清楚最终产品想要的预期是什么。

一个好的需求应该具备以下条件：

- 分类（categorized）：使用 MuSCOW 法则对需求进行分类。
  - MUSTS：系统能够满足客户的基本需求，类似于最小可行产品（minimum viable product）
  - SHOULDS：满足客户基本或更高级的需求
  - COULDS：能够使系统变得更好的额外需求
  - WON'TS：系统不该做的事情
- 有优先级（prioritized）：对不同的需求指定一个优先级，需要遵守 MUSTS > SHOULDS > COULDS > WONT'S。
- 逻辑合理：需求之间的依赖关系需要合理，例如一个 MUST 需求不能依赖于一个 SHOULD 需求。

- 时间估计（time estimate）：一个需求应该需要在 15 天内完成，如果不能完成，应该把它分解成多个小需求。

1.3 原型设计

1.3.1 原型设计（Prototyping）

原型是指没有功能的模型，它不一定要包含一个程序的所有方面。

原型设计是与客户/用户沟通的一种工具，让客户/用户感觉他们自己也是设计者。原型设计的目的是为了验证 UI 是否能够可以被正常使用，让开发者能够以用户的角度思考问题。

1.3.2 纸上原型（Paper Prototyping）

纸上原型也称低保真原型（low fidelity prototyping），通过纸笔等工具设计，通常是 UI 设计的第一步。它具有制作过程快速、成本低、更改快速、效率高的特点。

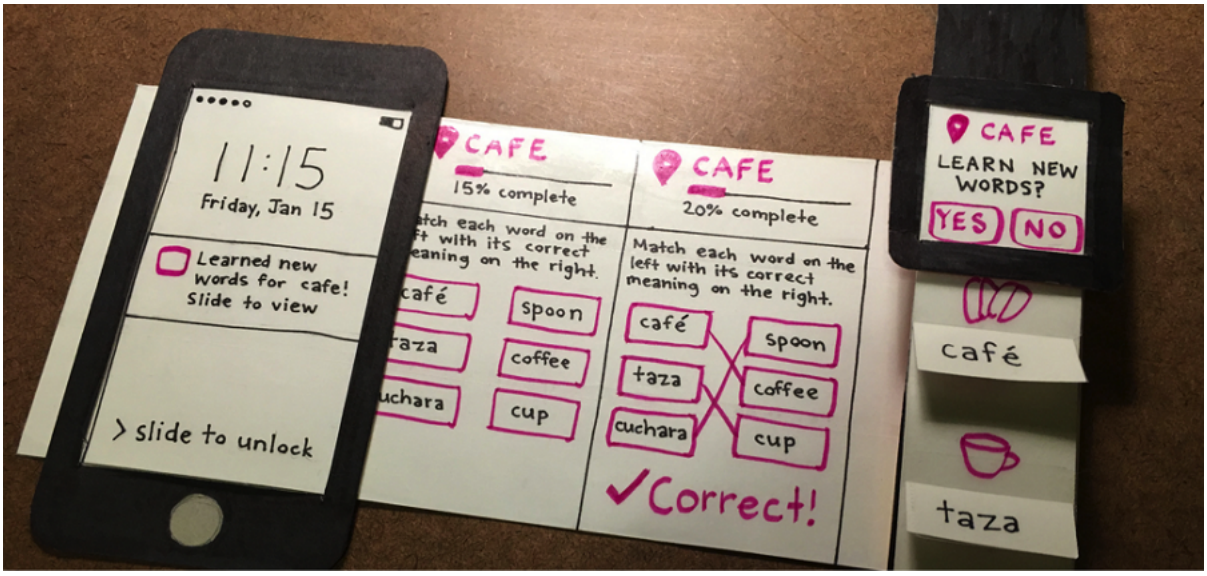


图 1.1: 纸上原型

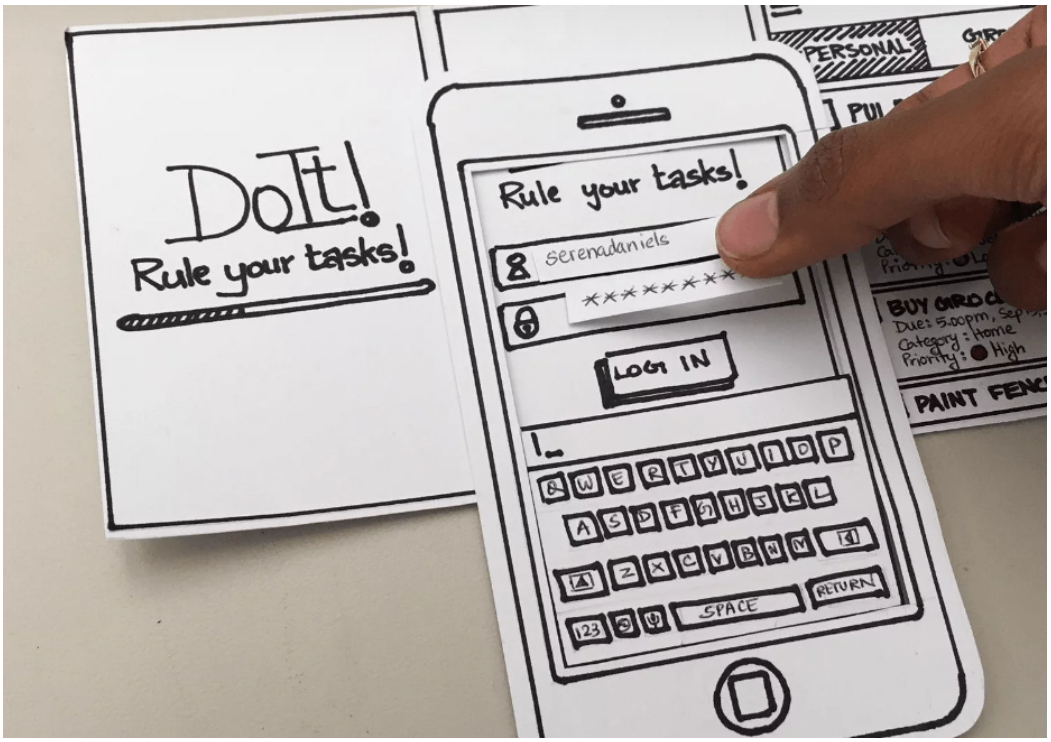


图 1.2: 纸上原型

纸上原型的目的并不是为了让 UI 更加美观，而是为了确保系统的流程是否存在问题。

在完成纸上原型后，需要让参与测试的人根据用例进行测试。在测试过程中，不要提供任何帮助或者指导，以此来验证用户交互是否流畅。

最后，记录下在纸上原型测试中哪些地方是正常的、哪些地方存在问题或遗漏的。

1.3.3 线框图（Wireframing）

线框图也称高保真原型（high fidelity prototyping），它看上去非常接近最终产品，几乎完全按照实物来制作，原型中甚至包含产品的细节、真实的交互、UI 等。

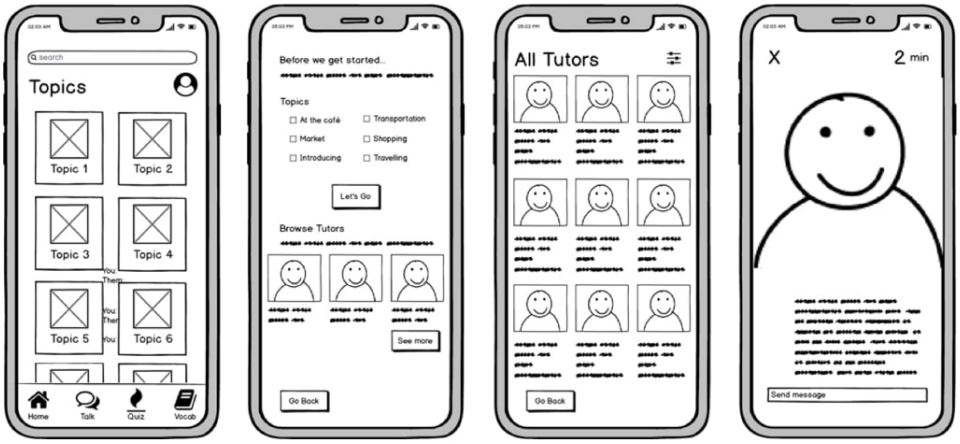


图 1.3: 线框图

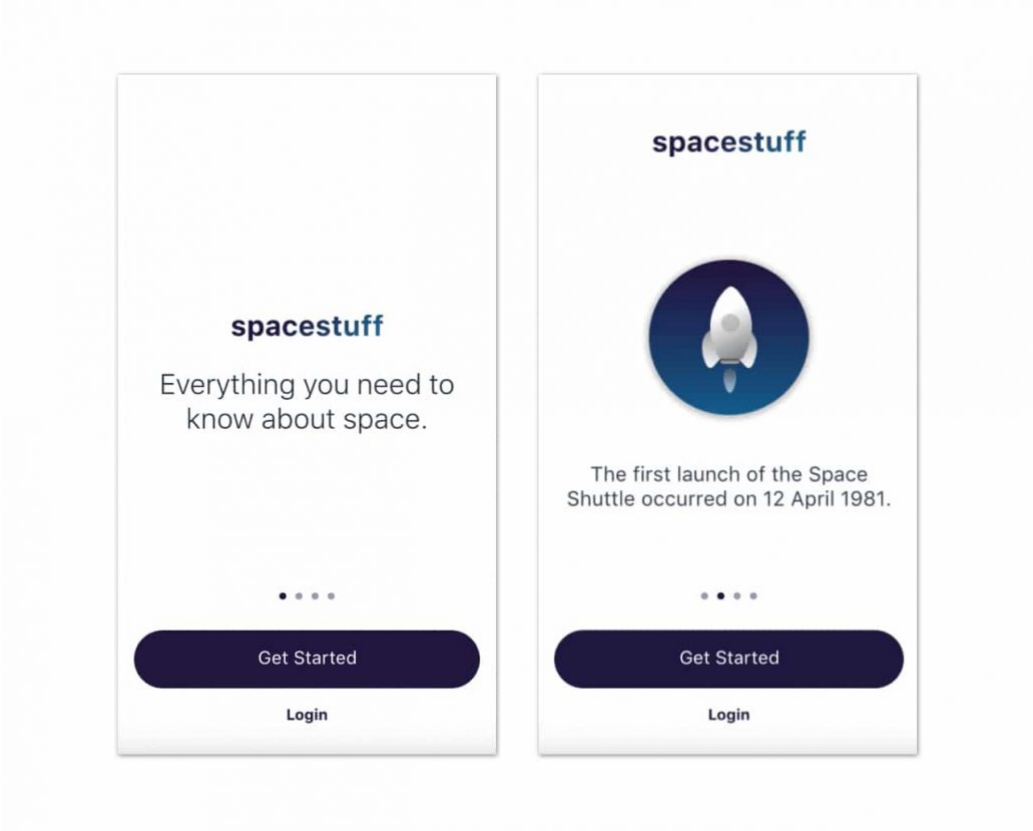


图 1.4: 线框图

高保真原型的目的在于让用户提供反馈，例如 UI 中控件的位置和 UI 的美观程度。

Chapter 2 软件过程

2.1 软件过程模型

2.1.1 瀑布模型（Waterfall）

软件工程是使用系统化的、严格约束的、可量化的方法开发、运行和维护软件。

瀑布模型是一个软件生命周期模型，开发过程是通过一系列阶段顺序开展的，从系统需求分析直到产品发布，项目开发从一个阶段流动到下一阶段，这也是瀑布模型名称的由来。直到 80 年代早期，它一直是唯一被广泛采用的软件开发模型。

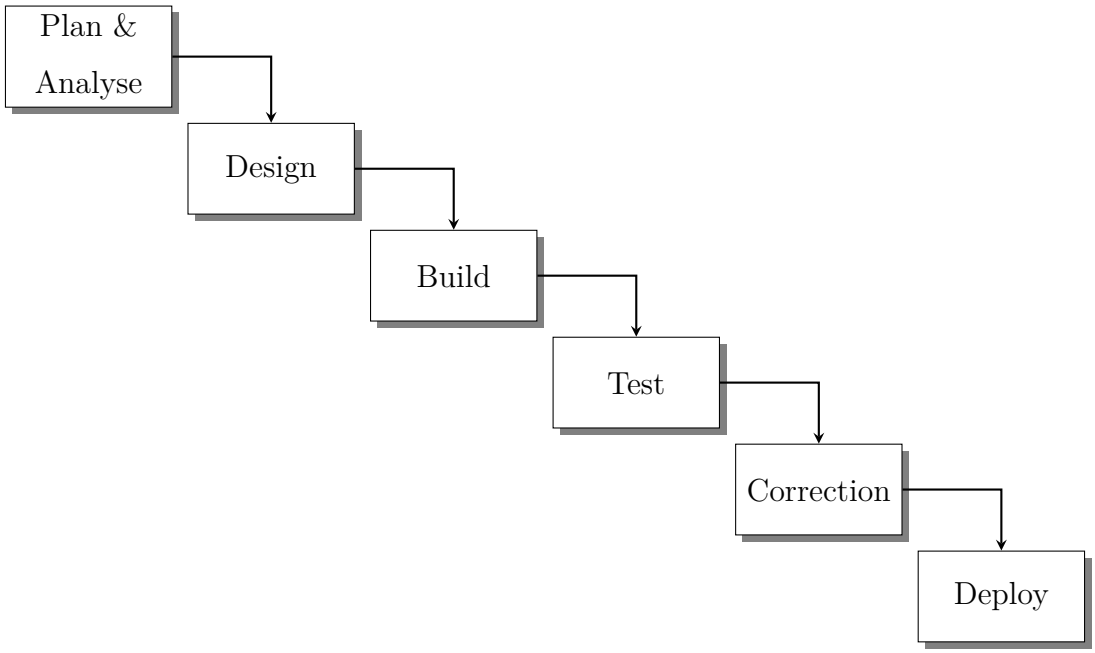


图 2.1: 瀑布模型

瀑布模型的特点是各阶段间具有顺序性和依赖性，必须等前一阶段的工作完成之后，才能开始后一阶段的工作。前一阶段的输出文档就是后一阶段的输入文档。

因此，瀑布模型是由文档驱动的，在可运行的软件产品交付给用户之前，用户只能通过文档来了解产品。瀑布模型几乎完全依赖于书面的规格说明，很可能导致

最终开发出的软件产品不能真正满足用户的需要。也不适合需求模糊的系统。

传统的瀑布模型过于理想化，人在工作过程中不可能不犯错误。因而产生了加入迭代过程的瀑布模型。

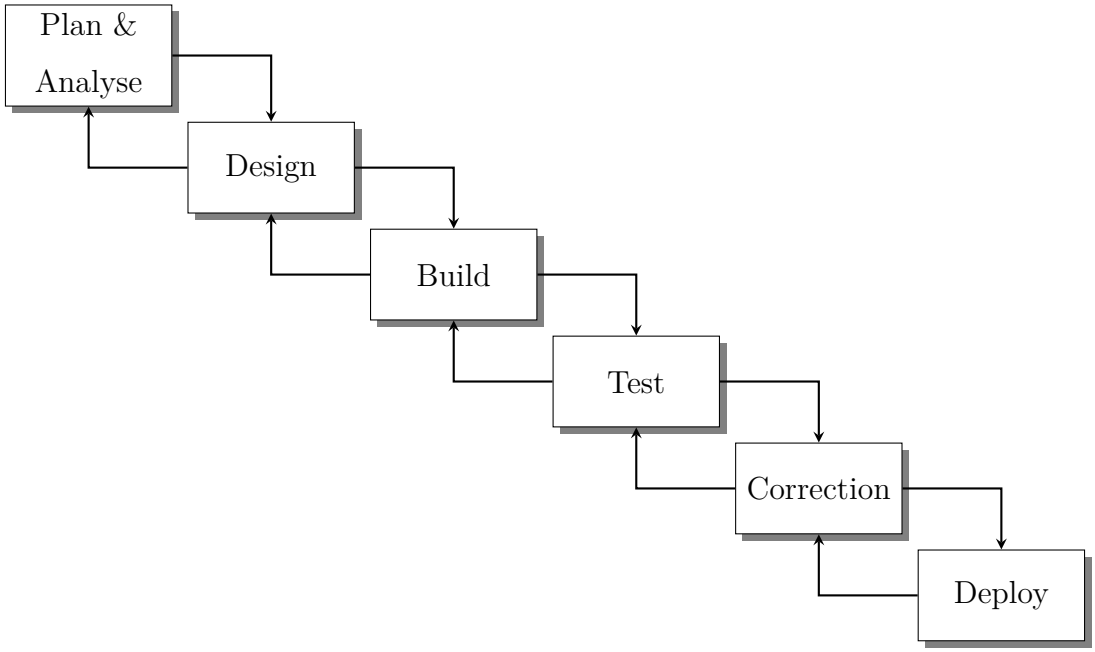


图 2.2: 加入迭代过程的瀑布模型

2.1.2 增量式开发（Incremental Development）

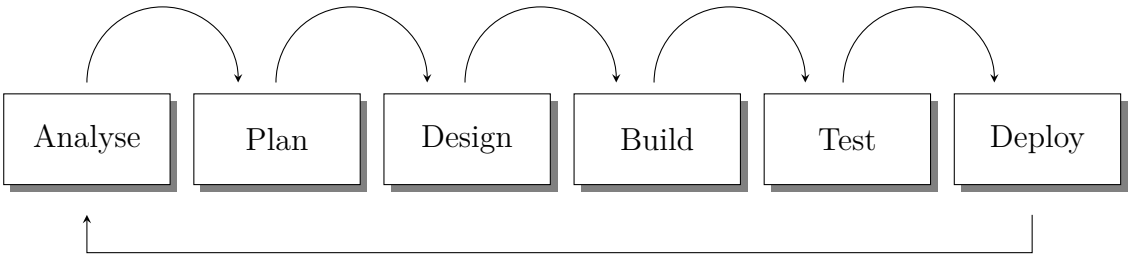


图 2.3: 增量式开发

增量式开发允许开发人员加以利用系统的早期交付版本，重新参考和修改以达到预期的结果。增量式开发较为灵活快速，在开发的每个周期都会产出一个能够交付的版本。这样较为频繁的交付，能够使客户持续地提供反馈，从而获得更高质量的产品。

2.2 敏捷开发

2.2.1 敏捷开发（Agile）

推行新的软件过程模型的原因在于，传统的开发方式有着更高的失败率，大部分项目的支出都远超出预算水平。

很多项目的开发阶段都超出了预期的时间，同时由于软件开发的特性，反而会导致"Adding people to a late project just makes it later"。

敏捷开发方法是一种将精力集中在软件本身，而不是设计和文档上。它依赖迭代的方法来完成软件的描述、开发和交付。

敏捷方法的核心思想体现在：

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

这些思想主要表现为注重客户的参与，客户会在开发过程中紧密参与，为软件系统提供新的需求和评估。同时敏捷开发要求能够接受变更，设计的系统需要能够适应新的需求变化。

2.2.2 敏捷项目管理

Scrum 方法是一个通用的敏捷方法，它更加注重迭代开发的管理。

在 Scrum 中，每个开发周期都包含开发任务的分配、实现、当前版本演示和会议讨论。每个周期都有一个固定的长度，通常为 2~4 周，每个周期开发者会根据开发任务的清单，依据需求的优先级各自实现需求。在该周期无法完成的任务，会

被继续放入任务清单，在下个周期继续完成。

在每个周期结束后，项目团队的所有成员需要参加会议讨论，根据当前版本的软件，回顾开发过程、交流问题等。

Scrum 使得产品被分解为多个可管理的部分，整个团队所做的任务都是可见的，有助于改善团队见的沟通。同时也能阶段性地交付产品给用户，在下个周期能够根据用户反馈进行修改。

Chapter 3 UML

3.1 UML

3.1.1 UML（Unified Modeling Language）

统一建模语言 UML 是一种为面向对象系统的产品进行说明、可视化和编制文档的一种标准语言。

UML 中包含了一系列不同类型的图：

- 类图
- 组件图
- 部署图
- 对象图
- 封装图
- 复合结构图
- 剖面图
- 用例图
- 活动图
- 状态机图
- 序列图
- 通讯图
- 交互概览图
- 时序图

这些图主要可分为三大类：

1. 功能模型（functional model）：从用户的角度展示系统的功能，如用例图。
2. 对象模型（object model）：采用对象、属性、操作、关联等展示系统的结构，如类图、对象图。
3. 动态模型（dynamic model）：展现系统的内部行为，如时序图、活动图、状态图。

3.2 用例图

3.2.1 用例图（Use Case Diagram）

用例图用于描述从用户角度所看到的系统功能。通过用例图，人们可以获知系统不同种类的用户和用例。

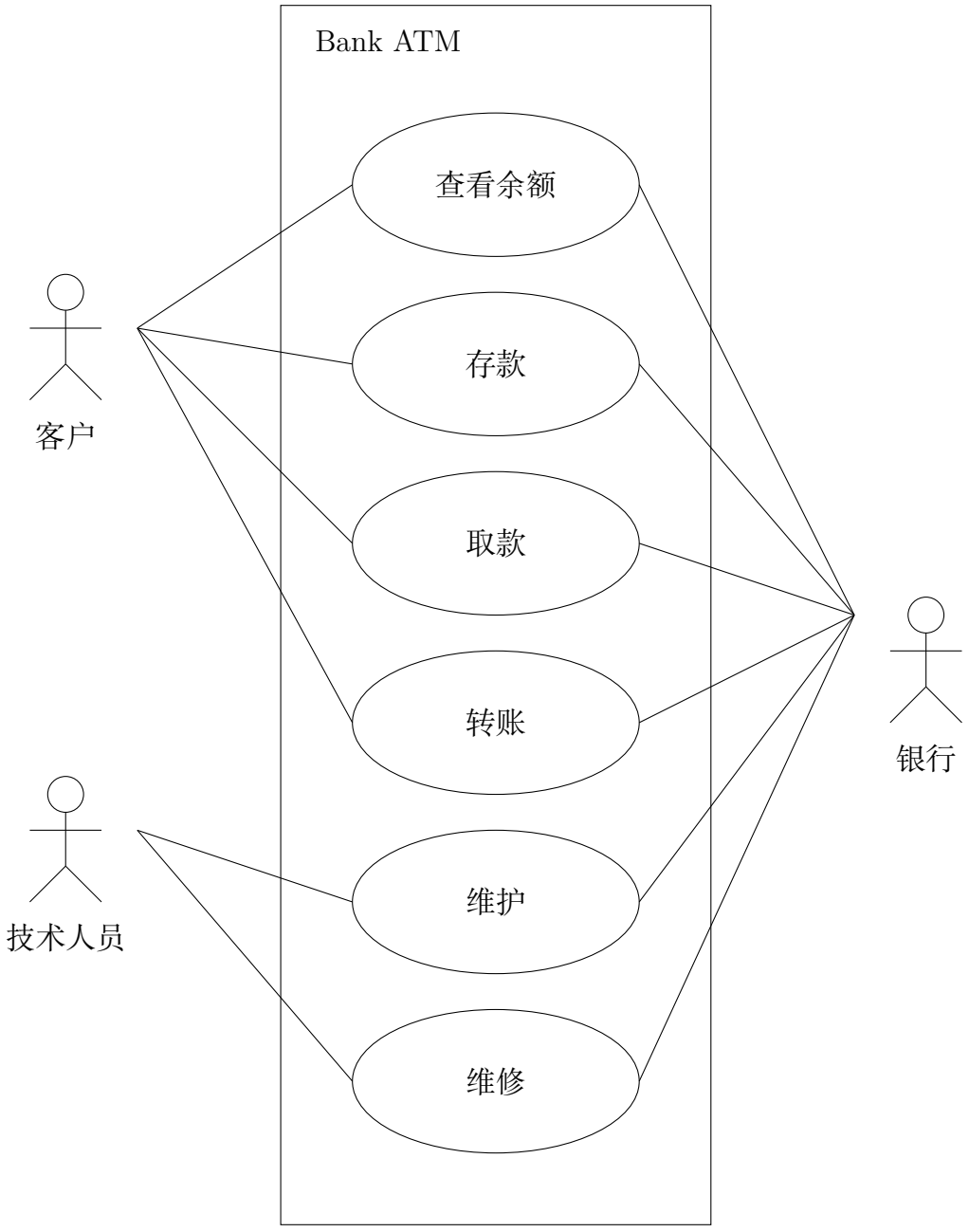


图 3.1: 银行 ATM 系统

3.3 类图

3.3.1 类图（Class Diagram）

类图用于显示模型的静态结构，例如类的内部结构以及类与类之间的关系。

一个类包含属性和方法两部分，在类图中 public 属性和方法使用“+”表示，private 属性和方法使用“-”表示，protected 属性和方法使用“#”表示。

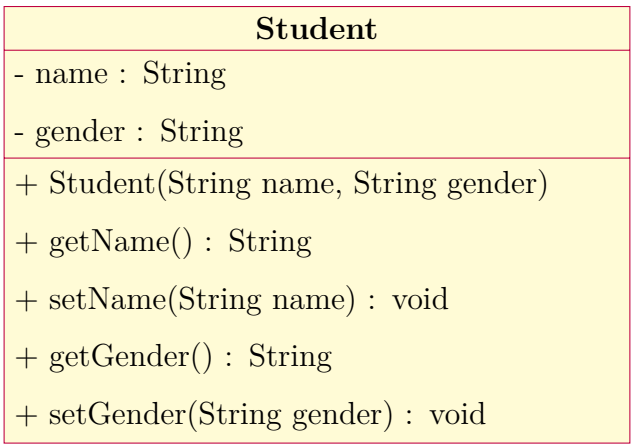


图 3.2: Student 类

3.3.2 继承（Inheritance）

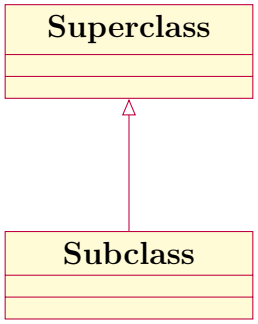


图 3.3: 继承

子类和父类之间存在“is a”的关系，例如 Dog 和 Cat 都属于 Animal 的子类。

3.3.3 关联（Association）

图 3.4: 关联

关联关系使一个类能够知道另一个类的属性和方法。关联可以是双向的，也可以是单向的。

关联关系还可以指定多重性（multiplicity）。例如一个 Student 可以和多个 Professor 存在关联。

一个 Professor 可以和多个 Student 存在关联。

多个 Professor 同样也可以和多个 Student 存在关联。

3.3.4 聚合（Aggregation）

聚合关系是关联联系的特例，用于表示整体和部分的关系，即 “has a”。但是整体和部分有各自独立的生命周期。

例如 Monther 有一个 Child，但是如果 Mother 死了，Child 仍然存活。

图 3.5: 聚合

3.3.5 组合（Composition）

组合用于表示“part of”的的关系，因此具有组合关系的两个类具有相同的声明周期。

例如血液细胞是身体的一部分，人死了，血液细胞也会死。

图 3.6: 组合

3.4 时序图

3.4.1 时序图（Sequence Diagram）

时序图通过描述对象之间发送消息的时间顺序显示多个对象之间的动态协作。

时序图中包括以下元素：

1. 角色（actor）/对象（object）：系统角色，可以是人或者其它子系统。
2. 生命线（lifeline）：表示对象在一段时期内的存在。
3. 控制焦点（activation）：在时序图中每条生命线上的窄矩形代表活动期。
4. 消息（message）：类角色通过发送和接受信息进行通信。

