南京大学
NANJING UNIVERSITY

《软件工程与计算II》

Ch16 设计模式

Main Contents

- Design Pattern Introduction
- Information Hiding AND Strategy
- Object Creation AND Factory, Factory Method, Abstract Factory, Singleton
- Programming to Interface AND Iterator

Design Patterns Introduction

Why?

- Designing OO software is hard
- Designing *reusable* OO software – harder
- Experienced OO designers make good design
- New designers tend to fall back on non-OO techniques used before
- Experienced designers know something – what is it?

Design Patterns Introduction

Why?

- Expert designers know *not* to solve every problem from first principles
- They *reuse* solutions
- These patterns make OO designs more flexible, elegant, and ultimately reusable

设计模式

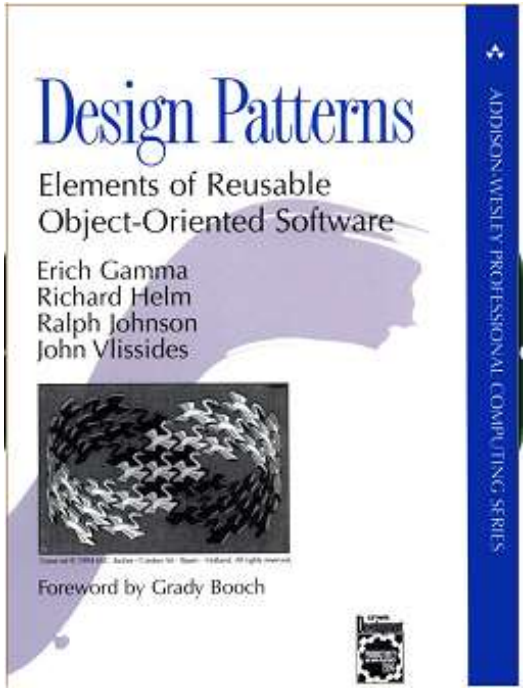
- “设计模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心”

  Christopher Alexander，《建筑的永恒之道》

- 是经过实践反复检验、能解决关键技术难题、有广泛应用前景和能够显著提高软件质量的有效设计经验总结

What is a Design Pattern?

- A design pattern
  - abstracts a recurring design structure
  - comprises class and/or object
    - dependencies,
    - structures,
    - interactions, or
    - conventions
  - distills design experience



Elements of Design Patterns

Design patterns have 4 essential elements:

- Pattern name: increases vocabulary of designers
- Problem: intent, context, when to apply
- Solution: UML-like structure, abstract code, responsibility, collaboration
- Consequences: results and tradeoffs

Main Contents

- Design Pattern Introduction
- Information Hiding AND Strategy
- Object Creation AND Factory, Factory Method, Abstract Factory, Singleton
- Programming to Interface AND Iterator

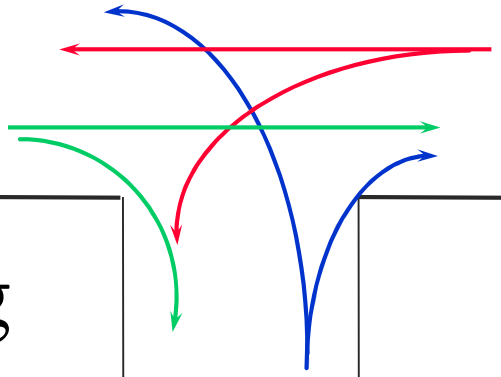
Information Hiding(1)

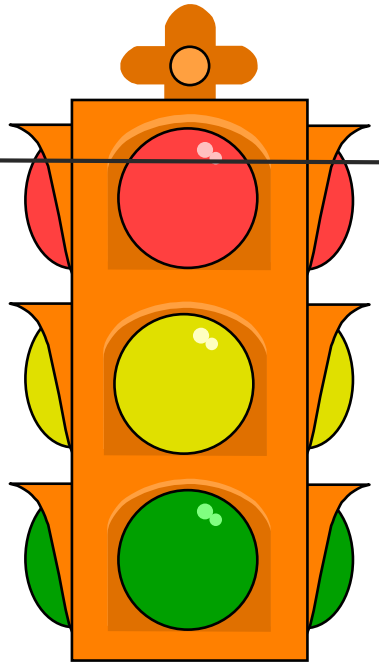
- Each module hides the implementation of an important design decision so that only the constituents of that module know the details

Information Hiding(2)

- Modules may have additional secrets: Changes
- Anticipate changes
  - You then separate each design secret by assigning it to its own class, subroutine, or other design unit.
  - Next you isolate (encapsulate) each secret so that if it does change, the change doesn't affect the rest of the program.

Intersection Traffic Lights Control





The light-switching policy changes by the hour

- The “dumb” policy: change the green route every 5 seconds
- Midnight policy: change to yellow always
- Rush hour policy: double the “green time” in the busy route

