面向对象设计模式　模板

曹东刚
caodg@pku.edu.cn

北京大学信息学院研究生课程 - 面向对象的分析与设计
http://sei.pku.edu.cn/~caodg/course/oo

从咖啡和茶的冲泡说起

咖啡冲泡法:

1. 把水煮沸
2. 用沸水冲泡咖啡
3. 把咖啡倒进杯子
4. 加糖和牛奶

从咖啡和茶的冲泡说起

咖啡冲泡法:

1. 把水煮沸
2. 用沸水冲泡咖啡
3. 把咖啡倒进杯子
4. 加糖和牛奶

茶冲泡法:

1. 把水煮沸
2. 用沸水浸泡茶叶
3. 把茶倒进杯子
4. 加柠檬

从咖啡和茶的冲泡说起

创建咖啡的代码

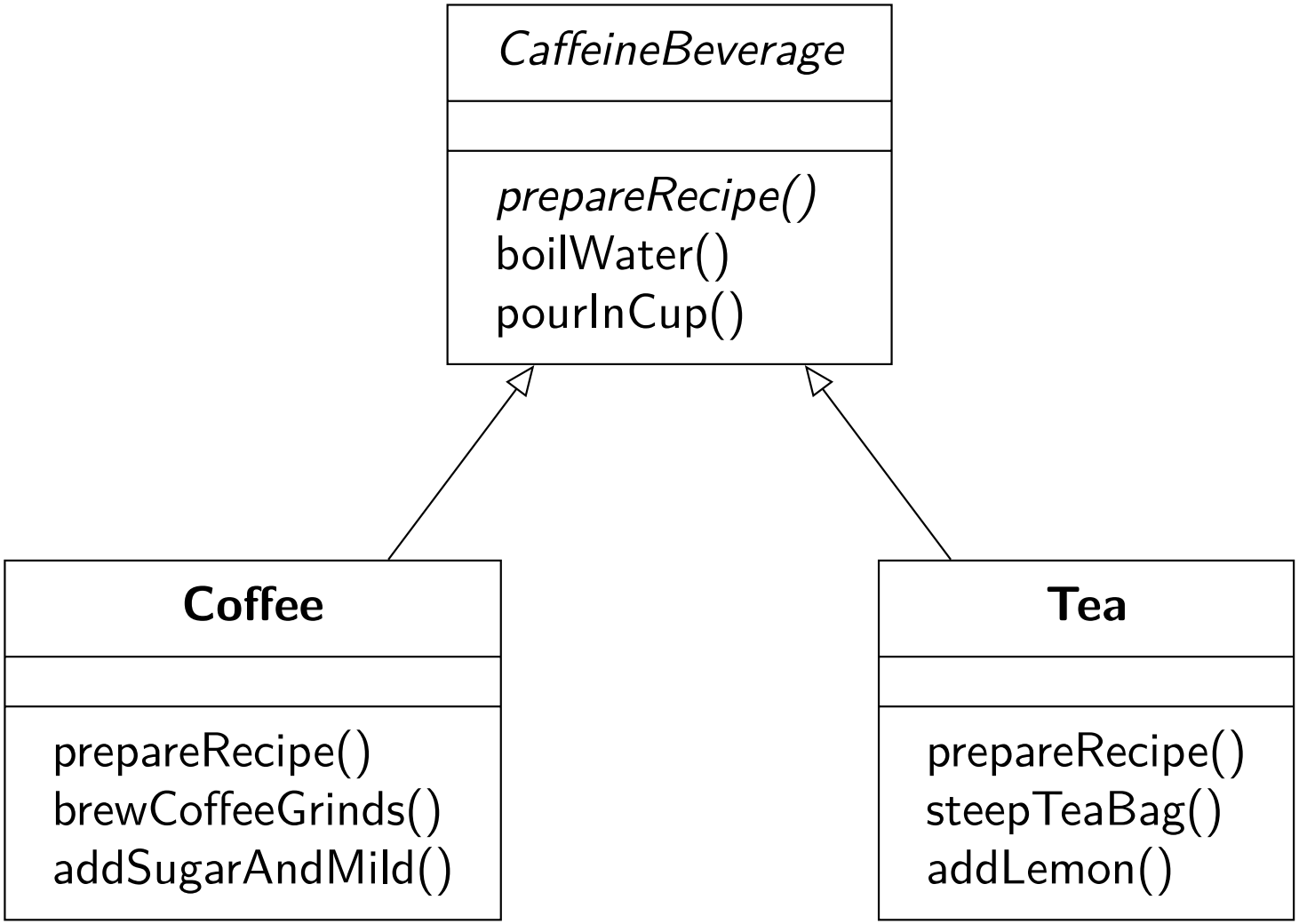
```
1  public class Coffee {
2    void prepareRecipe() {
3      boilWater();
4      brewCoffeeGrinds();
5      pourInCup();
6      addSugarAndMilk();
7    }
8    public void boilWater() {}
9    public void brewCoffeeGrinds() {}
10   public void pourInCup() {}
11   public void addSugarAndMilk() {}
12 }
```