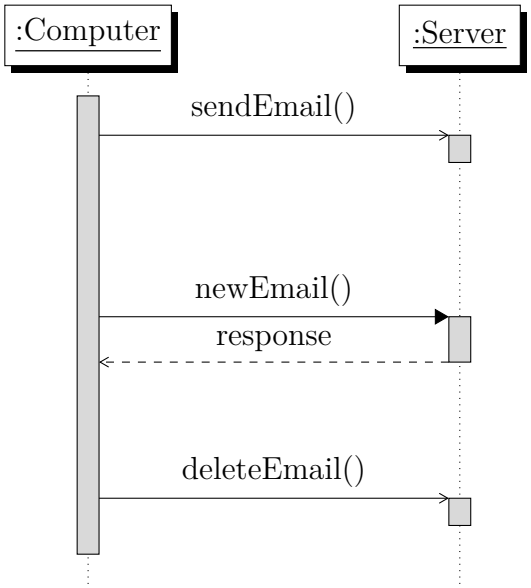


图 3.7: 时序图

Chapter 4 设计模式

4.1 设计模式

4.1.1 设计模式（Design Pattern）

设计模式代表了最佳的实践，通常被有经验的面向对象软件开发人员所采用。设计模式是开发人员在开发过程中面临的一般问题的解决方案，这些解决方案是众多开发人员经过相当长一段时间的试验和错误总结出来的。

设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了重用代码、让代码更容易被他人理解、保证代码可靠性。设计模式使代码编制真正工程化，是软件工程的基石。项目中合理地运用设计模式可以完美地解决很多问题，每种模式在现实中都有相应的原理来与之对应，每种模式都描述了周围不断重复发生的问题，以及该问题的核心解决方案，这也是设计模式能被广泛应用的原因。

4.1.2 GOF（Gang of Four）

1994 年，由 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 四人合著出版了一本名为 Design Patterns - Elements of Reusable Object-Oriented Software 的书，该书首次提到了软件开发中设计模式的概念。

四位作者合称 GOF，他们所提出的设计模式主要基于两个面向对象设计原则：

1. 对接口编程而不是对实现编程。
2. 优先使用对象组合而不是继承。

4.1.3 设计模式分类

Design Patterns - Elements of Reusable Object-Oriented Software 中提到的设计模式可以分为三大类:

1. 创建型模式（creational patterns）: 提供了一种在创建对象的同时隐藏创建逻辑的方式，主要特点是将对象的创建与使用分离。
    - 工厂模式
    - 抽象工厂模式
    - 单例模式
    - 建造者模式
    - 原型模式
2. 结构型模式（structural patterns）: 用于描述如何将类或对象按某种布局组成更大的结构。
    - 适配器模式
    - 桥接模式
    - 过滤器模式
    - 组合模式
    - 装饰器模式
    - 外观模式
    - 享元模式
    - 代理模式
3. 行为型模式（behavioral patterns）: 用于描述类或对象之间如何相互协作共同完成单个对象无法单独完成的任务以及如何分配职责。
    - 责任链模式
    - 命令模式
    - 解释器模式
    - 迭代器模式

- 中介者模式
- 备忘录模式
- 观察者模式
- 状态模式
- 空对象模式
- 策略模式
- 模板模式
- 访问者模式

4. J2EE 模式：特别关注表示层。

- MVC 模式
- 业务代表模式
- 组合实体模式
- 数据访问对象模式
- 前端控制器模式
- 拦截过滤器模式
- 服务定位器模式
- 传输对象模式

4.1.4 六大原则

1. 开闭原则（Open Close Principle）：对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。

2. 里氏代换原则（Liskov Substitution Principle）：只有当派生类可以替换掉基类，且功能不受到影响时，基类才能真正被复用，而派生类也能够在基类的基础上增加新的行为。

3. 依赖倒转原则（Dependence Inversion Principle）：针对接口编程，依赖于抽象而不依赖于具体。

4. 接口隔离原则（Interface Segregation Principle）：使用多个隔离的接口比使用单个接口要好，降低类之间的依赖和耦合，便于升级和维护。

5. 迪米特法则 / 最少知道原则（Demeter Principle）：一个实体应当尽量少地与其它实体之间发生相互作用，使得系统功能模块相对独立。

6. 合成复用原则（Composite Reuse Principle）：尽量使用合成/聚合的方式，而不是使用继承。

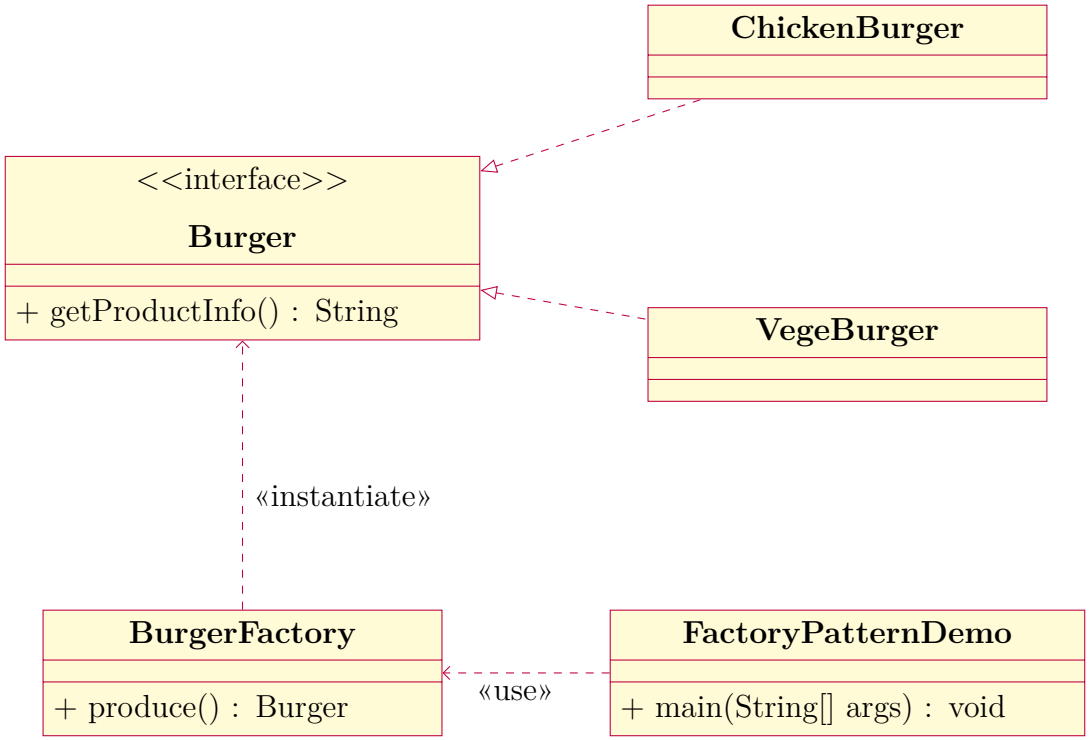
4.2 工厂模式

4.2.1 工厂模式（Factory Pattern）

工厂模式属于创建型模式。在工厂模式中，创建对象时不会对客户端暴露创建逻辑，而是通过使用一个共同的接口来指向新创建的对象。

工厂模式将客户类和工厂类分开，消费者任何时候需要某种产品，只需向工厂请求即可，消费者无须修改就可以接纳新产品。当多个子类共同继承同一父类，并由消费者的要求指定生产某一种产品时适用于工厂模式。例如，在麦当劳想吃麦辣鸡腿堡时，只需要对服务员说“要一个麦辣鸡腿堡”，过会一个麦辣鸡腿堡就会做好送过来。

工厂模式



Burger.java

```
1 public interface Burger {
2     String getProductInfo();
```

```
}
```

ChickenBurger.java

```
public class ChickenBurger implements Burger {
    @Override
    public String getProductInfo() {
        return "chicken burger";
    }
}
```

VegeBurger.java

```
public class VegeBurger implements Burger {
    @Override
    public String getProductInfo() {
        return "vegeburger";
    }
}
```

BurgerFactory.java

```
public class BurgerFactory {
    public Burger produce(String burgerType) {
        if(burgerType == null) {
            return null;
        }
        if(burgerType.equalsIgnoreCase("chicken burger")) {
            return new ChickenBurger();
        } else if(burgerType.equalsIgnoreCase("vegeburger")) {
            return new VegeBurger();
        }
        return null;
    }
}
```

FactoryPatternDemo.java

```
public class FactoryPatternDemo {
    public static void main(String[] args) {
```

```
3           BurgerFactory burgerFactory = new BurgerFactory();
4
5           Burger burger1 = burgerFactory.produce("chicken burger");
6           System.out.println(burger1.getProductInfo());
7
8           Burger burger2 = burgerFactory.produce("vegeburger");
9           System.out.println(burger2.getProductInfo());
10      }
11  }
```

运行结果

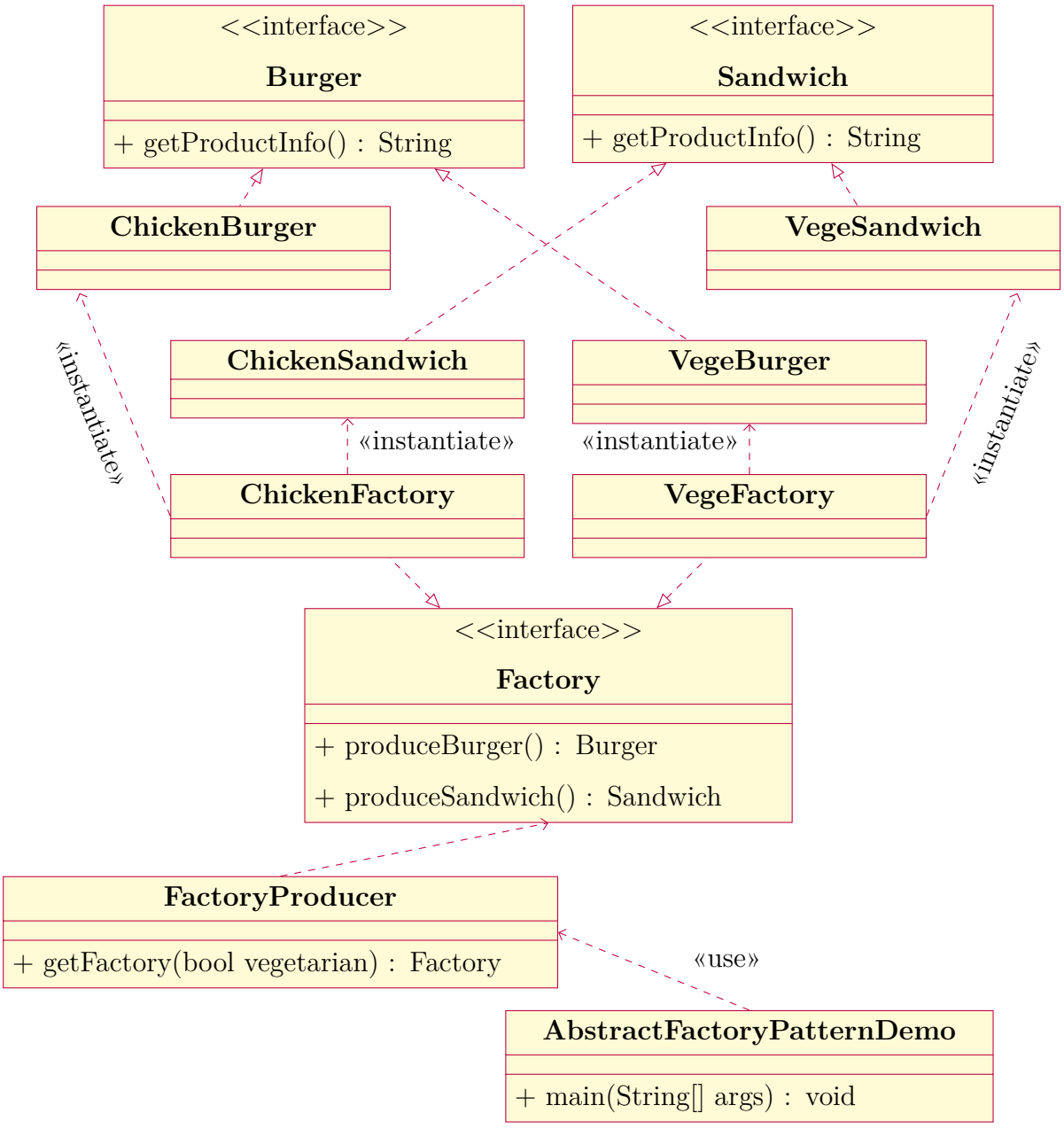
```
chicken burger
vegeburger
```

4.3 抽象工厂模式

4.3.1 抽象工厂模式（Abstract Factory Pattern）

抽象工厂模式属于创建型模式。抽象工厂模式提供了一个创建一系列相关或者相互依赖对象的接口，每个具体工厂都提供了多个工厂方法用于创建多种不同类型的具体对象。

抽象工厂模式



Burger.java

```java
public interface Burger {
    String getProductInfo();
}
```

Sandwich.java

```java
public interface Sandwich {
    String getProductInfo();
}
```

ChickenBurger.java

```java
public class ChickenBurger implements Burger {
    @Override
    public String getProductInfo() {
        return "chicken burger";
    }
}
```

VegeBurger.java

```java
public class VegeBurger implements Burger {
    @Override
    public String getProductInfo() {
        return "vegeburger";
    }
}
```

ChickenSandwich.java

```java
public class ChickenSandwich implements Sandwich {
    @Override
    public String getProductInfo() {
        return "chicken sandwich";
    }
}
```

VegeSandwich.java

```java
public class VegeSandwich implements Sandwich {
```

```
    @Override
    public String getProductInfo() {
        return "vegetarian sandwich";
    }
}
```

ChickenSandwich.java

```
public class ChickenSandwich implements Sandwich {
@Override
public String getProductInfo() {
return "chicken sandwich";
}
}
```

Factory.java

```
public interface Factory {
    Burger produceBurger();
    Sandwich produceSandwich();
}
```

ChickenFactory.java

```
public class ChickenFactory implements Factory {
    @Override
    public Burger produceBurger() {
        return new ChickenBurger();
    }

    @Override
    public Sandwich produceSandwich() {
        return new ChickenSandwich();
    }
}
```

VegeFactory.java

```
public class VegeFactory implements Factory {
    @Override
```

```
    public Burger produceBurger() {
        return new VegeBurger();
    }

    @Override
    public Sandwich produceSandwich() {
        return new VegeSandwich();
    }
}
```

FactoryProducer.java

```
public class FactoryProducer {
    public static Factory getFactory(boolean vegetarian) {
        if(vegetarian) {
            return new VegeFactory();
        } else {
            return new ChickenFactory();
        }
    }
}
```

AbstractFactoryPatternDemo.java

```
public class AbstractFactoryPatternDemo {
    public static void main(String[] args) {
        Factory factory1 = FactoryProducer.getFactory(false);
        Burger burger1 = factory1.produceBurger();
        System.out.println(burger1.getProductInfo());
        Sandwich sandwich1 = factory1.produceSandwich();
        System.out.println(sandwich1.getProductInfo());

        Factory factory2 = FactoryProducer.getFactory(true);
        Burger burger2 = factory2.produceBurger();
        System.out.println(burger2.getProductInfo());
        Sandwich sandwich2 = factory2.produceSandwich();
        System.out.println(sandwich2.getProductInfo());
    }
}
```

运行结果

```
chicken burger
chicken sandwich
vegeburger
vegetarian sandwich
```

4.4 单例模式

4.4.1 单例模式（Singleton Pattern）

单例模式属于创建型模式。单例模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建，单例模式提供了一种访问其唯一的对象的方式。单例模式主要解决一个全局使用的类被频繁地创建与销毁问题。

单例模式需要满足：

1. 单例类只能有一个实例。
2. 单例类必须自己创建自己的唯一实例。
3. 单例类必须给所有其它对象提供这一实例。

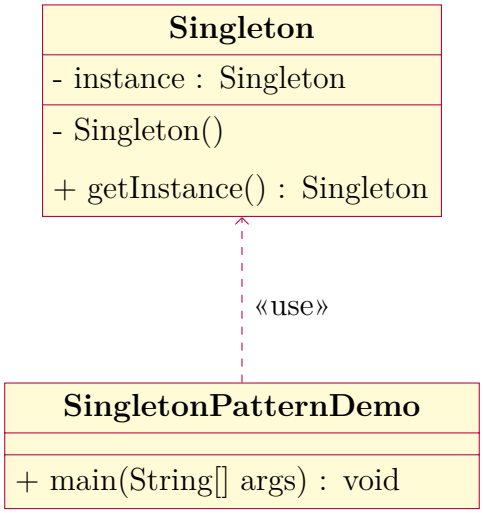


图 4.1: 单例模式

4.4.2 懒汉模式

要想让一个类只能构建一个对象，自然不能让它随便去做 new 操作，因此 Singleton 类的构造方法是私有的。instance 是 Singleton 类的静态成员，也就是单例对象。

getInstance() 是获取单例对象的方法，如果单例对象初始值是 null，还未构建，则构建单例对象并返回。这种写法属于单例模式中的懒汉模式。

懒汉模式

Singleton.java

```java
public class Singleton {
    private static Singleton instance = null;

    private Singleton() {}

    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

SingletonPatternDemo.java

```java
public class SingletonPatternDemo {
    public static void main(String[] args) {
        Singleton singleton1 = Singleton.getInstance();
        System.out.println(singleton1);
        Singleton singleton2 = Singleton.getInstance();
        System.out.println(singleton2);
    }
}
```

运行结果

```
Singleton@776ec8df
Singleton@776ec8df
```

4.4.3 饿汉模式

Singleton.java

```
1 public class Singleton {
2     private static Singleton instance = new Singleton();
3
4     private Singleton() {}
5
6     public static Singleton getInstance() {
7         return instance;
8     }
9 }
```

SingletonPatternDemo.java

```
1 public class SingletonPatternDemo {
2     public static void main(String[] args) {
3         Singleton singleton1 = Singleton.getInstance();
4         System.out.println(singleton1);
5         Singleton singleton2 = Singleton.getInstance();
6         System.out.println(singleton2);
7     }
8 }
```

运行结果

```
Singleton@776ec8df
Singleton@776ec8df
```

4.4.4 线程安全

但是以上实现的单例模式并不是线程安全的。假设 Singleton 类刚刚被初始化，instance 对象还是空，这时候两个线程同时访问 getInstance()：

```
1 public class Singleton {
```

```
    private static Singleton instance = null;

    private Singleton() {}

    public static Singleton getInstance() {
        if(instance == null) {      // <-- 线程A、B同时访问
            instance = new Singleton();
        }
        return instance;
    }
}
```

因为 instance == null，所以两个线程同时通过了条件判断，开始执行 new 操作：

```
public class Singleton {
    private static Singleton instance = null;

    private Singleton() {}

    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();     // <-- 线程A、B都new了对象
        }
        return instance;
    }
}
```

这样一来，instance 就被构建了两次。为了防止 new Singleton() 被执行多次，因此在 new 操作之前加上 synchronized 同步锁。进入 synchronized 临界区后，还要再做一次判空。因为当两个线程同时访问的时候，线程 A 构建完对象，线程 B 也可能已经通过了最初的判空验证，不做第二次判空的话，线程 B 还是会再次构建 instance 对象。像这样两次判空的机制叫做双重检测机制。

为了避免 JVM 编译器优化导致指令重排，需要在 instance 对象前面增加修饰符 volatile。

线程安全

```
1  public class Singleton {
2      private static volatile Singleton instance = null;
3
4      private Singleton() {}
5
6      public static Singleton getInstance() {
7          if(instance == null) {
8              synchronized(Singleton.class) {
9                  if(instance == null) {
10                     instance = new Singleton();
11                 }
12             }
13         }
14         return instance;
15     }
16 }
```

4.5 建造者模式

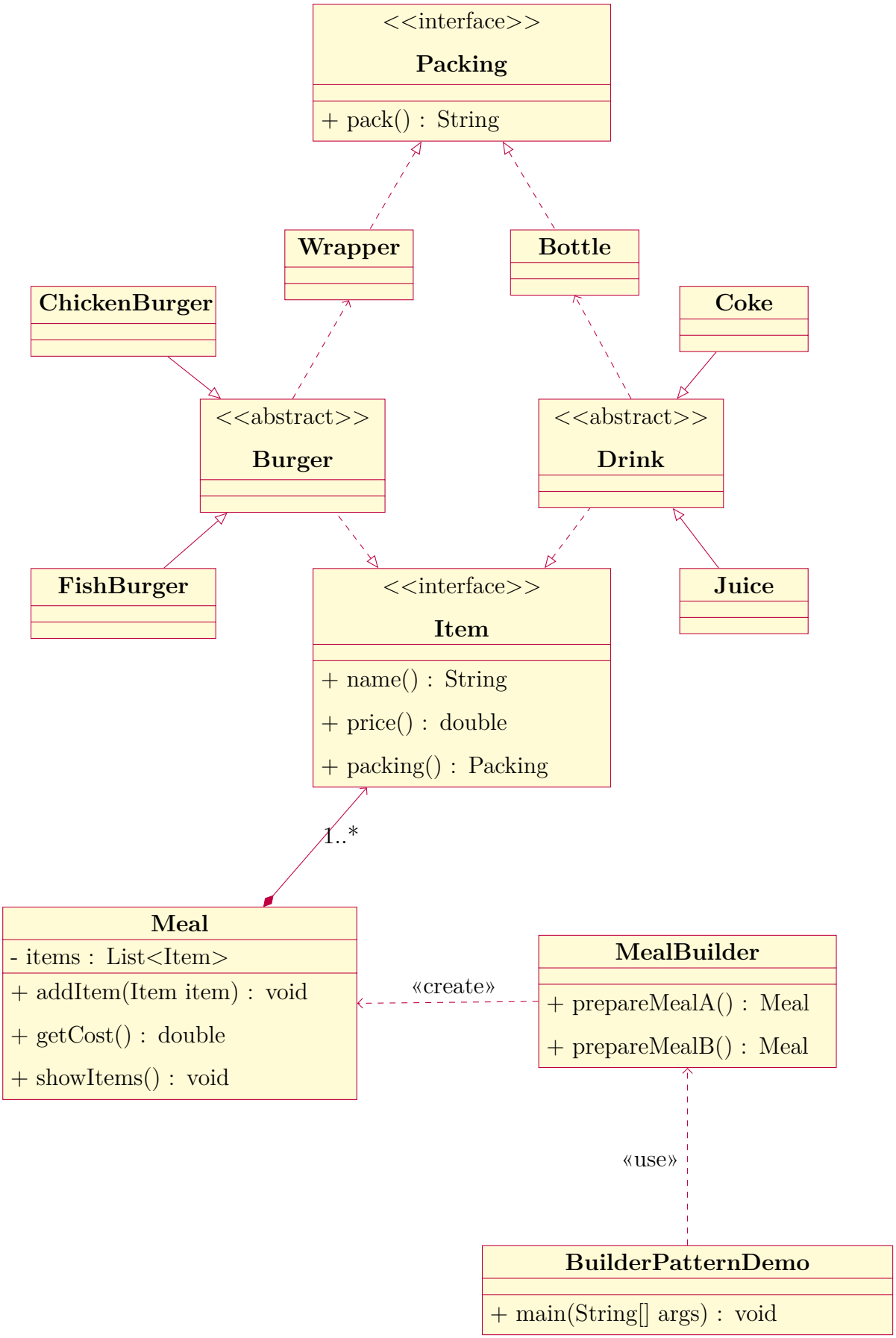
4.5.1 建造者模式（Builder Pattern）

建造者模式属于创建型模式。建造者模式将一个复杂的构建与其表示相分离，使用多个简单的对象一步一步构建成一个复杂的对象。与工厂模式的区别是在于建造者模式更加关注与零件装配的顺序。

建造者模式主要解决在软件系统中，有时候面临着一个复杂对象的创建工作，其通常由各个部分的子对象用一定的算法构成。由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定。

例如在 KFC，汉堡、可乐、薯条、鸡翅等是不变的，而其组合是经常变化的，生成出各类套餐。一个典型的套餐可以是一个汉堡和一杯饮料。汉堡可以是鸡肉汉堡或鱼肉汉堡，它们用纸盒包装；饮料可以是可乐或果汁，它们用瓶子包装。

建造者模式


<<interface>>
Packing
+ pack() : String
Wrapper
Bottle
ChickenBurger
Coke
<<abstract>>
Burger
<<abstract>>
Drink
FishBurger
<<interface>>
Item
+ name() : String
+ price() : double
+ packing() : Packing
Juice
1..*
Meal
- items : List<Item>
+ addItem(Item item) : void
+ getCost() : double
+ showItems() : void
«create»
MealBuilder
+ prepareMealA() : Meal
+ prepareMealB() : Meal
«use»
BuilderPatternDemo
+ main(String[] args) : void

Packing.java

```java
public interface Packing {
    String pack();
}
```

Wrapper.java

```java
public class Wrapper implements Packing {
    @Override
    public String pack() {
        return "wrapper";
    }
}
```

Bottle.java

```java
public class Bottle implements Packing {
    @Override
    public String pack() {
        return "bottle";
    }
}
```

Item.java

```java
public interface Item {
    String name();
    float price();
    Packing packing();
}
```

Burger.java

```java
public abstract class Burger implements Item {
    @Override
    public Packing packing() {
        return new Wrapper();
    }

    @Override
```

```
8       public abstract String name();
9
10      @Override
11      public abstract float price();
12  }
```

Drink.java

```
1   public abstract class Drink implements Item {
2       @Override
3       public Packing packing() {
4           return new Bottle();
5       }
6
7       @Override
8       public abstract String name();
9
10      @Override
11      public abstract float price();
12  }
```

ChickenBurger.java

```
1   public class ChickenBurger extends Burger {
2       @Override
3       public String name() {
4           return "chicken burger";
5       }
6
7       @Override
8       public float price() {
9           return 18.5f;
10      }
11  }
```

FishBurger.java

```
1   public class FishBurger extends Burger {
2       @Override
3       public String name() {
```

```
4           return "fish burger";
5       }
6
7       @Override
8       public float price() {
9           return 14.5f;
10      }
11  }
```

Coke.java

```
1   public class Coke extends Drink {
2       @Override
3       public String name() {
4           return "coke";
5       }
6
7       @Override
8       public float price() {
9           return 8.0f;
10      }
11  }
```

Juice.java

```
1   public class Juice extends Drink {
2       @Override
3       public String name() {
4           return "juice";
5       }
6
7       @Override
8       public float price() {
9           return 10.5f;
10      }
11  }
```

Meal.java

```java
import java.util.ArrayList;
import java.util.List;

public class Meal {
    private List<Item> items = new ArrayList<Item>();

    public void addItem(Item item) {
        items.add(item);
    }

    public float getCost() {
        float cost = 0.0f;
        for(Item item : items) {
            cost += item.price();
        }
        return cost;
    }

    public void showItems() {
        for(Item item : items) {
            System.out.println(
                "[" + item.packing().pack() + "] "
                + item.name() +
                " (" + item.price() + ")"
            );
        }
    }
}
```

MealBuilder.java

```java
public class MealBuilder {
    public Meal prepareMealA() {
        Meal meal = new Meal();
        meal.addItem(new ChickenBurger());
        meal.addItem(new Coke());
        return meal;
    }
```

```
8
9       public Meal prepareMealB() {
10          Meal meal = new Meal();
11          meal.addItem(new FishBurger());
12          meal.addItem(new Juice());
13          return meal;
14      }
15  }
```

BuilderPatternDemo.java

```
1   public class BuilderPatternDemo {
2       public static void main(String[] args) {
3           MealBuilder mealBuilder = new MealBuilder();
4
5           Meal mealA = mealBuilder.prepareMealA();
6           System.out.println("Meal A");
7           mealA.showItems();
8           System.out.println("====================");
9           System.out.println("Total cost: " + mealA.getCost());
10          System.out.println();
11
12          Meal mealB = mealBuilder.prepareMealA();
13          System.out.println("Meal B");
14          mealB.showItems();
15          System.out.println("====================");
16          System.out.println("Total cost: " + mealB.getCost());
17          System.out.println();
18      }
19  }
```

运行结果

```
Meal A
[wrapper] chicken burger (18.5)
[bottle] coke (8.0)
====================
Total cost: 26.5