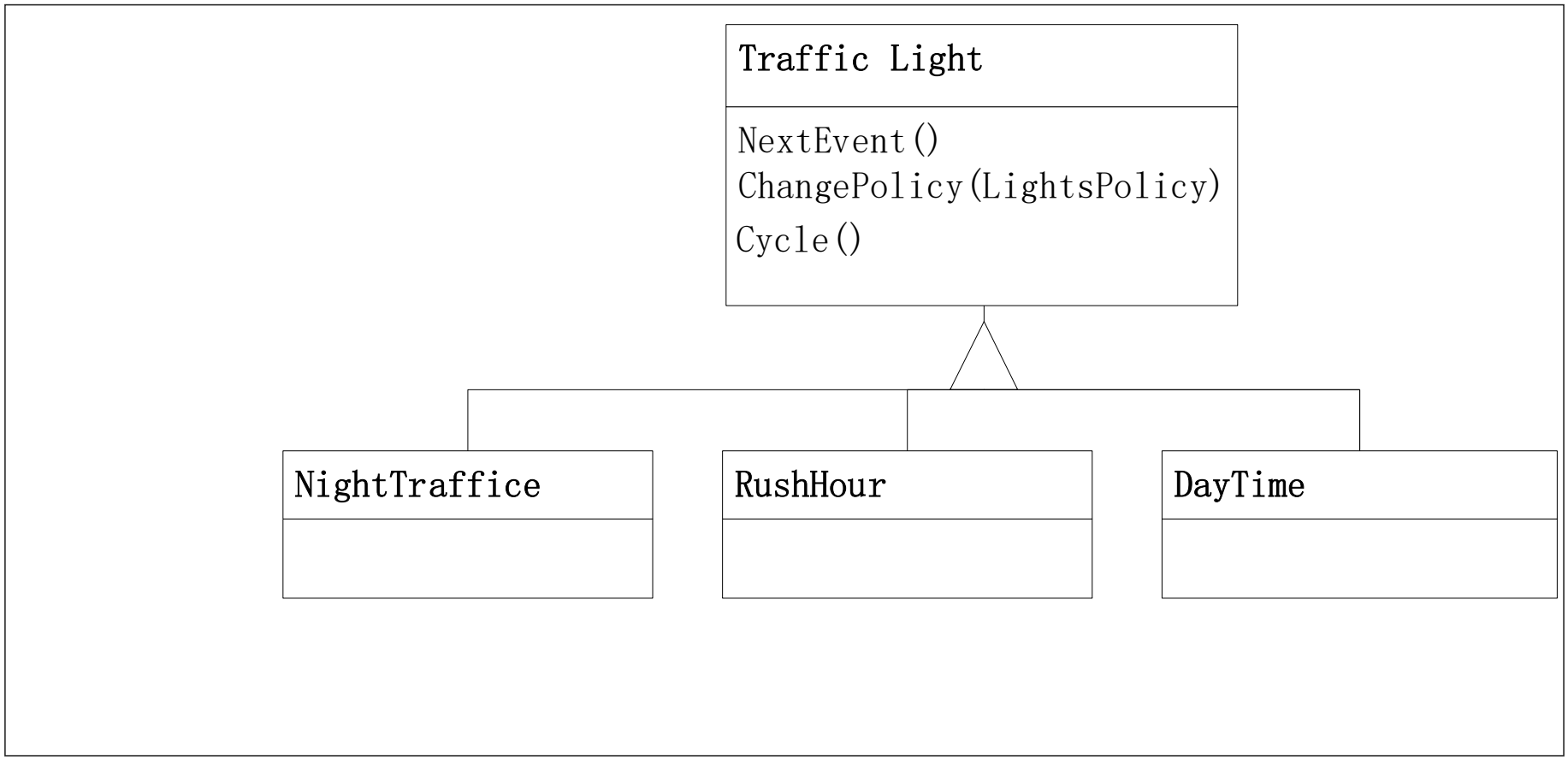
A Bad Solution

Use multiple conditional statement to set the behavior according to current policy

```
intersection::next_green(the_green_route r)
{
  switch (current_policy) {

    case dumb:  next_time := time + constant_time;

    case midnight: if (not the_green_route.is_busy)
       then …

    case rush_hour: …
  }

}
```

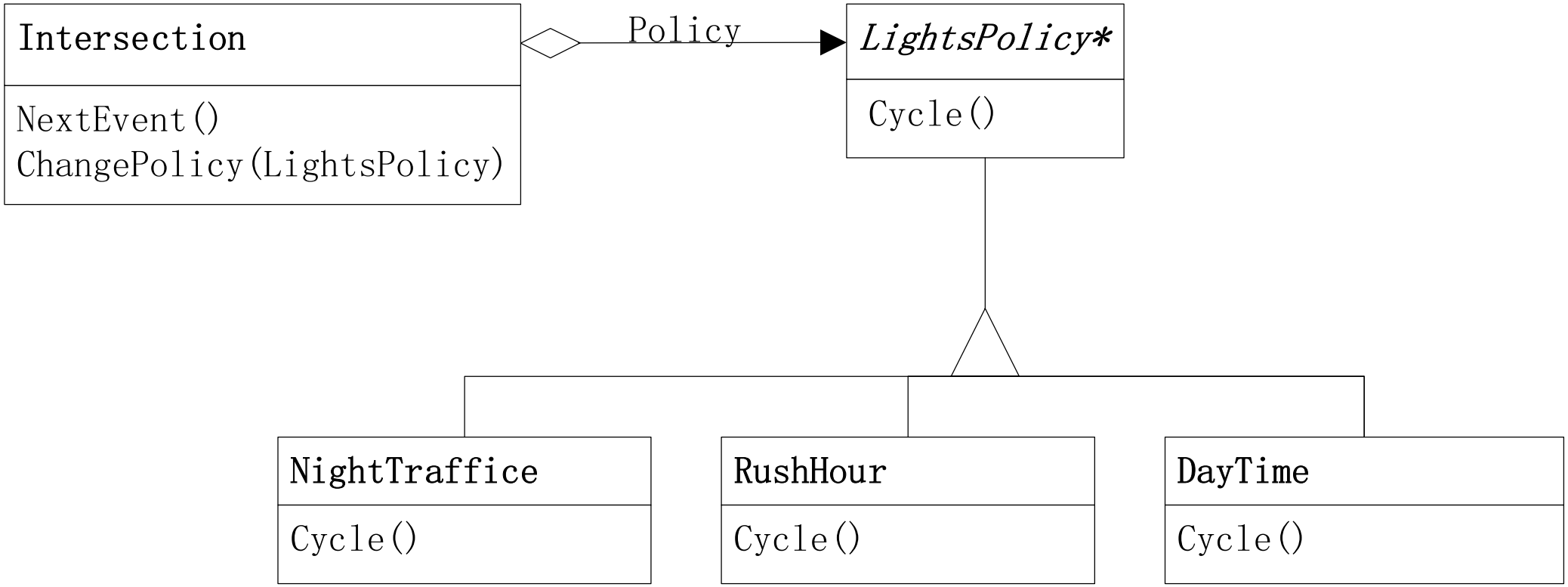
Better, but not enough

Plan the change



Strategy Pattern

Traffic Lights Management



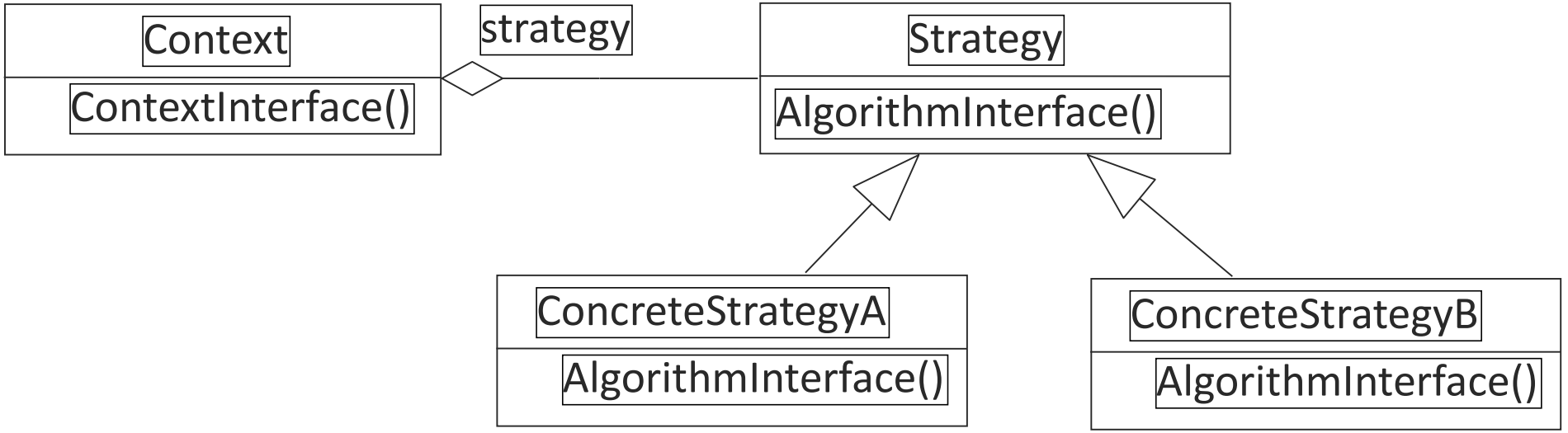
Strategy Pattern

Problem

- Use the Strategy pattern when
  - many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors
  - a class defines many behaviors, and these appear as multiple conditional statements in its operations
  - you need different variants of algorithm
  - an algorithms uses data that client shouldn’t know about

Strategy Pattern

STRATEGY Solution



Strategy Pattern

STRATEGY - Consequences

- Changeability and Reusability
  - Families of related algorithms for extend
  - An alternative to sub-classing
    - you can subclass **Context** class directly to give it different behaviors
- Complex for comprehension
  - Good: Strategies eliminate conditional statements
  - Bad: Logic is separated, Clients must be aware of different Strategies
    - a client must understand how strategies differ before it can select the appropriate one

