

Design Patterns

宋 杰

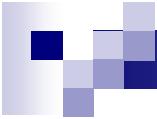
Song Jie

东北大学 软件学院

Software College, Northeastern
University

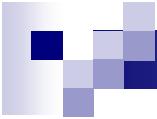


1、Principles Of Object Oriented Design



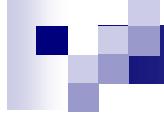
The Beauty of Software

- The beauty of software is in its function, in its internal structure, and in the way in which it is created by a team.
 - To a user, a program with just the right features presented through an intuitive and simple interface, is beautiful.
 - To a software designer, an internal structure that is partitioned in a simple and intuitive manner, and that minimizes internal coupling, is beautiful.
 - To developers and managers, a motivated team of developers making significant progress every week, and producing defect-free code, is beautiful.
-



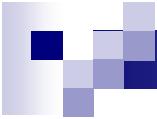
Ugly Software

- We know that software can be ugly. We know that:
 - It can be hard to use, unreliable, and carelessly structured.
 - There are software systems whose tangled and careless internal structures make them expensive and difficult to change.
 - There are software systems that present their features through an awkward and cumbersome interface.
 - There are software systems that crash and misbehave.
-



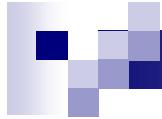
Our Goals

- As a profession, software developers should create much more beauty than ugliness.
 - 作为一种职业，软件开发人员所创建出来的美的东西因该多于丑的东西。
 - Let's start to study how to create the beautiful things.
-



Seven Deadly Sins of Software Design

- R rigidity (僵化) – make it hard to change
 - F ragility (脆弱) – make it easy to break
 - I mmobility (固化) – make it hard to reuse
 - V iscosity (黏滯) – make it hard to do the right thing
 - N eedless Complexity – over design
 - N eedless Repetition – error prone
 - N ot doing any design
-



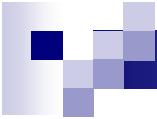
Let 's start from...

■ The Principles Of Object Oriented Design

- SRP: Single Responsibility Principle 单一职责原则
 - OCP: Open-Closed Principle 开放-封闭原则
 - LSP: Liskov Substitution Principle 里氏替换原则
 - DIP: Dependence Inversion Principle 依赖倒转原则
 - ISP: Interface Segregation Principle 接口隔离原则
 - CRP: Composite/Aggregate Reuse Principle 组合/聚合复用原则
-

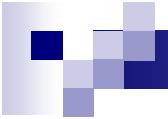


SRP: Single Responsibility Principle



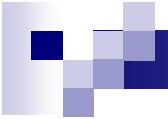
SRP : Definition

- SRP: Single Responsibility Principle
 - A class should have one reason to change
 - A responsibility is a reason to change
 - 单一职责原则及内聚性（Cohesion），表示一个模块的组成元素之间的功能相关性。从软件变化的角度来看，就一个类而言，应该仅有一个让他发生变化的原因。
-



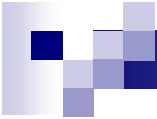
SRP : Description

- Single Responsibility = Increased Cohesion
 - Multiple Responsibilities = Increased Coupling
 - Harmful for reusing;
 - Changing one responsibility will effect the others,
the class is friable for changes .
-



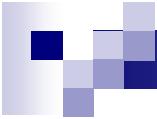
SRP Example (abstract aspect): Modem

```
public interface Modem{  
  
    public void dial(String pno);  
    public void hangUp();  
    public void send(char c);  
    public void recv();  
}
```

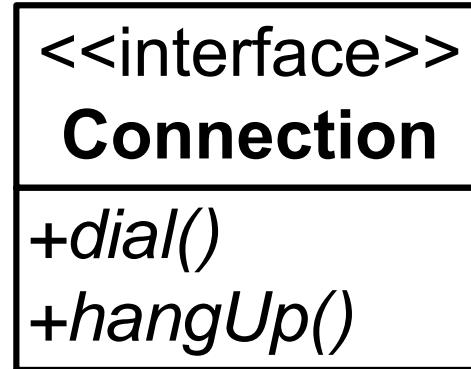
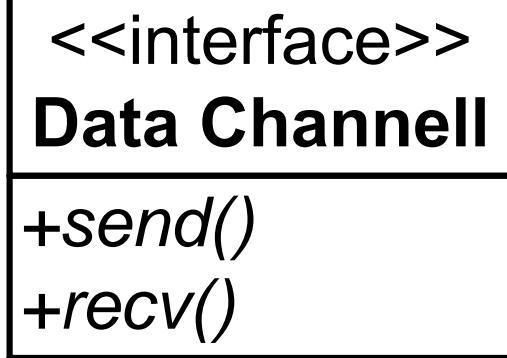


SRP Example (abstract aspect): Modem

- Modem have two responsibilities
 - Connection management:
 - *dial*
 - *hangUp*
 - Communications
 - *send*
 - *recv*
 - Whether two responsibilities should be separated is relay on whether they are changed together.
-

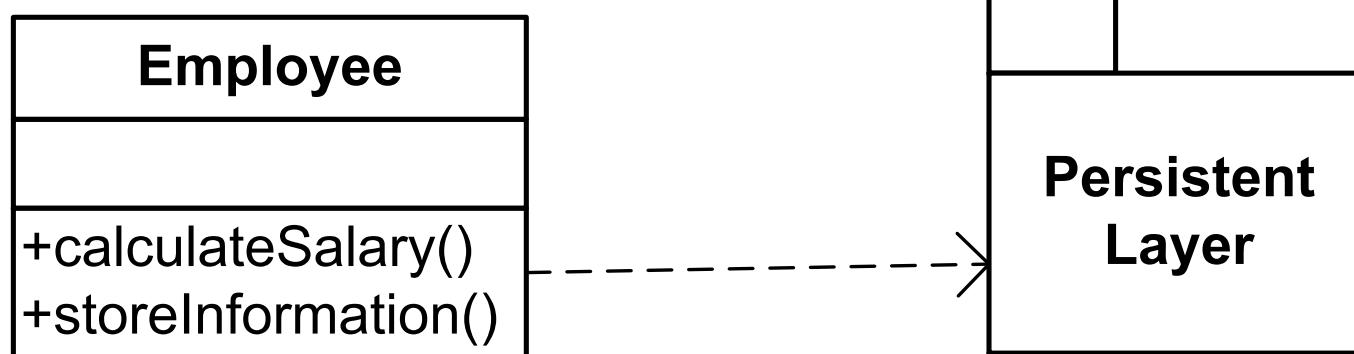
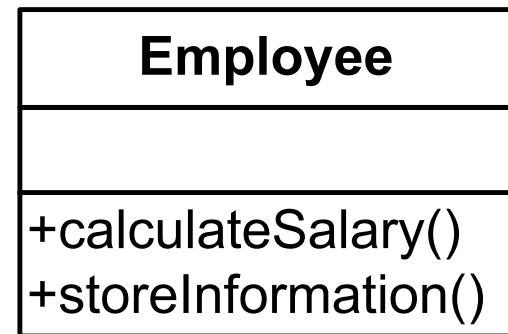


SRP Example (abstract aspect): Modem

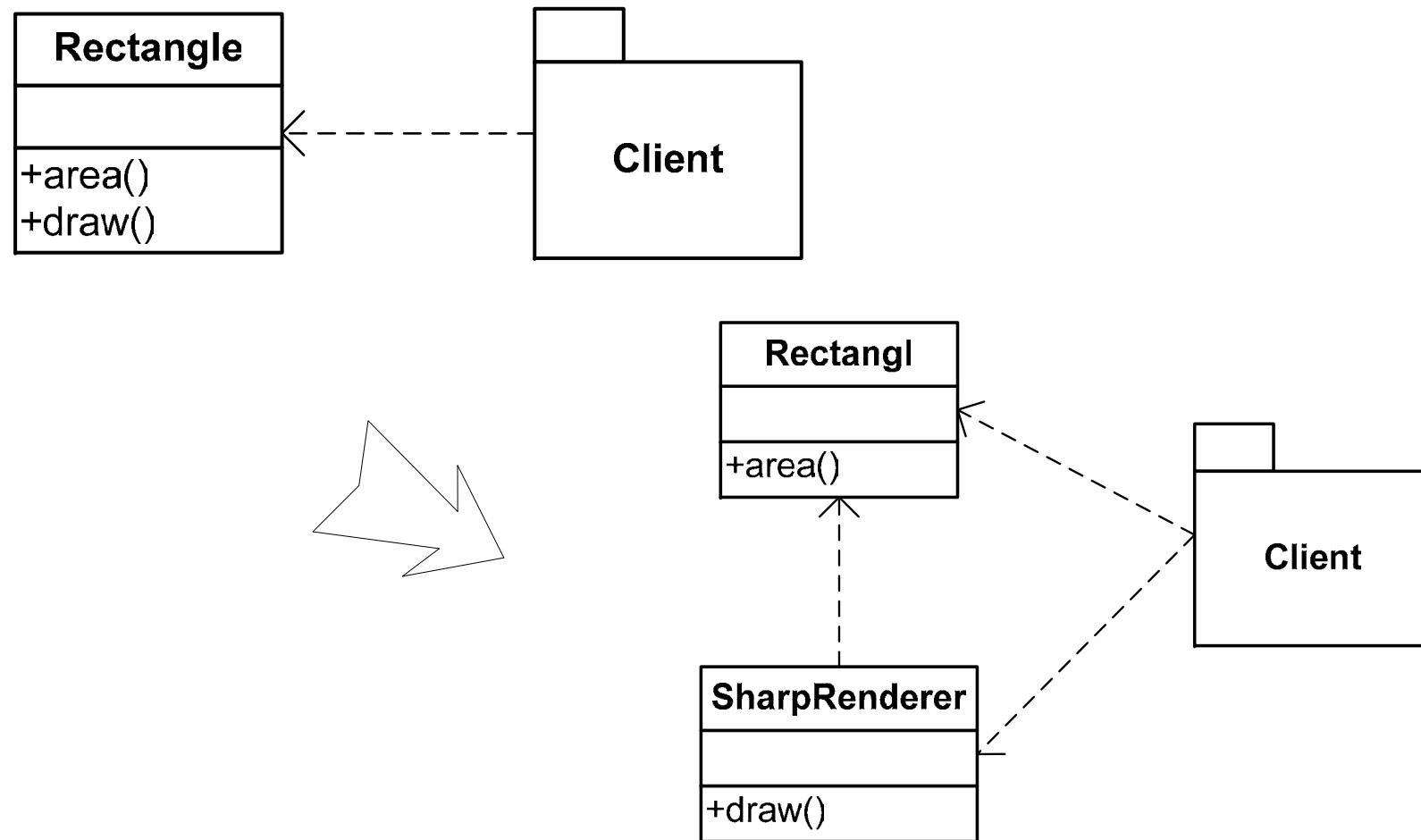


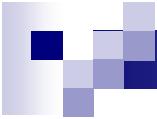
Thinking: Where is the **Modem**?
How to implement the **Modem**?

SRP Example (implemented aspect): business and persistent methods



SRP Example (both two aspects): Rectangle



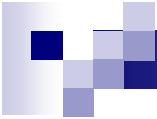


SRP: Kernel

- SRP is a simple and intuitive principle, but in practice it is sometimes hard to get it right.
 - In abstract aspect, The correct abstraction is the key issues for SRP.
 - In implemented aspect, move the codes (responsibility) to another class, then use them by invocation.
-

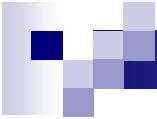


OCP: Open-Closed Principle



OCP: Definition

- OCP: Open-Closed Principle
 - Software entities (Classes, Modules, Methods, etc.) should be open for extension, but closed for modification.
 - Open For Extension: Satisfying the new requirements by adding new modules.
 - Closed For Modification: No need and can not modify current modules for new requirements .
 - 软件实体（类、模块、函数等等）应该是可以扩展的，但是不可修改的。
-

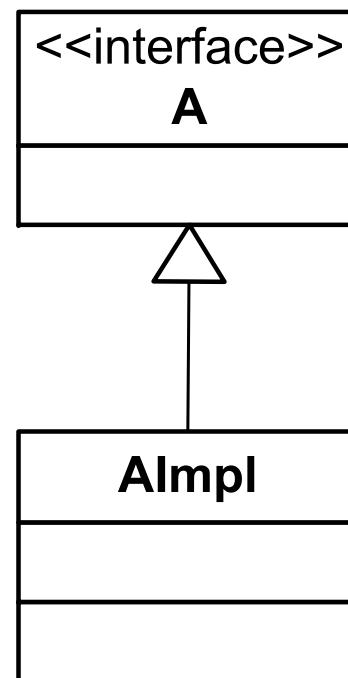


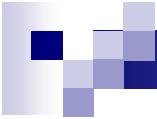
OCP: Description

- The software which satisfy OCP have two advantages
 - By extending the existing system, the software can provide new functions to satisfied new requirements, thus the software have strong adaptability and flexibility.
 - The existing modules, especially the most important abstract modules, are no need to modified, thus the software have strong stability and persistency.
-

OCP: Implementation

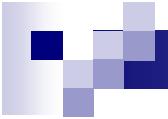
- Relay on:
 - Abstraction
 - Polymorphism
 - Inheritance
 - Interface





OCP: Implementation

- **Interface and abstract class** are the abstraction which are fixed but have many possible behaviors
 - These behaviors are presented as the implemented class or inherited class.
 - A **Interface** is open for extension because it have flexible number of implementations;
 - A **Interface** is closed for modification because it is pre-defined.
 - Modifying a **Interface** brings lots of cascaded changes in its implementations.
-

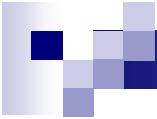


OCP Example: Document Processor

- A system which can process the documents one by one;
- There is three kinds of **Documents**, including **Paper**, **Report** and **Notice**.

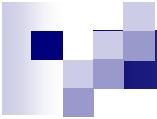
```
public class Document {  
  
    private String type;  
    private String title;  
    private String content;  
    private boolean processed;  
}
```

```
public static void process(List<Document> docList) {  
    for (Document doc : docList) {  
        if (doc.getType() .equals ("Paper")) {  
            System.out.println ("论文：" + doc.getContent ())  
            doc.setProcessed(true);  
        } else if (doc.getType() .equals ("Report")) {  
            System.out.println ("报告：" + doc.getContent ())  
            doc.setProcessed(true);  
        } else if (doc.getType() .equals ("Notice")) {  
            System.out.println ("通知：" + doc.getContent ())  
            doc.setProcessed(true);  
        } else {  
            System.out.println ("无法识别的文档");  
            doc.setProcessed(false);  
        }  
    }  
}
```



OCP: Kernel

- The key of OCP is the reasonable abstraction of class;
 - Generally, OCP can not be satisfied completely, there are always some functional extensions which can not be extended without modifying the existing codes;
 - OCP should be supported in a reasonable degree;
 - The designer should predict the potential changes of the modules, then build the corresponding abstraction to support them.
-

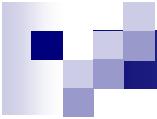


OCP: Conclusion

- The way of designing by OCP:
 - Traditional way: “Please do not introduce the changes to the system”,“千万别给系统增加新的需求”.
 - OCP way: “What kind of changes are supported without re-designing the system”. “在不重新设计的前提下系统支持什么样的变化”;
 - OCP is the kernel of OOD (Object Oriented Design), abstraction is the kernel of OCP;
 - It is bad idea to over-consider OCP, We should abstract the modules which are changed frequently, avoiding meaningless abstraction is the same important as abstraction itself.
 - It is impossible the every modules of system satisfy OCP, but we should try to minimize the number of modules which do not satisfy OCP;
 - OCP means the better reusability and maintainability.
-

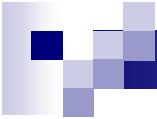


LSP: Liskov Substitution Principle



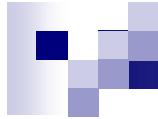
LSP: Definition

- LSP: Liskov Substitution Principle
 - If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T ."
 - 若对每个类型S的对象 o_1 。都存在一个类型T的对象 o_2 ，使得在所有针对T编写的程序P中。用 o_1 替换 o_2 后，程序P行为功能不变，则S是T的子类型。
-



Or in English

- Any subclass should always be usable instead of its parent class.
 - All derived classes must honour the contracts of their base classes
 - IS A = same public behavior
 - Pre-conditions is weaker
 - Post-conditions is stronger
-

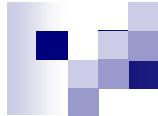


LSP : Kernel

- IS-A relationship is based on the **Behavior**, not **Concepts**.
 - The behavior is relay on the context and applied situation, Some concepts are obviously satisfy IS-A relationship but not inherited relationship because theirs behaviors are inconsistent in some situation.
 - When define inheritance, define it carefully.
-