Meal B
[wrapper] chicken burger (18.5)
[bottle] coke (8.0)
====================
Total cost: 26.5
```

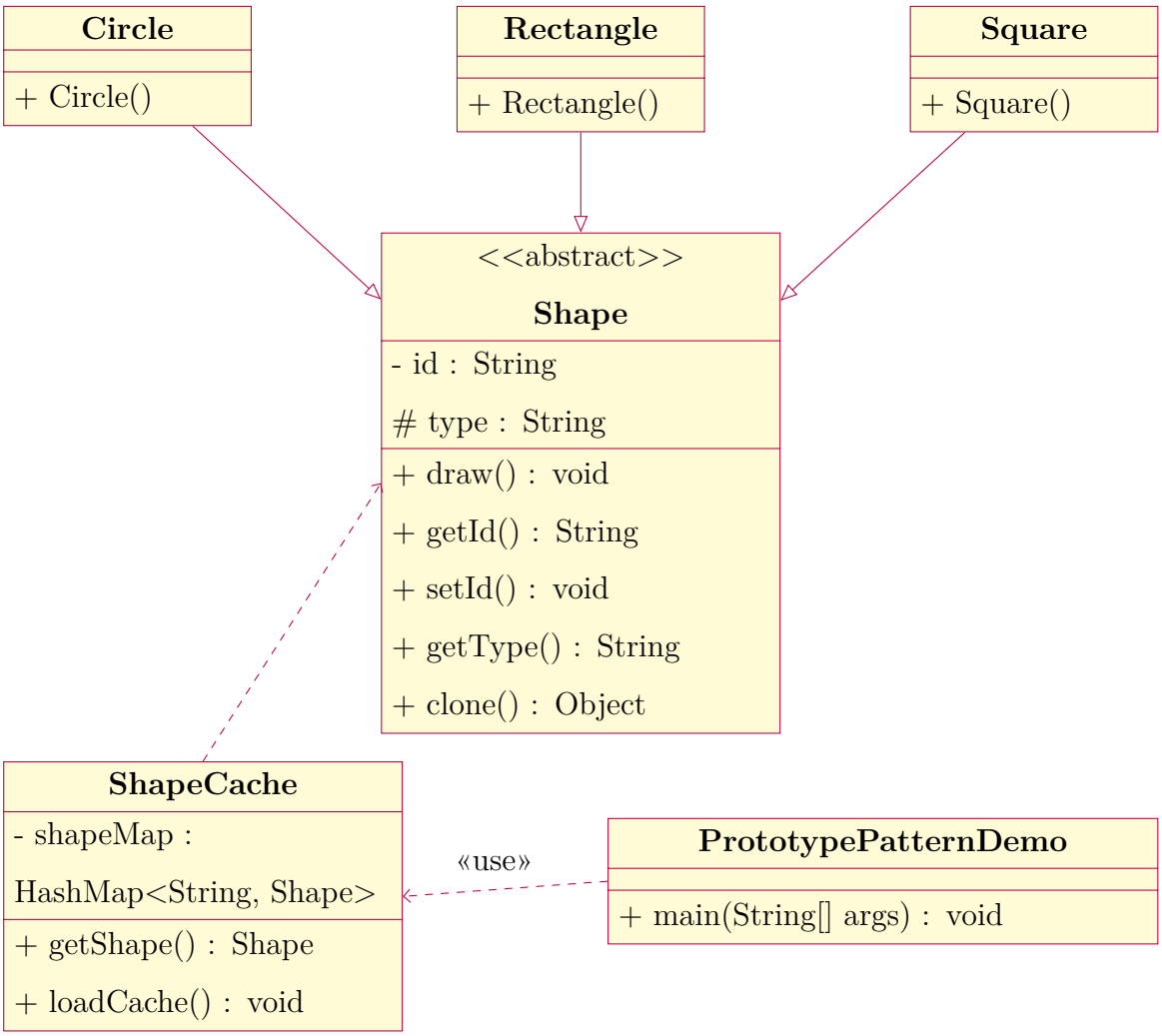
4.6 原型模式

4.6.1 原型模式（Prototype Pattern）

原型模式属于创建型模式。原型模式用于创建重复对象，同时又能保证性能。原型模式使用原型实例指定创建对象的种类，并且通过拷贝原型对象创建新对象。

原型模式提供了一个通过已存在对象进行新对象创建的接口。在初始化的信息不发生变化的情况下，克隆是最好的办法，既隐藏了对象创建的细节，又大大提高了性能。因为每次都需要执行一次比较复杂的构造函数，那么多次的执行初始化操作就非常低效。

原型模式



Shape.java

```java
public abstract class Shape implements Cloneable {
    private String id;
    protected String type;

    public abstract void draw();

    public String getType() {
        return type;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public Object clone() {
        Object clone = null;
        try {
            clone = super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return clone;
    }
}
```

Circle.java

```java
public class Circle extends Shape {
    public Circle() {
        type = "Circle";
    }

    @Override
```

```
7       public void draw() {
8           System.out.println("Inside Circle::draw() method.");
9       }
10  }
```

Rectangle.java

```
1   public class Rectangle extends Shape {
2       public Rectangle() {
3           type = "Rectangle";
4       }
5
6       @Override
7       public void draw() {
8           System.out.println("Inside Rectangle::draw() method.");
9       }
10  }
```

Square.java

```
1   public class Square extends Shape {
2       public Square() {
3           type = "Square";
4       }
5
6       @Override
7       public void draw() {
8           System.out.println("Inside Square::draw() method.");
9       }
10  }
```

ShapeCache.java

```
1   import java.util.Hashtable;
2
3   public class ShapeCache {
4       private static Hashtable<String, Shape> shapeMap
5                       = new Hashtable<String, Shape>();
6
7       public static Shape getShape(String shapeId) {
```

```
        Shape cachedShape = shapeMap.get(shapeId);
        return (Shape) cachedShape.clone();
    }

    // for each shape run database query and create shape
    // shapeMap.put(shapeKey, shape);
    // for example, we are adding three shapes
    public static void loadCache() {
        Circle circle = new Circle();
        circle.setId("1");
        shapeMap.put(circle.getId(),circle);

        Square square = new Square();
        square.setId("2");
        shapeMap.put(square.getId(),square);

        Rectangle rectangle = new Rectangle();
        rectangle.setId("3");
        shapeMap.put(rectangle.getId(), rectangle);
    }
}
```

PrototypePatternDemo.java

```
public class PrototypePatternDemo {
    public static void main(String[] args) {
        ShapeCache.loadCache();

        Shape clonedShape = (Shape) ShapeCache.getShape("1");
        System.out.println("Shape : " + clonedShape.getType());

        Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
        System.out.println("Shape : " + clonedShape2.getType());

        Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
        System.out.println("Shape : " + clonedShape3.getType());
    }
}
```

运行结果

```
Shape: Circle
Shape: Square
Shape: Rectangle
```

4.7 适配器模式

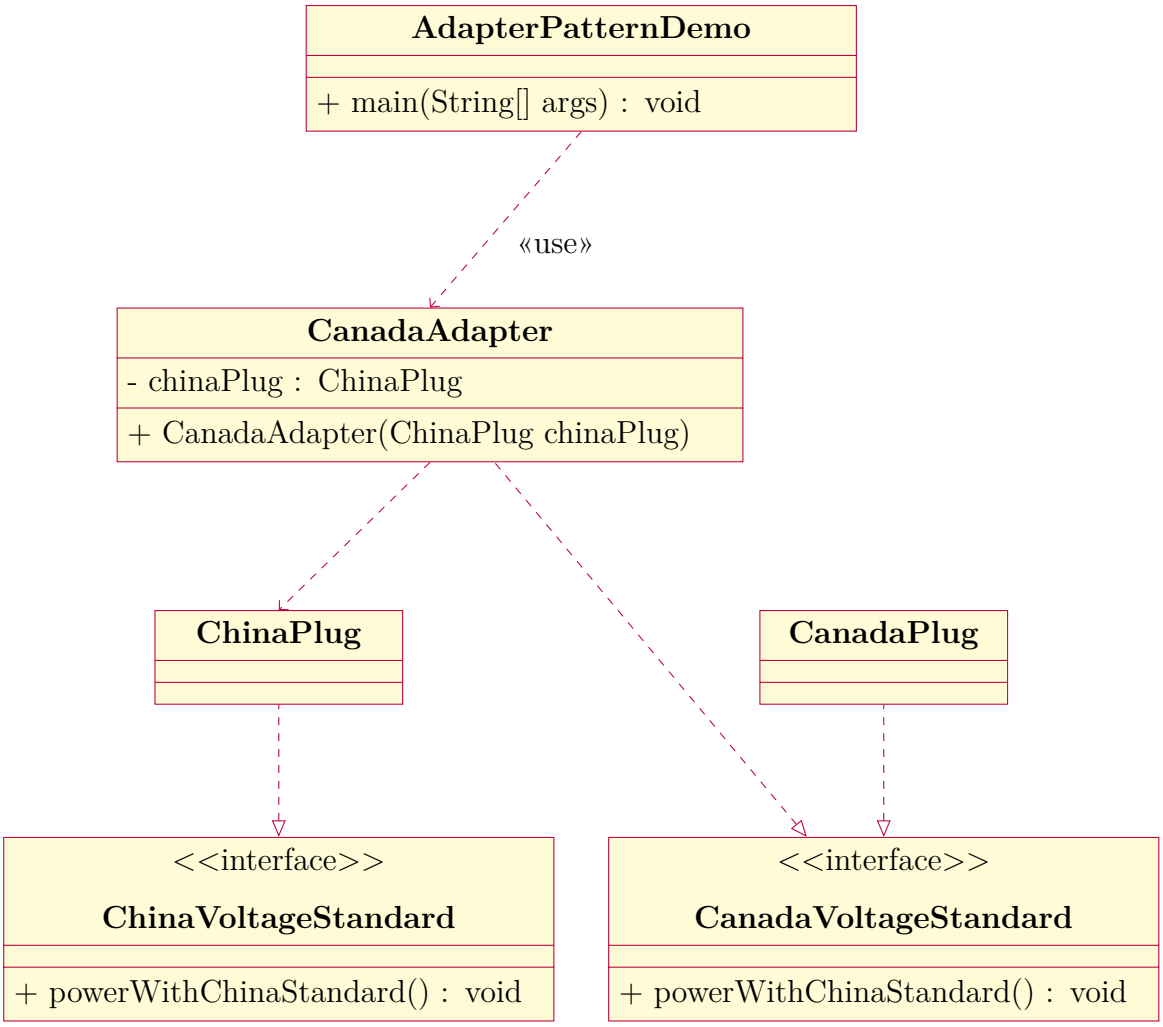
4.7.1 适配器模式（Adapter Pattern）

适配器在生活中无处不在，比如电脑、读卡器、电源的转换头，它们的共同点就是接口标准不一样，需要通过适配器转换后才能使用。就拿读卡器来说，存储卡的接口只能适配相机或者手机的卡槽。而电脑普遍为 USB 接口，那么如何在电脑上使用存储卡呢？可以用读卡器，一头卡槽能够插入存储卡，另一头 USB 可以插在电脑上。还有个例子就是变压器，中国电器的电压是 220V，加拿大的电压是 110V，因此需要变压器将 110V 电压转换为 220V 电压才能使用。通过适配器可以解决接口不兼容的问题。

图 4.2: 电压转换

适配器模式是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能，将一个类的接口转换成希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

适配器模式



ChinaVoltageStandard.java

```java
public interface ChinaVoltageStandard {
    void powerWithChinaStandard();
}
```

ChinaPlug.java

```java
public class ChinaPlug implements ChinaVoltageStandard {
    @Override
    public void powerWithChinaStandard() {
        System.out.println("Using China standard (220V)...");
    }
}
```

CanadaVoltageStandard.java

```java
public interface CanadaVoltageStandard {
    void powerWithCanadaStandard();
}
```

CanadaPlug.java

```java
public class CanadaPlug implements CanadaVoltageStandard {
    @Override
    public void powerWithCanadaStandard() {
        System.out.println("Using Canada standard (110V)...");
    }
}
```

CanadaAdapter.java

```java
public class CanadaAdapter implements CanadaVoltageStandard {
    private ChinaPlug chinaPlug;

    public CanadaAdapter(ChinaPlug chinaPlug) {
        this.chinaPlug = chinaPlug;
    }

    @Override
    public void powerWithCanadaStandard() {
        chinaPlug.powerWithChinaStandard();
    }
}
```

AdapterPatternDemo.java

```java
public class AdapterPatternDemo {
    public static void main(String[] args) {
        CanadaPlug canadaPlug = new CanadaPlug();
        canadaPlug.powerWithCanadaStandard();

        ChinaPlug chinaPlug = new ChinaPlug();
        CanadaAdapter adapter = new CanadaAdapter(chinaPlug);
        adapter.powerWithCanadaStandard();
    }
}
```

运行结果

```
Using Canada standard (110V)...
Using China standard (220V)...
```

4.8 桥接模式

4.8.1 桥接模式（Bridge Pattern）

桥接模式是一种结构型设计模式，用于将一个大类或一系列紧密相关的类拆分为抽象和实现两个独立的层次结构，从而能在开发时分别使用，而降低了抽象和实现这两个可变维度的耦合度。

桥接模式通过将继承改为组合的方式来解决这个问题，也就是抽取其中一个维度并使之成为独立的类层次，这样就可以在初始类中引用这个新层次的对象，从而使得一个类不必拥有所有的状态和行为。

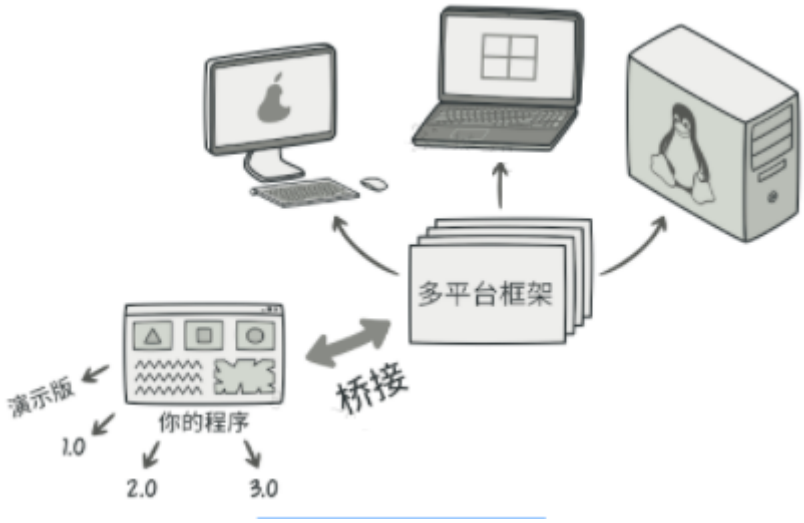
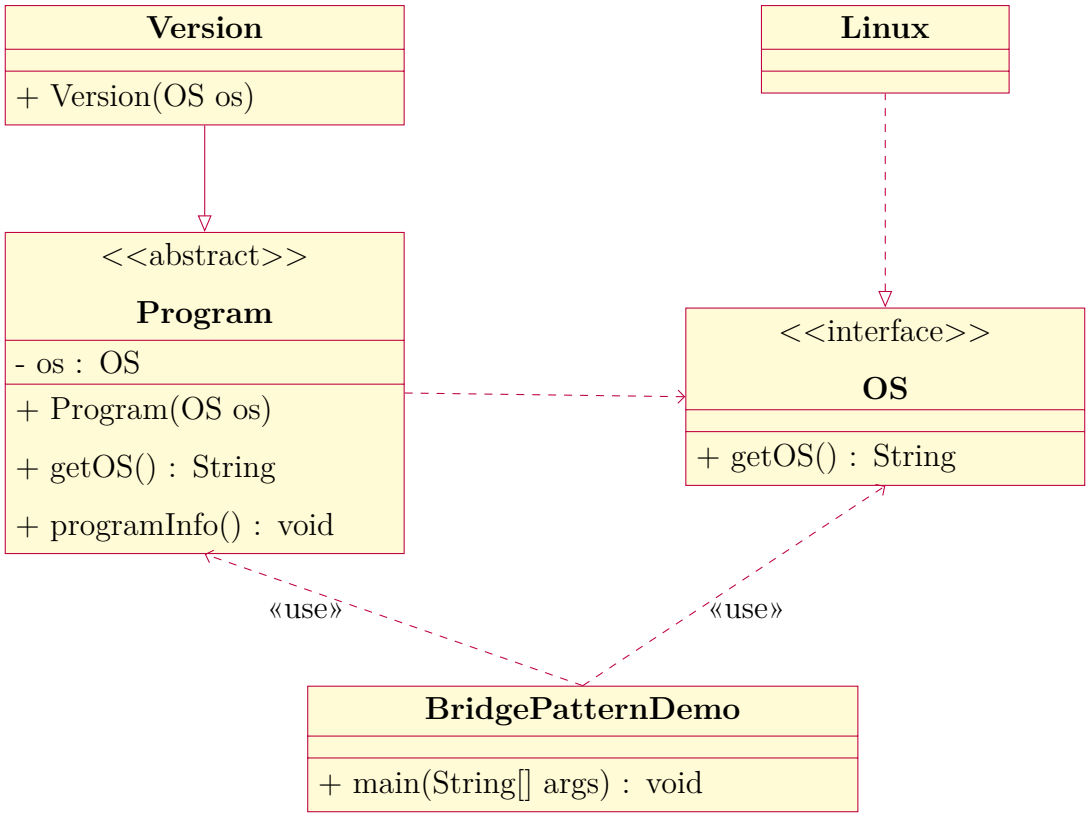


图 4.3: 跨平台程序

桥接模式



OS.java

```java
public interface OS {
    String getOS();
}
```

Linux.java

```java
public class Linux implements OS {
    @Override
    public String getOS() {
        return "Linux";
    }
}
```

Program.java

```java
public abstract class Program {
    private OS os;

    public Program(OS os) {
```

```
5           this.os = os;
6       }
7
8       public String getOS() {
9           return os.getOS();
10      }
11
12      public abstract void programInfo();
13  }
```

Version.java

```
1   public class Version extends Program {
2       public Version(OS os) {
3           super(os);
4       }
5
6       @Override
7       public void programInfo() {
8           System.out.println(
9               "Program v1.0 running on " + super.getOS()
10          );
11      }
12  }
```

BridgePatternDemo.java

```
1   public class BridgePatternDemo {
2       public static void main(String[] args) {
3           OS os = new Linux();
4           Program program = new Version1(os);
5           program.programInfo();
6       }
7   }
```

运行结果

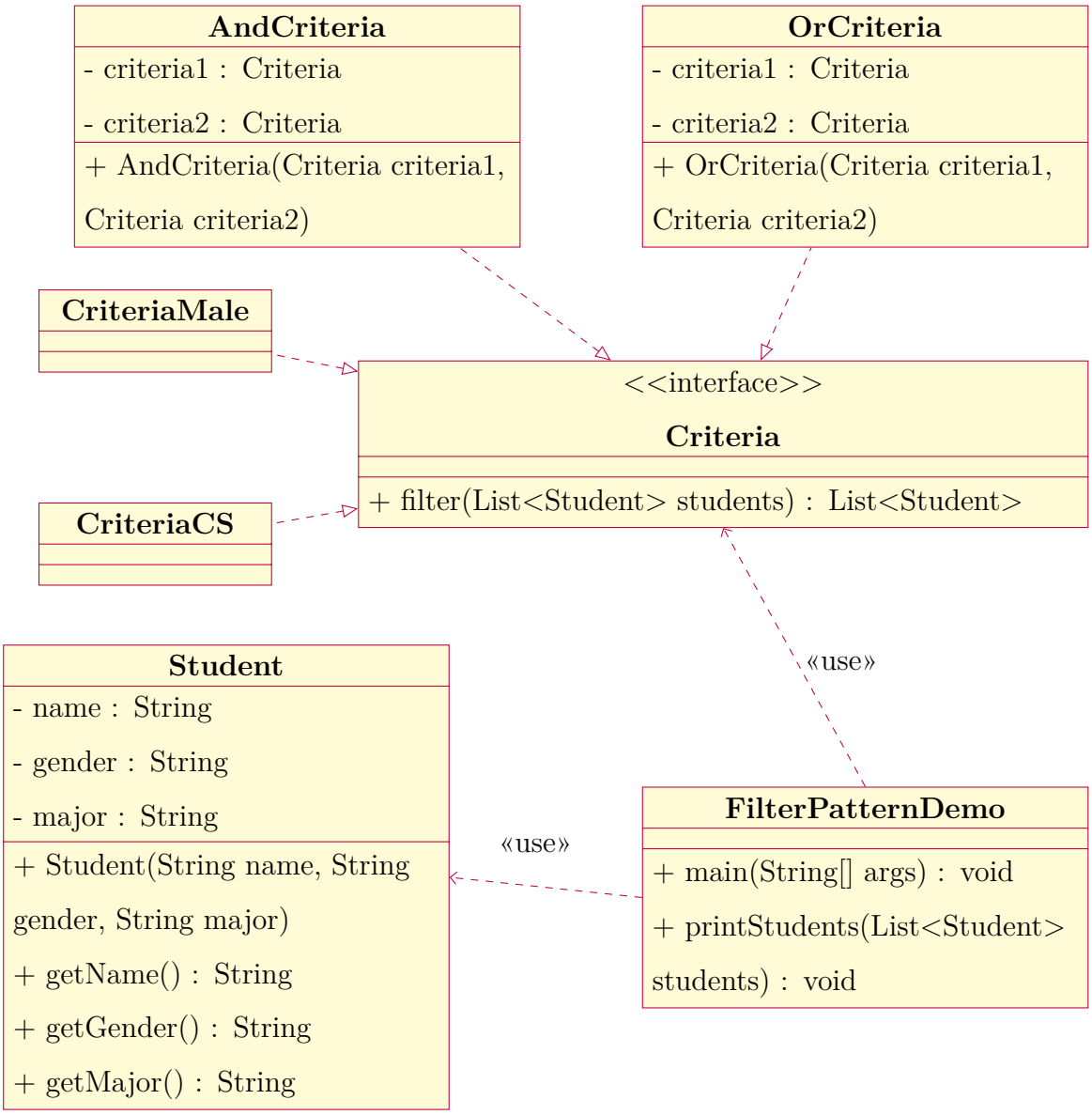
```
Program v1.0 running on Linux
```

4.9 过滤器模式

4.9.1 过滤器模式（Filter Pattern）

过滤器模式也称标准模式（Criteria Pattern），这种模式允许开发人员使用不同的标准来过滤一组对象，通过逻辑运算以解耦的方式把它们连接起来。过滤器模式属于结构型模式，它结合多个标准来获得单一标准。

过滤器模式



Student.java

```java
public class Student {
    private String name;
    private String gender;
    private String major;

    public Student(String name, String gender, String major) {
        this.name = name;
        this.gender = gender;
        this.major = major;
    }

    public String getName() {
        return name;
    }

    public String getGender() {
        return gender;
    }

    public String getMajor() {
        return major;
    }
}
```

Criteria.java

```java
import java.util.List;

public interface Criteria {
    List<Student> filter(List<Student> students);
}
```

CriteriaMale.java

```java
import java.util.ArrayList;
import java.util.List;

public class CriteriaMale implements Criteria {
```

```
    @Override
    public List<Student> filter(List<Student> students) {
        List<Student> maleStudents = new ArrayList<Student>();
        for(Student student : students) {
            if(student.getGender().equalsIgnoreCase("MALE")) {
                maleStudents.add(student);
            }
        }
        return maleStudents;
    }
}
```

CriteriaCS.java

```
import java.util.ArrayList;
import java.util.List;

public class CriteriaCS implements Criteria {
    @Override
    public List<Student> filter(List<Student> students) {
        List<Student> csStudents = new ArrayList<Student>();
        for(Student student : students) {
            if(student.getMajor().equalsIgnoreCase("CS")) {
                csStudents.add(student);
            }
        }
        return csStudents;
    }
}
```

AndCriteria.java

```
import java.util.List;

public class AndCriteria implements Criteria {
    private Criteria criteria1;
    private Criteria criteria2;

    public AndCriteria(Criteria criteria1, Criteria criteria2) {
```

```
        this.criteria1 = criteria1;
        this.criteria2 = criteria2;
    }

    @Override
    public List<Student> filter(List<Student> students) {
        return criteria2.filter(criteria1.filter(students));
    }
}
```

OrCriteria.java

```
import java.util.List;

public class OrCriteria implements Criteria {
    private Criteria criteria1;
    private Criteria criteria2;

    public OrCriteria(Criteria criteria1, Criteria criteria2) {
        this.criteria1 = criteria1;
        this.criteria2 = criteria2;
    }

    @Override
    public List<Student> filter(List<Student> students) {
        List<Student> list1 = criteria1.filter(students);
        List<Student> list2 = criteria2.filter(students);
        list1.removeAll(list2);
        list1.addAll(list2);
        return list1;
    }
}
```

FilterPatternDemo.java

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

```

```
5  public class FilterPatternDemo {
6      public static void main(String[] args) {
7          List<Student> students = new ArrayList<Student>();
8          students.add(new Student("Terry", "male", "CS"));
9          students.add(new Student("Bob", "male", "Math"));
10         students.add(new Student("Anna", "female", "Food Science"));
11         students.add(new Student("Lily", "female", "CS"));
12         students.add(new Student("Alice", "female", "Art"));
13
14         Criteria male = new CriteriaMale();
15         Criteria cs = new CriteriaCS();
16         Criteria maleAndCs = new AndCriteria(male, cs);
17         Criteria maleOrCs = new OrCriteria(male, cs);
18
19         System.out.println("MALE:");
20         printStudents(male.filter(students));
21
22         System.out.println("CS:");
23         printStudents(cs.filter(students));
24
25         System.out.println("Male and CS:");
26         printStudents(maleAndCs.filter(students));
27
28         System.out.println("Male or CS:");
29         printStudents(maleOrCs.filter(students));
30     }
31
32     public static void printStudents(List<Student> students) {
33         System.out.print("\t");
34         for(Student student : students) {
35             System.out.print(student.getName() + " ");
36         }
37         System.out.println();
38     }
39 }
```

运行结果

```
MALE:
    Terry Bob
CS:
    Terry Lily
Male and CS:
    Terry
Male or CS:
    Bob Terry Lily
```

4.10 组合模式

4.10.1 组合模式（Composite Pattern）

在现实生活中，存在很多“整体-部分”的关系，例如，大学的学院与部门、公司的分公司与部门。在软件开发中也是这样，例如，文件夹与文件、容器与控件等。对这些简单对象与复合对象的处理，如果用组合模式来实现会很方便。

组合模式属于结构型模式，用于把一组相似的对象当作一个单一的对象，组合模式将对象组合成树形结构以表示“整体-部分”的层次结构。组合模式模糊了简单元素和复杂元素的概念，可以像处理简单元素一样来处理复杂元素，从而使得复杂元素的内部结构解耦。