Strategy Pattern

STRATEGY -Summary

- If there are complex and vary algorithms, then there may be Strategy Pattern
- If there is a changeable behavior in some fixed characteristics, then there may be Strategy Pattern
- If there are complex conditional statements, then there may be Strategy Pattern

典型问题

- 在一个大规模的连锁超市中雇员的薪水支付可以分为很多种。其中雇员的薪酬支付方式和支付频率就有好几种：
  - 有些雇员是钟点工，按时薪来支付。薪水=时薪*工作小时数。每周三支付。
  - 有些雇员按月薪支付。薪水=固定月薪。每月21日支付。
  - 有些雇员是提成制。薪水=销售额*提成比率。每隔一周的周三支付。

最糟糕的实现：Conditional Statement

```
class PaymentStrategy{
    //拥有每个雇员的支付相关的数据
    ArrayList<DOUBLE> hourList = new ArrayList< DOUBLE >();
    ArrayList< DOUBLE > hourRateList = new ArrayList< DOUBLE >();
    ArrayList< DOUBLE > contractValueList = new ArrayList< DOUBLE >();
    ArrayList< DOUBLE > commissionRateList = new ArrayList< DOUBLE >();
    ...
    //计算需要支付的金额
    public double calculatePayment(int employeeID){
        switch(e.getPaymentClassification()){
            case HOURLY:
                return hourList.get (employeeID)* hourRateList.get (emplyeeID);
                break;
            case COMMISSIONED:
                return contractValueList.get (employeeID)* commissionRateList.get
        (emplyeeID);
                break;
            case SALARIED: …
        }
    }
    public boolean isPayDay(int employeeID){
        switch(e.getPaymentSchedule()){
            case MONTHLY: …
            case WEEKLY: …
            case BIWEEKLY: …
        }
    }
}
```

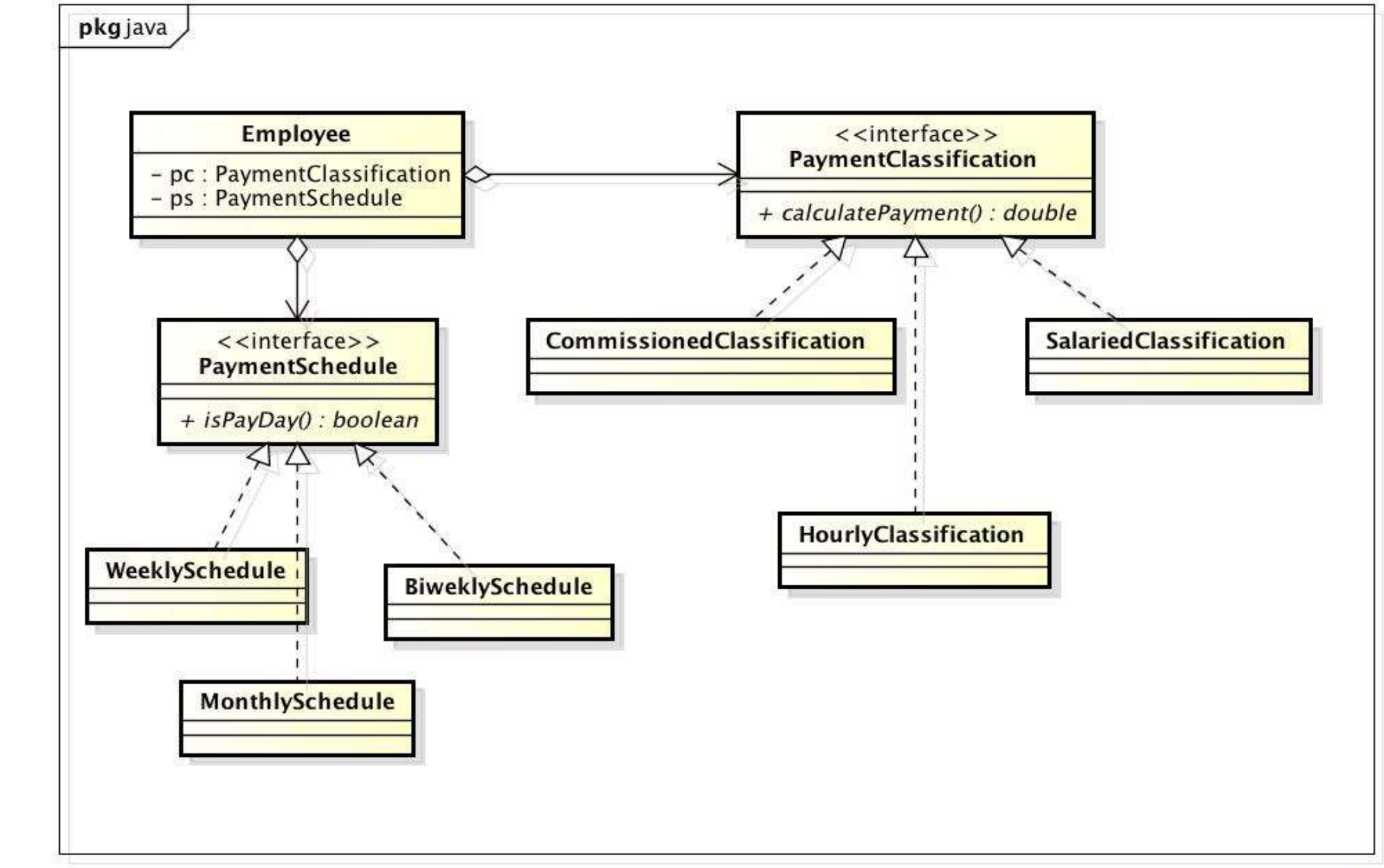
潜在的变化

- 钟点工可能两星期支付一次；
  - (M)实现的可修改性
- 现在是时薪以后可能会变为月薪；
  - (C)实现的灵活性
- 也有可能出现新的薪水支付方式和支付频率。
  - (E)实现的可扩展性

Secret 分析

- Payment
  - 需求：支付功能
  - 变更1：支付方式
  - 变更2：支付频率

案例：策略模式实现


pkg java
Employee
- pc : PaymentClassification
- ps : PaymentSchedule
<<interface>>
PaymentClassification
+ calculatePayment() : double
<<interface>>
PaymentSchedule
+ isPayDay() : boolean
CommissionedClassification
SalariedClassification
HourlyClassification
WeeklySchedule
BiweklySchedule
MonthlySchedule
powered by astah*

策略模式深入分析

- 可变更：信息隐藏
  - （M）实现的可修改性
    - 对已有实现的修改
    - 例如：修改现有促销策略
  - （E）实现的可扩展性
    - 对新的实现的扩展
    - 例如：增加一条新的促销策略
  - （C）实现的灵活性
    - 对实现的动态配置
    - 例如：通过修改数据或配置更改某商品对应促销策略

如何实现可修改性、可扩展性、灵活性?