从咖啡和茶的冲泡说起

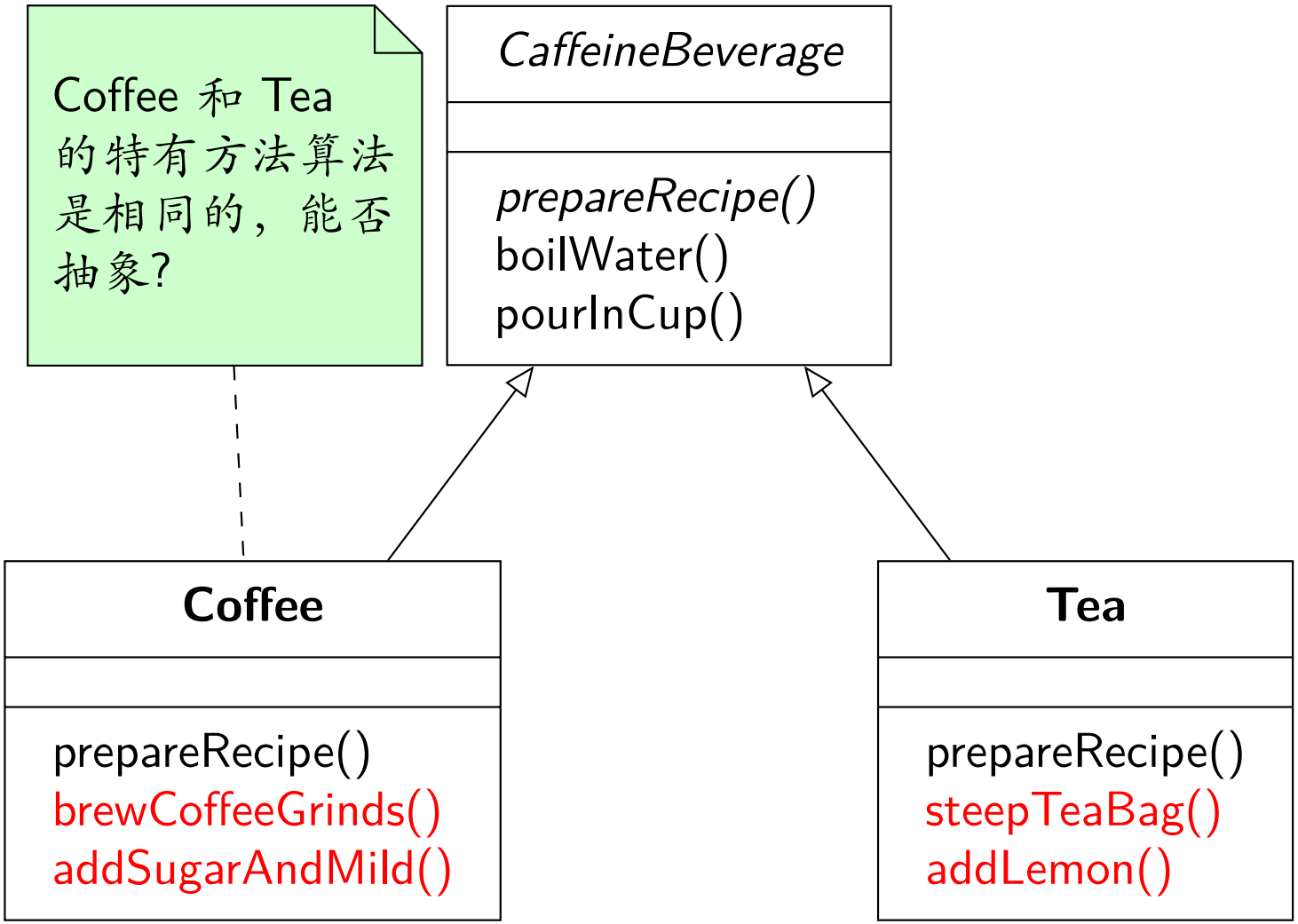
创建茶的代码

```
1  public class Tea {
2    void prepareRecipe() {
3      boilWater();
4      steepTeaBag();
5      pourInCup();
6      addLemon();
7    }
8    public void boilWater() {}
9    public void steepTeaBag() {}
10   public void addLemon() {}
11   public void pourInCup() {}
12 }
```