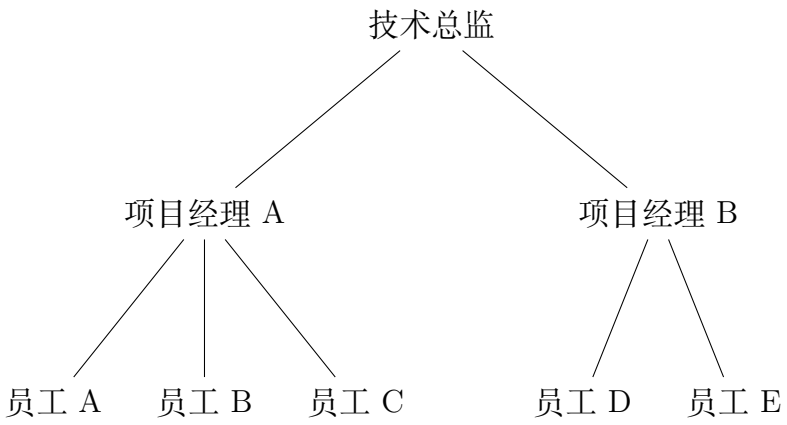
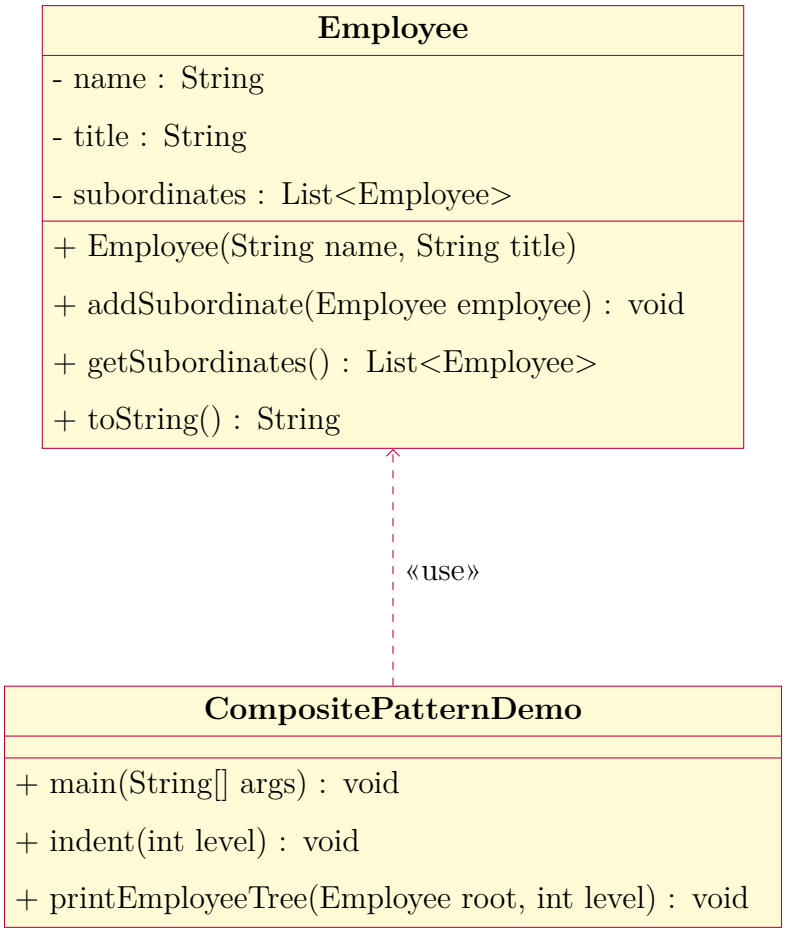


图 4.4: 职级关系

组合模式



Employee.java

```java
import java.util.ArrayList;
import java.util.List;

public class Employee {
    private String name;
    private String title;
    private List<Employee> subordinates;

    public Employee(String name, String title) {
        this.name = name;
        this.title = title;
        this.subordinates = new ArrayList<Employee>();
    }

    public void addSubordinate(Employee employee) {
        subordinates.add(employee);
```

```
    }

    public List<Employee> getSubordinates() {
        return subordinates;
    }

    @Override
    public String toString() {
        return "Name: " + name + "\t\tTitle: " + title;
    }
}
```

CompositePatternDemo.java

```
public class CompositePatternDemo {
    public static void main(String[] args) {
        Employee CTO = new Employee("Terry", "CTO");

        Employee pm1 = new Employee("Henry", "Project Manager");
        Employee pm2 = new Employee("Bob", "Project Manager");

        Employee e1 = new Employee("Lily", "Java Engineer");
        Employee e2 = new Employee("Alice", "Python Engineer");
        Employee e3 = new Employee("Eric", "C Engineer");

        pm1.addSubordinate(e1);
        pm1.addSubordinate(e2);
        pm2.addSubordinate(e3);

        CTO.addSubordinate(pm1);
        CTO.addSubordinate(pm2);

        printEmployeeTree(CTO, 0);
    }

    public static void indent(int level) {
        for(int i = 0; i < level; i++) {
            System.out.print("\t");
```

```
25          }
26      }
27
28      public static void printEmployeeTree(Employee root, int level) {
29          System.out.println(root);
30          for(Employee e : root.getSubordinates()) {
31              indent(++level);
32              if(!e.getSubordinates().isEmpty()) {
33                  printEmployeeTree(e, level);
34              } else {
35                  System.out.println(e);
36              }
37              level--;
38          }
39      }
40  }
```

运行结果

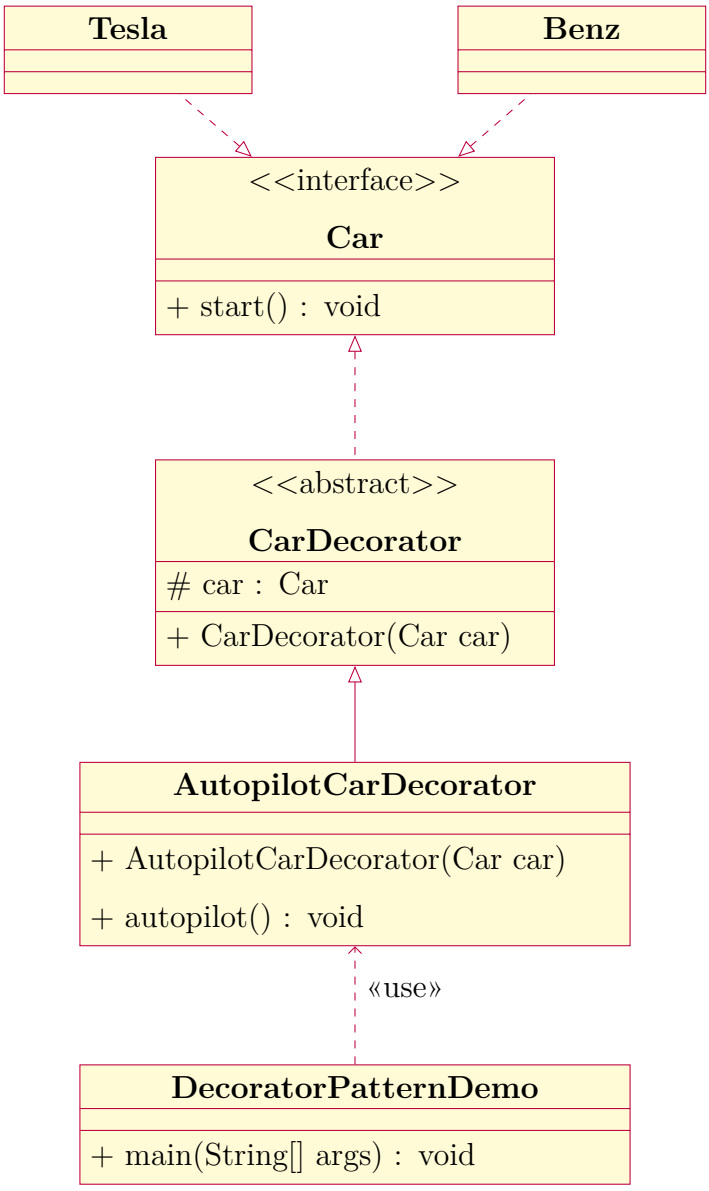
```
Name: Terry Title: CTO
    Name: Henry Title: Project Manager
        Name: Lily Title: Java Engineer
        Name: Alice Title: Python Engineer
    Name: Bob Title: Project Manager
        Name: Eric Title: C Engineer
```

4.11 装饰器模式

4.11.1 装饰器模式（Decorator Pattern）

装饰器模式允许向一个现有的对象添加新的功能，同时又不改变其结构装饰器模式属于结构型模式，它用于创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。

装饰器模式



Car.java

```java
public interface Car {
    void start();
}
```

Tesla.java

```java
public class Tesla implements Car {
    @Override
    public void start() {
        System.out.println("Tesla start engine...");
    }
}
```

Benz.java

```java
public class Benz implements Car {
    @Override
    public void start() {
        System.out.println("Benz start enginee...");
    }
}
```

CarDecorator.java

```java
public abstract class CarDecorator implements Car {
    protected Car car;

    public CarDecorator(Car car) {
        this.car = car;
    }

    public void start() {
        car.start();
    }
}
```

AutopilotCarDecorator.java

```java
public class AutopilotCarDecorator extends CarDecorator {
```

```
    public AutopilotCarDecorator(Car car) {
        super(car);
    }

    public void autopilot() {
        System.out.println("Start autopilot mode...");
    }

    @Override
    public void start() {
        car.start();
        autopilot();
    }
}
```

DecoratorPatternDemo.java

```
public class DecoratorPatternDemo {
    public static void main(String[] args) {
        Car benz = new Benz();
        benz.start();
        System.out.println("====================");
        Car autopilotTesla = new AutopilotCarDecorator(new Tesla());
        autopilotTesla.start();
    }
}
```

运行结果

```
Benz start enginee...
====================
Tesla start engine...
Start autopilot mode...
```

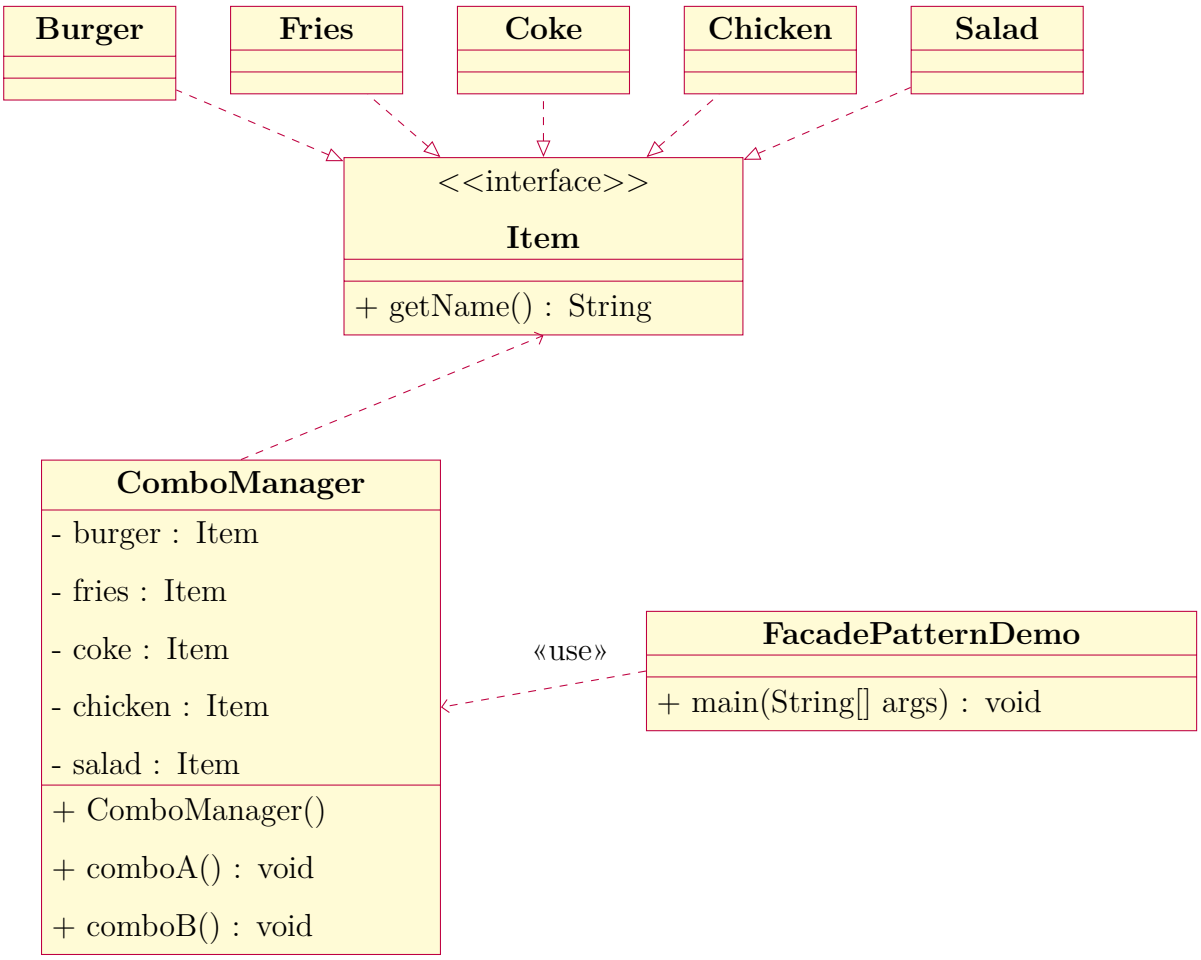
4.12 外观模式

4.12.1 外观模式（Facade Pattern）

外观模式属于结构型模式，它为子系统的众多接口提供了统一的高层接口，使子系统更容易使用。

例如 KFC 有众多基础产品，比如鸡翅、汉堡、薯条、沙拉、可乐等。这些琳琅满目的菜品虽好，但有些顾客犯了选择困难症，不知道该选什么好。于是 KFC 对这些菜品做了一定的组合，推出了各种各样的套餐。比如 A 套餐：汉堡 + 薯条 + 可乐；B 套餐：汉堡 + 鸡翅 + 沙拉。

外观模式



Item.java

```java
public interface Item {
    String getName();
}
```

Burger.java

```java
public class Burger implements Item {
    @Override
    public String getName() {
        return "Burger";
    }
}
```

Fries.java

```java
public class Fries implements Item {
    @Override
    public String getName() {
        return "Fries";
    }
}
```

Coke.java

```java
public class Coke implements Item {
    @Override
    public String getName() {
        return "Coke";
    }
}
```

Chicken.java

```java
public class Chicken implements Item {
    @Override
    public String getName() {
        return "Chicken";
    }
}
```

Salad.java

```java
public class Salad implements Item {
    @Override
    public String getName() {
        return "Salad";
    }
}
```

ComboManager.java

```java
public class ComboManager {
    private Item burger;
    private Item fries;
    private Item coke;
    private Item chicken;
    private Item salad;

    public ComboManager() {
        burger = new Burger();
        fries = new Fries();
        coke = new Coke();
        chicken = new Chicken();
        salad = new Salad();
    }

    public void comboA() {
        System.out.println("Combo A: "
                           + burger.getName() + ", "
                           + fries.getName() + ", "
                           + coke.getName());
    }

    public void comboB() {
        System.out.println("Combo B: "
                + burger.getName() + ", "
                + chicken.getName() + ", "
                + salad.getName());
    }
```

```
}
```

FacadePatternDemo.java

```java
public class FacadePatternDemo {
    public static void main(String[] args) {
        ComboManager comboManager = new ComboManager();
        comboManager.comboA();
        comboManager.comboB();
    }
}
```

运行结果

```
Combo A: Burger, Fries, Coke
Combo B: Burger, Chicken, Salad
```

4.13 享元模式

4.13.1 享元模式（Flyweight Pattern）

享元模式主要用于减少创建对象的数量，以减少内存占用和提高性能。享元模式尝试重用现有的同类对象，如果未找到匹配的对象，则创建新对象。享元模式属于结构型模式，它提供了减少对象数量从而改善应用所需的对象结构的方式。

享元模式

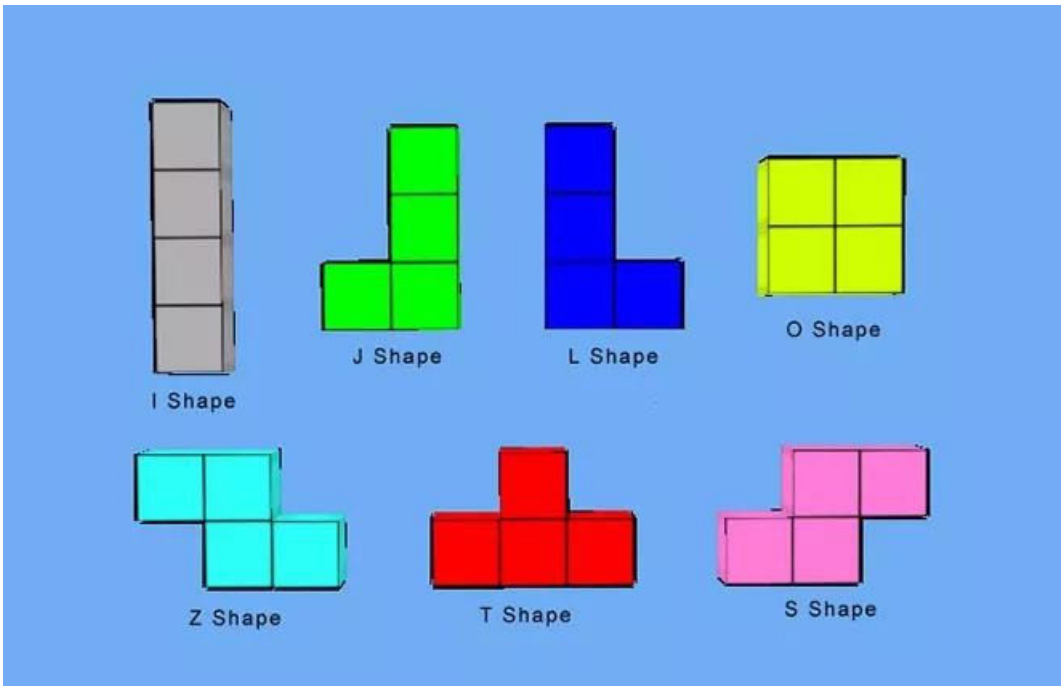
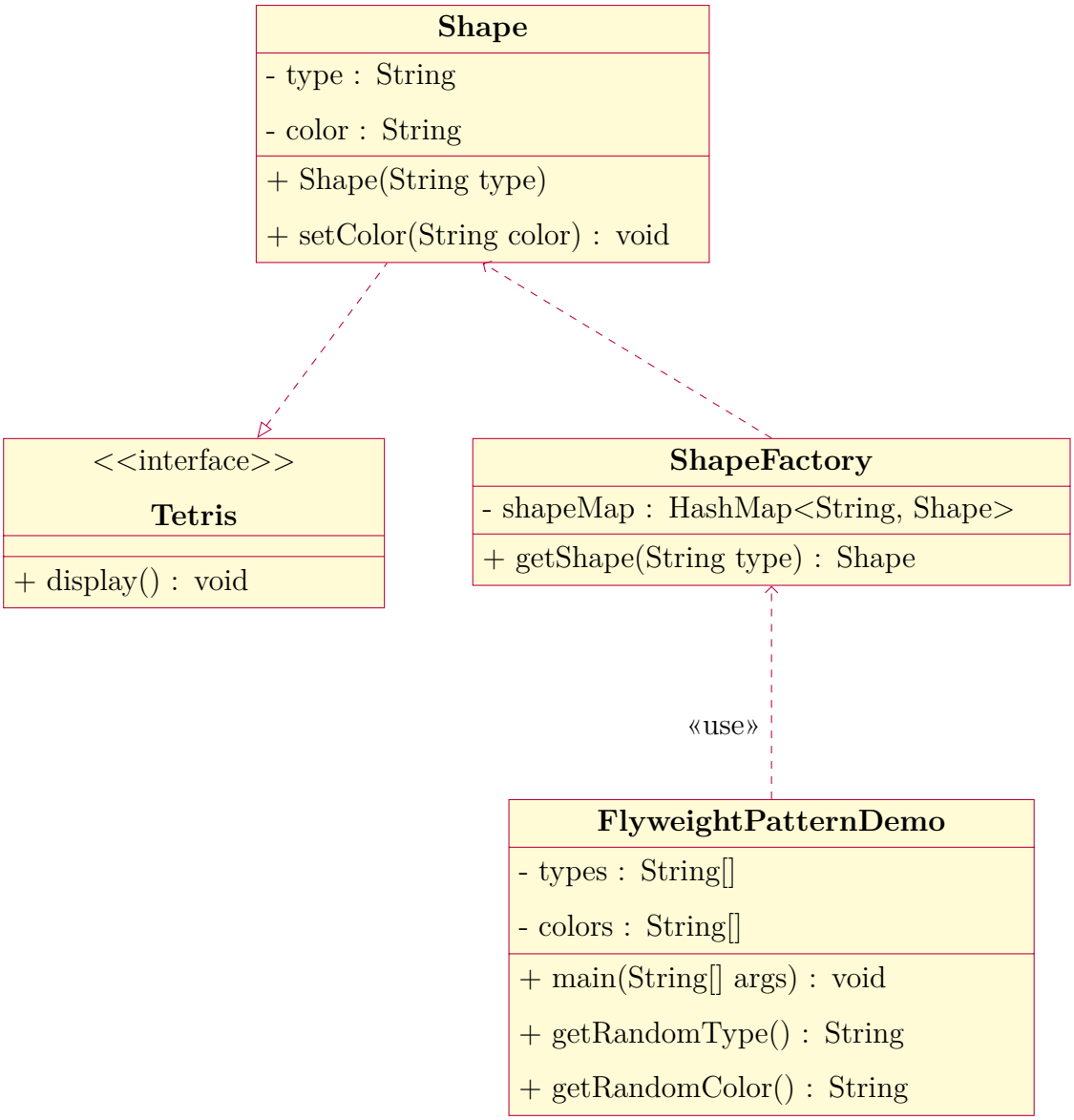


图 4.5: 俄罗斯方块



Tetris.java

```
1 public interface Tetris {
2     void display();
3 }
```

Shape.java

```
1 public class Shape implements Tetris {
2     private String type;
3     private String color;
4
5     public Shape(String type) {
```

```
        this.type = type;
    }

    public void setColor(String color) {
        this.color = color;
    }

    @Override
    public void display() {
        System.out.println("Shape: " + type + "\tColor: " + color);
    }
}
```

ShapeFactory.java

```
import java.util.HashMap;

public class ShapeFactory {
    private static HashMap<String, Shape> shapeMap = new HashMap<>();

    public static Shape getShape(String type) {
        Shape shape = shapeMap.get(type);
        if(shape == null) {
            shape = new Shape(type);
            shapeMap.put(type, shape);
            System.out.println("Shape " + type + " created");
        }
        return shape;
    }
}
```

FlyweightPatternDemo.java

```
public class FlyweightPatternDemo {
    private static final String[] types = {
            "I", "J", "L",
            "O", "S", "T", "Z"
    };

```

```
7       private static final String[] colors = {
8               "grey", "green", "blue",
9               "yellow", "indigo", "red", "pink"
10      };
11
12      public static void main(String[] args) {
13          for(int i = 0; i < 10; i++) {
14              Shape shape = ShapeFactory.getShape(getRandomType());
15              shape.setColor(getRandomColor());
16              shape.display();
17          }
18      }
19
20      public static String getRandomType() {
21          return types[(int)(Math.random() * types.length)];
22      }
23
24      public static String getRandomColor() {
25          return colors[(int)(Math.random() * colors.length)];
26      }
27  }
```

运行结果

```
Shape Z created
Shape: Z Color: red
Shape O created
Shape: O Color: blue
Shape: O Color: pink
Shape: Z Color: green
Shape I created
Shape: I Color: grey
Shape: I Color: grey
Shape: Z Color: green
Shape J created
Shape: J Color: pink
Shape S created
Shape: S Color: green
Shape: J Color: green
```

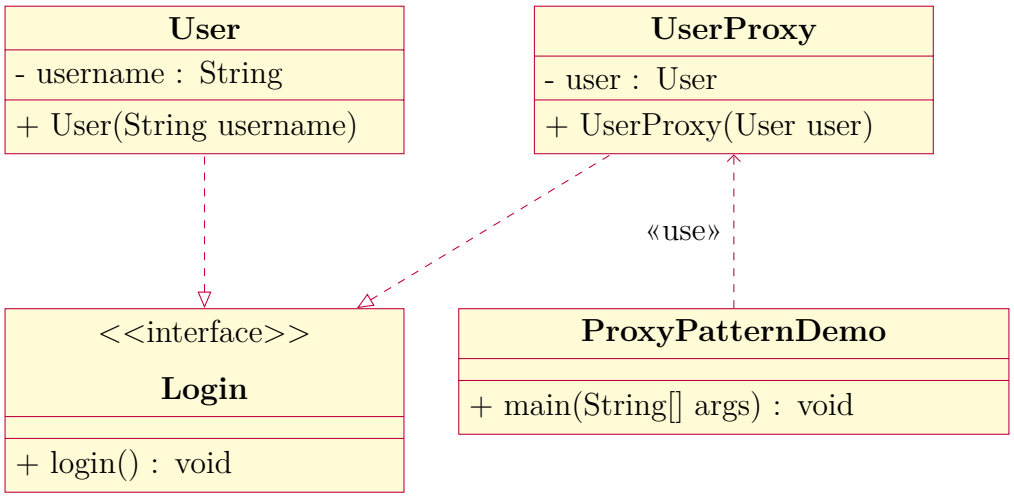
4.14 代理模式

4.14.1 代理模式（Proxy Pattern）

代理模式属于结构型模式，其核心是在被调用方和调用方之间增加一个中介者的角色，也就是代理。

在现实生活中，同样需要各种各样的代理人，例如房屋中介、留学中介等。引入代理的意义在于代理专门负责完成专业的事情，如果试图省去代理，只会带来更多的麻烦和成本。

代理模式



Login.java

```
1 public interface Login {
2     void login();
3 }
```

User.java

```
1 public class User implements Login {
2     private String username;
3 
4     public User(String username) {
```

```
        this.username = username;
    }

    @Override
    public void login() {
        System.out.println(username + " login succeed.");
    }
}
```

UserProxy.java

```
public class UserProxy implements Login {
    private User user;

    public UserProxy(User user) {
        this.user = user;
    }

    @Override
    public void login() {
        System.out.println("Check username / password...");
        user.login();
    }
}
```

ProxyPatternDemo.java

```
public class ProxyPatternDemo {
    public static void main(String[] args) {
        Login login = new UserProxy(new User("Admin"));
        login.login();
    }
}
```

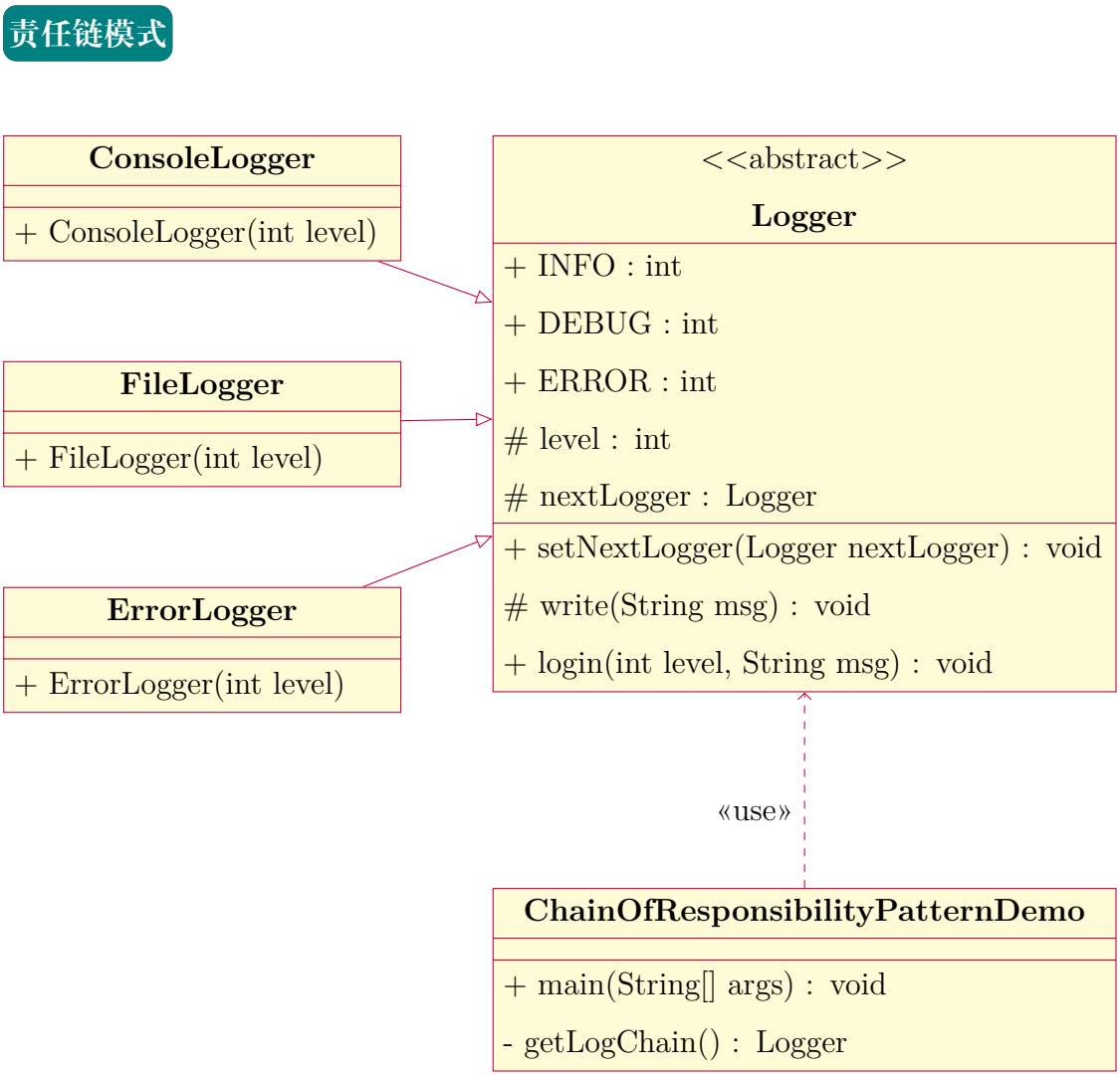
运行结果

```
Check username / password...
Admin login succeed.
```