- 可修改性、可扩展性
  - 符合DIP原则的抽象类（继承）机制
    - 通过定义抽象类与继承（实现）它的子类强制性地做到：接口与实现的分离，进而实现上述质量
      - 强制性地使用抽象类起到接口的作用
      - 强制性地使用子类起到实现的作用
- 灵活性
  - 组合（委托）机制
    - 动态调整所委托的类，实现灵活性

使用Interface的抽象类机制

- interface 定义了接口

```
public interface Interface_A {
    // 接口
    public void method_A();
}
```

- Class是接口的实现

```
public class Class_A1 implements Interface_A {
    public void method_A(){
        // 实现
        System.out.println("Class_A1 's method_A()!");
    }
}
```

使用继承的抽象类机制

- “父类”起到接口的作用
  - 父类定义的接口规约比其实现代码更加重要

```
public class Super_A{
    public void method_A(){
        // 父类的接口和父类的实现
        System.out.println(“Super_A ‘s method_A()!”);
    }
```

- “子类”是对父类所定义接口规约的实现
  - 子类所继承的父类实现代码不会扮演关键作用

```
public class Sub_A1 extends Super_A{
   public void method_A(){
        // 子类的实现
        System.out.println(“Sub_A ‘s method_A()!”);
   }
}
```

利用抽象类机制实现可修改性和可扩展性

- 只要方法的接口保持不变，方法的实现代码是比较容易修改的，不会产生连锁反应

- 通过简单修改创建新类的代码，就可以相当容易地做到扩展新的需求（不用修改大量与类方法调用相关的代码

利用委托机制实现灵活性

- 继承的缺陷：一旦一个对象被创建完成，它的类型就无法改变，这使得单纯利用继承机制无法实现灵活性（类型的动态改变）
- 利用组合（委托）机制可以解决这个问题

利用委托机制实现灵活性

只需要调整Frontend的委托类（Backend），就可以实现Client与Backend之间的灵活性

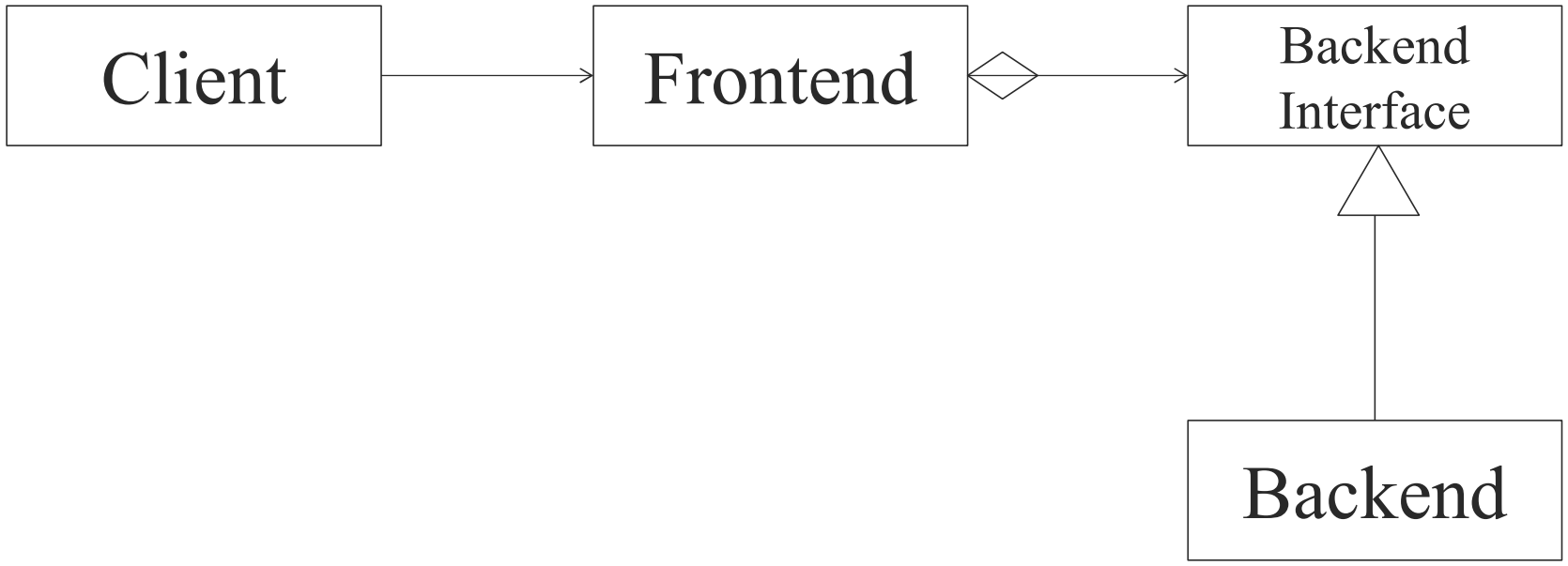
```
class Backend{
  public int method_2(){
  }
}
```

```
class Frontend{
  public Backend back = new Backend();
  public int method_2(){
     back.method_2();
  }
}
```

```
class Client{
  public static void main(String[] args){
     Frontend front = new Frontend();
     int i = front.method_2();
  }
}
```

更好的灵活性实现方案

- 委托与抽象类机制的结合



Strategy体现的设计原则？

- SRP
  - 将变化与需求职责分离
- Favorite Composition over Inheritance
  - 实现灵活性
- OCP，LSP，DIP
  - 将可修改和可扩展都统一为可扩展
- 可能会违反：To be Explicit
  - 如果由client来决定algorithm的创建，那么Context将不再能明确algorithm细节
    - 可变更性的必要代价

Main Contents

- Design Pattern Introduction
- Information Hiding AND Strategy
- Object Creation AND Factory, Factory Method, Abstract Factory, Singleton
- Programming to Interface AND Iterator

OO的对象创建与撤销问题

- 类型的创建与销毁 VS 对象的创建与销毁?
  - 声明即创建　　主动创建
  - 生命周期结束　　不再被需要时
  - 自动销毁　　主动销毁
- 为解决对象销毁问题，需要垃圾收集机制
- 解决对象创建问题，完全依赖程序员！
  - 依赖于很多具体方法的使用
    - 单件模式、工厂、工厂方法、抽象工厂
    - 原型模式、代理模式（虚拟创建）、容器