LSP Example: Square is a Rectangle

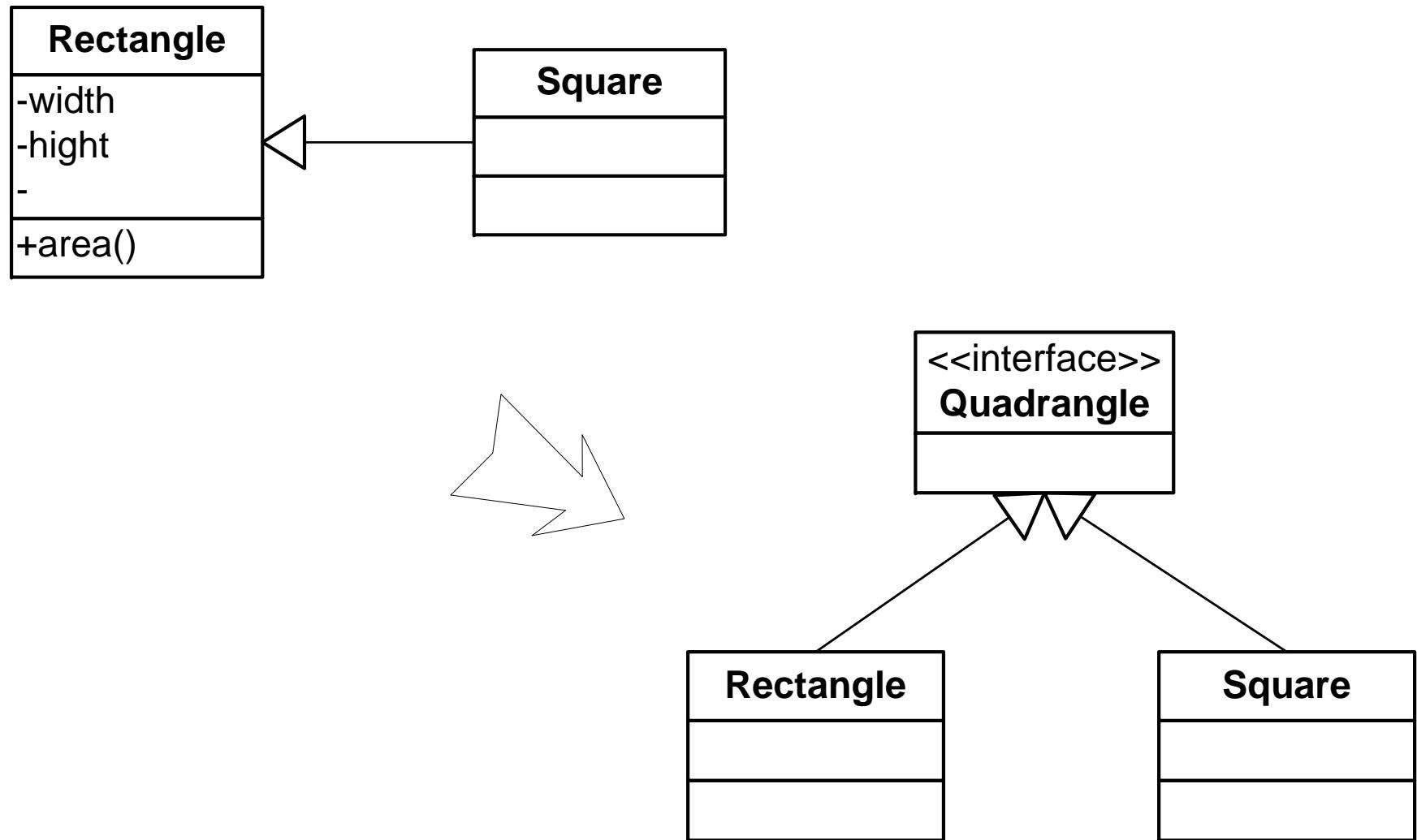
```
public class Square extends Rectangle{  
  
    public double setWidth (double value){  
        super.width = value;  
        super.height = value;  
    }  
  
    public double setHeight (double value){  
        super.width = value;  
        super.height = value;  
    }  
  
    public Square(double side){  
        super.height = side;  
        super.width = side;  
    }  
}
```



LSP Example: Square is a Rectangle

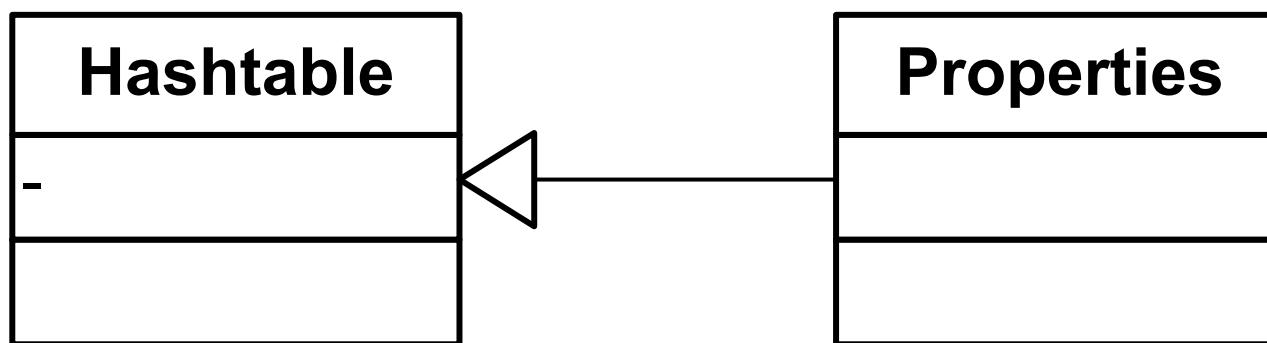
```
public static void assertArea(Rectangle rect) {
    rect.setWidth(4);
    rect.setHeight(5);
    assert(rect.area() == 20);
}

public static void reSize(Rectangle rect) {
    while (rect.Height >= rect.Width)
    {
        rect.width = rect.width++;
    }
}
```



LSP Example :

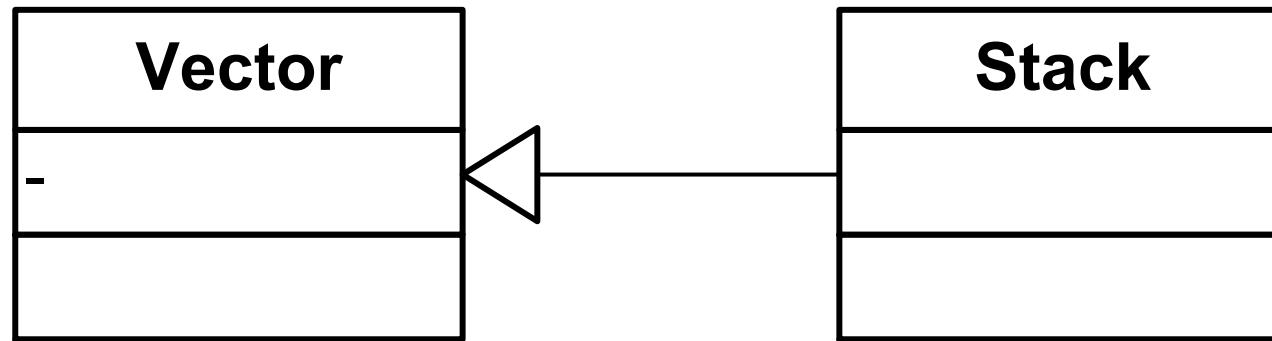
java.util.Properties is a java.util.Hashtable



- **Hashtable.** key:Object,value:Object
- **Properties.** key:String,value:String

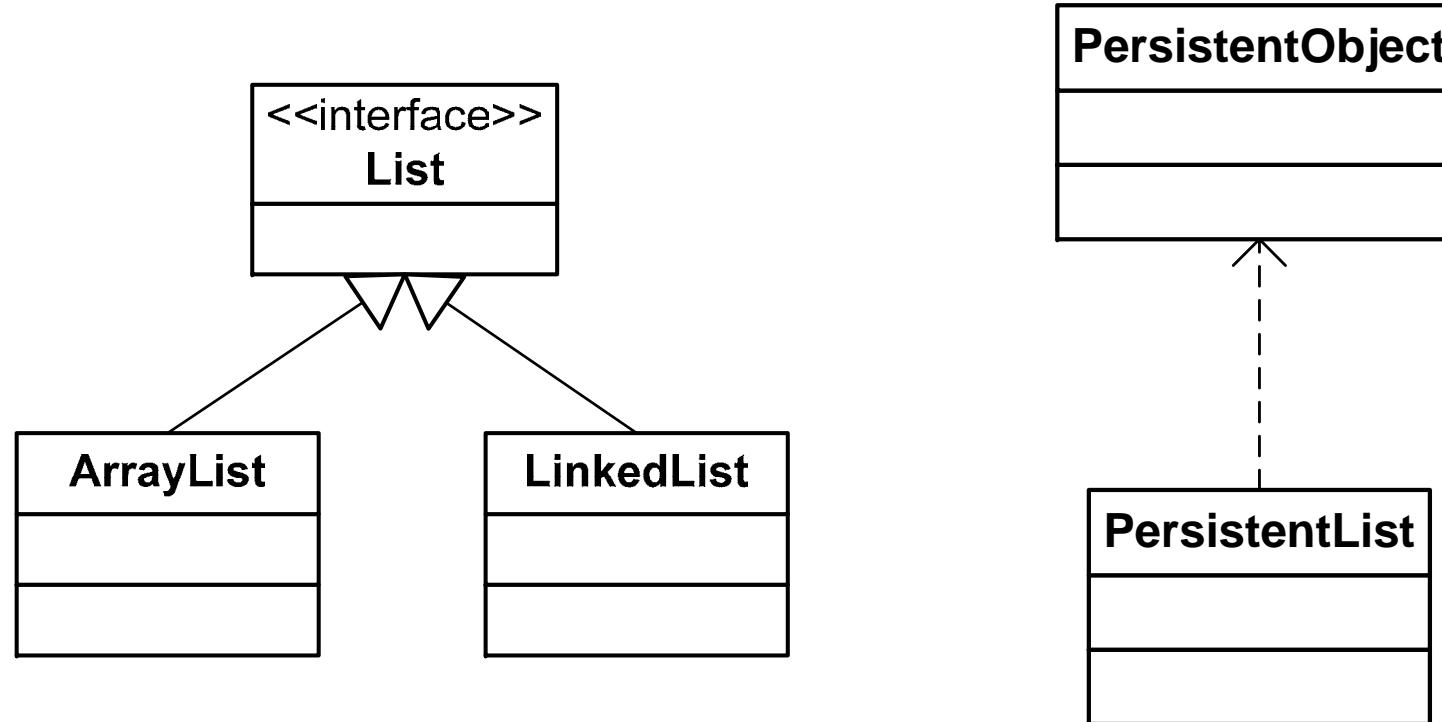
LSP Example :

java.util.Stack is a java.util.Vector

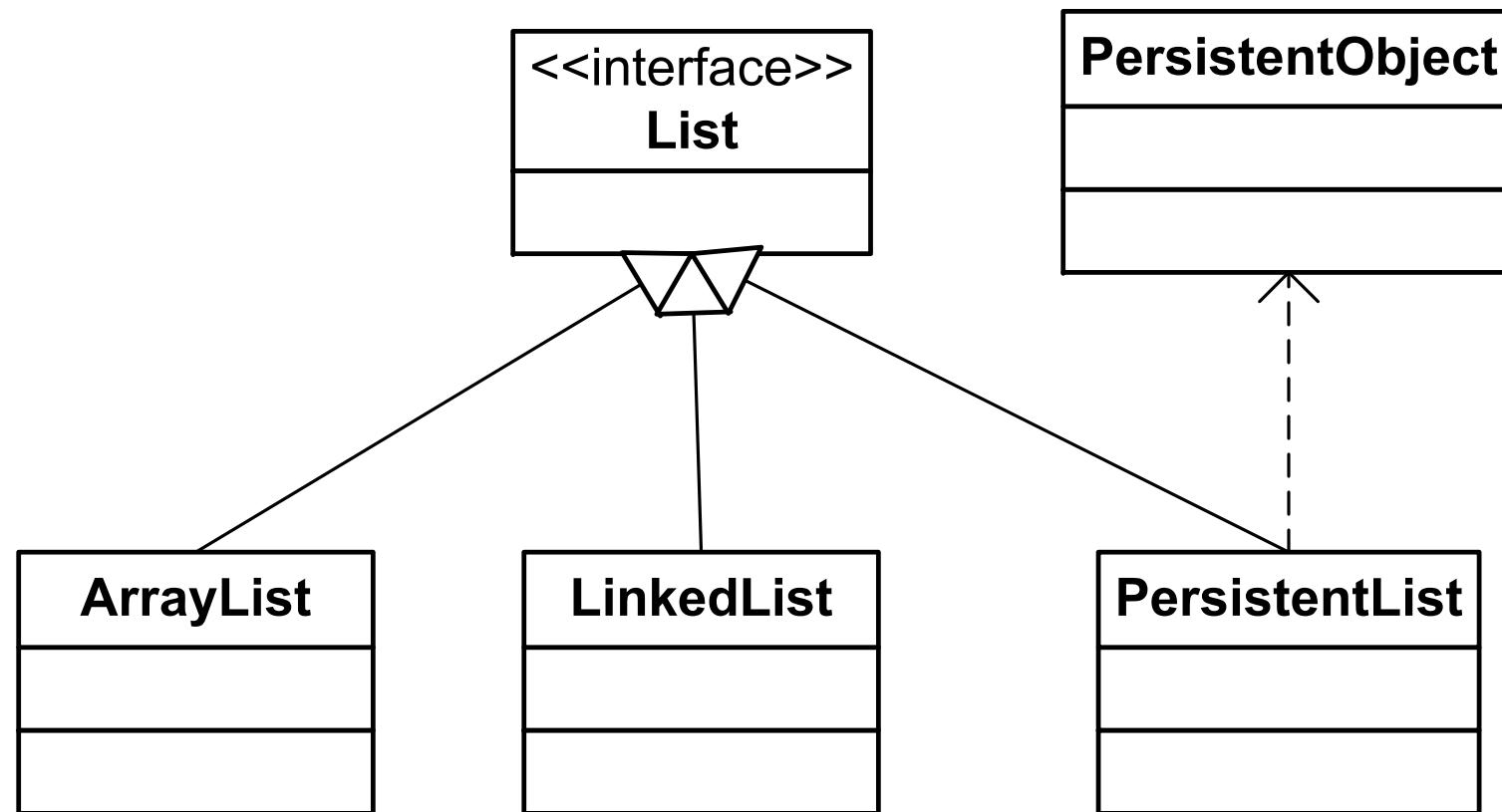


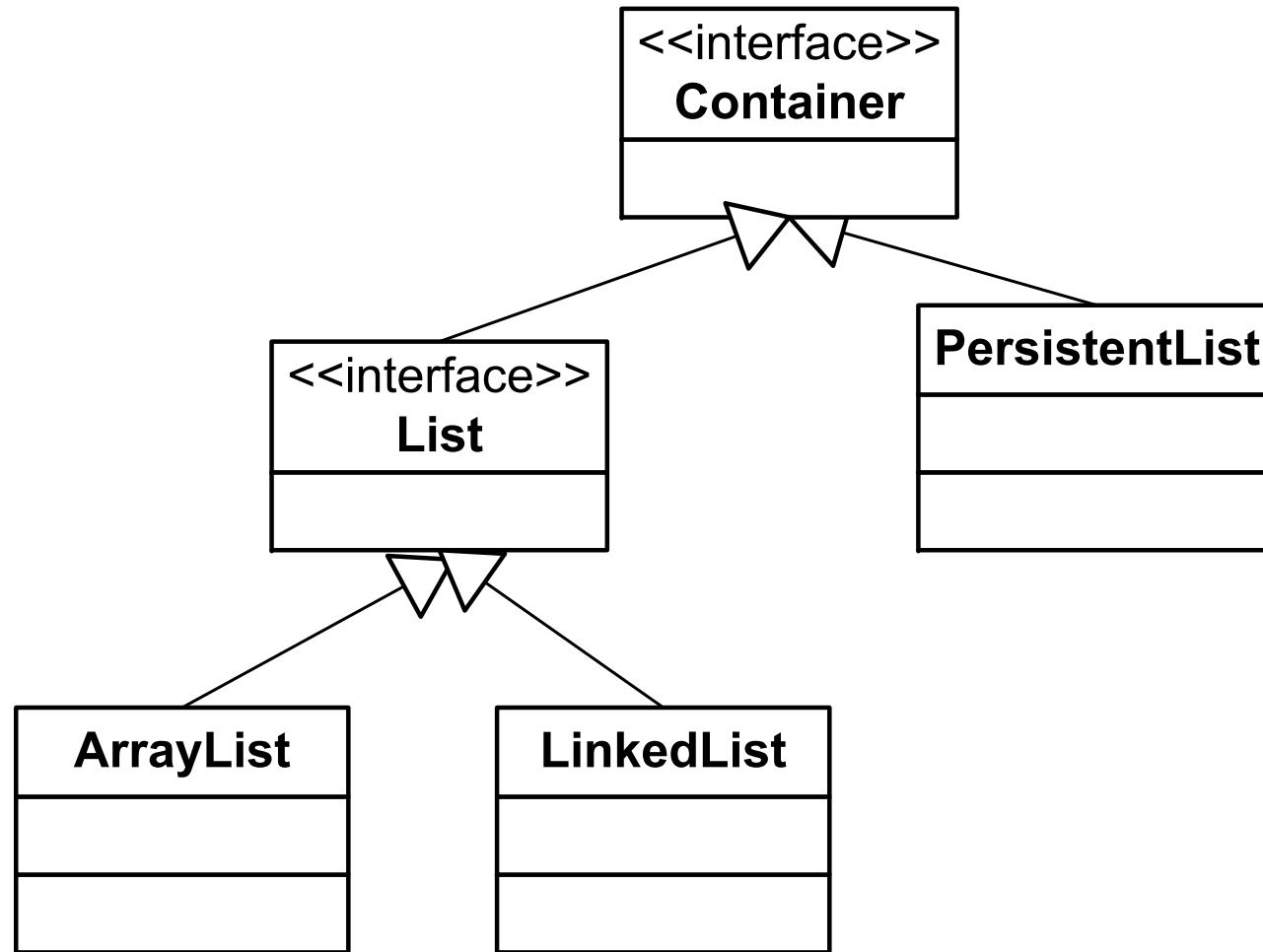
- **Vector:** FIFO
- **Stack:** FILO

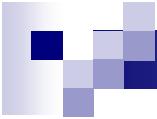
LSP Example: List



```
PersistentList.add(T node) {  
    //If node is not PersistentObject, throws an exception  
}
```