4.15 责任链模式

4.15.1 责任链模式（Chain of Responsibility Pattern）

责任链模式属于行为型模式，该模式构造一系列分别担当不同职责的类对象来共同完成一个任务，这些类的对象之间像链条一样紧密相连。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。



Logger.java

```
1 public abstract class Logger {
2     public static int INFO = 1;
3     public static int DEBUG = 2;
4     public static int ERROR = 3;
```

```
5
6       protected int level;
7       protected Logger nextLogger;
8
9       public void setNextLogger(Logger nextLogger) {
10          this.nextLogger = nextLogger;
11      }
12
13      protected abstract void write(String msg);
14
15      public void log(int level, String msg) {
16          if(this.level <= level) {
17              write(msg);
18          }
19          if(nextLogger != null) {
20              nextLogger.log(level, msg);
21          }
22      }
23  }
```

ConsoleLogger.java

```
1   public class ConsoleLogger extends Logger {
2       public ConsoleLogger(int level) {
3           this.level = level;
4       }
5
6       @Override
7       protected void write(String msg) {
8           System.out.println("Console: " + msg);
9       }
10  }
```

FileLogger.java

```
1   public class FileLogger extends Logger {
2       public FileLogger(int level) {
3           this.level = level;
4       }
```

```
5
6       @Override
7       protected void write(String msg) {
8           System.out.println("File: " + msg);
9       }
10  }
```

ErrorLogger.java

```
1   public class ErrorLogger extends Logger {
2       public ErrorLogger(int level) {
3           this.level = level;
4       }
5
6       @Override
7       protected void write(String msg) {
8           System.out.println("Error: " + msg);
9       }
10  }
```

ChainOfResponsibilityPatternDemo.java

```
1   public class ChainOfResponsibilityPatternDemo {
2       public static void main(String[] args) {
3           Logger logChain = getLogChain();
4
5           logChain.log(Logger.INFO, "[INFO] Hello world!");
6           logChain.log(Logger.DEBUG, "[DEBUG] Hello world!");
7           logChain.log(Logger.ERROR, "[ERROR] Hello world!");
8       }
9
10      private static Logger getLogChain() {
11          Logger errorLogger = new ErrorLogger(Logger.ERROR);
12          Logger fileLogger = new FileLogger(Logger.DEBUG);
13          Logger consoleLogger = new ConsoleLogger(Logger.INFO);
14
15          errorLogger.setNextLogger(fileLogger);
16          fileLogger.setNextLogger(consoleLogger);
17          return errorLogger;
```

```
18      }
19  }
```

运行结果

```
Console: [INFO] Hello world!
File: [DEBUG] Hello world!
Console: [DEBUG] Hello world!
Error: [ERROR] Hello world!
File: [ERROR] Hello world!
Console: [ERROR] Hello world!
```

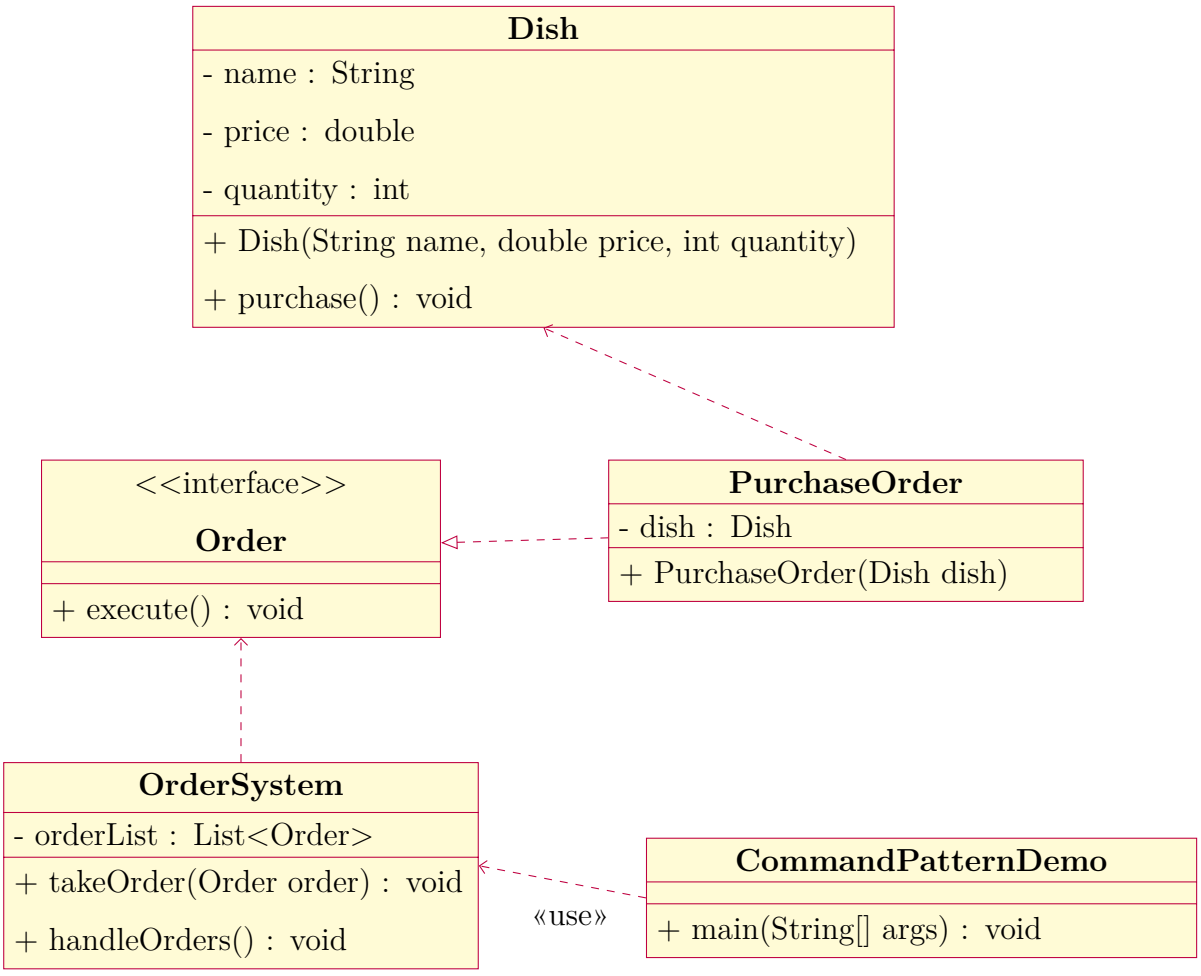
4.16 命令模式

4.16.1 命令模式（Command Pattern）

命令模式属于行为型模式，请求以命令的形式包裹在对象中，并传给调用对象。调用对象寻找可以处理该命令的合适的对象，并把该命令传给相应的对象，该对象执行命令。

在软件系统中，行为请求者与行为实现者通常是一种紧耦合的关系，但某些场合，比如需要对行为进行记录、撤销或重做、事务等处理时，这种无法抵御变化的紧耦合的设计就不太合适。

命令模式



Order.java

```java
public interface Order {
    void execute();
}
```

Dish.java

```java
public class Dish {
    private String name;
    private double price;
    private int quantity;

    public Dish(String name, double price, int quantity) {
        this.name = name;
        this.price = price;
        this.quantity = quantity;
    }

    public void purchase() {
        System.out.println(
            String.format("[%s]\t$%.2f\t*%d", name, price, quantity)
        );
    }
}
```

PurchaseOrder.java

```java
public class PurchaseOrder implements Order {
    private Dish dish;

    public PurchaseOrder(Dish dish) {
        this.dish = dish;
    }

    @Override
    public void execute() {
        dish.purchase();
    }
}
```

OrderSystem.java

```java
import java.util.ArrayList;
import java.util.List;

public class OrderSystem {
    private List<Order> orderList = new ArrayList<>();

    public void takeOrder(Order order) {
        orderList.add(order);
    }

    public void handleOrders() {
        for(Order order : orderList) {
            order.execute();
        }
        orderList.clear();
    }
}
```

CommandPatternDemo.java

```java
public class CommandPatternDemo {
    public static void main(String[] args) {
        PurchaseOrder purchaseOrder1 = new BuyOrder(
            new Dish("Boiled Fish", 20, 1)
        );
        PurchaseOrder purchaseOrder2 = new BuyOrder(
            new Dish("B.B.Q Pork", 15, 2)
        );
        PurchaseOrder purchaseOrder3 = new BuyOrder(
            new Dish("Lemon beef", 25, 1)
        );

        OrderSystem orderSystem = new OrderSystem();
        orderSystem.takeOrder(purchaseOrder1);
        orderSystem.takeOrder(purchaseOrder2);
        orderSystem.takeOrder(purchaseOrder3);

```

```
18          orderSystem.handleOrders();
19      }
20  }
```

运行结果

```
[Boiled Fish] $20.00 *1
[B.B.Q Pork] $15.00 *2
[Lemon beef] $25.00 *1
```

4.17 解释器模式

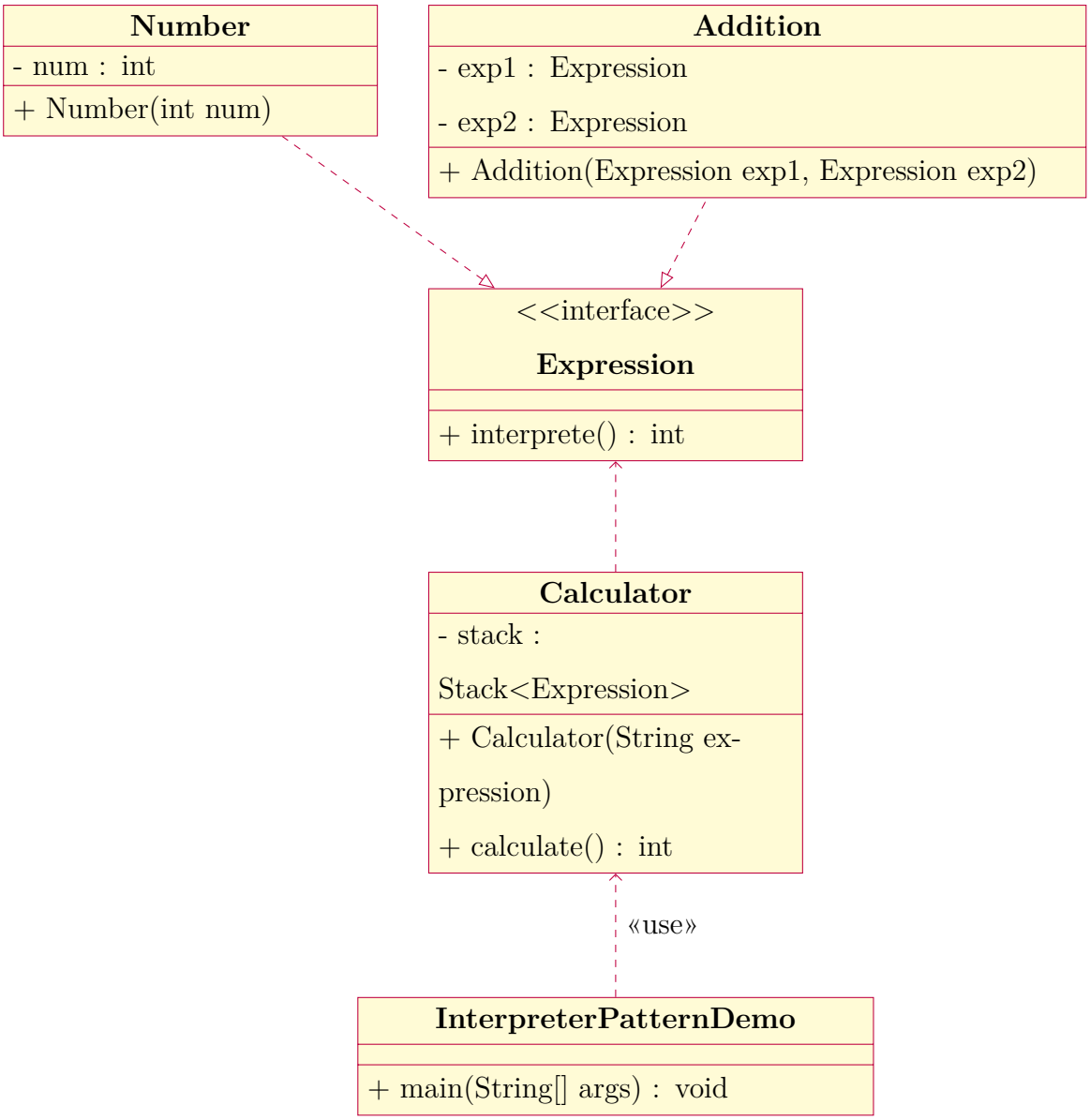
4.17.1 解释器模式（Interpreter Pattern）

在编译原理中，一个算术表达式通过词法分析器形成词法单元，而后这些词法单元再通过语法分析器构建语法分析树，最终形成一个抽象语法分析树。

如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子，这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。比如正则表达式就是解释器模型的一种应用，解释器为正则表达式定义了一个文法、如何表示一个特定的正则表达式、以及如何解释这个正则表达式。

解释器模式属于行为型模式，提供了评估语言的语法或表达式的方式，这种模式实现了一个表达式接口，该接口解释一个特定的上下文。解释器模式常被用于 SQL 解析、符号处理引擎等领域。

解释器模式



Expression.java

```java
public interface Expression {
    int interpret();
}
```

Number.java

```java
public class Number implements Expression {
    private int num;

    public Number(int num) {
        this.num = num;
```

```
    }

    @Override
    public int interpret() {
        return num;
    }
}
```

Addition.java

```
public class Addition implements Expression {
    private Expression exp1;
    private Expression exp2;

    public Addition(Expression exp1, Expression exp2) {
        this.exp1 = exp1;
        this.exp2 = exp2;
    }

    @Override
    public int interpret() {
        return exp1.interpret() + exp2.interpret();
    }
}
```

Calculator.java

```
import java.util.Stack;

public class Calculator {
    private Stack<Expression> stack = new Stack<>();

    public Calculator(String expression) {
        Expression exp1, exp2;

        String[] elements = expression.split(" ");
        for (int i = 0; i < elements.length; i++) {
            if (elements[i].charAt(0) == '+') {
                exp1 = stack.pop();
```

```
                exp2 = new Number(Integer.valueOf(elements[++i]));
                stack.push(new Addition(exp1, exp2));
            } else {
                stack.push(new Number(Integer.valueOf(elements[i])));
            }
        }
    }

    public int calculate() {
        return stack.pop().interpret();
    }
}
```

InterpreterPatternDemo.java

```
public class InterpreterPatternDemo {
    public static void main(String[] args) {
        Calculator calculator = new Calculator("1 + 22 + 333");
        System.out.println(calculator.calculate());
    }
}
```

运行结果

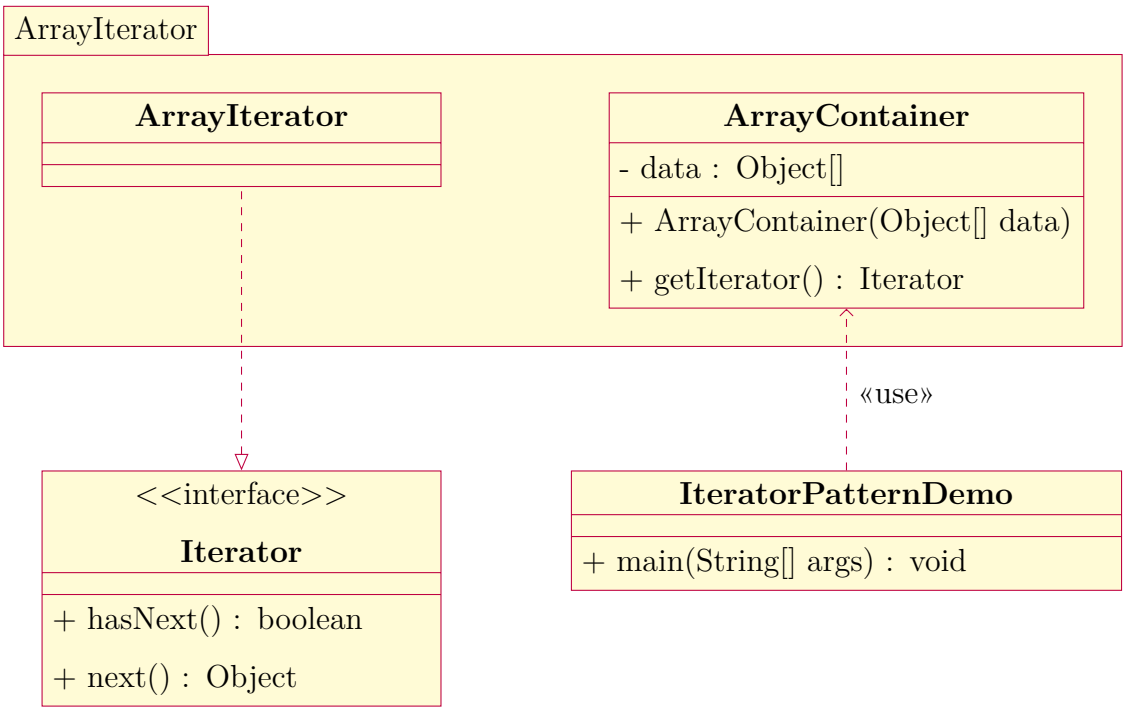
```
356
```

4.18 迭代器模式

4.18.1 迭代器模式（Iterator Pattern）

迭代器模式属于行为型模式，提供了一种方法顺序访问一个聚合对象中的各种元素，而又不暴露该对象的内部表示。

迭代器模式



Iterator.java

```
1 public interface Iterator {
2     boolean hasNext();
3     Object next();
4 }
```

ArrayContainer.java

```
1 public class ArrayContainer {
2     private Object[] data;
3
```

```
4       private class ArrayIterator implements Iterator {
5           private int index;
6
7           @Override
8           public boolean hasNext() {
9               return index < data.length;
10          }
11
12          @Override
13          public Object next() {
14              if(this.hasNext()) {
15                  return data[index++];
16              }
17              return null;
18          }
19      }
20
21      public ArrayContainer(Object[] data) {
22          this.data = data;
23      }
24
25      public Iterator getIterator() {
26          return new ArrayIterator();
27      }
28  }
```

IteratorPatternDemo.java

```
1   public class IteratorPatternDemo {
2       public static void main(String[] args) {
3           String[] data = {"data1", "data2", "data3", "data4"};
4           ArrayContainer arrayContainer = new ArrayContainer(data);
5           Iterator iter = arrayContainer.getIterator();
6           while(iter.hasNext()) {
7               System.out.println(iter.next());
8           }
9       }
10  }
```

运行结果

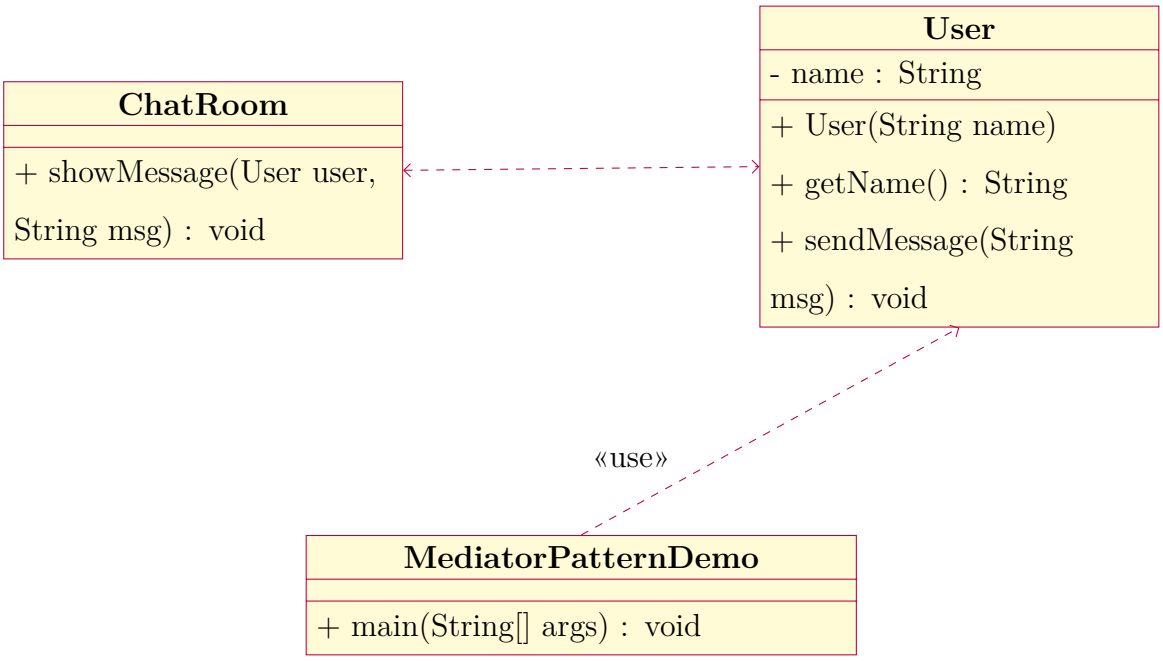
```
data1
data2
data3
data4
```

4.19 中介者模式

4.19.1 中介者模式（Mediator Pattern）

中介者模式属于行为型模式，用来降低多个对象和类之间的通信复杂性。中介者模式提供了一个中介类，该类通常处理不同类之间的通信，并支持松耦合，使代码易于维护。

中介者模式



ChatRoom.java

```
1  import java.util.Date;
2
3  public class ChatRoom {
4      public static void showMessage(User user, String msg) {
5          System.out.println(
6                  String.format("[%s] %s: %s",
7                                new Date().toString(),
8                                user.getName(),
9                                msg)
```

```
        );
    }
}
```

User.java

```
public class User {
    private String name;

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

    public String getName() {
        return name;
    }

    public void sendMessage(String msg) {
        ChatRoom.showMessage(this, msg);
    }
}
```

MediatorPatternDemo.java

```
public class MediatorPatternDemo {
    public static void main(String[] args) {
        User user1 = new User("Terry");
        User user2 = new User("Lily");

        user1.sendMessage("Hi, Lily!");
        user2.sendMessage("Hello, Terry!");
    }
}
```

运行结果

```
[Wed Dec 01 17:54:29 CST 2021] Terry: Hi, Lily!
[Wed Dec 01 17:54:29 CST 2021] Lily: Hello, Terry!
```

4.20 备忘录模式

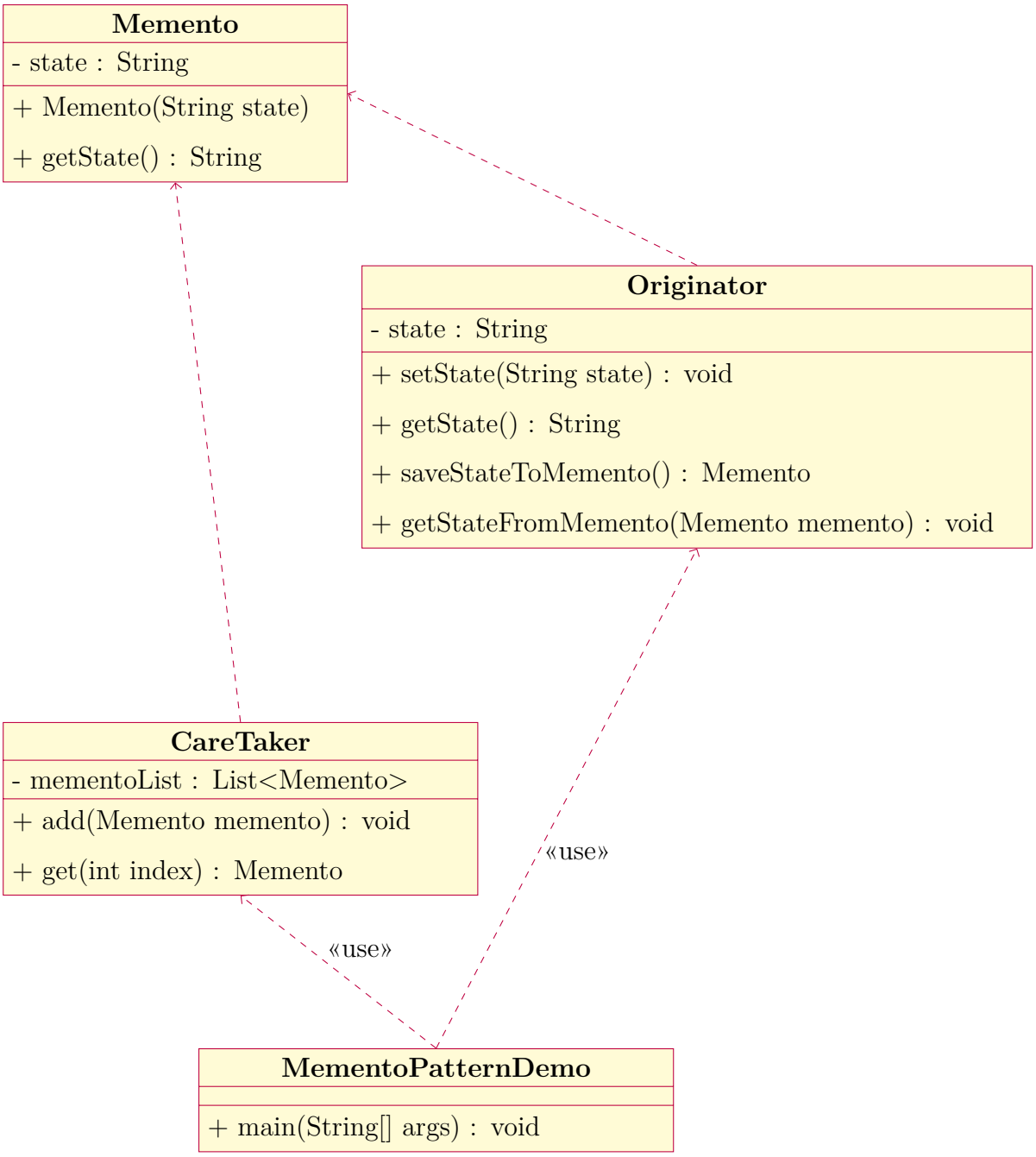
4.20.1 备忘录模式（Memento Pattern）

备忘录模式属于行为型模式，允许在不暴露对象实现细节的情况下保存和恢复对象之前的状态。

例如一款文字编辑器，除了简单的文字编辑功能外，编辑器中还要有让用户能撤销施加在文本上的任何操作，给用户提供了一种可以恢复状态的机制，可以使用户能够比较方便地回到某个历史的状态。程序在执行任何操作前会记录所有的对象状态，并将其保存下来。当用户此后需要撤销某个操作时，程序将从历史记录中获取最近的快照，然后使用它来恢复所有对象的状态。