重构，以消除重复代码


CaffeineBeverage
prepareRecipe()
boilWater()
pourInCup()
Coffee
prepareRecipe()
brewCoffeeGrinds()
addSugarAndMild()
Tea
prepareRecipe()
steepTeaBag()
addLemon()

重构，以消除重复代码


Coffee 和 Tea 的特有方法算法是相同的，能否抽象?
CaffeineBeverage
prepareRecipe()
boilWater()
pourInCup()
Coffee
prepareRecipe()
brewCoffeeGrinds()
addSugarAndMild()
Tea
prepareRecipe()
steepTeaBag()
addLemon()

重构，以消除重复代码

Coffee

```
void prepareRecipe() {
  boilWater();
  brewCoffeeGrinds();
  pourInCup();
  addSugarAndMilk();
}
```

Tea

```
void prepareRecipe() {
  boilWater();
  steepTeaBag();
  pourInCup();
  addLemon();
}
```

重构，以消除重复代码

抽象后的新框架

```
void prepareRecipe() {
  boilWater();
  brew();
  pourInCup();
  addCondiments();
}
```

重构，以消除重复代码

重写 CaffeineBeverage

```
1  public abstract class CaffeineBeverage {
2    final void prepareRecipe() {
3      boilWater();
4      brew();
5      pourInCup();
6      addCondiments();
7    }
8    abstract void brew();
9    abstract void addCondiments();
10   void boilWater() { }
11   void pourInCup() { }
12 }
```

重构，以消除重复代码

重写 Tea

```
1  public class Tea extends CaffeineBeverage
2  {
3    public void brew() {
4      System.out.println("Steeping the tea");
5    }
6
7    public void addCondiments() {
8      System.out.println("Adding Lemon");
9    }
10 }
```

认识模板方法

- CaffeineBeverage 类定义并拥有算法，主导一切
- CaffeineBeverage 的模板方法提供了一个框架，允许其他子类咖啡因饮料加入
- CaffeineBeverage 专注算法本身，由子类提供完整实现
- Coffee 和 Tea 子类通过继承父类，复用代码
- 算法只存在于一个地方，很容易修改