Object creation: Simple

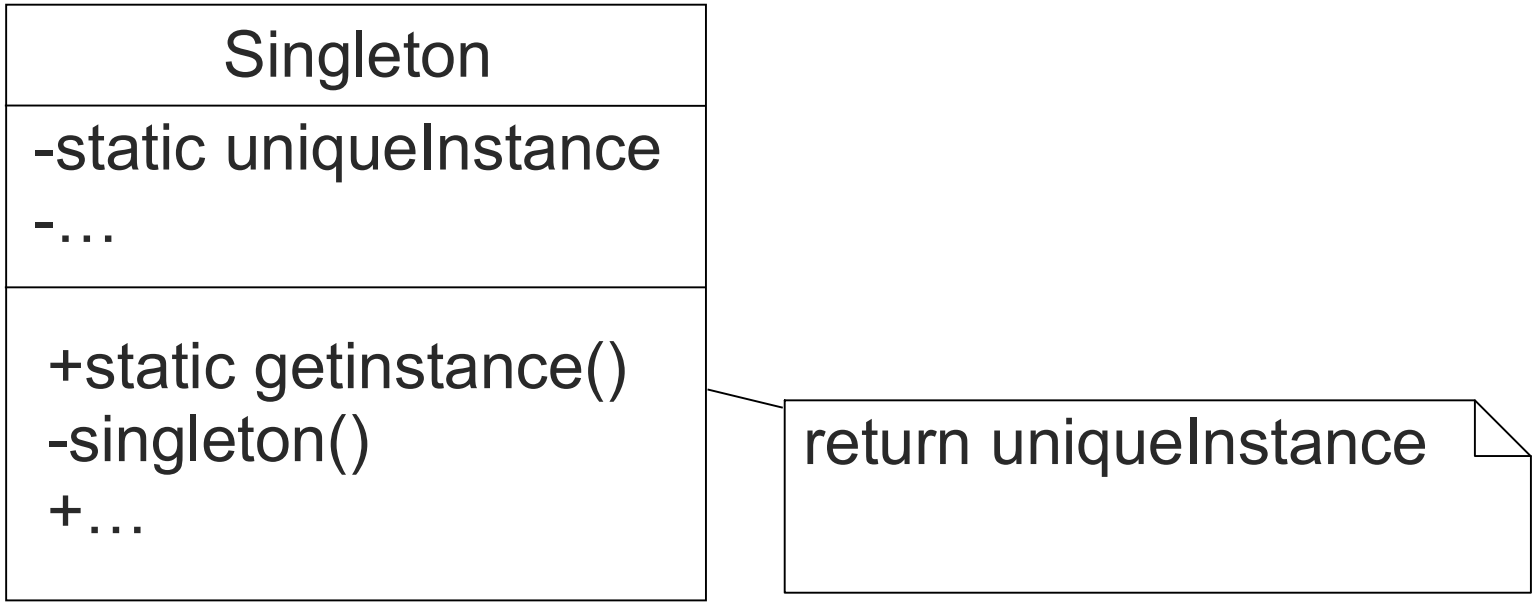
- Creational connections with others
  - Unlimited instances
  - Creating one type
  - Simple instantiation and initialization
- Methods： Creator Pattern
  - 高内聚、低耦合
    - 如果在A的所有关联类中,B和A具有最强的耦合，那么优先选择让B 创建A
    - 创建关系也是一种耦合，强度类似于 聚集耦合

Object creation: Complex (1)

- Limited instances
- Scenario 1: only one instance permitted

- Pattern: Singleton
  - problem: sometimes we will really only ever need one instance of a particular class
    - examples: keyboard reader, bank data collection
    - we'd like to make it illegal to have more than one, just for safety's sake

Singleton: structure



Implementing Singleton

- make constructor(s) private so that they can not be called from outside
- declare a single static private instance of the class
- write a public getInstance() or similar method that allows access to the single instance
  - possibly protect / synchronize this method to ensure that it will work in a multi-threaded program

Singleton example

- consider a singleton class RandomGenerator that generates random numbers

```
public class RandomGenerator {
  private static RandomGenerator gen;

  public static RandomGenerator getInstance() {
     if (gen == null)
      gen = new RandomGenerator();
    return gen;
  }

  private RandomGenerator() {}

  public double nextNumber() {
    return Math.random();
  }
}
```

Singleton模式体现了哪些思想?

- 封装(信息隐藏：外部抽象与内部结构)
  - Principle#1 最小化访问
- 实现软件需求需要的是“对象创建方法”，不是“构造方法”
  - 如果“构造方法”与“对象创建方法”效果相同，可以直接替代
  - 如果“构造方法”与“对象创建方法”效果不同，需要将“对象创建方法”作为外部表现，并加工“构造方法”以实现内部结构
- 操作符重载?
- 有没有违反： Global Variables Consider Harmful?

Object creation: Complex (2)

- Scenario 2: type variations

```
Pizza OrderPizza(String type) {
    Pizza pizza;

    if (type.equals("cheese")) {
        pizza = new CheesePizza();
    } else if (type.equals("greek")) {
        pizza = new GreekPizza();
    } else if (type.equals("pepperoni")) {
        pizza = new PepperoniPizza();
    }

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();

    return pizza;
}
```

This is what varies. As the pizza selection change over time, you'll have to modify this code

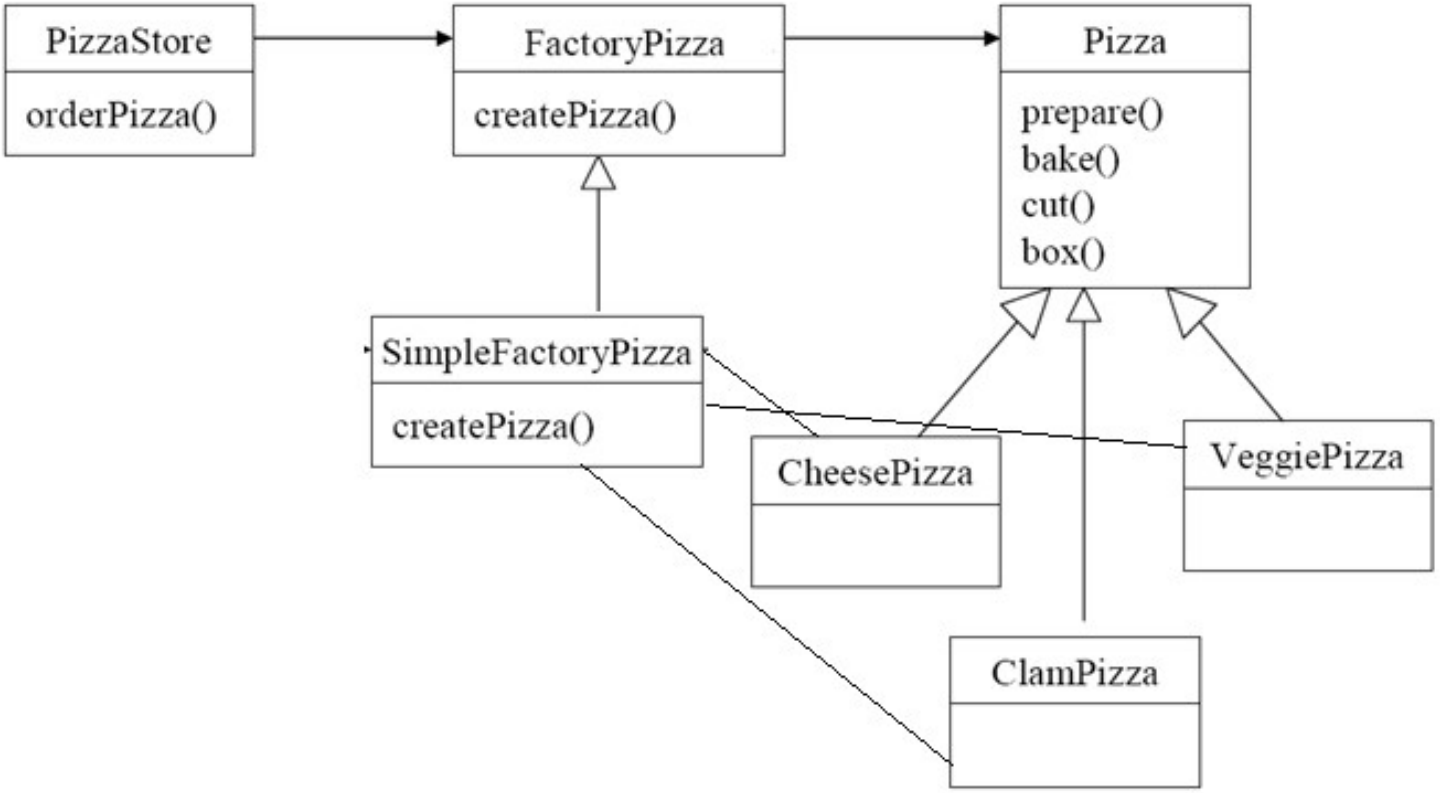
Encapsulating object creation

```
Pizza OrderPizza(String type) {
    Pizza pizza;

    if (type.equals("cheese")) {
        pizza = new CheesePizza();
    } else if (type.equals("greek")) {
        pizza = new GreekPizza();
    } else if (type.equals("pepperoni")) {
        pizza = new PepperoniPizza();
    }

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();

    return pizza;
}
```