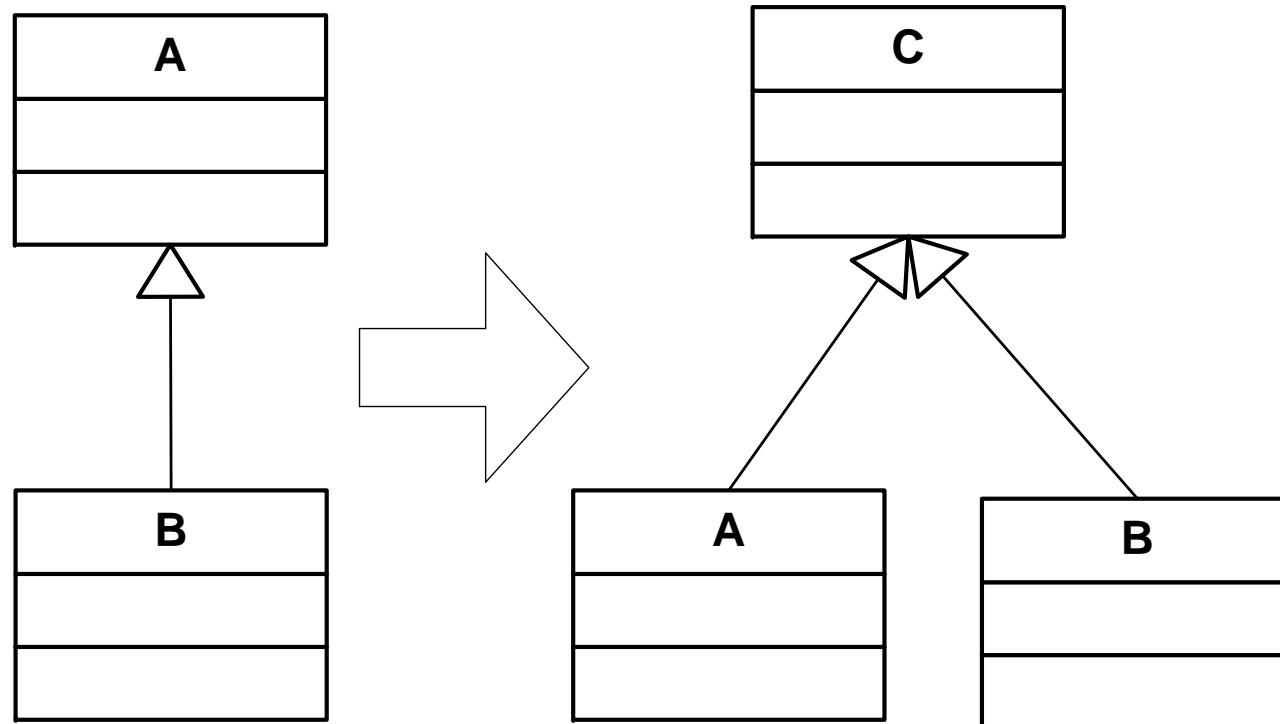




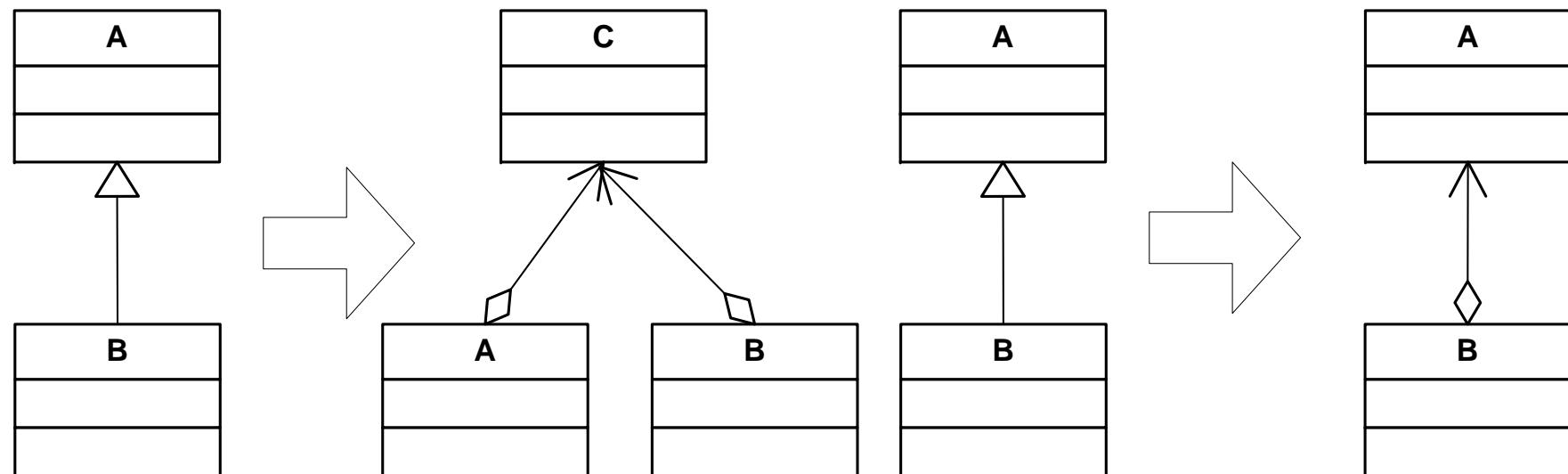
LSP Extension: Refactoring

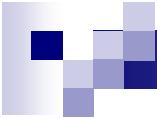
- Class **A** and Class **B** have same functions (duplicated codes), then:
 - Generally, **A** and **B** are not inherited because they have own behaviors.
 - If both **A** and **B** satisfy LSP with **C**, then move the duplicated parts (code) of **A** and **B** to **C**, duplicated method (signature) to **interface IC**. Let **A** and **B** inherit from **C**, and **C** implement **IC**. generally **C** is an abstract class.
 - Or (better) change the inherited relationship into delegates relationship, **A** and **B** delegate **C** to perform the common functions.

LSP Extension: Refactoring



LSP : Refactoring



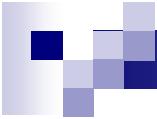


LSP : Conclusion

- LSP is the principles of using inheritance.
 - IS-A is based on behavior, not concepts.
 - LSP is relay on the applied situation.
 - LSP is theoretic and rigorous, sometime breaking LSP a little is reasonable and beneficial, anyhow LSP should be well considered when a inherited relationship is designed.
-

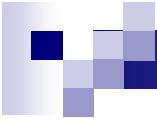


DIP: Dependence Inversion Principle



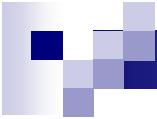
DIP : Definition

- DIP: Dependence Inversion Principle
 - Higher layer modules should NOT depend on lower layer modules, both should depend on abstractions (**interfaces** or abstract classes).
 - Abstractions should NOT depend on details, details should depend on abstractions .
 - 高层模块不应该依赖于低层模块，二者都应该依赖于抽象。进一步的，抽象不应该依赖于细节，细节应该依赖于抽象。
-



DIP : Description

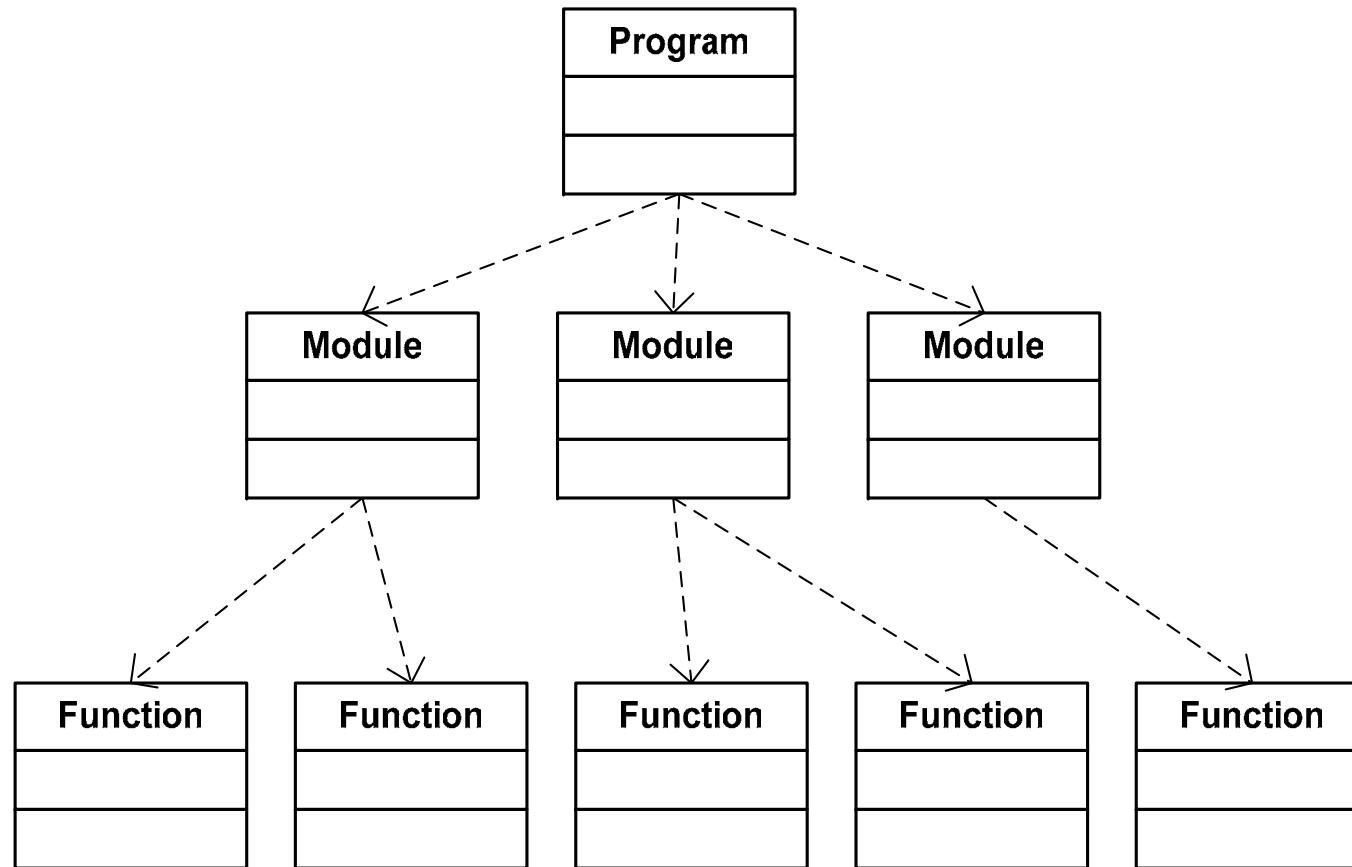
- Increase loose coupling
 - Abstract interfaces don't change
 - Concretions implement interfaces
 - Concretions easy to throw away and replace
 - Increase flexible
 - Increase isolation
 - Decrease rigidity
 - Increase testability
 - Increase maintainability
-



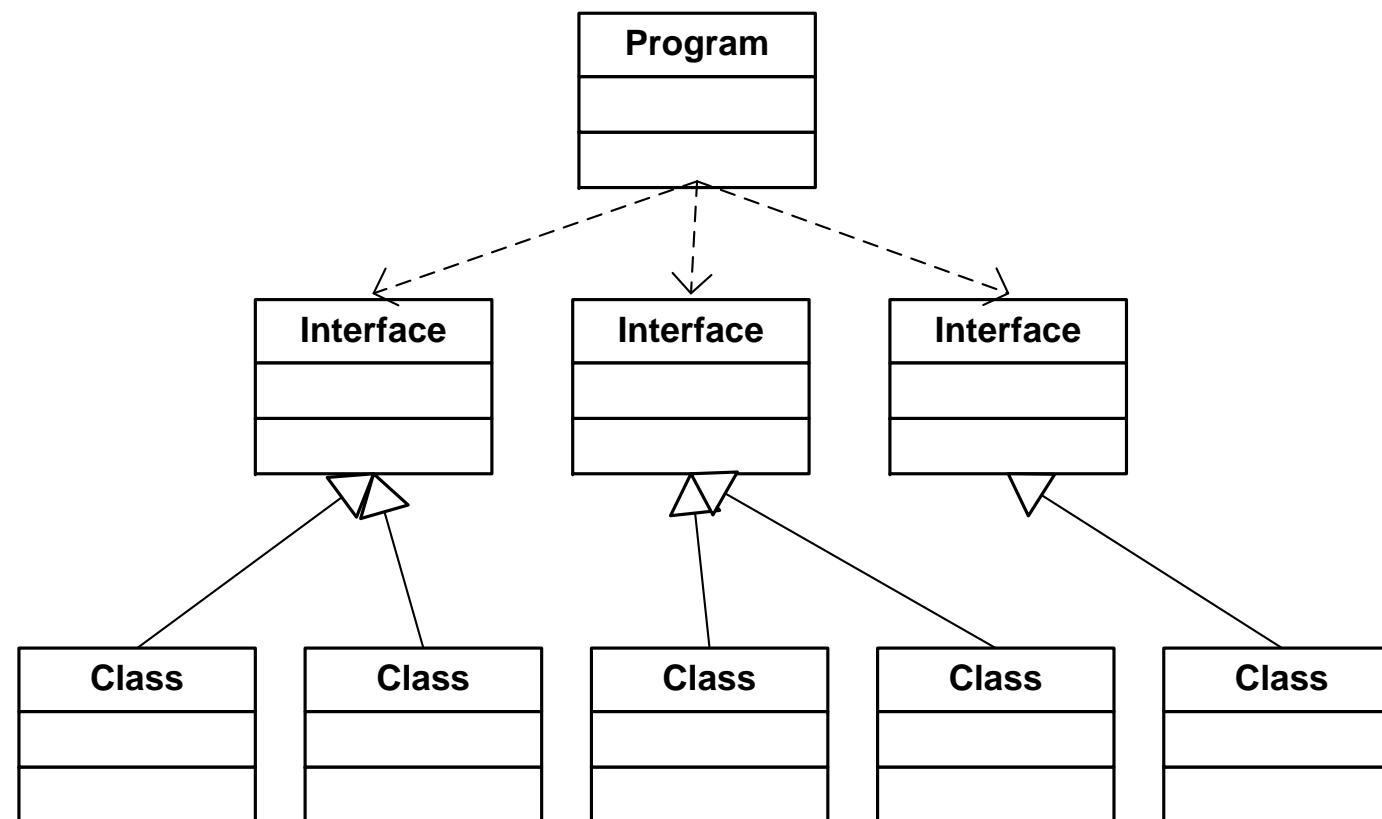
DIP : Implementation

- Higher layer specify interfaces for the required services;
 - Lower layer implements these interfaces;
 - Higher layer using the services of lower level through these interfaces. So that higher layer do not depend on lower layer ;
 - On the contrary, lower layer depends on the service interfaces which are specified by the higher layer;
 - The dependency is inverted.
-