图 4.6: 快照

备忘录模式



Memento.java

```
1  public class Memento {
2      private String state;
3
4      public Memento(String state) {
5          this.state = state;
6      }
7
```

```
    public String getState() {
        return state;
    }
}
```

Originator.java

```
public class Originator {
    private String state;

    public void setState(String state) {
        this.state = state;
    }

    public String getState() {
        return state;
    }

    public Memento saveStateToMemento() {
        return new Memento(state);
    }

    public void getStateFromMemento(Memento memento) {
        state = memento.getState();
    }
}
```

CareTaker.java

```
import java.util.ArrayList;
import java.util.List;

public class CareTaker {
    private List<Memento> mementoList = new ArrayList<>();

    public void add(Memento memento) {
        mementoList.add(memento);
    }

```

```
    public Memento get(int index) {
        return mementoList.get(index);
    }
}
```

MementoPatternDemo.java

```
public class MementoPatternDemo {
    public static void main(String[] args) {
        CareTaker careTaker = new CareTaker();
        Originator originator = new Originator();

        originator.setState("State 1");
        originator.setState("State 2");
        careTaker.add(originator.saveStateToMemento());

        originator.setState("State 3");
        careTaker.add(originator.saveStateToMemento());

        originator.setState("State 4");
        System.out.println(
            "Current State: " + originator.getState()
        );

        originator.getStateFromMemento(careTaker.get(0));
        System.out.println(
            "First saved state: " + originator.getState()
        );

        originator.getStateFromMemento(careTaker.get(1));
        System.out.println(
            "Second saved state: " + originator.getState()
        );
    }
}
```

运行结果

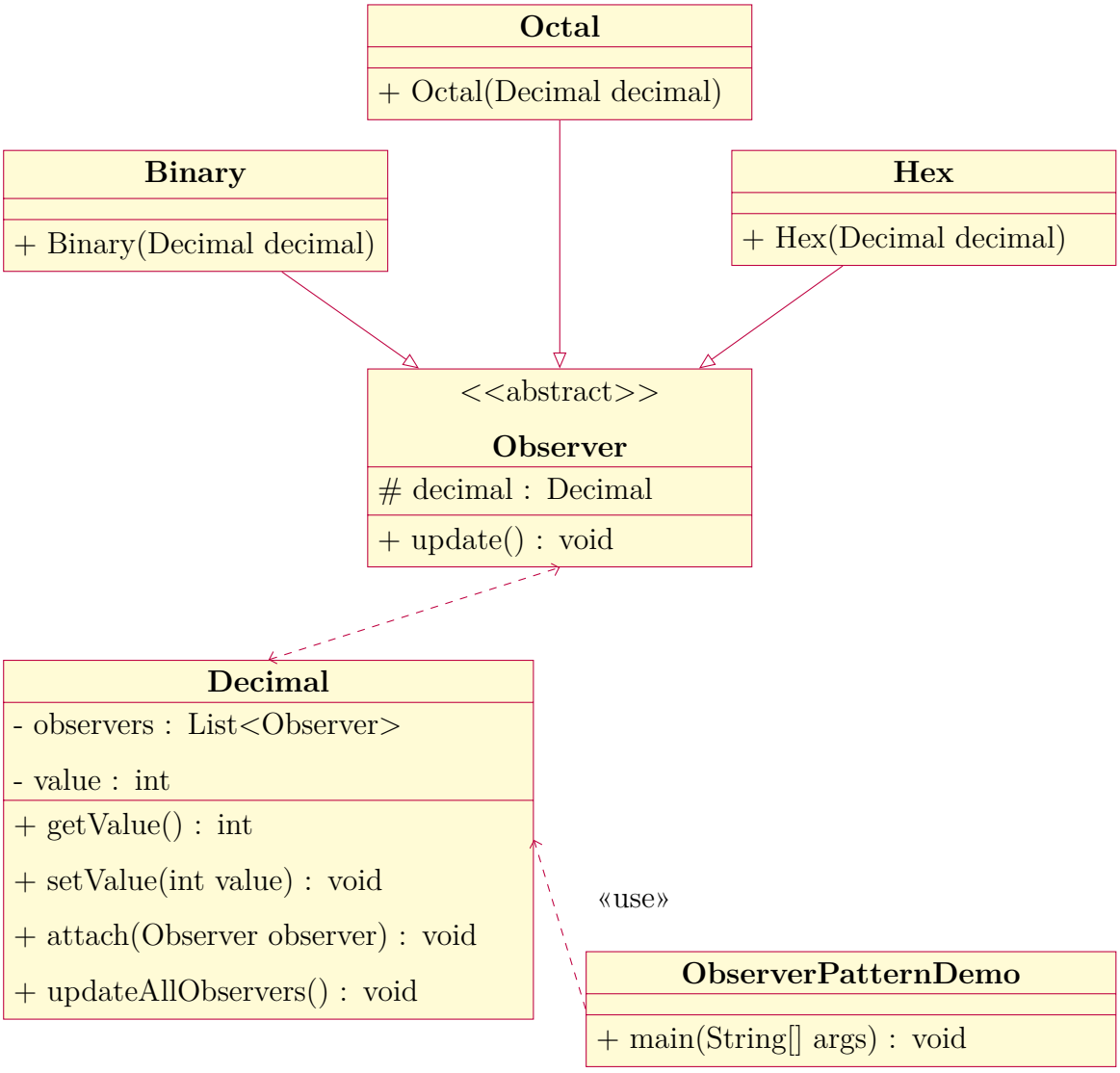
```
Current State: State 4
First saved state: State 2
Second saved state: State 3
```

4.21 观察者模式

4.21.1 观察者模式（Observer Pattern）

观察者模式属于行为型模式，是一种基于事件和响应的设计模式，常用于窗体应用程序和游戏开发。对象间存在一对多的依赖关系，当一个对象的状态发生改变，所有依赖于它的对象都得到通知并被自动更新。观察者模式主要解决一个对象状态改变给其它对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作。

观察者模式



Decimal.java

```
import java.util.ArrayList;
import java.util.List;

public class Decimal {
    private List<Observer> observers = new ArrayList<>();
    private int value;

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
        updateAllObservers();
    }

    public void attach(Observer observer){
        observers.add(observer);
    }

    public void updateAllObservers(){
        for(Observer observer : observers) {
            observer.update();
        }
    }
}
```

Observer.java

```
public abstract class Observer {
    protected Decimal decimal;
    public abstract void update();
}
```

Binary.java

```
public class Binary extends Observer {
    public Binary(Decimal decimal) {
```

```
        this.decimal = decimal;
        this.decimal.attach(this);
    }

    @Override
    public void update() {
        System.out.println(
            "Binary: " +
            Integer.toBinaryString(decimal.getValue())
        );
    }
}
```

Octal.java

```
public class Octal extends Observer {
    public Octal(Decimal decimal) {
        this.decimal = decimal;
        this.decimal.attach(this);
    }

    @Override
    public void update() {
        System.out.println(
            "Octal: " +
            Integer.toOctalString(decimal.getValue())
        );
    }
}
```

Hex.java

```
public class Hex extends Observer {
    public Hex(Decimal decimal) {
        this.decimal = decimal;
        this.decimal.attach(this);
    }

    @Override
```

```
    public void update() {
        System.out.println(
            "Hex: " +
            Integer.toHexString(decimal.getValue())
        );
    }
}
```

ObserverPatternDemo.java

```
public class ObserverPatternDemo {
    public static void main(String[] args) {
        Decimal decimal = new Decimal();
        new Binary(decimal);
        new Octal(decimal);
        new Hex(decimal);

        decimal.setValue(10);
        decimal.setValue(20);
    }
}
```

运行结果

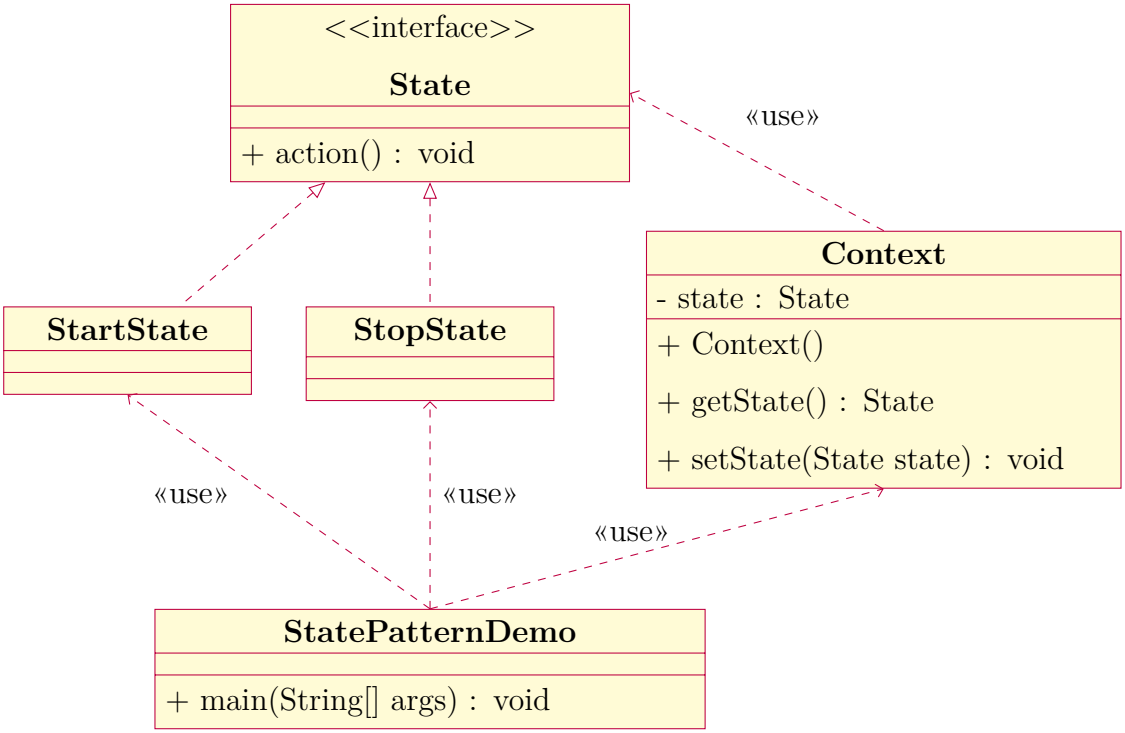
```
Binary: 1010
Octal: 12
Hex: a
Binary: 10100
Octal: 24
Hex: 14
```

4.22 状态模式

4.22.1 状态模式（State Pattern）

状态模式属于行为型模式，在状态模式中，允许一个对象在其内部状态改变时改变它的行为，类的行为是基于它的状态改变的。

状态模式



State.java

```
1 public interface State {
2     void action(Context context);
3 }
```

Context.java

```
1 public class Context {
2     private State state;
3 
4     public Context() {
```

```
        state = null;
    }

    public void setState(State state) {
        this.state = state;
    }

    public State getState() {
        return state;
    }
}
```

StartState.java

```
public class StartState implements State {
    @Override
    public void action(Context context) {
        context.setState(this);
        System.out.println("In start state ...");
    }

    @Override
    public String toString() {
        return "StartState";
    }
}
```

StopState.java

```
public class StopState implements State {
    @Override
    public void action(Context context) {
        context.setState(this);
        System.out.println("In stop state ...");
    }

    @Override
    public String toString() {
        return "StopState";
```

```
    }
}
```

StatePatternDemo.java

```
public class StatePatternDemo {
    public static void main(String[] args) {
        Context context = new Context();

        StartState startState = new StartState();
        startState.action(context);
        System.out.println(context.getState());

        StopState stopState = new StopState();
        stopState.action(context);
        System.out.println(context.getState());
    }
}
```

运行结果

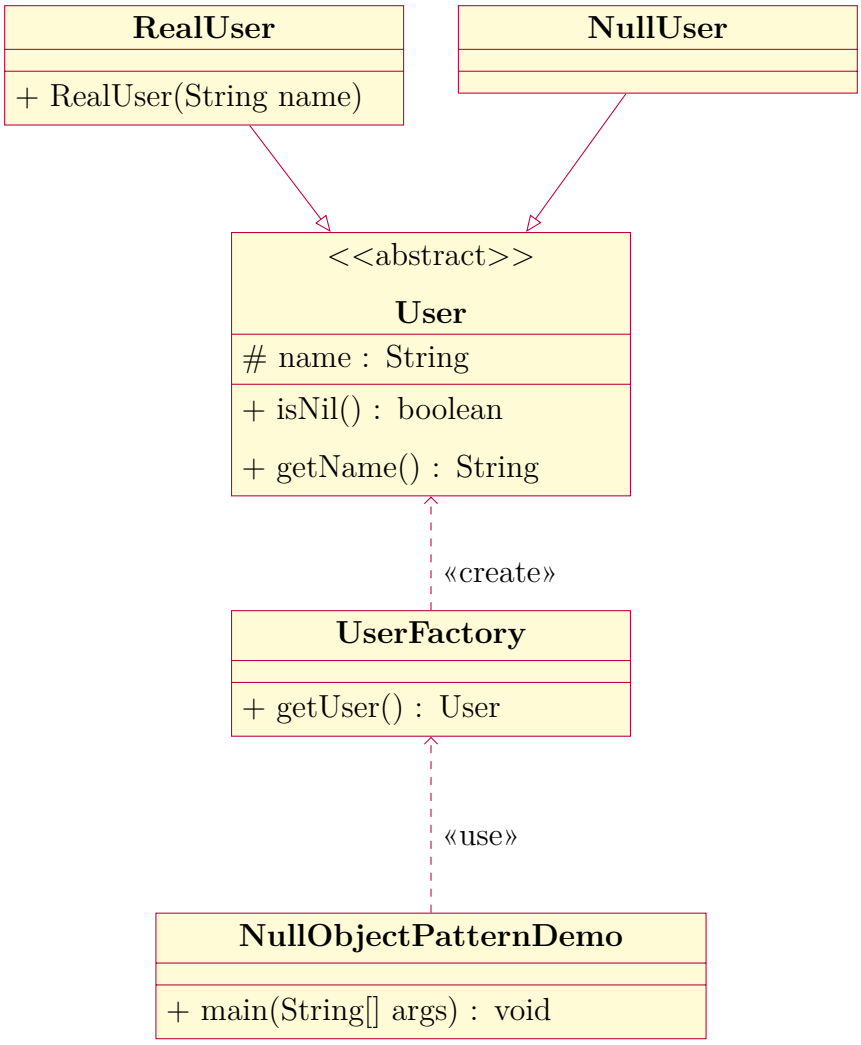
```
In start state ...
StartState
In stop state ...
StopState
```

4.23 空对象模式

4.23.1 Null Object Pattern

平时开发中应该避免过多的判空检查，空对象模式很好地避免了这种情况出现。在空对象模式中，当为空或者不存在的时候返回一个空对象，避免发生程序 NullPointException 异常。这样的空对象也可以在数据不可用的时候提供默认的行为。

空对象模式



User.java

```
1 public abstract class User {
2     protected String name;
```

```
3
4       public abstract boolean isNil();
5
6       public abstract String getName();
7   }
```

RealUser.java

```
1   public class RealUser extends User {
2       public RealUser(String name) {
3           this.name = name;
4       }
5
6       @Override
7       public boolean isNil() {
8           return false;
9       }
10
11      @Override
12      public String getName() {
13          return name;
14      }
15  }
```

NullUser.java

```
1   public class NullUser extends User {
2       @Override
3       public boolean isNil() {
4           return true;
5       }
6
7       @Override
8       public String getName() {
9           return "N/A";
10      }
11  }
```

UserFactory.java

```java
public class UserFactory {
    public static final String[] names = {
        "Terry", "Lily", "Henry", "Bob"
    };

    public static User getUser(String name) {
        for(int i = 0; i < names.length; i++) {
            if(names[i].equalsIgnoreCase(name)) {
                return new RealUser(name);
            }
        }
        return new NullUser();
    }
}
```

NullObjectPatternDemo.java

```java
public class NullObjectPatternDemo {
    public static void main(String[] args) {
        User user1 = UserFactory.getUser("Terry");
        User user2 = UserFactory.getUser("John");
        User user3 = UserFactory.getUser("Alice");
        User user4 = UserFactory.getUser("Lily");

        System.out.println(user1.getName());
        System.out.println(user2.getName());
        System.out.println(user3.getName());
        System.out.println(user4.getName());
    }
}
```

运行结果

```
Terry
N/A
N/A
Lily
```

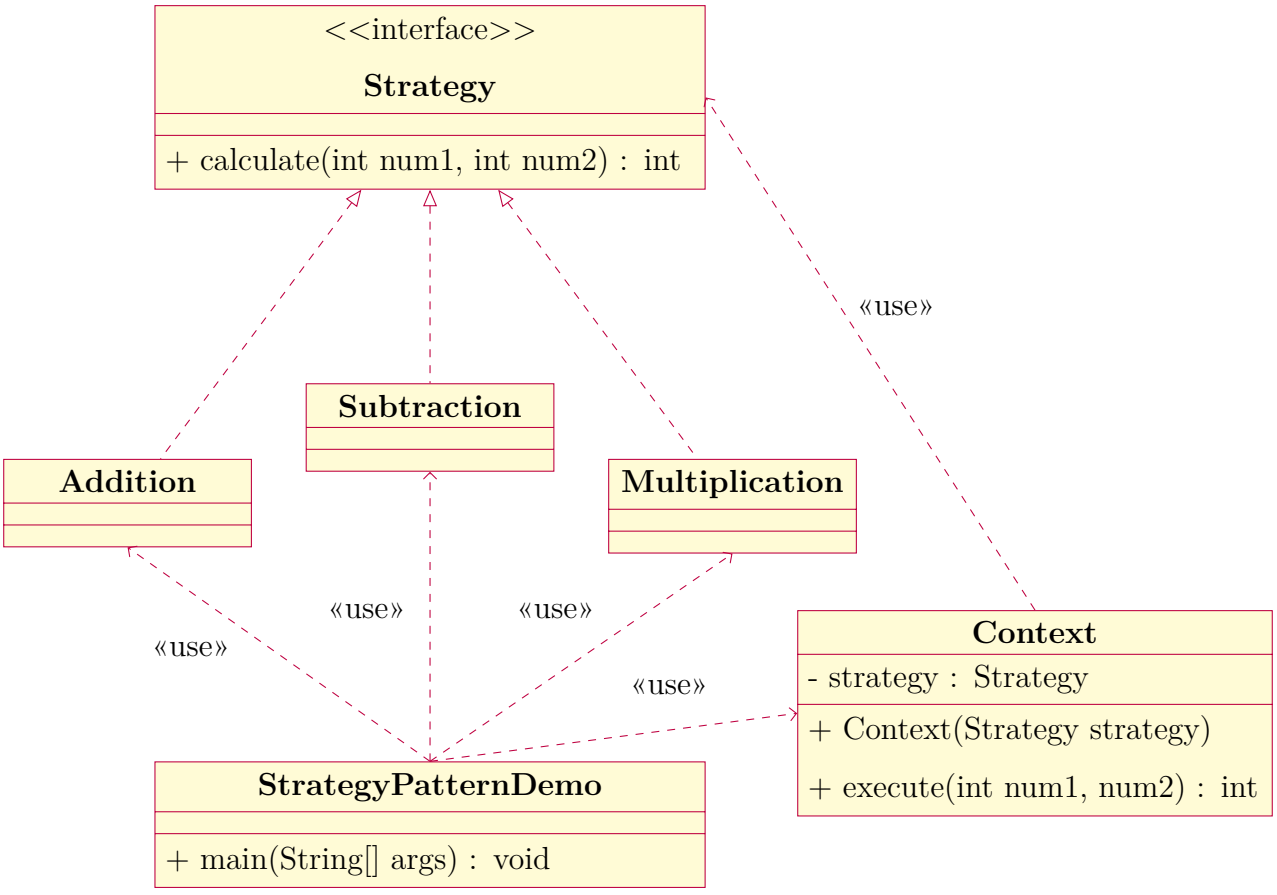
4.24 策略模式

4.24.1 策略模式（Strategy Pattern）

策略模式属于行为型模式，在策略模式中一个类的行为或其算法可以在运行时更改。策略指的是可以实现目标的方案集合，在某些特定情况下，策略之间是可以相互替换的。

策略模式将每一个算法封装起来，并让它们可以相互替换。策略模式只适用管理一组同类型的算法，并且这些算法是完全互斥的情况，也就是说任何时候，多个策略中只有一个可以生效。

策略模式



Strategy.java

```java
public interface Strategy {
    int calculate(int num1, int num2);
}
```

Addition.java

```java
public class Addition implements Strategy {
    @Override
    public int calculate(int num1, int num2) {
        return num1 + num2;
    }
}
```

Subtraction.java

```java
public class Subtraction implements Strategy {
    @Override
    public int calculate(int num1, int num2) {
        return num1 - num2;
    }
}
```

Multiplication.java

```java
public class Multiplication implements Strategy {
    @Override
    public int calculate(int num1, int num2) {
        return num1 * num2;
    }
}
```

Context.java

```java
public class Context {
    private Strategy strategy;

    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

```

```
    public int execute(int num1, int num2) {
        return strategy.calculate(num1, num2);
    }
}
```

Context.java

```
public class Context {
    private Strategy strategy;

    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public int execute(int num1, int num2) {
        return strategy.calculate(num1, num2);
    }
}
```

StrategyPatternDemo.java

```
public class StrategyPatternDemo {
    public static void main(String[] args) {
        Context context = new Context(new Addition());
        System.out.println("10 + 5 = " + context.execute(10, 5));

        context = new Context(new Subtraction());
        System.out.println("10 - 5 = " + context.execute(10, 5));

        context = new Context(new Multiplication());
        System.out.println("10 * 5 = " + context.execute(10, 5));
    }
}
```

运行结果

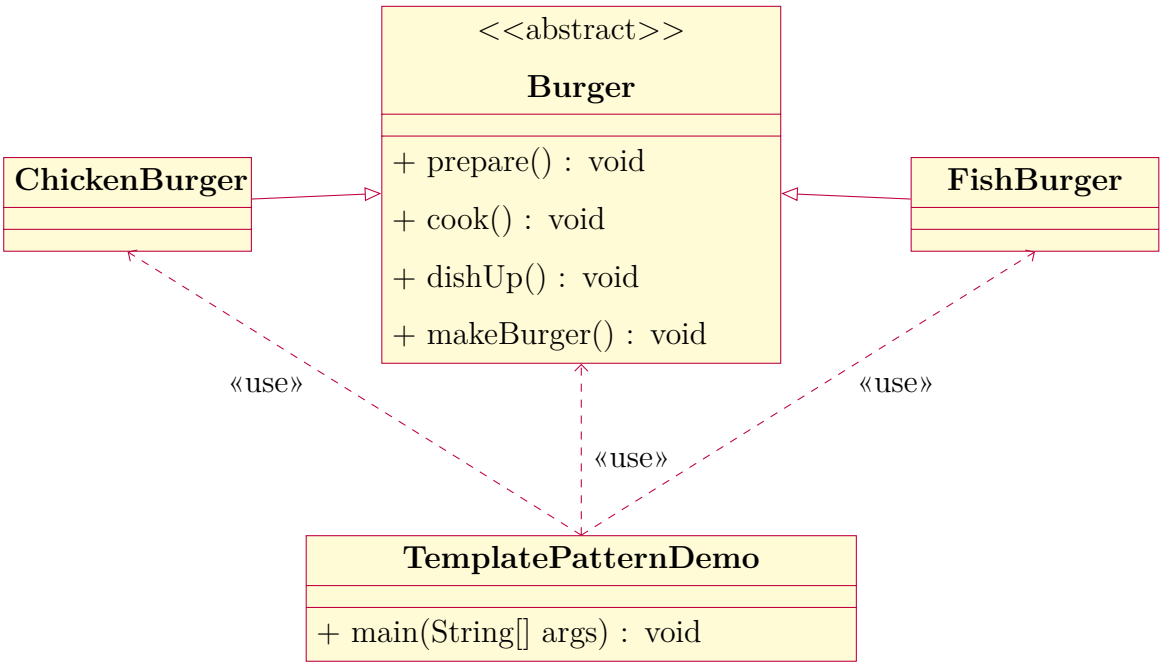
```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
```

4.25 模板模式

4.25.1 模板模式（Template Pattern）

模板模式属于行为型模式，一个抽象类公开定义了执行它的方法的方式，它的子类可以按需要重写方法实现，但调用将以抽象类中定义的操作进行。

模板模式



Burger.java

```
1  public abstract class Burger {
2      public abstract void prepare();
3
4      public abstract void cook();
5
6      public abstract void dishUp();
7
8      public void makeBurger() {
9          prepare();
10         cook();
```

```
11          dishUp();
12      }
13  }
```

ChickenBurger.java

```
1   public class ChickenBurger extends Burger {
2       @Override
3       public void prepare() {
4           System.out.println("Preparing buns, chicken, cheese ...");
5       }
6
7       @Override
8       public void cook() {
9           System.out.println("Cooking for 10 mins ...");
10      }
11
12      @Override
13      public void dishUp() {
14          System.out.println("Chicken burger done ...");
15      }
16  }
```

FishBurger.java

```
1   public class FishBurger extends Burger {
2       @Override
3       public void prepare() {
4           System.out.println("Preparing buns, fish, vegetable ...");
5       }
6
7       @Override
8       public void cook() {
9           System.out.println("Cooking for 5 mins ...");
10      }
11
12      @Override
13      public void dishUp() {
14          System.out.println("Fish burger done ...");
```

```
    }
}
```

TemplatePatternDemo.java

```
public class TemplatePatternDemo {
    public static void main(String[] args) {
        Burger burger = new ChickenBurger();
        burger.makeBurger();
        System.out.println("--------------------");
        burger = new FishBurger();
        burger.makeBurger();
    }
}
```

运行结果

```
Preparing buns, chicken, cheese ...
Cooking for 10 mins ...
Chicken burger done ...
--------------------
Preparing buns, fish, vegetable ...
Cooking for 5 mins ...
Fish burger done ...
```