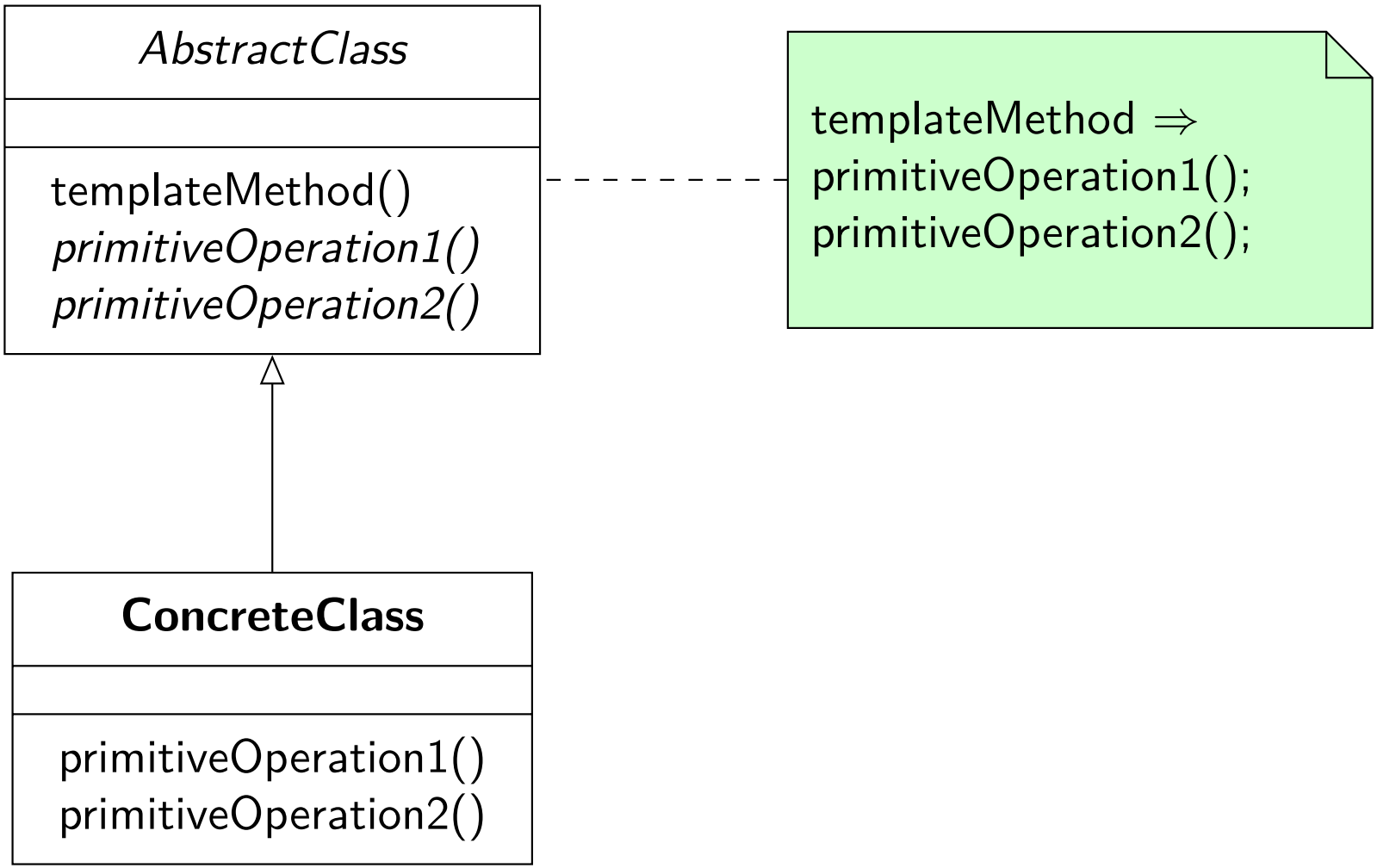
定义模板方法模式

模板方法模式 (Template Method Pattern)

模板方法模式在一个方法中定义一个算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤

模板就是一个方法，该方法将算法定义成一组步骤，其中的任何步骤都可以是抽象的，由子类负责实现

定义模板方法模式


AbstractClass
templateMethod()
primitiveOperation1()
primitiveOperation2()
templateMethod ⇒
primitiveOperation1();
primitiveOperation2();
ConcreteClass
primitiveOperation1()
primitiveOperation2()