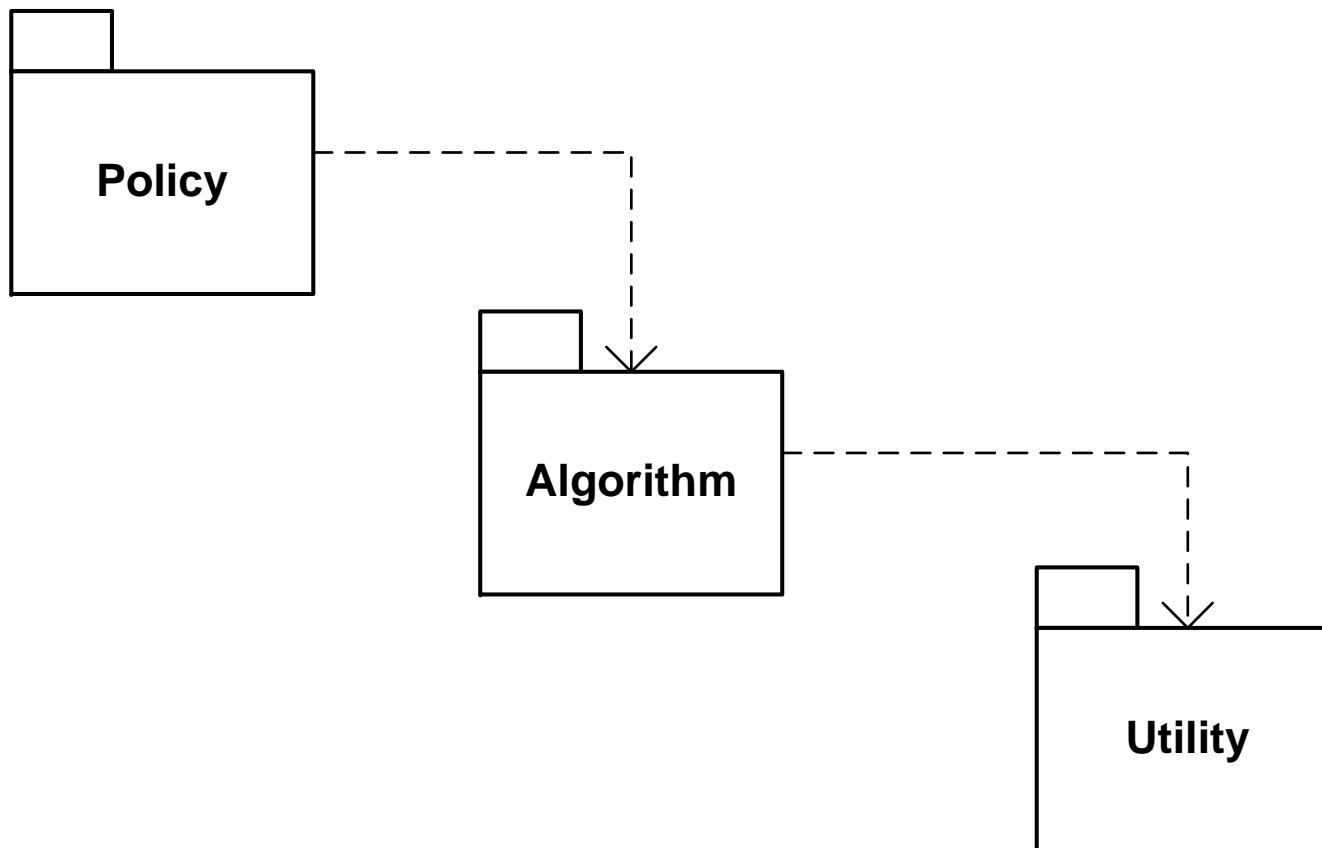
Procedural Design

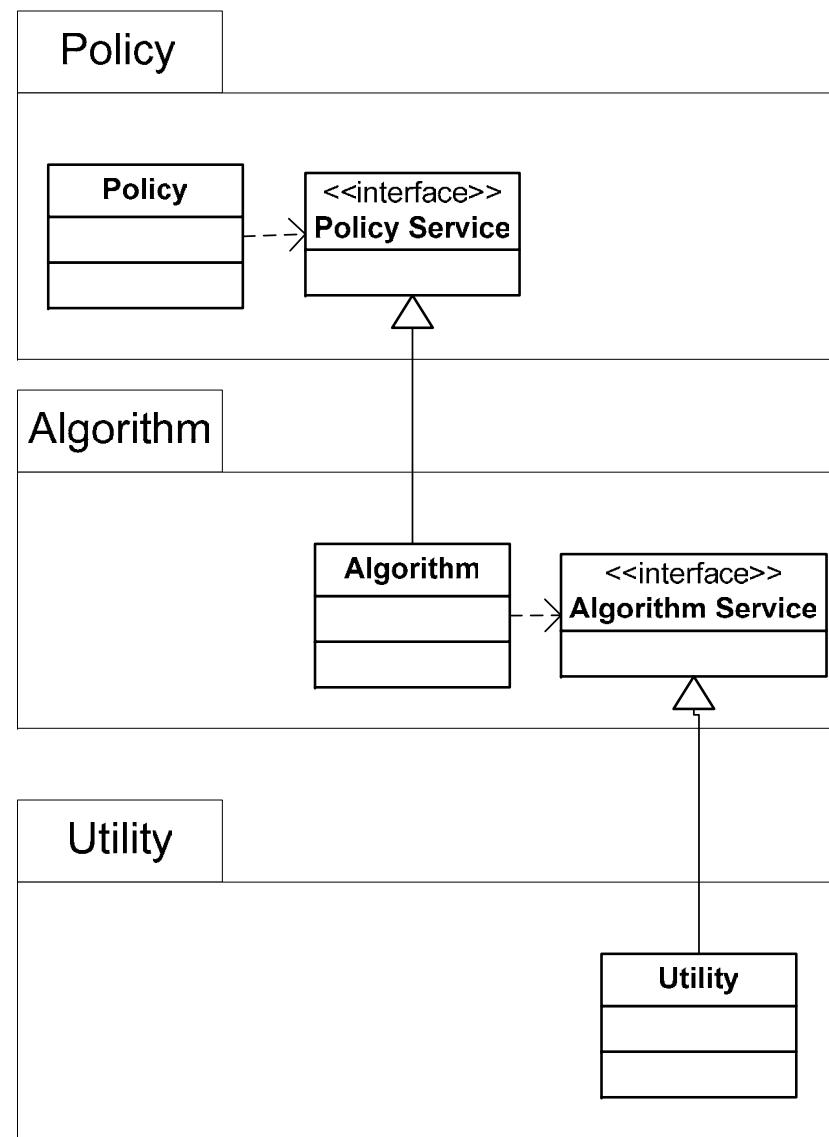


DIP (OOD)

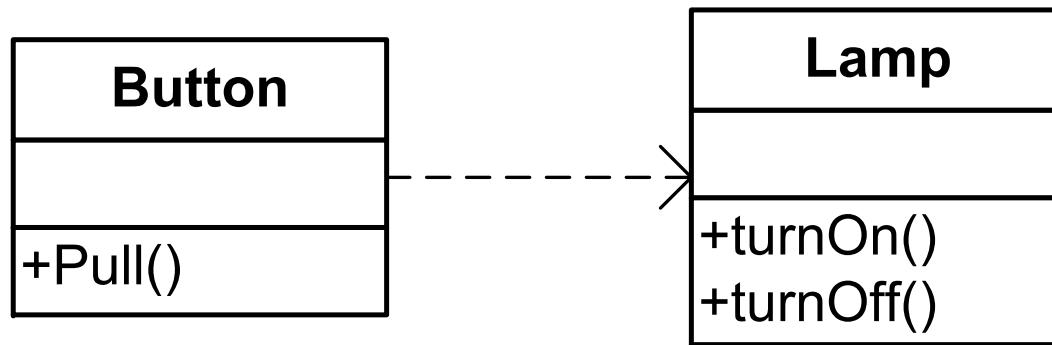


DIP Example: General layers of an module

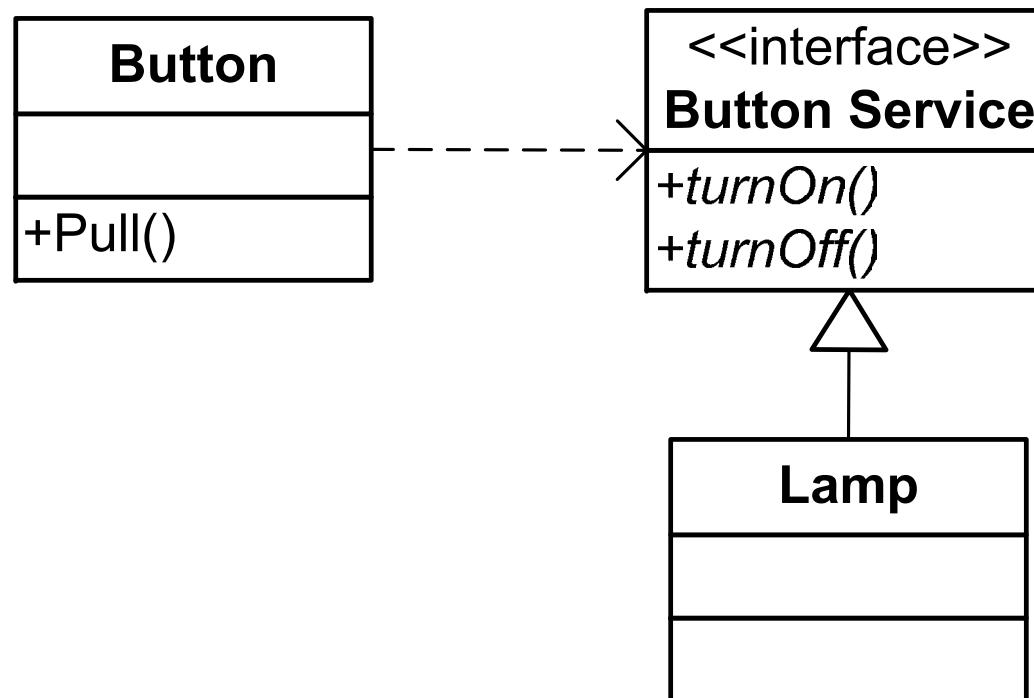




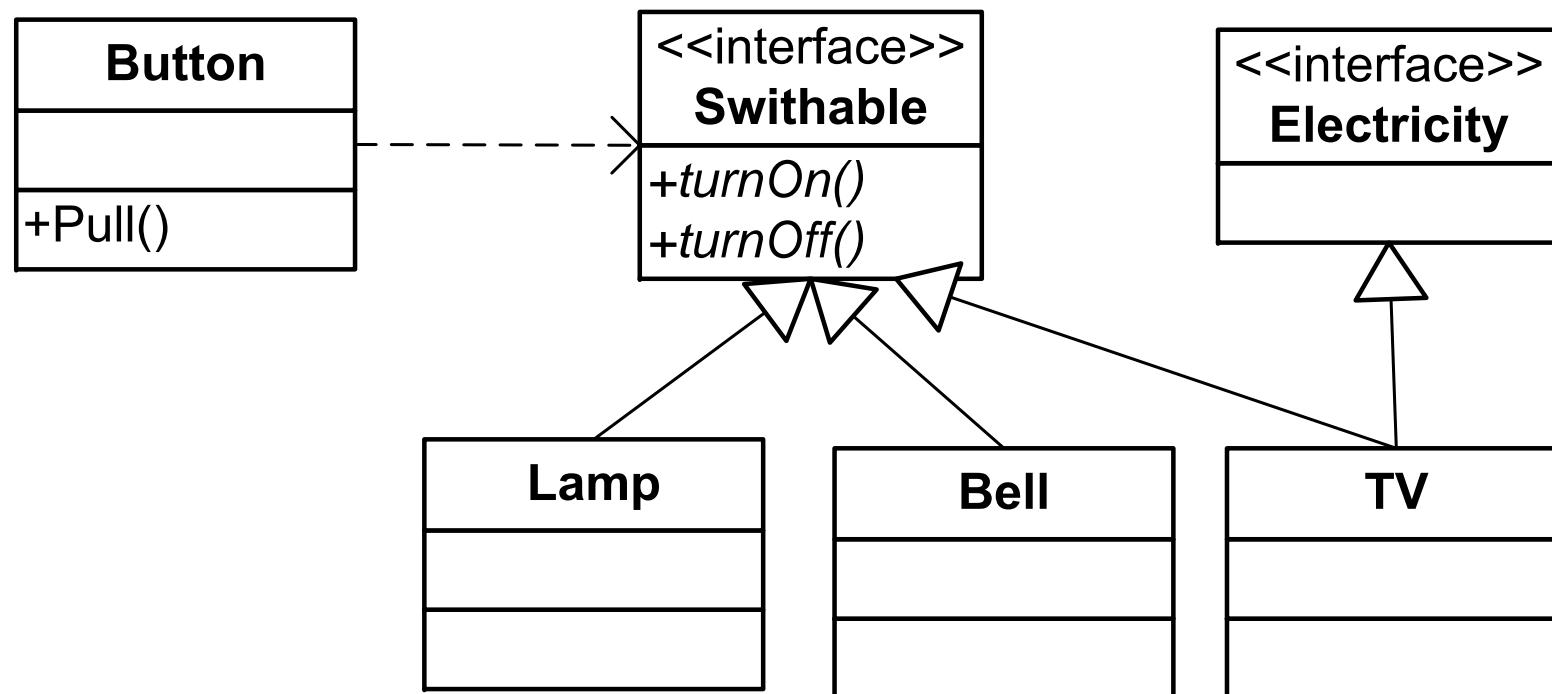
DIP Example: Button and Lamp

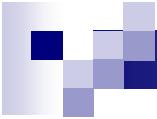


DIP Example: Button and Lamp



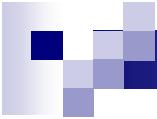
DIP Example: Button and Lamp





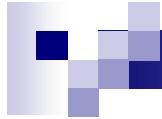
DIP : Rules

- Any variables should NOT hold a reference which refer to a concrete class, but a **interface** or abstract class;
 - Any classes should NOT inherit from a concrete class;
 - Any methods should NOT override the base-methods which has been implemented in base class.
 - The rules of DIP is over-strict.
-



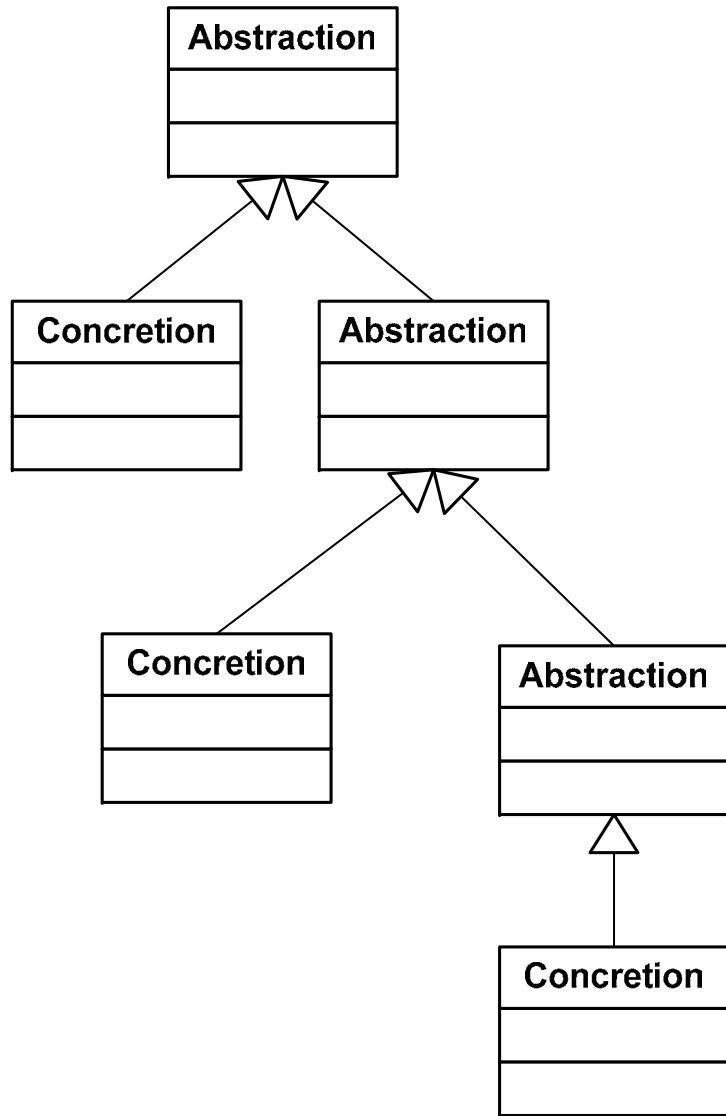
DIP : Kernel

- DIP is used for de-coupling, further for OCP.
 - DIP proposes that always use interface instead of concrete class.(针对接口编程)
 - DIP is used when the classes are NOT stable;
 - Most classes are variable, they contain potential changes, except some utilized classes (Tools) or final classes (String);
 - In most cases, DIP should be well adopted;
-

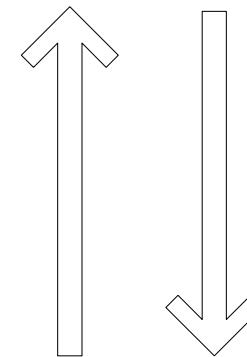


DIP : Kernel

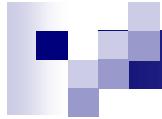
- In a inherited hierarchy:
 - Shared codes should move to the abstract class in the abstract level (higher level of hierarchy)
 - Private data should move to the concrete class in the implemented level (lower level of hierarchy)
 - Code presents the logic, which contains commonness;
 - Data presents strong privacy, which is various ;
-



Codes are
centralized
to

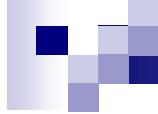


Data is
centralized
to



DIP Extension: Coupling

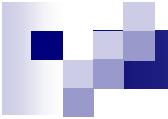
- Three types of Coupling in OOD
 - Nil Coupling
 - Concrete Coupling
 - Abstract Coupling
 - Abstract Coupling increases both flexibility and complexity;
 - Concrete Coupling is better than Abstract Coupling when class is stable.
-



DIP: Interface is everything?