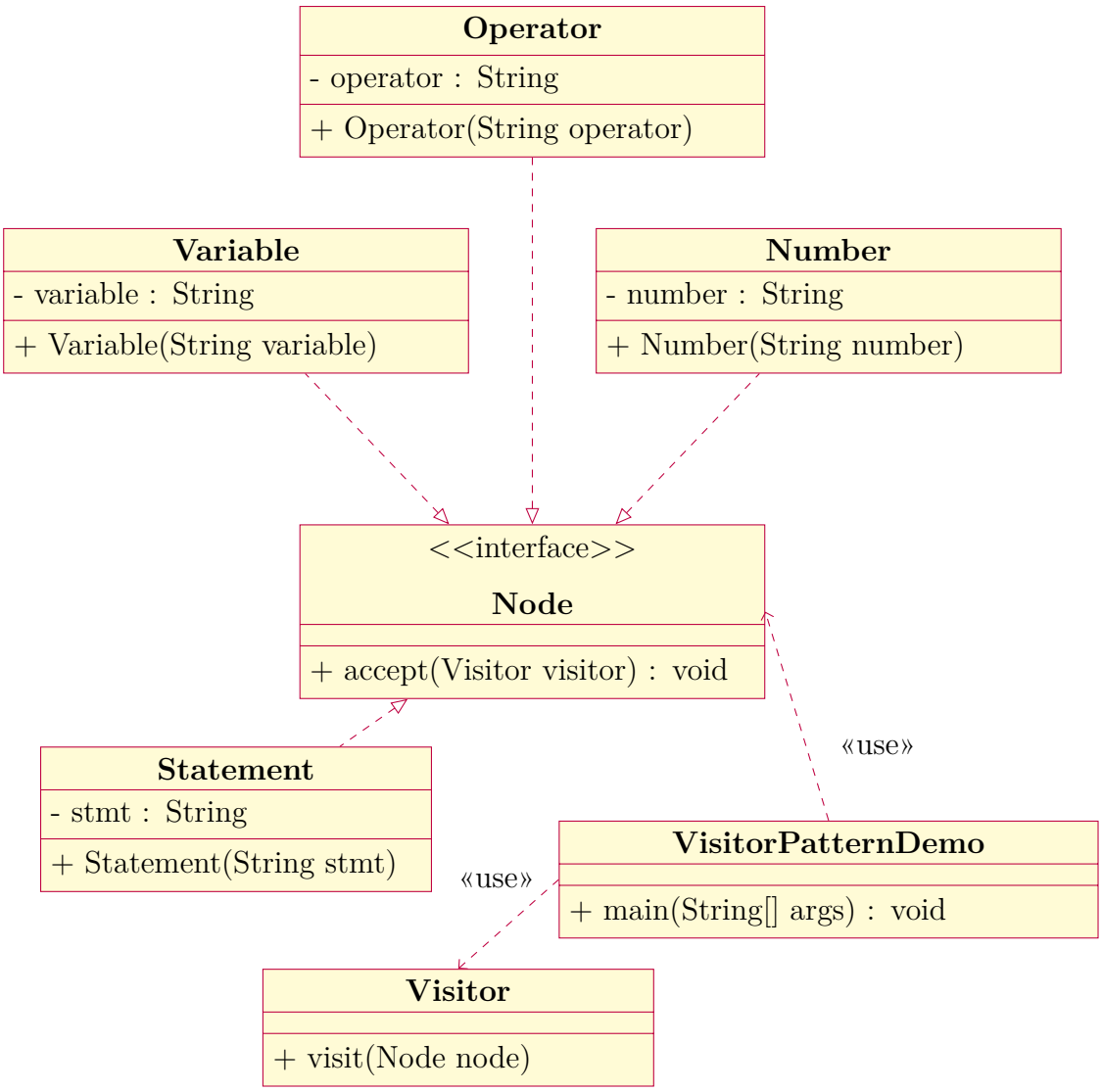
4.26 访问者模式

4.26.1 访问者模式（Visitor Pattern）

访问者模式属于行为型模式，通过使用一个访问者类，改变了元素类的执行算法，因此元素的执行算法可以随着访问者改变而改变。

访问者模式能把处理方法从数据结构中分离出来，并可以根据需要增加新的处理方法，且不用修改原来的程序代码与数据结构，这提高了程序的扩展性和灵活性。

访问者模式



Node.java

```java
public interface Node {
    void accept(Visitor visitor);
}
```

Variable.java

```java
public class Variable implements Node {
    private String variable;

    public Variable(String variable) {
        this.variable = variable;
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }

    @Override
    public String toString() {
        return variable + "\t(Variable)";
    }
}
```

Operator.java

```java
public class Operator implements Node {
    private String operator;

    public Operator(String operator) {
        this.operator = operator;
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }

```

```
13      @Override
14      public String toString() {
15          return operator + "\t(Operator)";
16      }
17  }
```

Number.java

```
1   public class Number implements Node {
2       private String number;
3
4       public Number(String number) {
5           this.number = number;
6       }
7
8       @Override
9       public void accept(Visitor visitor) {
10          visitor.visit(this);
11      }
12
13      @Override
14      public String toString() {
15          return number + "\t(Number)";
16      }
17  }
```

Statement.java

```
1   public class Statement implements Node {
2       private String stmt;
3
4       public Statement(String stmt) {
5           this.stmt = stmt;
6       }
7
8       @Override
9       public void accept(Visitor visitor) {
10          for(String token : stmt.split(" ")) {
11              // 合法変量名
```

```
            if(token.matches("[a-zA-Z_][a-zA-Z0-9_]*")) {
                new Variable(token).accept(visitor);
            }
            // 运算符: = + - * /
            else if(token.matches("[=]")) {
                new Operator(token).accept(visitor);
            }
            // 运算数: 实数
            else if(token.matches(
                "((\\+|-)?([0-9]+)(\\.[0-9]+)?)|((\\+|-)?\\.?[0-9]+)"
            )) {
                new Number(token).accept(visitor);
            }
        }
    }
}
```

Visitor.java

```
public class Visitor {
    public void visit(Node node) {
        System.out.println(node);
    }
}
```

VisitorPatternDemo.java

```
public class VisitorPatternDemo {
    public static void main(String[] args) {
        Node node = new Statement("PI = 3.1415 * radius * radius");
        node.accept(new Visitor());
    }
}
```

运行结果

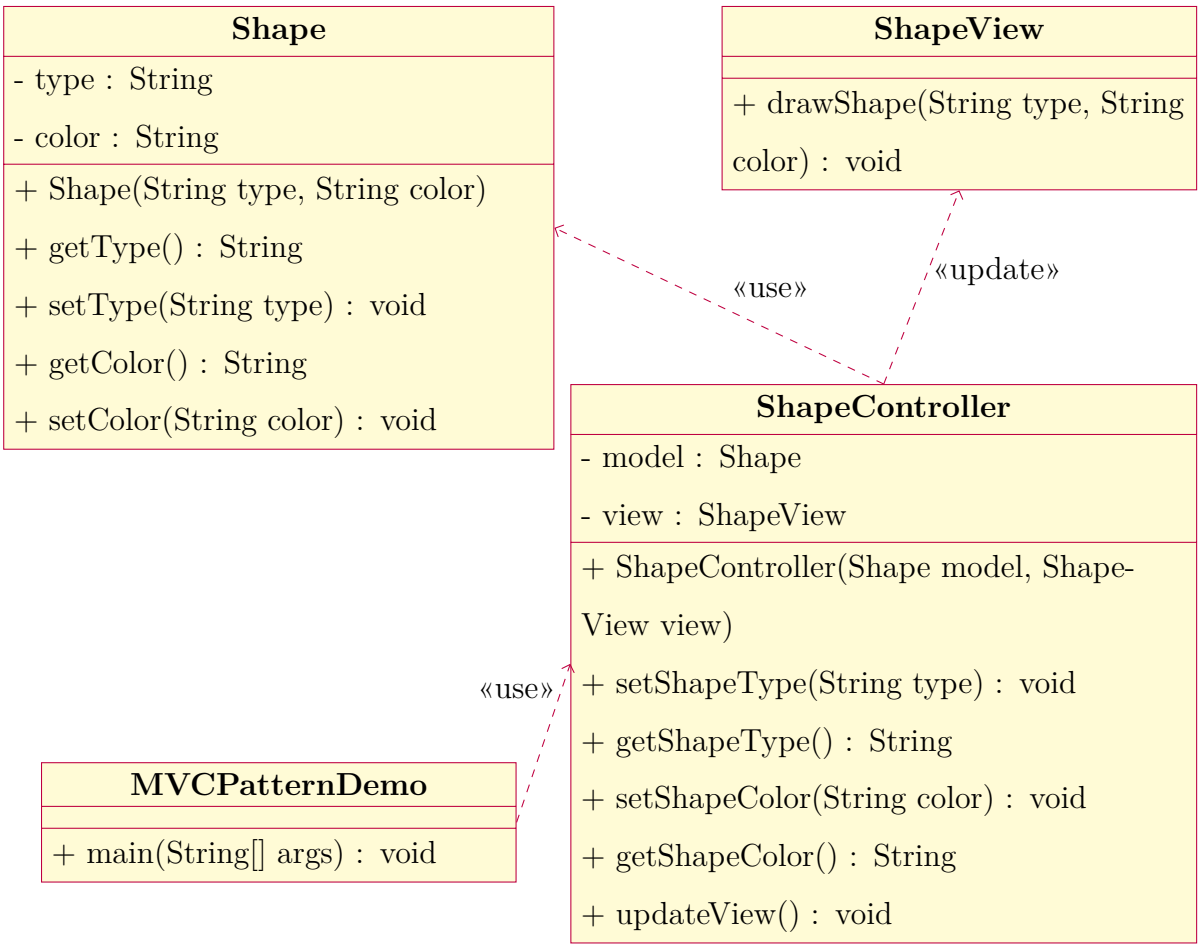
```
PI (Variable)
= (Operator)
3.1415 (Number)
radius (Variable)
radius (Variable)
```

4.27 MVC 模式

4.27.1 MVC 模式（Model View Controller Pattern）

MVC 模式用一种业务逻辑、数据与界面显示分离的方法来组织代码，将众多的业务逻辑聚集到一个部件里面，在需要改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑，达到减少编码的时间，提高代码复用性。

MVC 模式



Shape.java

```
1 public class Shape {
2     private String type;
3     private String color;
4
```

```
    public Shape(String type, String color) {
        this.type = type;
        this.color = color;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
}
```

ShapeView.java

```
public class ShapeView {
    public void drawShape(String type, String color) {
        System.out.println(type + " (" + color + ")");
    }
}
```

ShapeController.java

```
public class ShapeController {
    private Shape model;
    private ShapeView view;

    public ShapeController(Shape model, ShapeView view) {
        this.model = model;
        this.view = view;
```

```
    }

    public void setShapeType(String type) {
        model.setType(type);
    }

    public String getShapeType() {
        return model.getType();
    }

    public void setShapeColor(String color) {
        model.setColor(color);
    }

    public String getShapeColor() {
        return model.getColor();
    }

    public void updateView() {
        view.drawShape(model.getType(), model.getColor());
    }
}
```

MVCPatternDemo.java

```
public class MVCPatternDemo {
    public static void main(String[] args) {
        Shape shape = new Shape("circle", "red");
        ShapeView view = new ShapeView();
        ShapeController controller = new ShapeController(shape, view);

        controller.updateView();
        controller.setShapeType("square");
        controller.setShapeColor("blue");
        controller.updateView();
    }
}
```

运行结果

```
circle (red)
square (blue)
```

4.28 业务代表模式

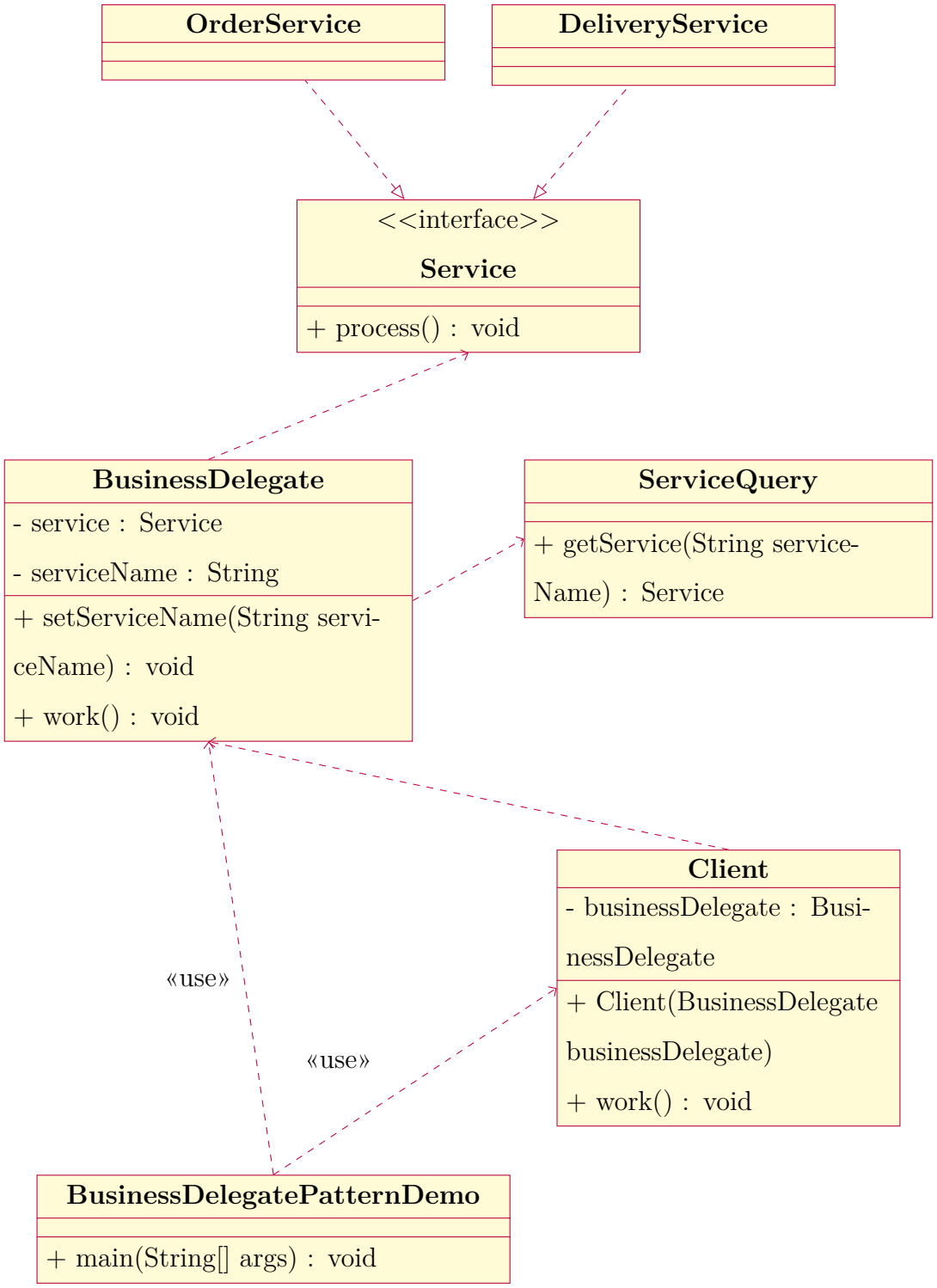
4.28.1 业务代表模式（Business Delegate Pattern）

业务代表模式用于对表示层和业务层解耦，它用来减少通信或对表示层代码中的业务层代码的远程查询功能。

在业务层中包含以下实体：

- 客户端：表示层代码可以是 JSP 或 UI 代码。
- 业务代表：一个为客户端实体提供的入口类，它提供了对业务服务方法的访问。
- 查询服务：查找服务对象负责获取相关的业务实现，并提供业务对象对业务代表对象的访问。
- 业务服务：业务服务接口，实现了该业务服务的实体类，提供了实际的业务实现逻辑。

业务代表模式



Service.java

```java
public interface Service {
    void process();
}
```

OrderService.java

```java
public class OrderService implements Service {
    @Override
    public void process() {
        System.out.println("处理订单服务");
    }
}
```

DeliveryService.java

```java
public class DeliveryService implements Service {
    @Override
    public void process() {
        System.out.println("处理配送服务");
    }
}
```

ServiceQuery.java

```java
public class ServiceQuery {
    public static Service getService(String serviceName) {
        if(serviceName.equalsIgnoreCase("order")) {
            return new OrderService();
        } else {
            return new DeliveryService();
        }
    }
}
```

BusinessDelegate.java

```java
public class BusinessDelegate {
    private Service service;
    private String serviceName;

    public void setServiceName(String serviceName) {
        this.serviceName = serviceName;
    }

    public void work() {
```

```
        service = ServiceQuery.getService(serviceName);
        service.process();
    }
}
```

Client.java

```
public class Client {
    private BusinessDelegate businessDelegate;

    public Client(BusinessDelegate businessDelegate) {
        this.businessDelegate = businessDelegate;
    }

    public void work() {
        businessDelegate.work();
    }
}
```

BusinessDelegatePatternDemo.java

```
public class BusinessDelegatePatternDemo {
    public static void main(String[] args) {
        BusinessDelegate businessDelegate = new BusinessDelegate();
        Client client = new Client(businessDelegate);

        businessDelegate.setServiceName("order");
        client.work();

        businessDelegate.setServiceName("delivery");
        client.work();
    }
}
```

运行结果

处理订单服务

处理配送服务

4.29 组合实体模式

4.29.1 组合实体模式（Composite Entity Pattern）

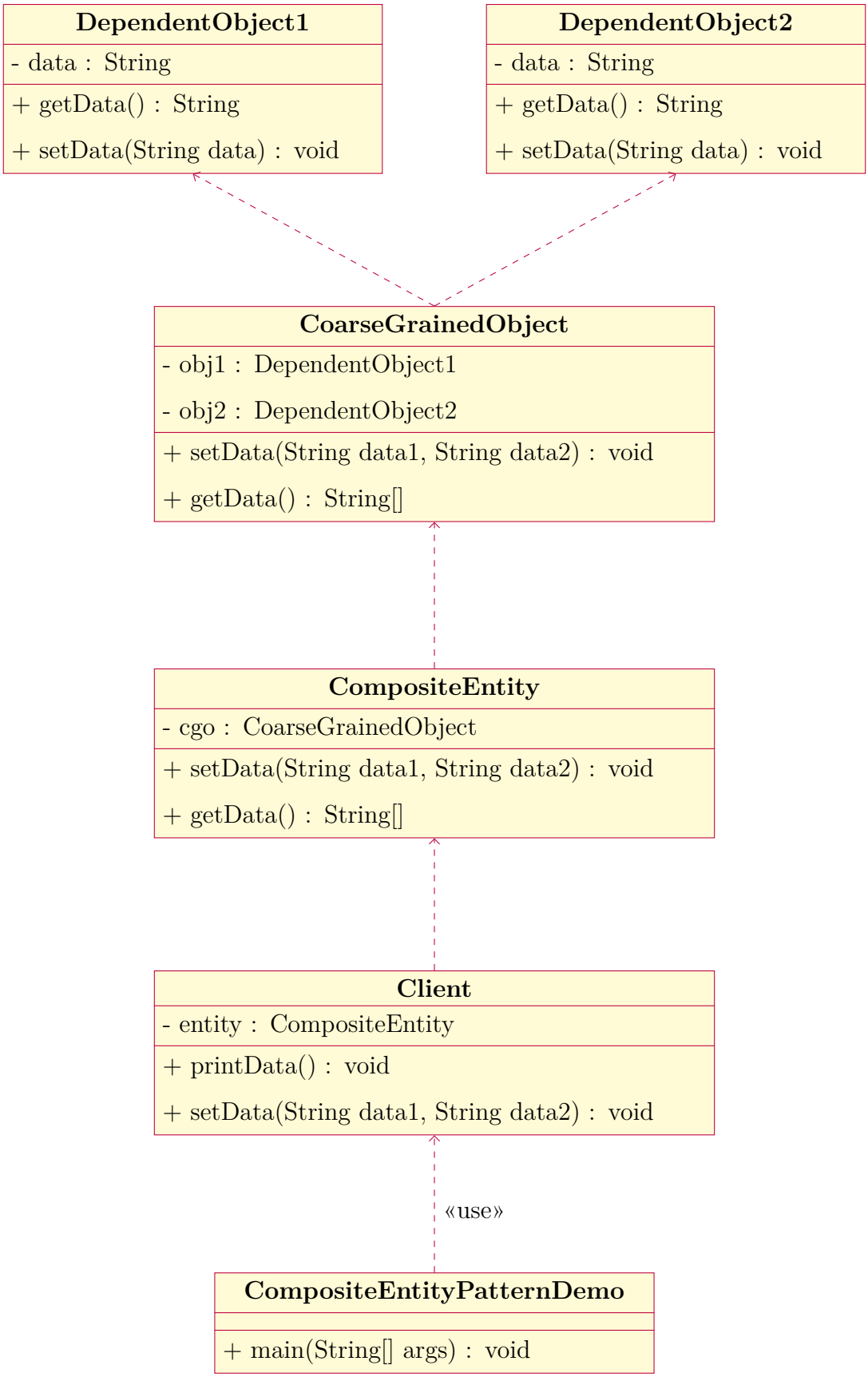
在组合实体模式中，一个组合实体是一个实体，当更新一个组合实体时，内部依赖对象会自动更新，因为它们是由实体管理的。

组合实体包含以下部分：

- 组合实体：主要的实体，它可以是粗粒的，或者可以包含一个粗粒度对象，用于持续生命周期。
- 粗粒度对象（coarse-grained object）：该对象包含依赖对象，它有自己的生命周期，也能管理依赖对象的生命周期。
- 依赖对象（dependent object）：依赖对象是一个持续生命周期依赖于粗粒度对象的对象。
- 策略：表示如何实现组合实体。

粒度是根据项目模块划分的细致程度区分的，一个项目模块分得越多，每个模块越小，负责的工作越细，就说粒度越细，否则为粗粒度。

组合实体模式


DependentObject1
- data : String
+ getData() : String
+ setData(String data) : void
DependentObject2
- data : String
+ getData() : String
+ setData(String data) : void
CoarseGrainedObject
- obj1 : DependentObject1
- obj2 : DependentObject2
+ setData(String data1, String data2) : void
+ getData() : String[]
CompositeEntity
- cgo : CoarseGrainedObject
+ setData(String data1, String data2) : void
+ getData() : String[]
Client
- entity : CompositeEntity
+ printData() : void
+ setData(String data1, String data2) : void
«use»
CompositeEntityPatternDemo
+ main(String[] args) : void

DependentObject1.java

```java
public class DependentObject1 {
    private String data;

    public String getData() {
        return data;
    }

    public void setData(String data) {
        this.data = data;
    }
}
```

DependentObject2.java

```java
public class DependentObject2 {
    private String data;

    public String getData() {
        return data;
    }

    public void setData(String data) {
        this.data = data;
    }
}
```

CoarseGrainedObject.java

```java
public class CoarseGrainedObject {
    DependentObject1 obj1 = new DependentObject1();
    DependentObject2 obj2 = new DependentObject2();

    public void setData(String data1, String data2) {
        obj1.setData(data1);
        obj2.setData(data2);
    }

    public String[] getData() {
```

```
        return new String[] {obj1.getData(), obj2.getData()};
    }
}
```

CompositeEntity.java

```
public class CompositeEntity {
    private CoarseGrainedObject cgo = new CoarseGrainedObject();

    public void setData(String data1, String data2) {
        cgo.setData(data1, data2);
    }

    public String[] getData() {
        return cgo.getData();
    }
}
```

Client.java

```
public class Client {
    private CompositeEntity entity = new CompositeEntity();

    public void printData() {
        for(int i = 0; i < entity.getData().length; i++) {
            System.out.println("Data: " + entity.getData()[i]);
        }
    }

    public void setData(String data1, String data2) {
        entity.setData(data1, data2);
    }
}
```

CompositeEntityPatternDemo.java

```
public class CompositeEntityPatternDemo {
    public static void main(String[] args) {
        Client client = new Client();
        client.setData("test1", "test2");
```

```
5           client.printData();
6           client.setData("test3", "test4");
7           client.printData();
8       }
9   }
```

运行结果

```
Data: test1
Data: test2
Data: test3
Data: test4
```

4.30 数据访问对象模式

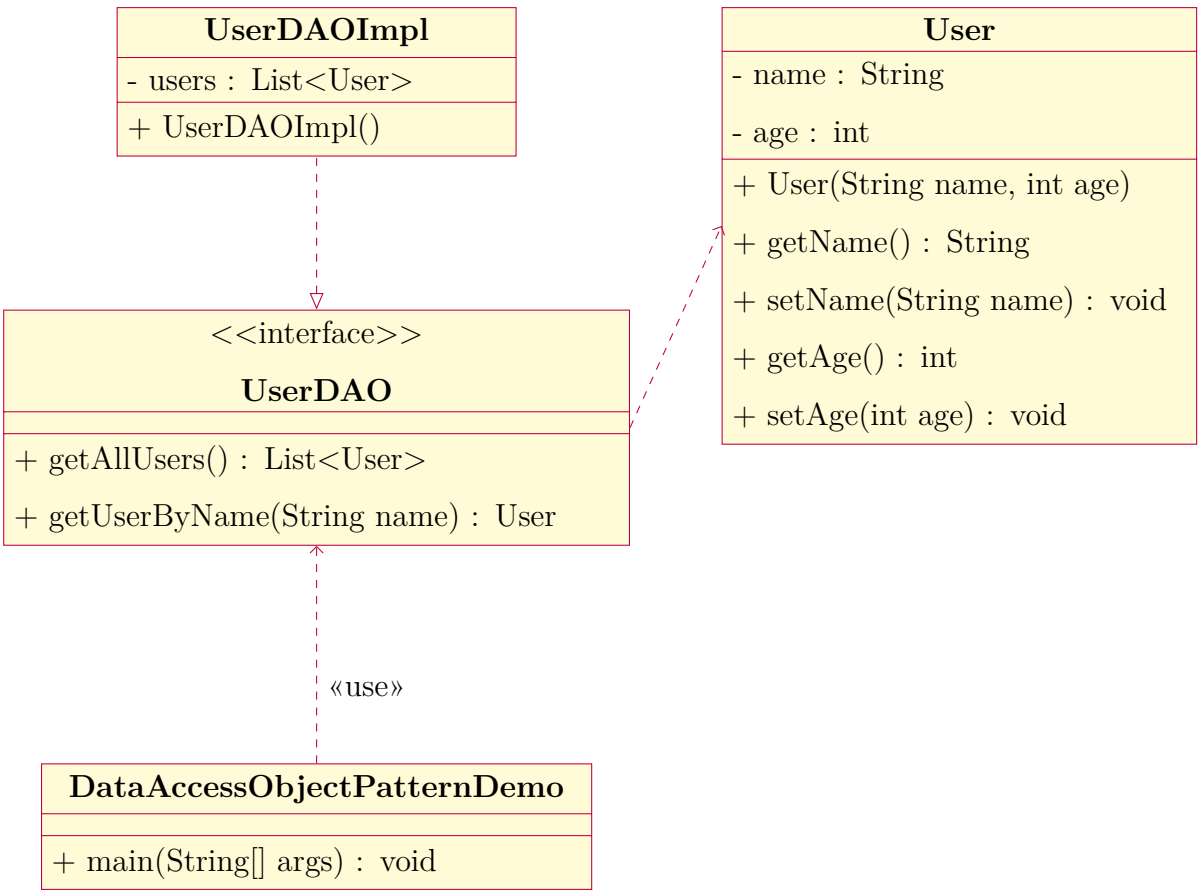
4.30.1 数据访问对象模式（Data Access Object Pattern）

数据访问对象模式用于把低级的数据访问 API 或操作从高级的业务服务中分离出来。

数据访问对象模式有以下参与者：

- 数据访问对象接口：定义了在一个模型对象上要执行的标准操作。
- 数据访问对象实体类：实现数据访问对象接口，负责从数据源获取数据，数据源可以是数据库、XML 或其它存储机制。
- 模型对象 / 数值对象：该对象包含了 getter / setter 来存储通过使用 DAO 类检索到的数据。

数据访问对象模式



User.java

```java
public class User {
    private String name;
    private int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "User{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

UserDAO.java

```java
import java.util.List;
```

```
2
3  public interface UserDAO {
4      List<User> getAllUsers();
5
6      User getUserByName(String name);
7  }
```

UserDAOImpl.java

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class UserDAOImpl implements UserDAO {
5      private List<User> users;
6
7      public UserDAOImpl() {
8          users = new ArrayList<>();
9          users.add(new User("Terry", 22));
10         users.add(new User("Henry", 19));
11     }
12
13     @Override
14     public List<User> getAllUsers() {
15         return users;
16     }
17
18     @Override
19     public User getUserByName(String name) {
20         for(User user : users) {
21             if(name.equals(user.getName())) {
22                 return user;
23             }
24         }
25         return null;
26     }
27 }
```

DataAccessObjectPatternDemo.java

```
1  public class DataAccessObjectPatternDemo {
2      public static void main(String[] args) {
3          UserDAO userDAO = new UserDAOImpl();
4
5          for(User user : userDAO.getAllUsers()) {
6              System.out.println(user);
7          }
8
9          System.out.println(userDAO.getUserByName("Terry"));
10     }
11 }
```

运行结果

```
User{name='Terry', age=22}
User{name='Henry', age=19}
User{name='Terry', age=22}
```