SimplePizzaFactory

Factory

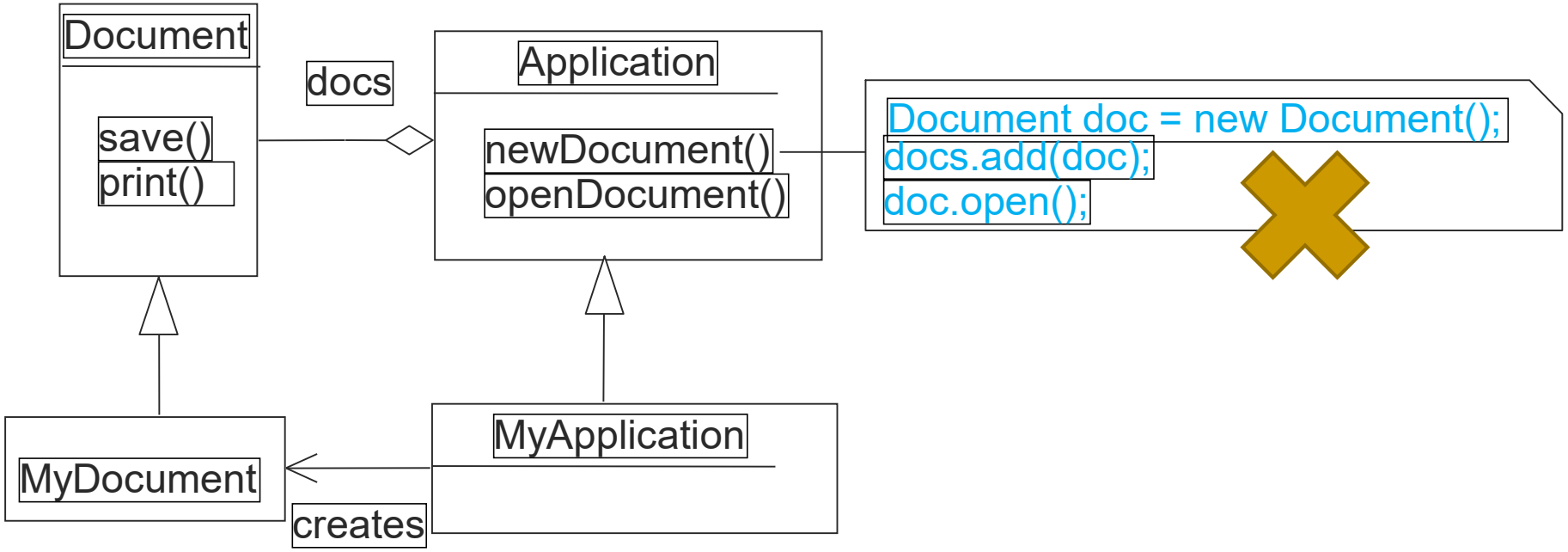
- Factory: a class which's responsibility is to creating other class with vary types



Factory体现了哪些思想?

- Do not repeat!
- 封装（信息隐藏）：将对象创建抽象为单独职责，并隐藏创建细节
  - 有利于对象创建的变更
- 特别是Conditional Statements

A More complex scenario: type variations



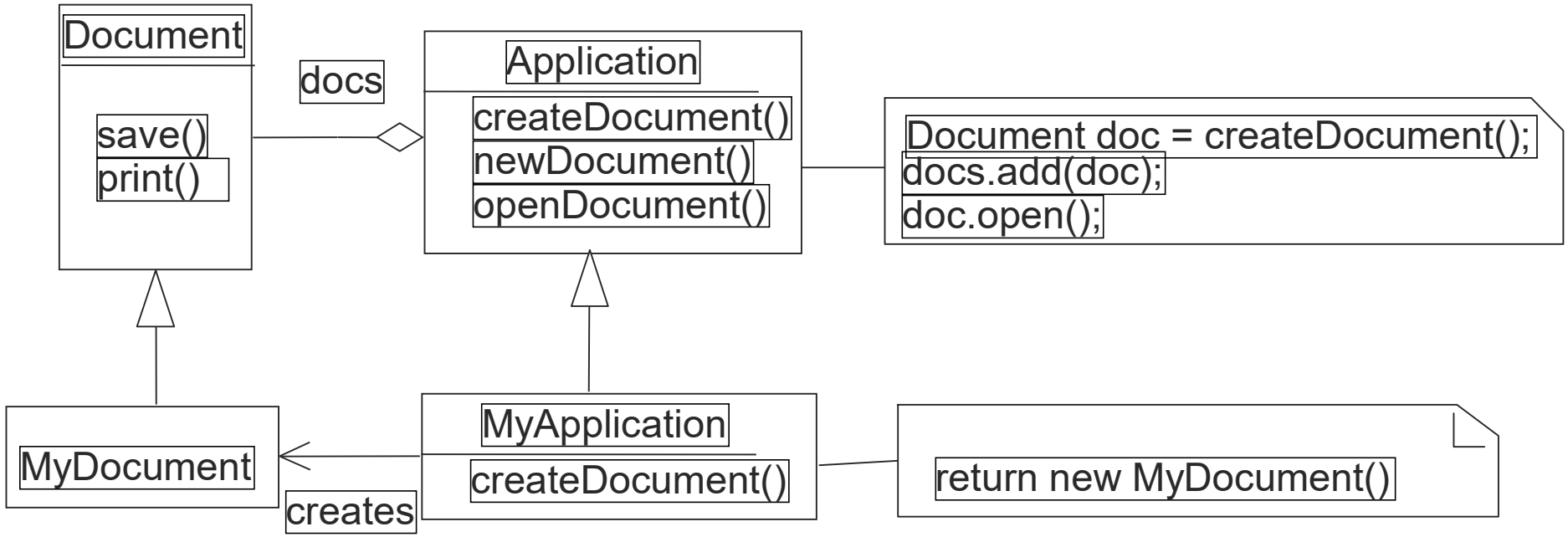
`Application` class is responsible for creation (and management) of `Documents`

Problem:

- `Application` class knows: WHEN a new document should be created
- `Application` class doesn't know: WHAT KIND of document to create

A More complex scenario: type variations

- Solution:
  - Application defines a virtual function, createDocument()
  - MyApplication makes sure that createDocument() will create a product (Document) of the correct type.