■ Interface is not silver bullet

- Unstable interface will break the isolation between abstraction and implementation;
 - Interface should be defined by the service requester, not services provider;
 - The various implementations of an interface are according to the clients, not implementations themselves;
 - Even the changes of interface should also be proposed by the clients, not service providers.
-

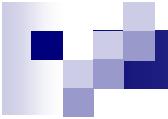


DIP: Conclusion

- DIP is the basic principle of object oriented design, the inversion of dependency is the key idea of OOD;
 - DIP defines:
 - The dependency between modules should be isolated by abstraction;
 - How to extract the abstraction;
 - How to implement the abstraction.
-

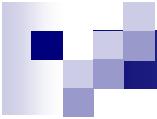


ISP: Interface Segregation Principle



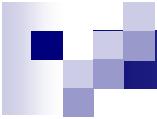
ISP : Definition

- ISP: Interface Segregation Principle
 - The dependency of one class to another one should depend on the smallest possible interface.
 - Interface should be atomic, cohesive, it presents an independent role, or provides independent services;
 - Many client specific interfaces are better than one general purpose interface;
-



ISP : Description

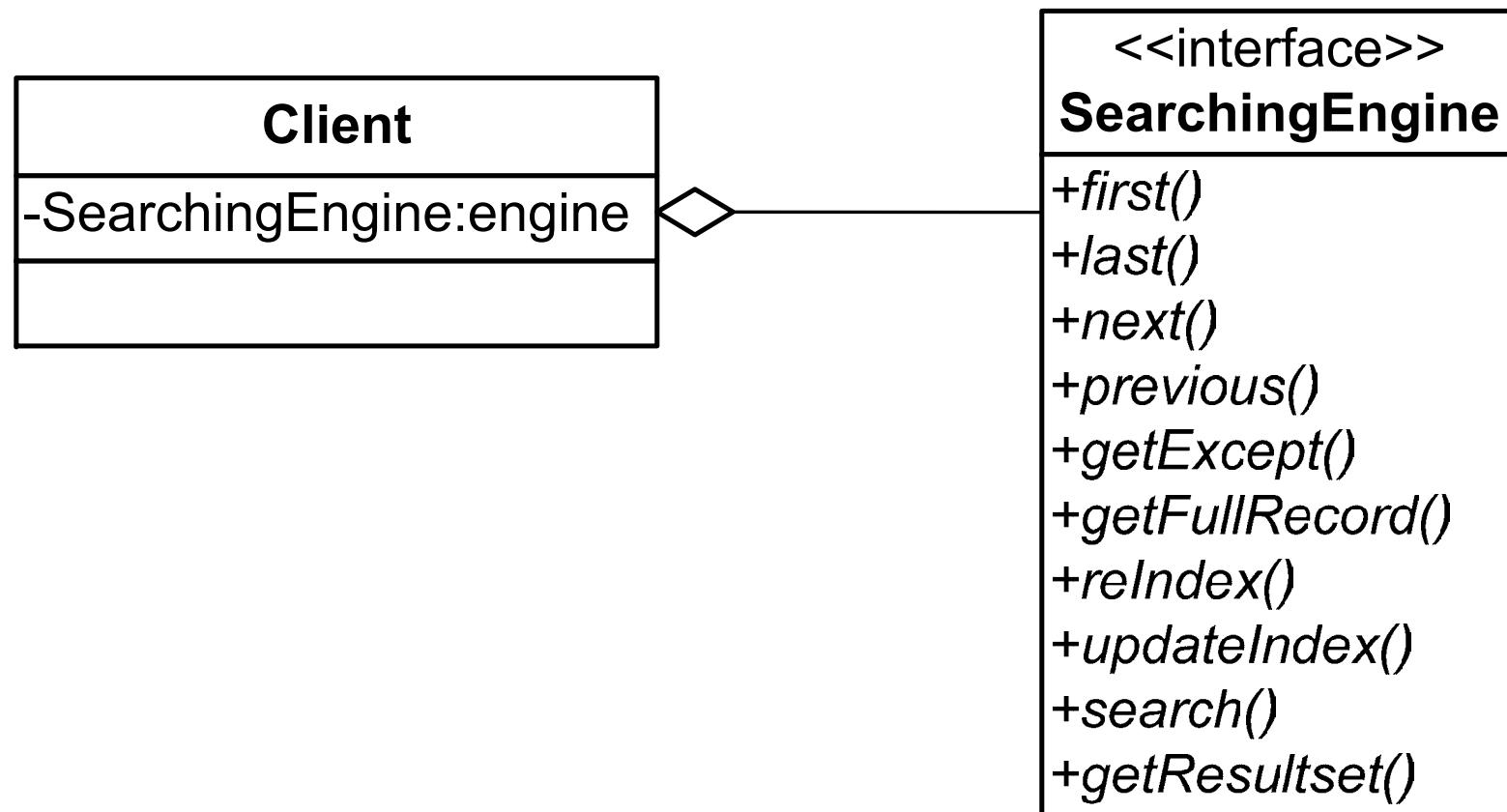
- Make fine grained interfaces that are client specific.
 - Clients should not be forced to depend upon interfaces that they don't use.
 - Create an interface per client type not per client, avoid needless coupling to clients
 - 接口隔离原则，其“隔离”并不是准确的翻译，真正的意图是“分离”接口(的功能)。
-



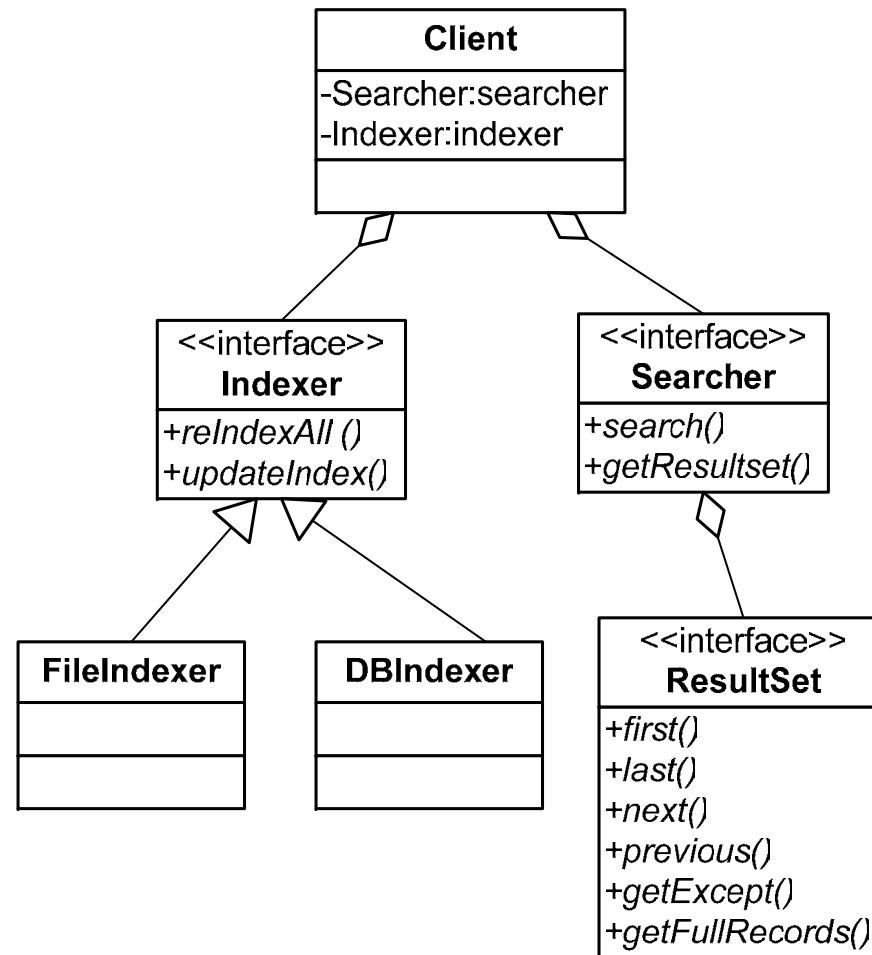
ISP : Interface Pollution

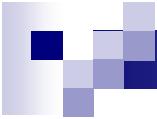
- Fat interface is interface pollution;
- Saving the number of **interfaces** can not reduce the code-amount, but pollute the interfaces.
- Interface should be thin, it is also reasonable even there is no method defined in a interface.
 - Single-method-interface: (function pointer)
 - **Runnable**
 - Flag interface (Indicate **interface**)
 - **Cloneable, Serializable, Remote**

ISP : Example



ISP : Example



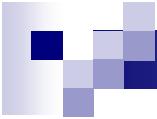


ISP : Kernel

- The intention of ISP is avoid the coupling among clients.
 - The implementation of ISP is designing fine granularity interface.
 - Interface is the abstraction of the service contracts ;
 - Service contracts is based on service requirements of clients;
 - Service requirements of different type of clients should be separated;
 - Different kind of requirements of one client should also be separated;
 - ISP is SRP in interface version.
-

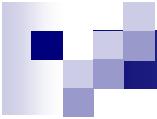


CRP: Composite Reuse Principle



CRP: Definition

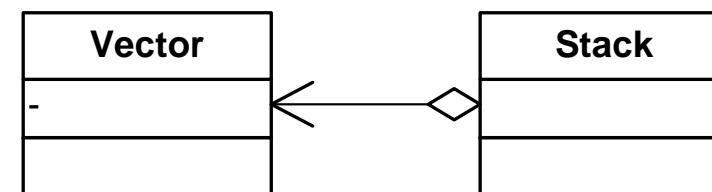
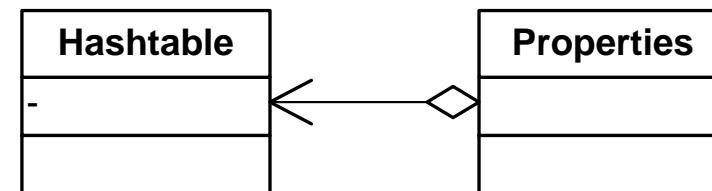
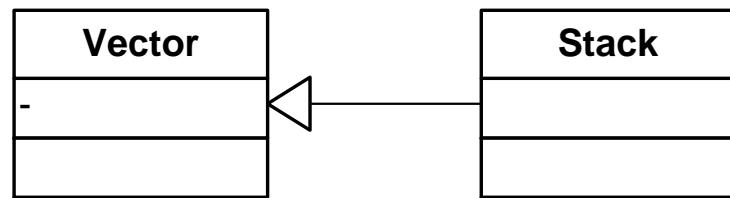
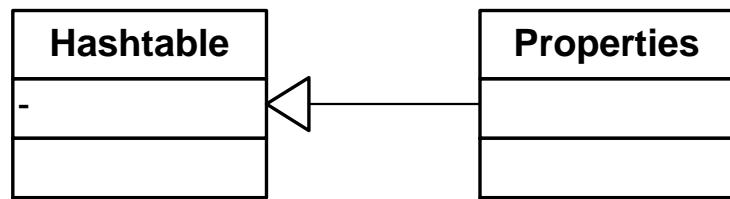
- Classes may achieve polymorphic behavior and code reuse by containing other classes which implement the desired functionality instead of through inheritance.
 - Favor delegation over inheritance as a reuse mechanism.
 - 组合/聚合复用原则就是在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分；新的对象通过向这些对象的委派达到复用已有功能的目的。
-



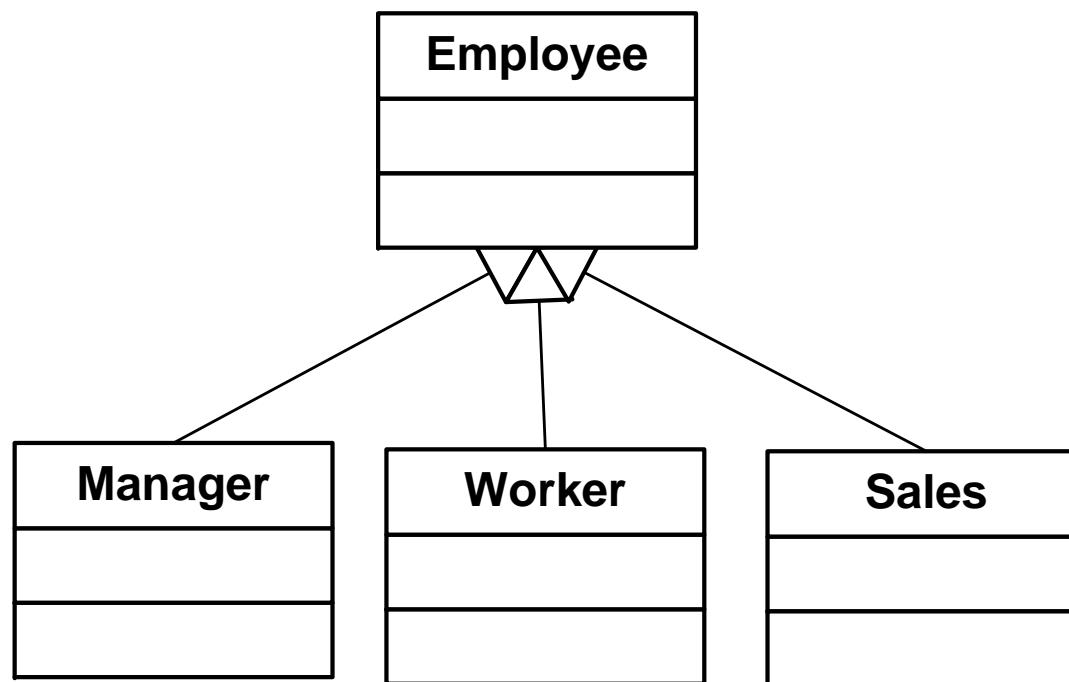
CRP: Description

- Aggregation and Composition are special relationship of Association.
 - Aggregation presents HAS-A relationship;
 - Composition presents whole/part relationship .
 - OO beginners often over-use inheritance and end up with big, complicated, rigid class hierarchies. The CRP wants to remind you that Aggregation/Composition is an alternative, more flexible way of achieving reuse.
-

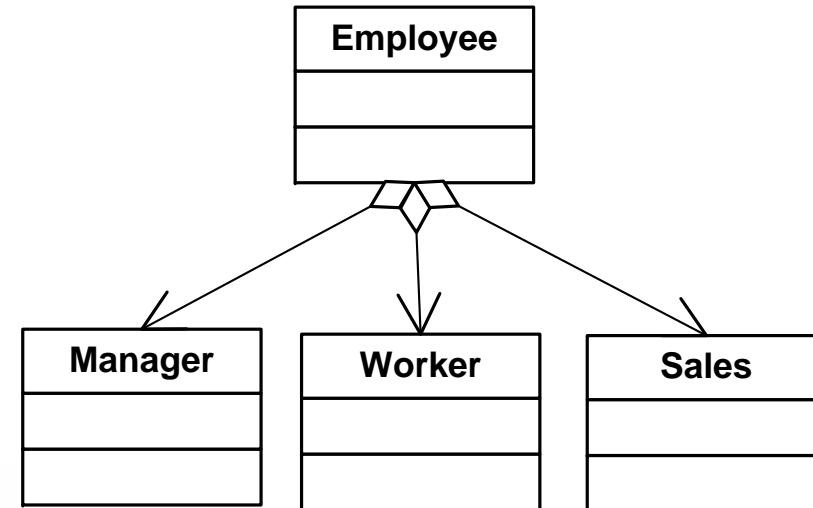
CRP Example (reuse): JDK Container



CRP Example (polymorphic) : Employee

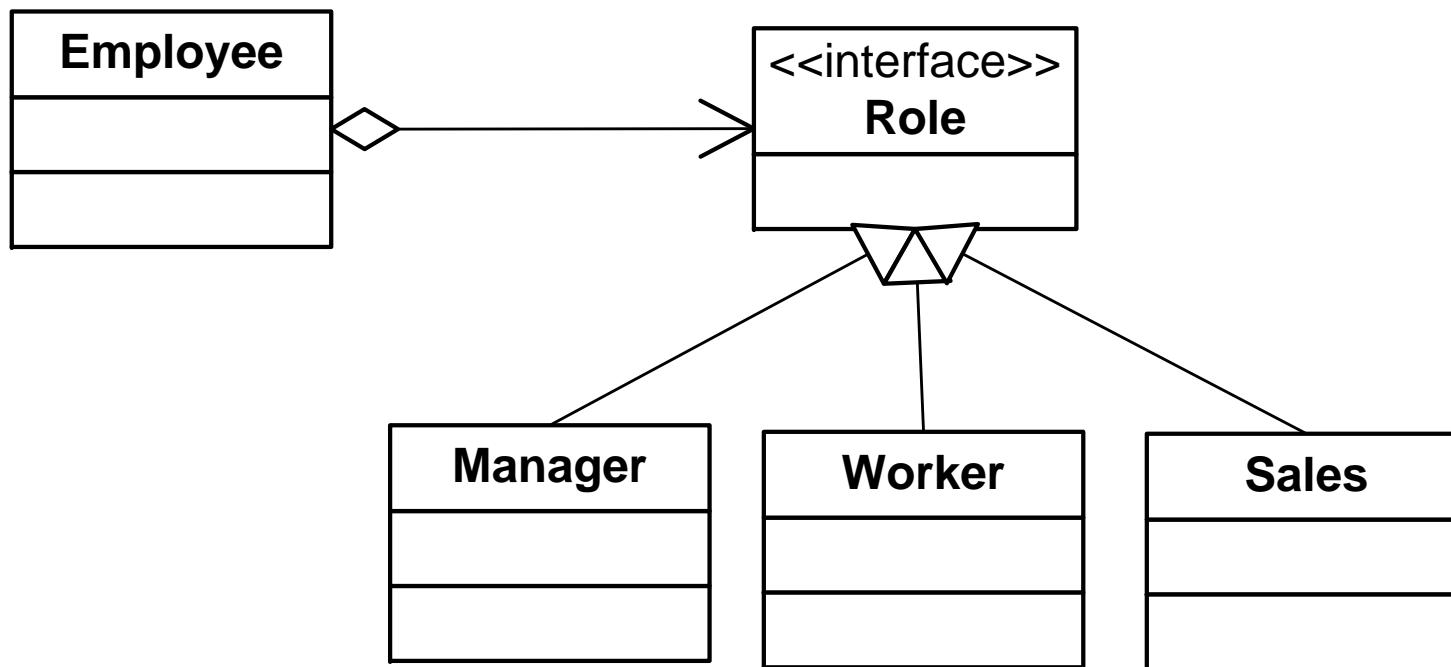


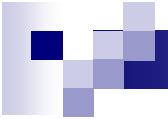
Employee



```
public class Employee {  
  
    private Worker worker = new Worker();  
    private Manager manager = new Manager();  
    private Sales sales = new Sales();  
  
    public void something() {  
        this.worker.something();  
        //this.manager.something();  
        //this.sales.something();  
    }  
}
```

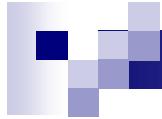
Employee





CRP: Kernel

- Generally, Aggregation/Composition is better than Inheritance.
 - For reusing:
 - Aggregation/Composition is “black box” reusing, the details of contained object is invisible for clients.
 - Inheritance is “white box” reusing, strong coupling, the code is reused statically.
 - For polymorphism:
 - Aggregation/Composition is flexible polymorphism for not only implementations but also abstractions.
 - Inheritance is fixed, the interfaces, the implementing rules are all fixed, only implementations is extendable.
-



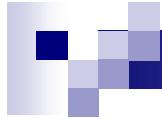
CRP Extension: Inheritance

■ Advantages

- It is easy to introduce the new implementation.
- It is easy to implement new sub-class because most of methods have inherited from base-class.