4.31 前端控制器模式

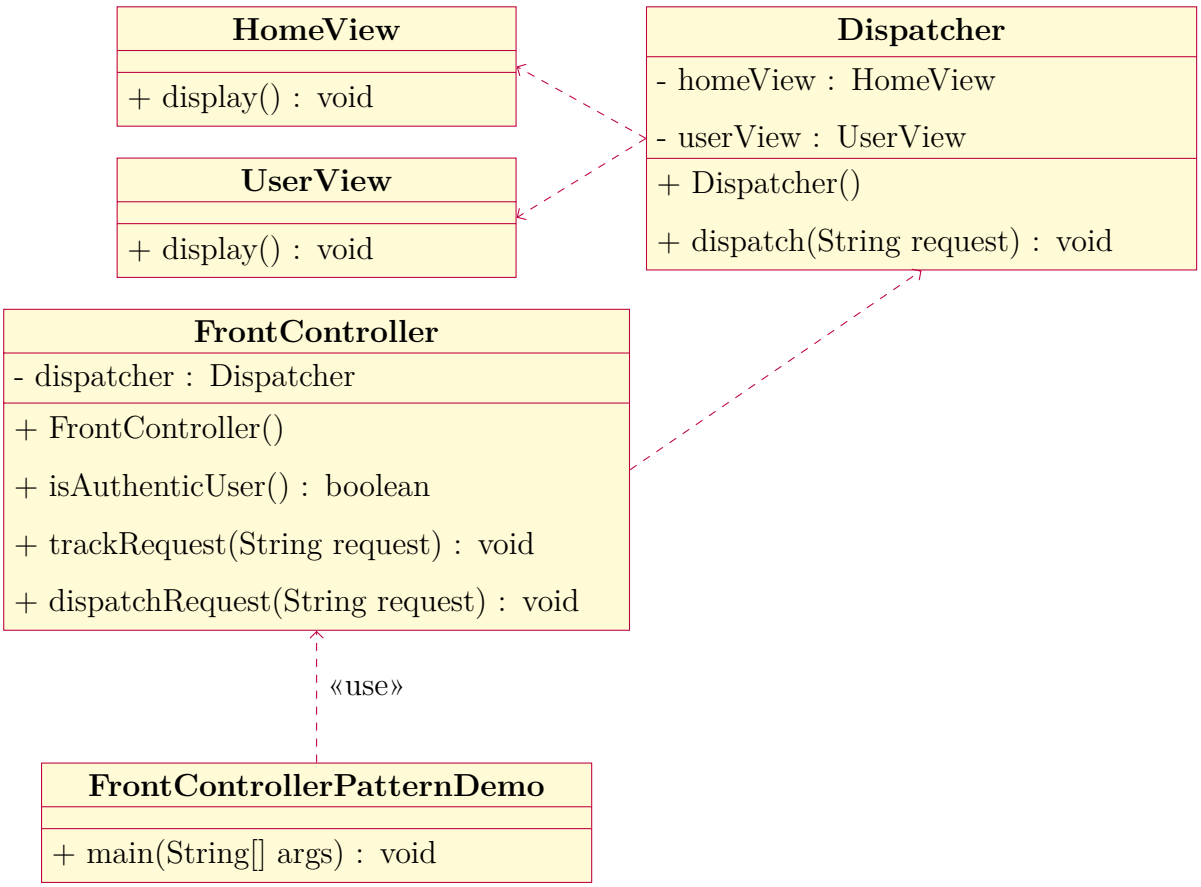
4.31.1 前端控制器模式（Front Controller Pattern）

前端控制器模式用来提供一个集中的请求处理机制，所有的请求都将由一个单一的处理程序处理。该处理程序可以做认证/授权/记录日志，或者跟踪请求，然后把请求传给相应的处理程序。

前端控制器模式包含以下实体：

- 前端控制器：处理应用程序所有类型请求的单个处理程序，应用程序可以是基于 Web 的应用程序，也可以是基于桌面的应用程序。
- 调度器（dispatcher）：前端控制器使用调度器来调度请求到相应处理程序。
- 视图（view）：为请求而创建的对象。

前端控制器模式



HomeView.java

```
1  public class HomeView {
2      public void display() {
3          System.out.println("Displaying home page ...");
4      }
5  }
```

UserView.java

```
1  public class UserView {
2      public void display() {
3          System.out.println("Displaying user page ...");
4      }
5  }
```

Dispatcher.java

```
1  public class Dispatcher {
2      private HomeView homeView;
3      private UserView userView;
4
5      public Dispatcher() {
6          homeView = new HomeView();
7          userView = new UserView();
8      }
9
10     public void dispatch(String request) {
11         if(request.equalsIgnoreCase("USER")) {
12             userView.display();
13         } else {
14             homeView.display();
15         }
16     }
17 }
```

FrontController.java

```
1  public class FrontController {
2      private Dispatcher dispatcher;
3
```

```
    public FrontController(){
        dispatcher = new Dispatcher();
    }

    private boolean isAuthenticUser() {
        System.out.println("User is authenticated successfully");
        return true;
    }

    private void trackRequest(String request) {
        System.out.println("Page requested: " + request);
    }

    public void dispatchRequest(String request) {
        // log each request
        trackRequest(request);

        // authenticate the user
        if(isAuthenticUser()) {
            dispatcher.dispatch(request);
        }
    }
}
```

FrontControllerPatternDemo.java

```
public class FrontControllerPatternDemo {
    public static void main(String[] args) {
        FrontController frontController = new FrontController();
        frontController.dispatchRequest("HOME");
        frontController.dispatchRequest("USER");
    }
}
```

运行结果

```
Page requested: HOME
User is authenticated successfully
Displaying home page ...
Page requested: USER
User is authenticated successfully
Displaying user page ...
```

4.32 拦截过滤器模式

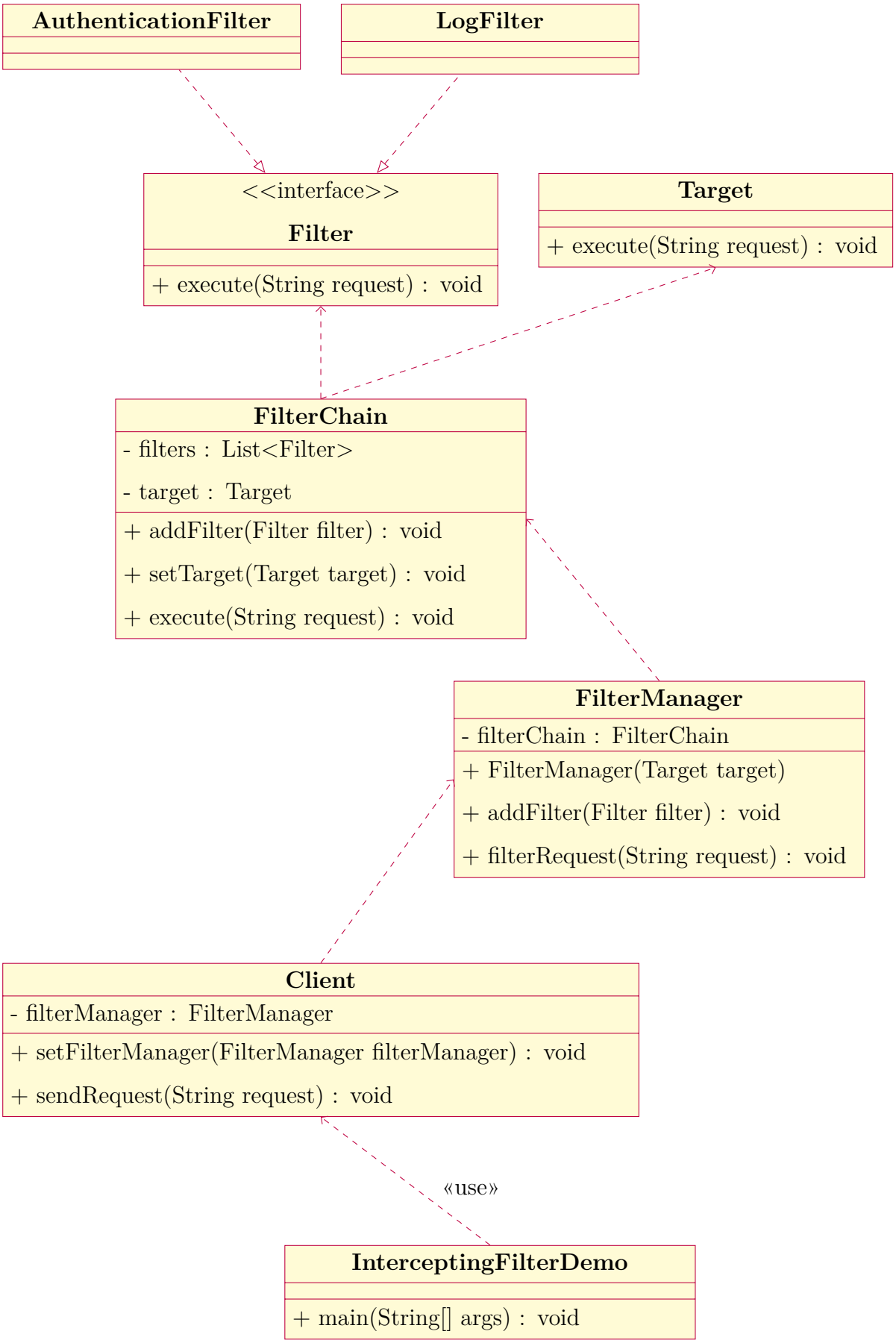
4.32.1 拦截过滤器模式（Intercepting Filter Pattern）

拦截过滤器模式用于对应用程序的请求或响应做一些预处理/后处理。过滤器可以做认证/授权/记录日志，或者跟踪请求，然后把请求传给相应的处理程序。

拦截过滤器模式包含以下实体：

- 过滤器：在请求处理程序执行请求之前或之后，执行某些任务。
- 过滤器链：过滤器链带有多个过滤器，并按照定义的顺序执行这些过滤器。
- Target：请求处理的程序。
- 过滤管理器：过滤管理器管理过滤器和过滤器链。
- 客户端：向 Target 发送请求的对象。

拦截过滤器模式


AuthenticationFilter
LogFilter
<<interface>>
Filter
+ execute(String request) : void
Target
+ execute(String request) : void
FilterChain
- filters : List<Filter>
- target : Target
+ addFilter(Filter filter) : void
+ setTarget(Target target) : void
+ execute(String request) : void
FilterManager
- filterChain : FilterChain
+ FilterManager(Target target)
+ addFilter(Filter filter) : void
+ filterRequest(String request) : void
Client
- filterManager : FilterManager
+ setFilterManager(FilterManager filterManager) : void
+ sendRequest(String request) : void
«use»
InterceptingFilterDemo
+ main(String[] args) : void

Filter.java

```java
public interface Filter {
    public void execute(String request);
}
```

AuthenticationFilter.java

```java
public class AuthenticationFilter implements Filter {
    @Override
    public void execute(String request) {
        System.out.println("Authenticating request: " + request);
    }
}
```

LogFilter.java

```java
public class LogFilter implements Filter {
    @Override
    public void execute(String request) {
        System.out.println("Logging request: " + request);
    }
}
```

Target.java

```java
public class Target {
    public void execute(String request) {
        System.out.println("Executing request: " + request);
    }
}
```

FilterChain.java

```java
import java.util.ArrayList;
import java.util.List;

public class FilterChain {
    private List<Filter> filters = new ArrayList<>();
    private Target target;

```

```
8       public void addFilter(Filter filter) {
9           filters.add(filter);
10      }
11
12      public void setTarget(Target target) {
13          this.target = target;
14      }
15
16      public void execute(String request) {
17          for(Filter filter : filters) {
18              filter.execute(request);
19          }
20          target.execute(request);
21      }
22  }
```

FilterManager.java

```
1   public class FilterManager {
2       FilterChain filterChain;
3
4       public FilterManager(Target target) {
5           filterChain = new FilterChain();
6           filterChain.setTarget(target);
7       }
8
9       public void addFilter(Filter filter) {
10          filterChain.addFilter(filter);
11      }
12
13      public void filterRequest(String request) {
14          filterChain.execute(request);
15      }
16  }
```

Client.java

```
1   public class Client {
2       FilterManager filterManager;
```

```
3
4       public void setFilterManager(FilterManager filterManager) {
5           this.filterManager = filterManager;
6       }
7
8       public void sendRequest(String request) {
9           filterManager.filterRequest(request);
10      }
11  }
```

InterceptingFilterDemo.java

```
1   public class InterceptingFilterDemo {
2       public static void main(String[] args) {
3           FilterManager filterManager = new FilterManager(new Target());
4           filterManager.addFilter(new AuthenticationFilter());
5           filterManager.addFilter(new LogFilter());
6
7           Client client = new Client();
8           client.setFilterManager(filterManager);
9           client.sendRequest("USER");
10      }
11  }
```

运行结果

```
Authenticating request: USER
Logging request: USER
Executing request: USER
```

4.33 服务定位器模式

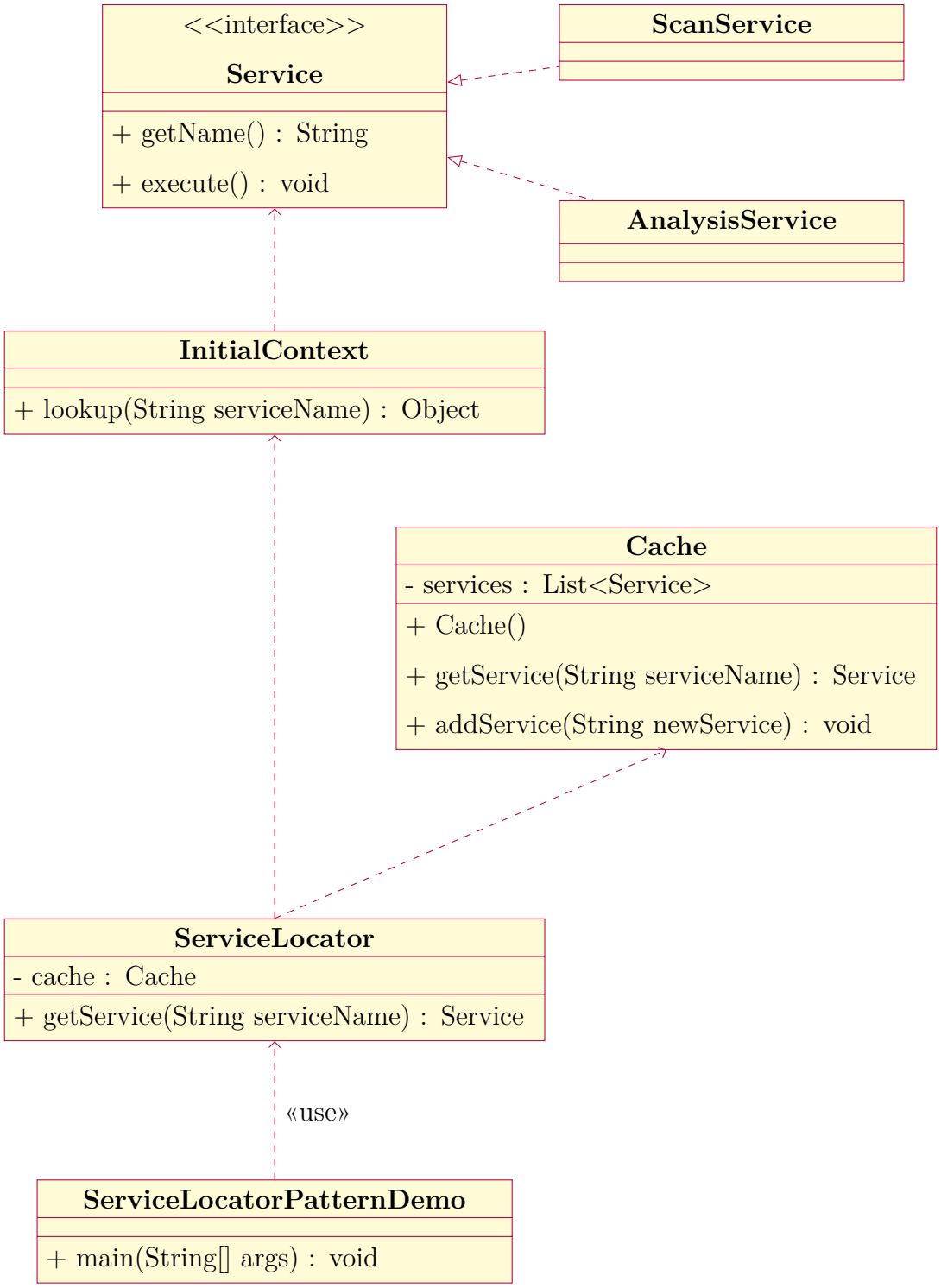
4.33.1 服务定位器模式（Service Locator Pattern）

服务定位器模式用在查询定位各种服务的时候，考虑到为某个服务查找的代价很高，服务定位器模式充分利用了缓存技术。在首次请求某个服务时，服务定位器查找服务，并缓存该服务对象。当再次请求相同的服务时，服务定位器会在它的缓存中查找，这样可以在很大程度上提高应用程序的性能。

拦截过滤器模式包含以下实体：

- 服务：实际处理请求的服务。
- Initial Context：带有对要查找的服务的引用。
- 服务定位器：服务定位器是通过查找和缓存服务来获取服务。
- 缓存：缓存存储服务的引用以便复用。
- 客户端：通过服务定位器调用服务的对象。

服务定位器模式



Service.java

```java
public interface Service {
    String getName();

    void execute();
```

```
}
```

ScanService.java

```
public class ScanService implements Service {
    @Override
    public String getName() {
        return "SCAN";
    }

    @Override
    public void execute() {
        System.out.println("Scanning ...");
    }
}
```

AnalysisService.java

```
public class AnalysisService implements Service {
    @Override
    public String getName() {
        return "ANALYSIS";
    }

    @Override
    public void execute() {
        System.out.println("Analyzing ...");
    }
}
```

InitialContext.java

```
public class InitialContext {
    public Object lookup(String serviceName) {
        if(serviceName.equalsIgnoreCase("SCAN")) {
            System.out.println(
                "Looking up and creating a new SCAN object."
            );
            return new ScanService();
        } else if(serviceName.equalsIgnoreCase("ANALYSIS")) {
```

```
            System.out.println(
                "Looking up and creating a new ANALYSIS object."
            );
            return new AnalysisService();
        }
        return null;
    }
}
```

Cache.java

```
import java.util.ArrayList;
import java.util.List;

public class Cache {
    private List<Service> services;

    public Cache() {
        services = new ArrayList<>();
    }

    public Service getService(String serviceName) {
        for(Service service : services) {
            if(service.getName().equalsIgnoreCase(serviceName)) {
                System.out.println(
                    "Returning cached " + serviceName + " object."
                );
                return service;
            }
        }
        return null;
    }

    public void addService(Service newService) {
        boolean exists = false;
        for(Service service : services) {
            if(service.getName().equalsIgnoreCase(
                newService.getName())
```

```
            ) {
                exists = true;
                break;
            }
        }

        if(!exists) {
            services.add(newService);
        }
    }
}
```

ServiceLocator.java

```
public class ServiceLocator {
    private static Cache cache;

    static  {
        cache = new Cache();
    }

    public static Service getService(String serviceName) {
        Service service = cache.getService(serviceName);

        if(service == null) {
            InitialContext context = new InitialContext();
            service = (Service)context.lookup(serviceName);
            cache.addService(service);
        }

        return service;
    }
}
```

ServiceLocatorPatternDemo.java

```
public class ServiceLocatorPatternDemo {
    public static void main(String[] args) {
        Service service = ServiceLocator.getService("SCAN");
```

```
4           service.execute();
5
6           service = ServiceLocator.getService("ANALYSIS");
7           service.execute();
8
9           service = ServiceLocator.getService("SCAN");
10          service.execute();
11
12          service = ServiceLocator.getService("ANALYSIS");
13          service.execute();
14      }
15  }
```

运行结果

```
Looking up and creating a new SCAN object.
Scanning ...
Looking up and creating a new ANALYSIS object.
Analyzing ...
Returning cached SCAN object.
Scanning ...
Returning cached ANALYSIS object.
Analyzing ...
```

4.34 传输对象模式

4.34.1 传输对象模式（Transfer Object Pattern）

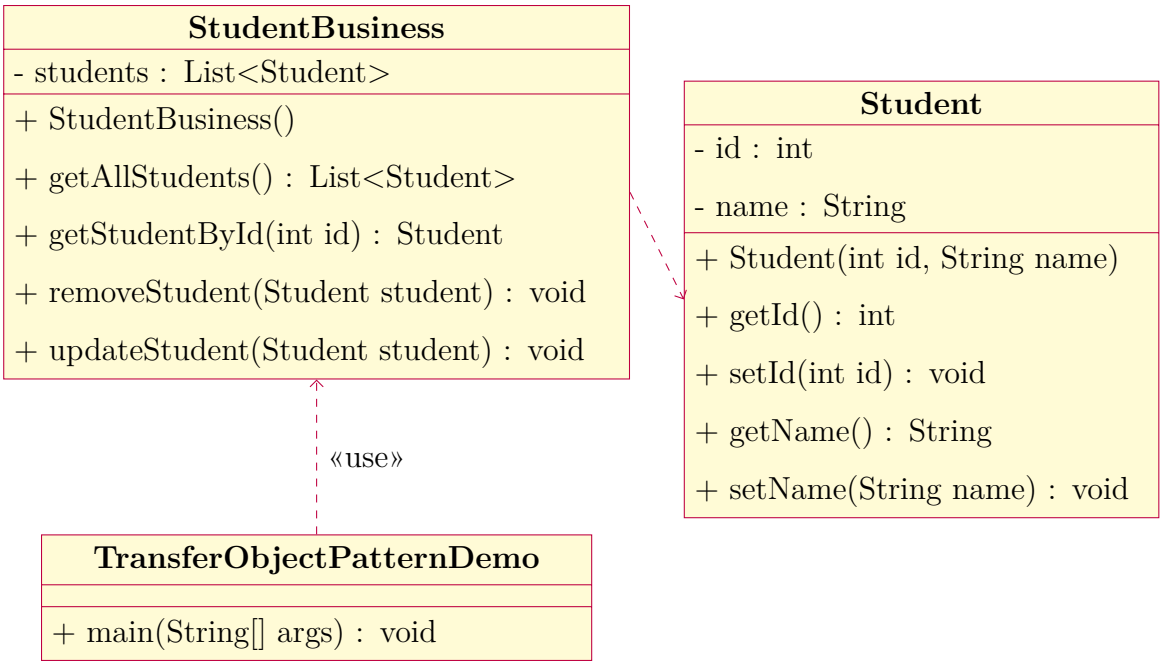
传输对象模式用于从客户端向服务器一次性传递带有多个属性的数据。传输对象是一个具有 getter / setter 的类，它是可序列化的，所以它可以通过网络传输。

服务器端的业务类通常从数据库读取数据，并把它发送到客户端。对于客户端，传输对象是只读的。客户端可以创建自己的传输对象，并把它传递给服务器，以便一次性更新数据库中的数值。

传输对象模式包含以下实体：

- 业务对象：为传输对象填充数据的业务服务。
- 传输对象：具有 getter / setter 的类。
- 客户端：客户端可以发送请求或者发送传输对象到业务对象。

传输对象模式



Student.java

```java
public class Student {
    private int id;
    private String name;

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                '}';
    }
}
```

StudentBusiness.java

```java
import java.util.ArrayList;
```

```
2  import java.util.List;
3
4  public class StudentBusiness {
5      // working as a database
6      private List<Student> students;
7
8      public StudentBusiness() {
9          students = new ArrayList<>();
10         students.add(new Student(0, "Terry"));
11         students.add(new Student(1, "Henry"));
12         students.add(new Student(2, "Lily"));
13     }
14
15     public List<Student> getAllStudents() {
16         return students;
17     }
18
19     public Student getStudentById(int id) {
20         return students.get(id);
21     }
22
23     public void removeStudent(Student student) {
24         students.remove(student.getId());
25         System.out.println(
26             "Student (id=" + student.getId() + ") DELETED."
27         );
28     }
29
30     public void updateStudent(Student student) {
31         students.get(student.getId()).setName(student.getName());
32         System.out.println(
33             "Student (id=" + student.getId() + ") UPDATED."
34         );
35     }
36 }
```

TransferObjectPatternDemo.java

```
public class TransferObjectPatternDemo {
    public static void main(String[] args) {
        StudentBusiness studentBusiness = new StudentBusiness();

        for(Student student : studentBusiness.getAllStudents()) {
            System.out.println(student);
        }

        Student student1 = studentBusiness.getStudentById(1);
        student1.setName("Alice");
        studentBusiness.updateStudent(student1);

        Student student2 = studentBusiness.getStudentById(0);
        studentBusiness.removeStudent(student2);

        for(Student student : studentBusiness.getAllStudents()) {
            System.out.println(student);
        }
    }
}
```

运行结果

```
Student{id=0, name='Terry'}
Student{id=1, name='Henry'}
Student{id=2, name='Lily'}
Student (id=1) UPDATED.
Student (id=0) DELETED.
Student{id=1, name='Alice'}
Student{id=2, name='Lily'}
```

Chapter 5 软件质量

5.1 软件质量

5.1.1 软件质量（Software Quality）

一个质量好的软件从一个优秀的团队和开发过程开始，并且正确地使用系统架构和设计模式，后期严格的审查和测试都能够有效地增加软件的质量。

软件质量的特征包括：

- 功能性
- 可靠性
- 易使用性
- 效率
- 可维护性
- 可移植性

客户/用户而言，他们的满足度取决于软件的质量、产品的一致性和是否在预算和预期时间内交付。

一个高质量的软件应该符合以下特征：

- 易于使用
- 保证安全和隐私
- 较少的维护成本
- 较少的 bug
- 客户满意度所带来的收益

- 在未来的产品中可以复用技术

5.1.2 软件质量保证 (SQA/QA, Software Quality Assurance)

一个公司通常有一个独立的 SQA 小组与开发团队合作，评估正在生产的产品质量。即使在软件开发过程中注重错误的预防，也不能减少对 QA 的需求。

SQA 小组的职责主要包括：

- 准备 SQA 计划：评估标准、审查、错误跟踪。
- 协助开发过程：分析软件的缺陷和流程。

5.2 缺陷预防

5.2.1 代码评审（Code Review）

软件缺陷很大一部分是来自于对需求的定义和理解不正确，code review 是最有效的去除缺陷的手段。越早去除这些潜在的缺陷，所需的代价也越小。

Code review 不是为了去刻意批斗某个开发者，而是为了团队成员之间相互了解学习，加深成员对系统的理解，使团队成员的代码更加健壮，提早发现代码缺陷。

Code review 由一组技术人员组成，主要是为了发现软件中功能和逻辑的漏洞，验证功能是否满足了客户/用户的需求，保持项目的可维护性。

在 code review 过程中，可以根据检查清单（checklist）对产品进行检查，并记录下存在的问题。Code review 的目的在于发现问题，而不是解决问题。

Code review 的好处包括：

- 提升系统的可维护性
- 及早发现潜在 bug，降低事故成本。
- 促进团队内部知识共享，提高团队整体水平。
- 对于评审人员来说是一种思路重构的过程，可以帮助更多的人理解系统。
- 彼此能熟悉对方模块。

5.3 缺陷检测

5.3.1 缺陷检测（Defect Detection

缺陷检测包括三个过程：

1. 测试（testing）：根据测试用例发现错误。
2. 调试（debugging）：查找并消除故障的原因。
3. 监控（monitoring）：监视有关状态和行为的信息。

缺陷检测可以通过静态分析（static analysis）和动态分析（dynamic analysis）两种方式进行。

静态分析包括 code review、向他人解释代码流程、借助自动化的工具检查语法语义错误及代码规范。

动态分析包括：

- 黑盒测试（black-box testing）：测试子系统的输入/输出行为。
- 白盒测试（white-box testing）：测试子系统或类的内部逻辑。

5.3.2 测试

测试最好是由非开发软件的人进行，因为开发者一般会更加注重能够使程序正常工作的数据。当其他人使用程序时，往往会发现程序的问题。

因此一个测试人员非常有必要对系统由足够的了解，掌握各种测试方法和技术。

测试分为 4 种类型：

1. 单元测试（unit testing）：对每个类/子系统单独测试，确保每个模块的正确性。

2. 集成测试（integration testing）：将模块组装成子系统，确保各个模块连接在一起后也能正常工作。

3. 系统测试（system testing）：对整个系统进行测试。

4. 验收测试（acceptance test）：由客户/用户进行测试。