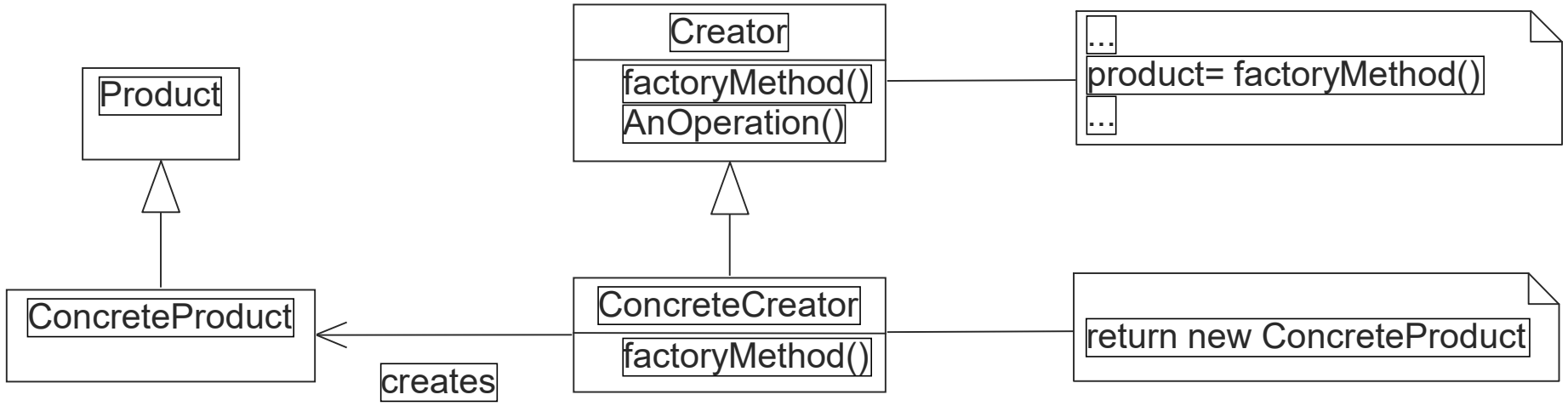
Factory Method: intent

Define an interface for creating an object (replace construction method), but let subclasses decide which class to instantiate.

Lets a class defer instantiation to subclasses.

Factory method is not a simple factory!
Factory method = A method like factory

Factory Method: structure



Factory Method: Consequences

- Changeability , Reusability
  - Concrete (Dynamic) types are isolated from the client code
  - Provides hooks for subclasses: the factory method gives subclasses a hook for providing an extended version of an object
  - Connects parallel class hierarchies: a clients can use factory methods to create a parallel class hierarchy
- Complex
  - Clients might have to subclass the Creator class just to create a particular ConcreteProduct object

Factory Method体现了哪些思想?

- 封装（职责抽象）
  - 区分“对象创建方法”与“构造方法”
- 通过将“对象创建方法”外置（置入Client继承结构中），实现了“对象创建”的多态
  - DIP, LSP, OCP

Factory Method与Factory的关键区别在哪里?

- 创建方法的执行时机与创建类型信息是否统一!
- 统一的时候使用Factory
  - 此时的创建方法不需要多态
- 不统一的时候使用Factory Method
  - 此时的创建方法需要多态

对象创建的更复杂问题（1）

- 而在软件系统中，经常面临着“多种差异对象”的创建工作。
- 比如汽车由引擎、轮胎、车身、车门等各部件组成。而每一部件都有很多种。

- 如何解决该创建问题?
  - 超级工厂?

```
CarFactory {
    CarBody creatCarBody (int cbtype){
        choose case cbtype
            case 1: return new CarBody1();
          break;
            …
        }
    Engine creatEngine(int etype){
            choose case etype
            case 1: return new Engine1();
                    break;
            …
        }
```

对象创建的更复杂问题（2）

- 而在软件系统中，经常面临着“多种差异对象”的创建工作，由于需求的变化，多种对象的具体实现有时候需要灵活组合。
- 汽车有很多类型，车的类型可以决定部件的类型。

- 改进超级工厂？

- CarFactory {

```
CarBody creatCarBody (int cbtype){
        choose case cbtype
                case 1: return new CarBody1();
                break;
                …
        }
Engine creatEngine(int etype){
        choose case etype
                case 1: return new Engine1();
                        break;
                …
        }
```

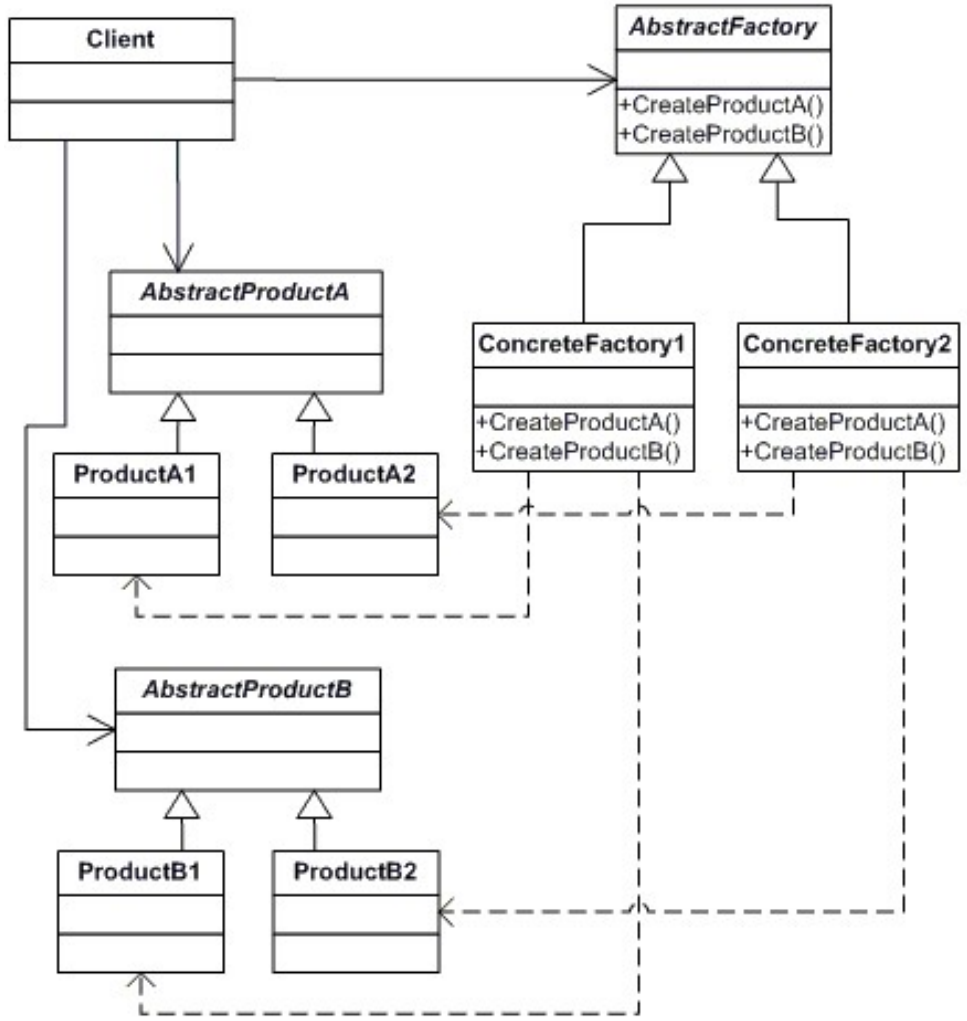
当cbtype与etype统一起来，会发生什么?

```
CarFactory1: CarFactory {
    CarBody creatCarBody (){
            return new CarBody1();
    }
    Engine creatEngine( ){
            return new Engine1();
    }
    ….
}
CarFactory2: CarFactory {
    …….
}
……
```