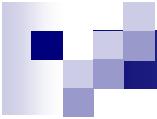
■ Disadvantages

- Inheritance break the encapsulation, the details of base-class is uncovered to the sub-class.
 - If base-class is modified, such modification will effect it sub-class level by level, like water waves when stone throwing in pool.
 - The implementation inherited from base-class is static, can not changed during runtime, it lacks flexibility.
-



CRP Extension: Aggregation/Composition

- Aggregation/Composition is better than inheritance in follows :
 - Accessing the aggregated object through its interface;
 - “black-box” reuse, the details of aggregated objects is transparent;
 - Wrapping is supported;
 - Less dependency;
 - Runtime aggregated the instances of aggregated class.
- The disadvantages of aggregation
 - Many objects which need to be well managed;
 - For being aggregated by other classes, the interface of aggregated class should be designed carefully.



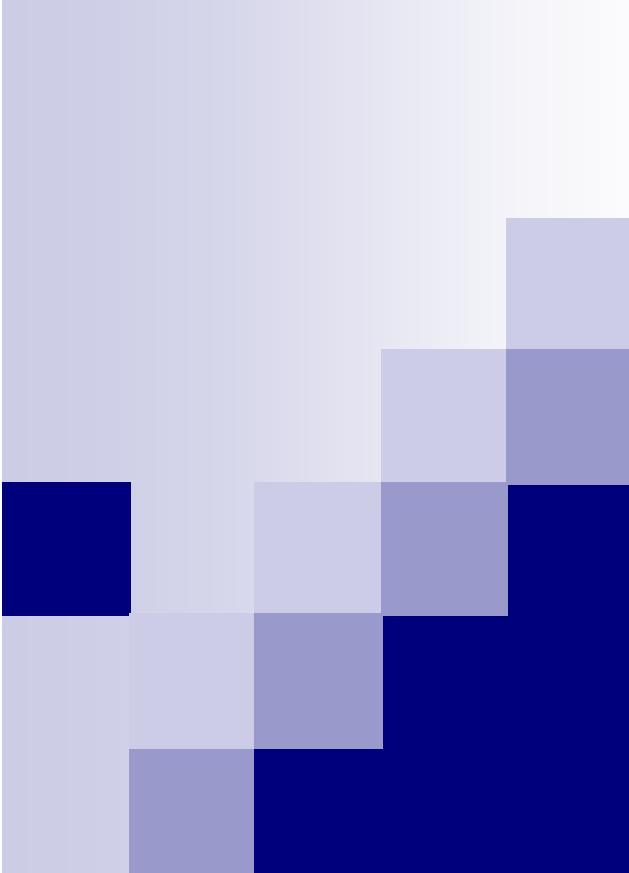
Final Example: Document Processor

- A system which can process the documents one by one;
 - There is three kinds of **Documents**, including **Paper**, **Report** and **Notice**.
 - 1. New types of **Document** may be introduced;
 - 2. Documents need to be sorted by their **name** before they are processed.
 - 3. The sorted rules may be optional;
 - 4. A certain actions should be done before and/or after the document is processed;
 - 5. The pre-processing and post-processing is optional to each document.
-



Let's go to next...





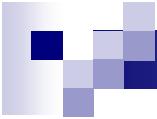
Design Patterns

宋 杰

Song Jie

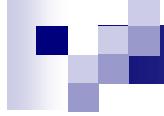
东北大学 软件学院

Software College, Northeastern
University



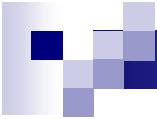
What are design patterns?

- Recurring solutions to design problems you see over and over. (Alpert et al, 1998)
 - A set of rules describing how to accomplish certain tasks in the realm of software development. (Pree, 1994)
 - Focus on reuse of recurring architectural design themes (Coplien and Schmidt, 1995)
 - Address a recurring design problem that arises in a specific context and presents a solution to it (Buschmann. et al, 1996)
 - Identify and specify abstractions that are above the level of single classes or instances, or of components. (GoF,1995)
-



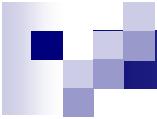
Why study design patterns?

- Up to now in your degree you have been taught the OO basics and some higher level principles,
 - BUT always building software from first principles is wasteful;
 - Better if you can use some existing software or patterns produced and tested by experts;
 - They help when you're in the situation where you think: "Oh I've seen this problem before and I solved it by ..."
-



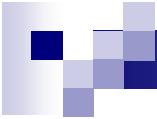
Why use design patterns?

- Reuse of design expertise;
 - Support software that is flexible to change (extension).
 - Improve communication between engineers: A common vocabulary;
-



Problems with design patterns?

- Trade-off: design can become a little more complicated.
 - Inexperienced users often try to use more design patterns than they need to
 - You need to ask yourself why you're using the design pattern.
-

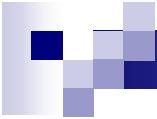


Patterns History

- Software design patterns were inspired by the work of Christopher Alexander (architect).
 - Developed a pattern language for describing architectural features in buildings (1977).
 - Seminal work of 23 software design patterns in:
 - E. Gamma, R. Helm, R. Johnson and J. Vlissades, Design patterns: Elements of reusable object-oriented software, Addison-Wesley, 1994
 - Informally known as the Gang of Four (GoF)
-

Pattern List - Three types of pattern

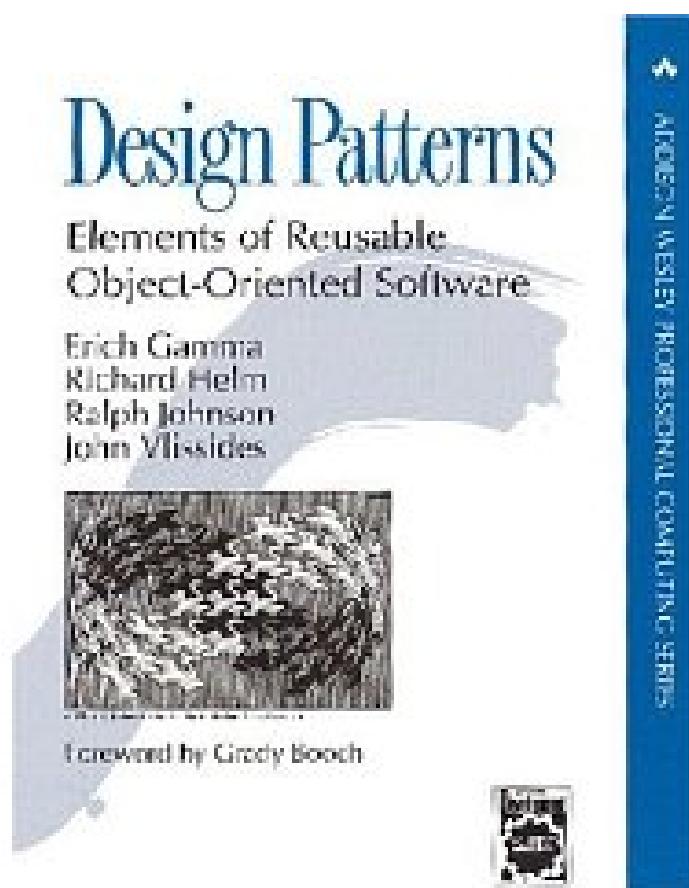
- **Creational Patterns** 创建型
 - Factory Method (工厂方法)
 - Abstract Factory (抽象工厂)
 - Singleton (单例)
 - Builder (创建)
 - Prototype (原型)
- **Structural Patterns** 结构型
 - Adapter (适配器)
 - Decorator(装饰器)
 - Composite(合成)
 - Facade(外观)
 - Flyweight(享元)
 - Proxy(代理)
 - Bridge(桥梁)
- **Behavioral Patterns** 行为型
 - Interpreter(解释器)
 - Template Method(模板方法)
 - Chain of Responsibility(责任链)
 - Command(命令)
 - Iterator(迭代器)
 - Mediator(调停者)
 - Memento(备忘录)
 - Observer(观察者)
 - State(状态)
 - Strategy (策略)
 - Visitor(访问者模式)



How to introduce a pattern

- Intent
 - Structure
 - Participants
 - Collaborations
 - Consequences
 - Applicability
 - Implementation
 - Sample Code
 - Examples
 - Variation
 - Extension
 - Related Patterns
-

Referenced Book



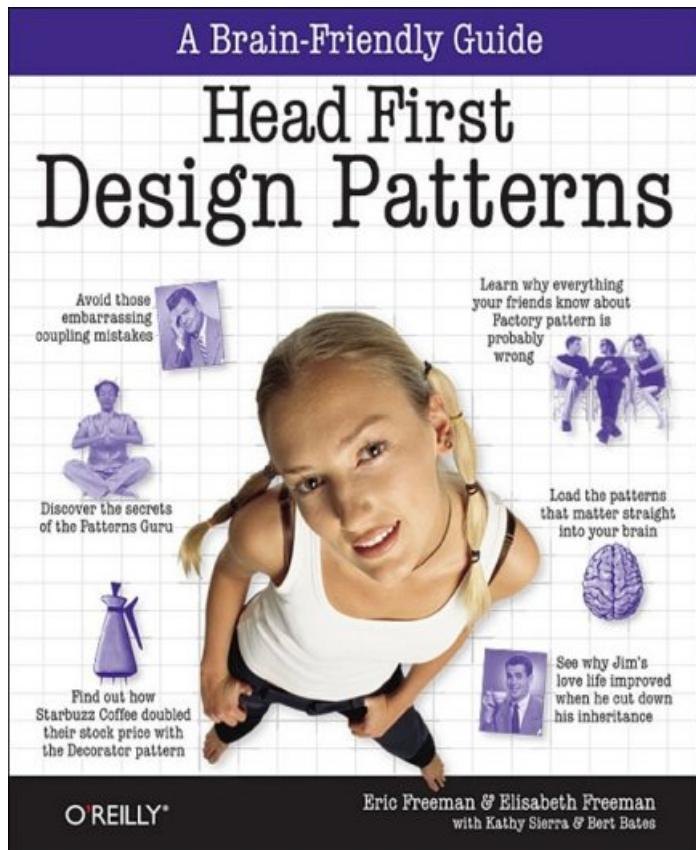
■ Authors (GoF):

- Erich Gamma
- Richard Helm
- Ralph Johnson
- John M. Vlissides

■ Published by:

- Addison Wesley 1994

Referenced Book



■ Authors:

□ Eric Freeman & Elisabeth Freeman with Sierra & Bert Bates

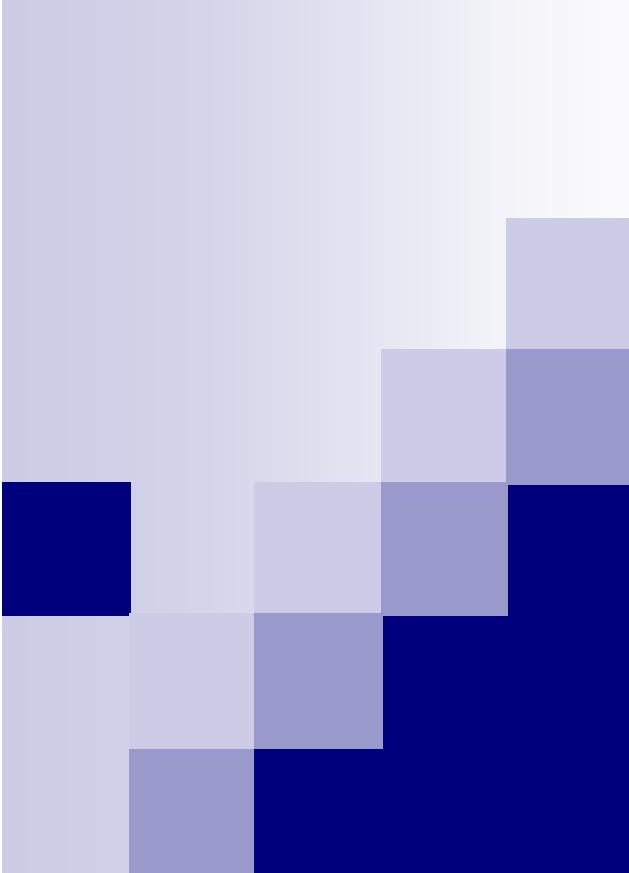
■ Published by:

□ O'REILLY 2005



Let's go to next...





Design Patterns

宋 杰

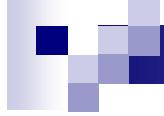
Song Jie

东北大学 软件学院

Software College, Northeastern
University



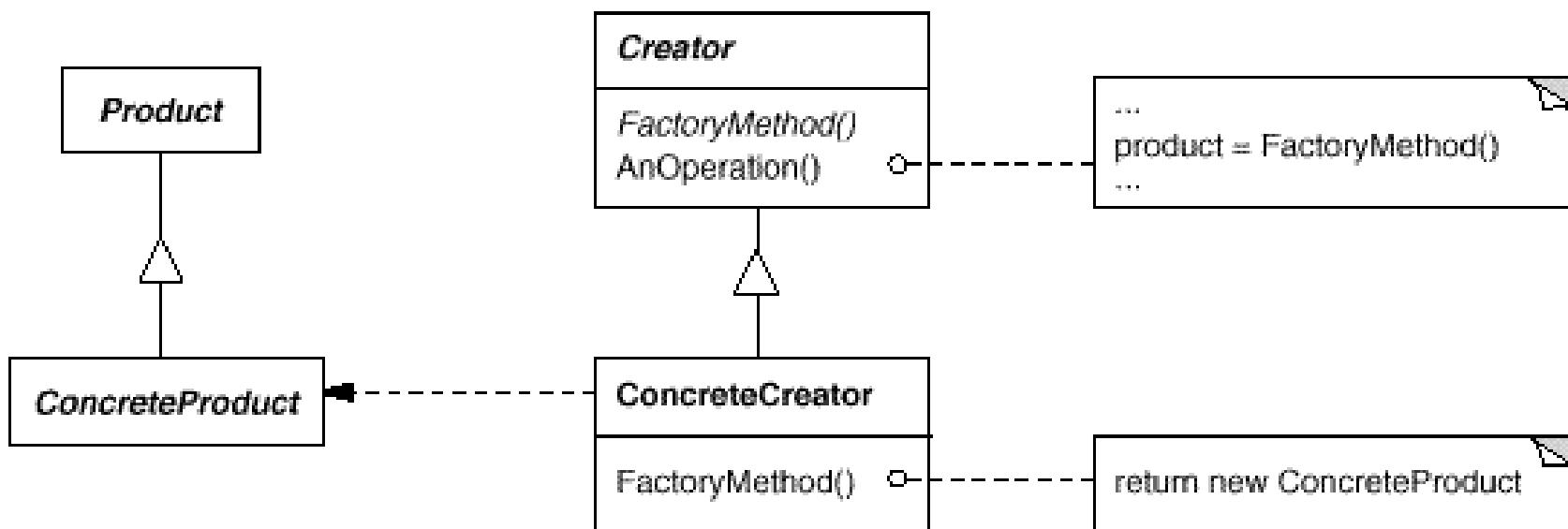
1. Factory Method Pattern

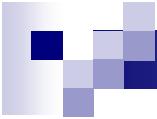


Intent

- Define an **interface** for **creating an object**, but let **subclasses** decide which class to instantiate.
 - Why need a method to **create an object**;
 - Why use **interface**?
 - The factory method return what kind of object?
 - Why let **subclasses** to decide the concrete class?
-