定义模板方法模式

抽象类的设计

```
1  abstract class AbstractClass {
2    final void templateMethod() {
3      primitiveOperation1();
4      primitiveOperation2();
5      concreteOperation();
6    }
7    abstract void primitiveOperation1();
8    abstract void primitiveOperation2();
9    void concreteOperation() {
10     // implementation here
11   }
12 }
```

定义模板方法模式

定义了钩子的抽象类

```
1  abstract class AbstractClass {
2    final void templateMethod() {
3      primitiveOperation1();
4      primitiveOperation2();
5      concreteOperation();
6      hook();
7    }
8    abstract void primitiveOperation1();
9    abstract void primitiveOperation2();
10   void concreteOperation() {
11     // implementation here
12   }
13   void hook() {}
14 }
```

钩子 (hook) 的作用

- 钩子是一种被声明在抽象类里的方法，但只有空的或默认的实现
- 钩子的存在，使得子类有能力对算法的不同点进行挂钩。要不要挂钩，由子类决定
- 如果子类必须提供算法的某个步骤，用抽象方法；如果某个步骤是可选的，用钩子
- 钩子可以让子类对模板方法中某些即将发生的步骤做出反应

钩子 (hook) 的作用

使用钩子的 CaffeineBeverage

```
1  public abstract class CaffeineBeverageWithHook {
2    final void prepareRecipe() {
3      boilWater();
4      brew();
5      pourInCup();
6      if (customerWantsCondiments()) {
7        addCondiments();
8      }
9    }
10   abstract void brew();
11   abstract void addCondiments();
12   boolean customerWantsCondiments() { return true ;}
13 }
```

设计原则：好莱坞原则

好莱坞原则

别调用我们，我们会调用你

- 好莱坞原则可以防止“依赖腐败”— 系统的各个构件彼此互相依赖，关系复杂，难以维护
- 好莱坞原则使低层构件可以挂钩到系统上，但由高层构件决定何时和如何使用低层构件，即，高层构件对待低层构件是：“别调用我们，我们会调用你”
- 模板方法模式体现了好莱坞原则

Java Arrays 的模板方法排序

java.util.Arrays 的部分排序代码

```
1  public static void sort(Object[] a) {
2    Object aux[] = (Object[])a.clone();
3    mergeSort(aux, a, 0, a.length, 0);
4  }
```

Java Arrays 的模板方法排序

java.util.Arrays 的部分排序代码

```
5  private static void mergeSort(Object src[], Object dest[],
6    int low, int high, int off)
7  {
8    for (int i=low; i<high; i++){
9      for (int j=i; j>low &&
10       ((Comparable)dest[j-1]).compareTo(
11         (Comparable)dest[j])>0; j--) {
12       swap(dest, j, j-1);
13     }
14   return;
15 }
```

Java Arrays 的模板方法排序

让鸭子支持排序

```
1  public class Duck implements Comparable {
2    String name;
3    int weight;
4    public int compareTo(Object object) {
5      Duck otherDuck = (Duck)object;
6      if (this.weight < otherDuck.weight) {
7        return -1;
8      } else if (this.weight == otherDuck.weight) {
9        return 0;
10     } else
11       return 1;
12   }
13 }
```

关于模板方法模式的小结

- 模板方法定义了算法的步骤，把步骤的实现延迟到了子类
- 模板方法模式提供了一种代码复用的重要技巧
- 模板方法的抽象类可以定义具体方法、抽象方法和钩子
- 抽象方法由子类实现
- 钩子在抽象类中要么不做事，要么做缺省的事；子类可以选择是否重载他们
- 为了防止子类改变模板方法中的算法，可以将模板方法声明为 **final**
- 好莱坞原则告诉我们，将决策权放到高层模块中，以便决定如何及何时调用低层模块

关于设计原则的小结

1. 封装变化
2. 多用聚合、少用继承
3. 针对接口编程，不针对实现编程
4. 尽最大可能将要交互的对象设计为松耦合的
5. 对扩展开放，对修改封闭
6. 依赖抽象，不要依赖具体类
7. 只和朋友交谈
8. 别找我，我会找你