Client需要（使用新的Factory来）创建CarFactory的具体类型

Abstract Factory: The Solutions

- **AbstractFactory**
  Declares an interface for operations that create abstract products
- **ConcreteFactory**
  - Implements the operations to create concrete product objects: usually instantiated as a Singleton
- **AbstractProduct**
  - Declares an interface for a type of product object; Concrete Factories produce the concrete products
- **ConcreteProduct**
  - Defines a product object to be created by the corresponding concrete factory



应用场景

- 抽象工厂模式可以帮助系统独立于如何对产品的创建、构成、表现。
- 抽象工厂模式可以让系统灵活配置拥有某多个产品族中的某一个。
- 一个产品族的产品应该被一起使用，抽象工厂模式可以强调这个限制。
- 如果你想提供一个产品的库，抽象工厂模式可以帮助暴露该库的接口，而不是实现。

应用注意点

- 隔离了客户和具体实现。客户可见的都是抽象的接口。
- 使得对产品的配置变得更加灵活。
- 可以使得产品之间有一定一致性。同一类产品可以很容易一起使用。
- 但是限制是对于新的产品的类型的支持是比较困难。抽象工厂的接口一旦定义好，就不容易变更了。
- 而这个场景的“代价”，或者是“限制”，是一个工厂中具体产品的种类是稳定的。

Abstract Factory体现了哪些思想?

- Double： Do not repeat
  - 对单一对象的创建代码可能会重复
  - 不同对象创建重复于同一个组合定义
- 封装（信息隐藏）：将对象创建抽象为单独职责，并隐藏创建细节
- OCP、DIP、LSP
  - 提供抽象的创建接口定义
  - 产品组合的可扩展性
- 轻度违反：To be Explicit
  - 对象组合信息被置入ConcreteFactory，在Client代码中将无从知晓

Main Contents

- Design Pattern Introduction
- Information Hiding AND Strategy
- Object Creation AND Factory, Factory Method, Abstract Factory, Singleton
- Programming to Interface AND Iterator

Programming to Interfaces, not Implementations

- “Connections that address or refer to a module as a whole by its name yield lower coupling than connections referring to the internal elements of another module”
  - Collections
  - Inheritance

Single values VS a collection of values

```
class Album {
  private List tracks =new ArrayList();
  public List getTracks() {
    return tracks;
   }
}
```

Client代码对Album的侵入：

- Album的tracks存储区
- 需要为该存储区建立一个访问接口
  - Liskov: 区分功能抽象与控制抽象

Iterator Pattern

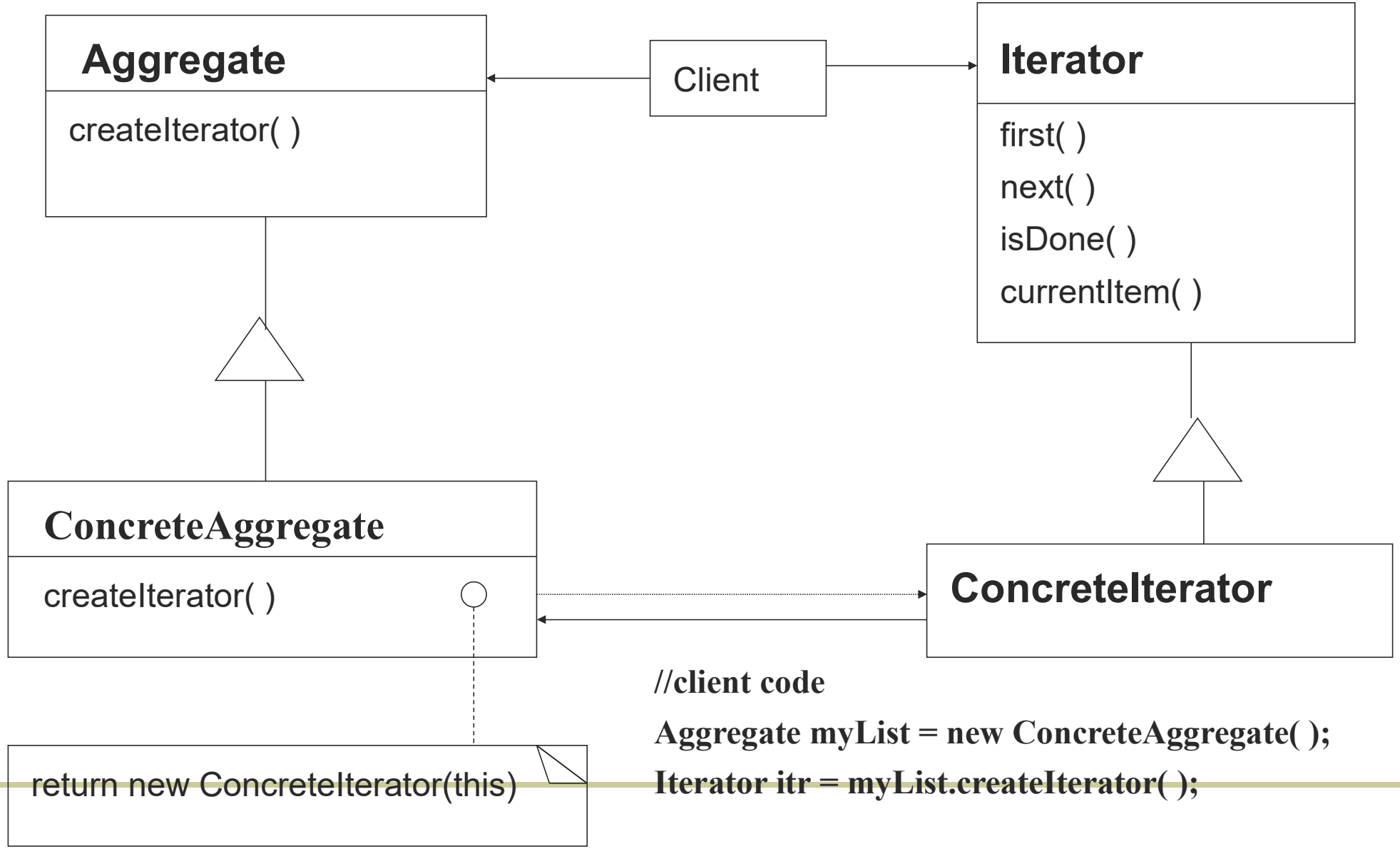
Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- Supports multiple traversals of aggregate objects.
- Provides a uniform interface for traversing different aggregate structures (it supports polymorphic iteration).

Iterator Pattern

Structure




```
//client code
Aggregate myList = new ConcreteAggregate( );
Iterator itr = myList.createIterator( );
```

Iterator Pattern

Consequences

1. Programming to interfaces and Information Hiding
2. It supports variations in the traversal of an aggregate. **For example, code generation may traverse the parse tree inorder or preorder. Iterators make it easy to change the traversal. Just replace the iterator instance with a different one.**
3. Iterators simplify the Aggregate interface. **Iterator's traversal interface obviates the need for a similar interface in Aggregate.**
4. More than one traversal can be pending on an aggregate.

Iterator Pattern体现了哪些思想?

- Programming to interface
- 封装（信息隐藏）
  - 隐藏内部的复杂结构
- ISP、SRP
  - 分离功能抽象与控制抽象
- OCP、LSP、DIP
  - 实现访问机制的可变更性

```
class Album {
  private List tracks =new ArrayList();
  public List getTracks() {
    return tracks;
   }
}
```