Structure





Participants

- **Product**: defines the interface of objects the factory creates.
 - **ConcreteProduct**: implements the **Product** interface.
 - **Creator**: declares the factory method, which returns an object of type **Product**.
 - **Creator** may also define a default implementation of the factory method that returns a default **ConcreteProduct** object.
 - **ConcreteCreator**: overrides the factory method to return an instance of a **ConcreteProduct**, referred by **Product**.
-

Codes – Product

```
interface Product {  
}
```

```
class DefaultProduct implements Product {  
}
```

```
class ProductA implements Product {  
}
```

```
class ProductB implements Product {  
}
```

Codes – Factory Method (version 1)

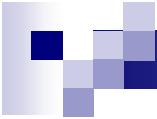
```
interface Factory {  
  
    public Product createProduct(String type);  
}  
  
class ConcreteFactory implements Factory {  
    public Product createProduct(String type) {  
        if (type.equals("A")) {  
            return new ProductA();  
        } else if (type.equals("B")) {  
            return new ProductB();  
        } else {  
            return new DefaultProduct();  
        }  
    }  
}
```

Codes – Factory Method (version 2)

```
interface Factory {  
  
    public Product createProductA();  
    public Product createProductB();  
    public Product createProduct();  
  
}  
  
class ConcreteFactory implements Factory {  
  
    public Product createProductA() {  
        return new ProductA();  
    }  
    public Product createProductB() {  
        return new ProductB();  
    }  
    public Product createProduct() {  
        return new DefaultProduct();  
    }  
}
```

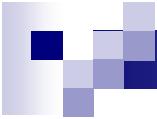
Codes - Client

```
class Client{
    public static void main(String[] args) {
        Factory factory = new ConcreteFactory();
        Product productA = factory.createProductA();
        Product productB = factory.createProductB();
    }
}
```



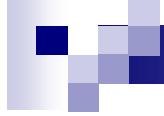
Consequences

- Factory methods eliminate the need to bind application-specific classes into your code.
 - The code only deals with the **Product** interface; therefore it can work with any user-defined **ConcreteProduct** classes.
 - Creating objects inside a class with a factory method is always more flexible than creating an object directly.
-



Applicability

- The concrete **products** are required not to be exposed to **clients**;
 - **Creator** can't decide the concrete **product** it must create;
 - **Creator** wants its subclasses to specify the product it creates.
 - **Creator** delegate the responsibility of creating instance to one of several subclasses.
-



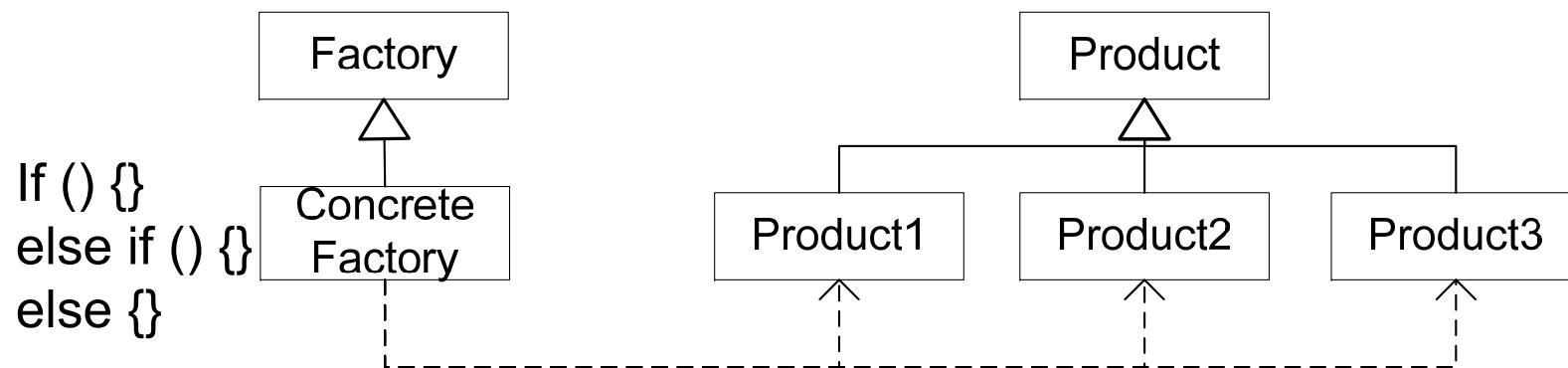
Implementation 1:

Factory methods in pure factory class or not

- Factory method can be contained in a business class which performed some business logic.
 - It always including the logic that using the products.
 - Factory method can be contained in a pure factory class which is responsible for nothing but creating products.
-

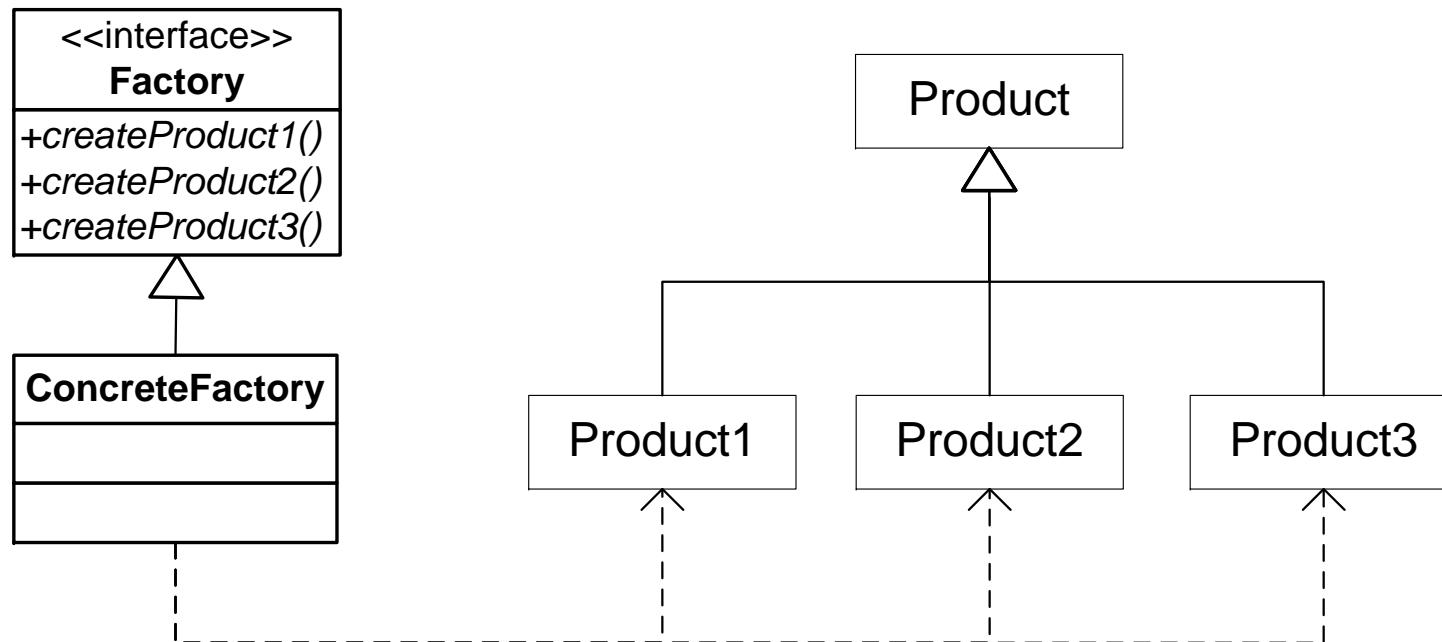
Implementation 2: Parameterized factory methods

- Factory method create multiple kinds of products by taking a parameter that identifies the kind of object to create.



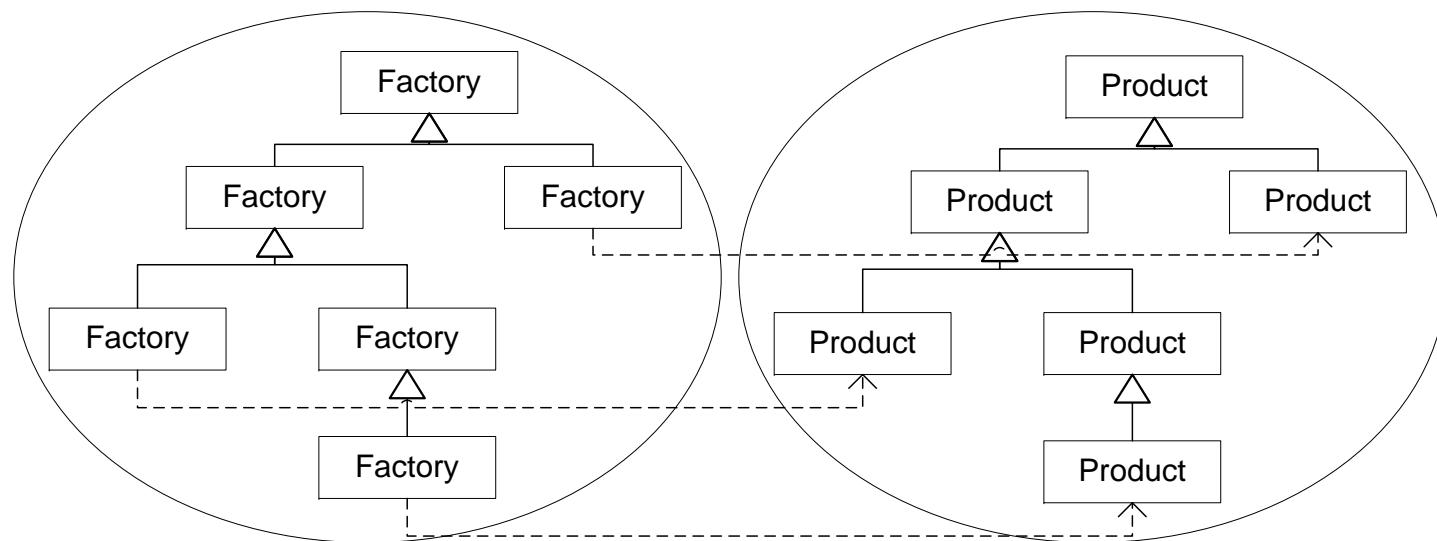
Implementation 3: Parallel factory methods

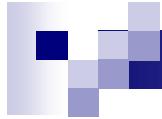
- A group of factory methods create multiple kinds of products by different methods signatures.



Implementation 4: Connects parallel class hierarchies

- Generally, there are many hierarchical products to be created by hierarchical factory. The factory and corresponded product a in same level of inherited hierarchy.



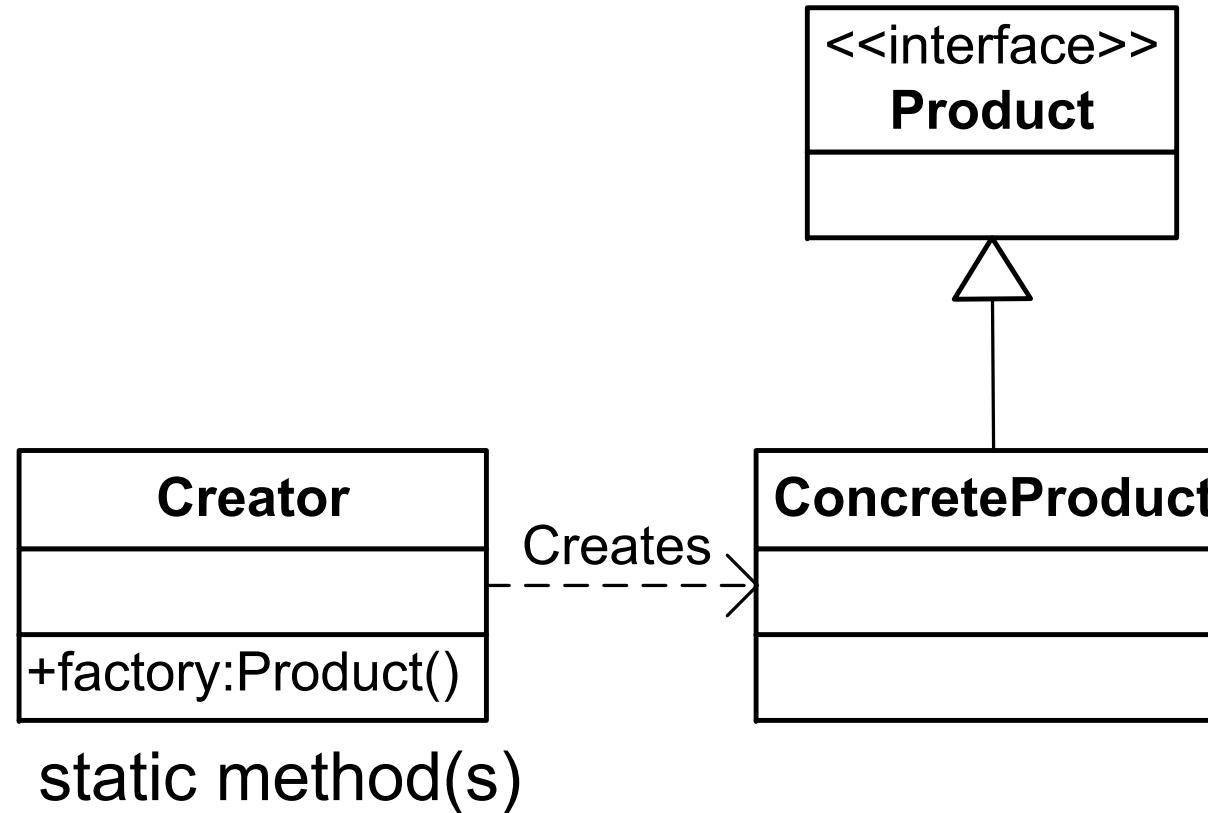


Implementation 5:

Default product providing by default factory

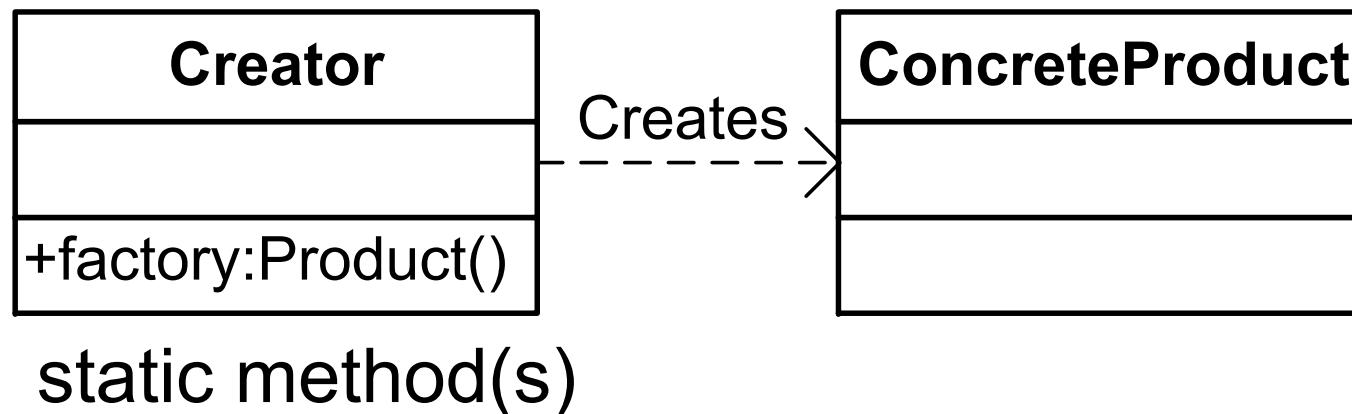
- **Creator** is an abstract class or interface and does not provide an implementation for the factory method it declares, **OR**
 - **Creator** is a concrete class and provides a default implementation for the factory method.
OR
 - An abstract **creator** that defines a default implementation providing default product .
-

Variation 1: Abstract Factory is omitted (Simple Factory Pattern)

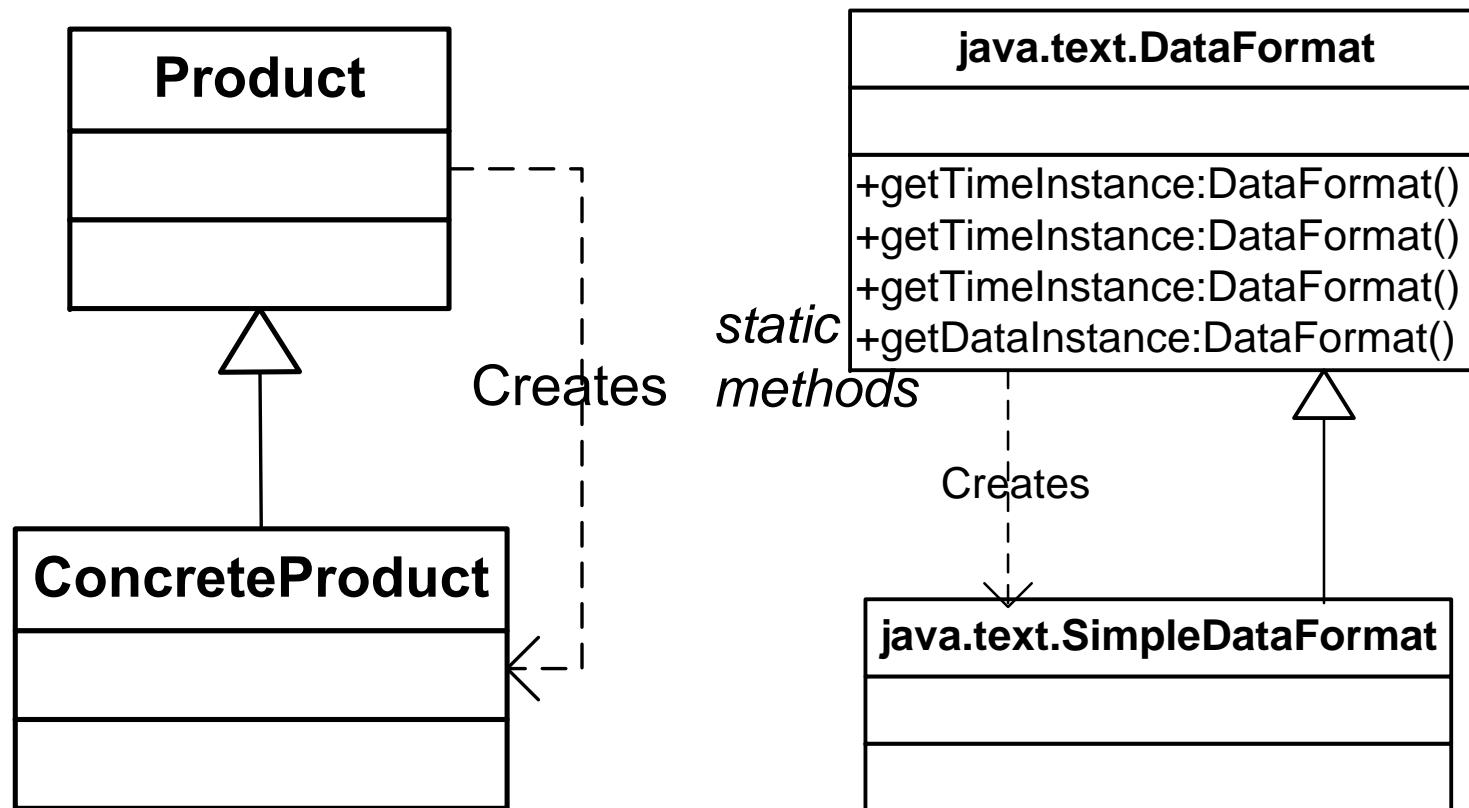


Variation 2: Abstract Factory and Product are omitted

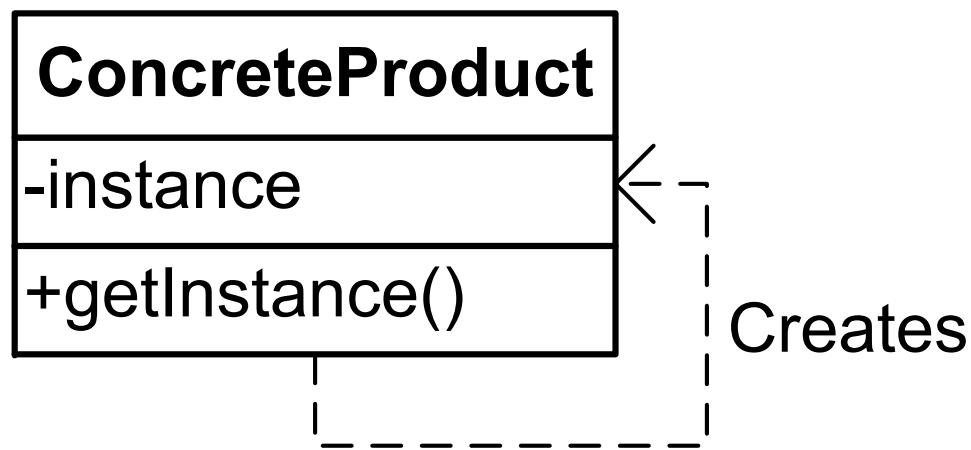
- If products are “unrelated”

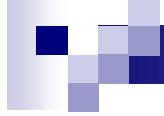


Variation 3: Products contains Factory method to create itself



Variation 4: Concrete Product creates itself (singleton)





Variation 5: Factory with registered instances (products) for instance reusing

- Factory store the created instances in an registered pool;
 - Reusing the registered instance when required;
 - Further more, the instance can not only be create by “new”, but also initialized from other resources, for example from database.
-

Example: JDBC

```
Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
String url="jdbc:oracle:thin:@localhost:1521:orcl";
//orcl为数据库的SID
String user="test";
String password="test";
Connection conn= DriverManager.getConnection(url,user,password);
Statement stmt = conn.createStatement();
```

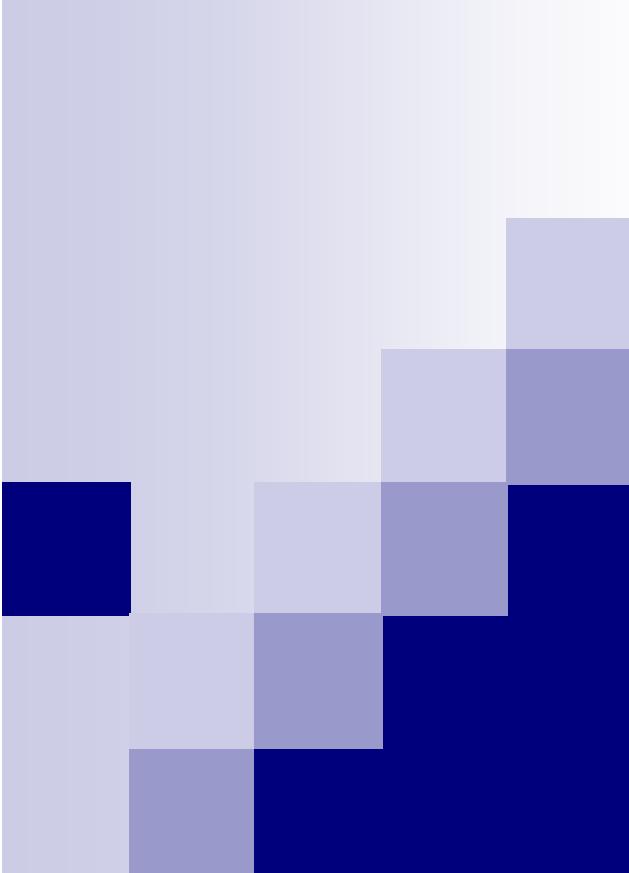
Example: JNDI and EJB

```
//JNDI naming context
Context ctx = new InitialContext();
//find EJB instance,factory method pattern
EmployeeHome home = (EmployeeHome) ctx.lookup("Employee");
//factory method pattern
Employee emp = home.create(1001, "Song", "Jie");
emp.setTel("13940348888");
```



Let's go to next...





Design Patterns

宋 杰

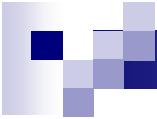
Song Jie

东北大学 软件学院

Software College, Northeastern
University



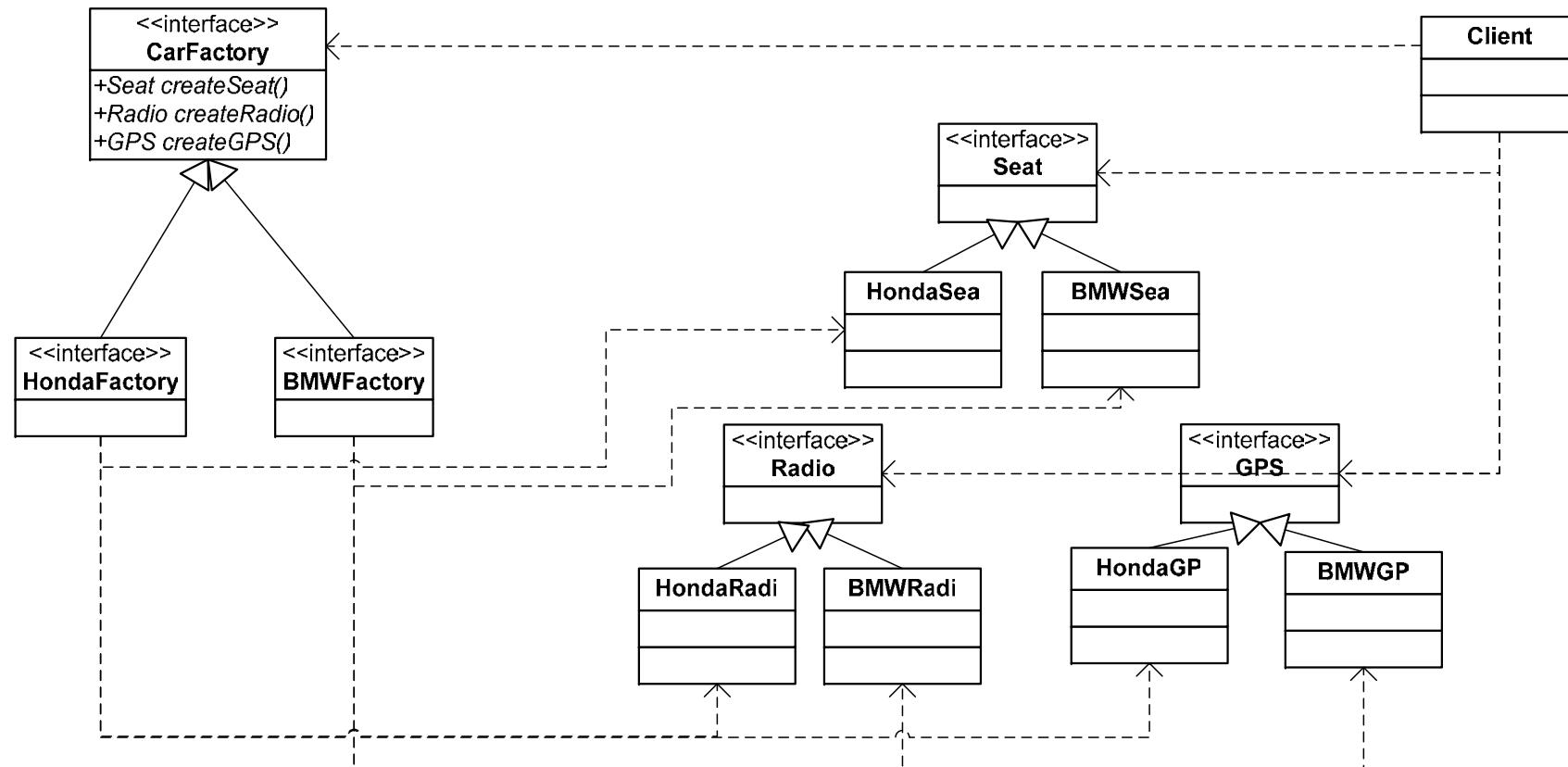
2. Abstract Factory Pattern



Intent

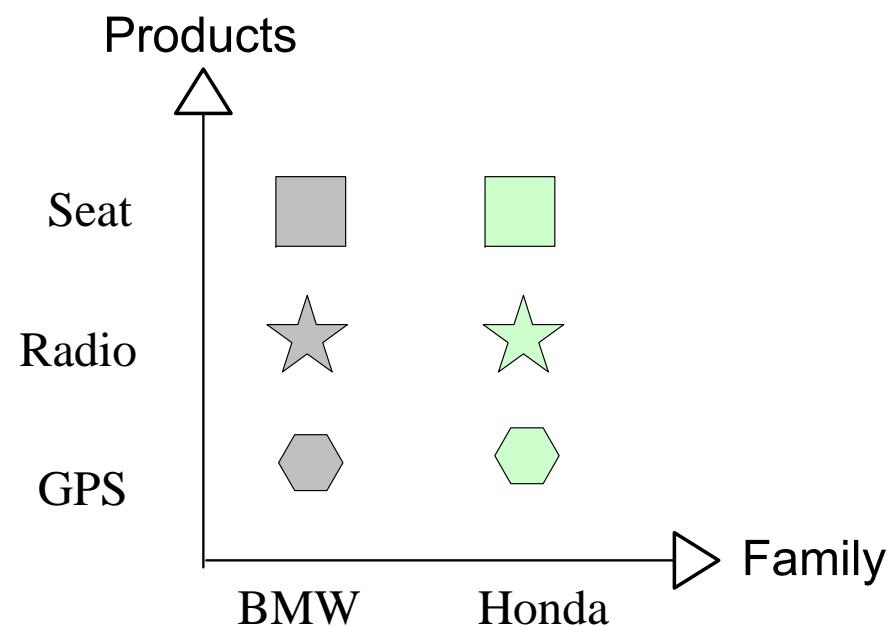
- Provide an **interface** for creating **families** of related or dependent **objects** without specifying their **concrete classes**.
 - Abstract level: Factory creates products in product-family.
 - Concrete level: Concrete factory create different concrete products in one product family, these products are in same inherited level.
 - “Abstract” means both factory and products are abstract.
-

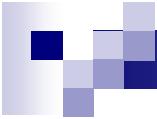
Product Family



Product Family

- Product Family have several related Products
- Every Products have same Concrete Products of each Product Family .
- 2-dimensions to classified the related products. The one is product, the other is family.

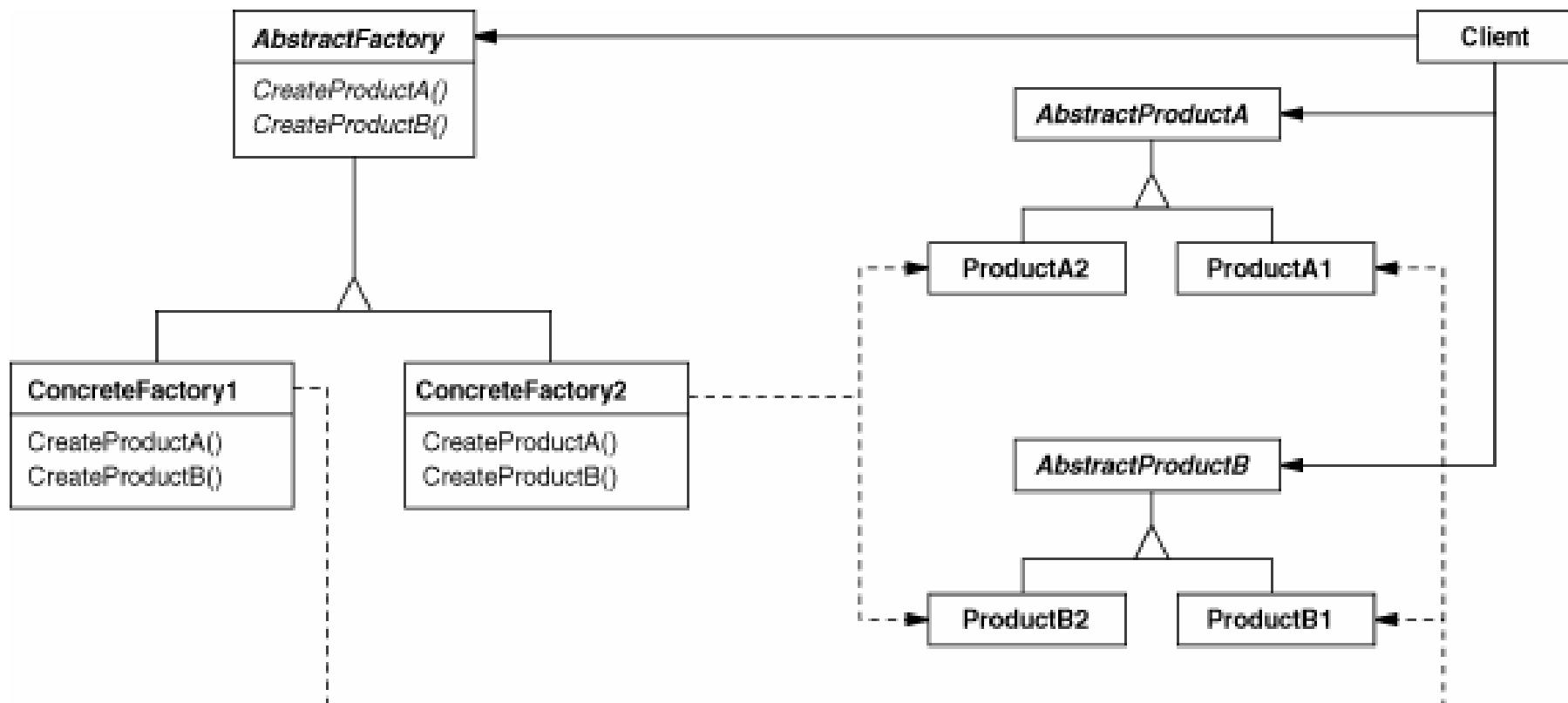


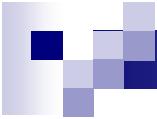


Factory and Product

- Factory method corresponds to product in the product family;
 - Factory class corresponds to all concrete products in product family;
 - Generally, the number of factory method match that of product. The number of concrete factory match that of product family.
-

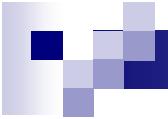
Structure





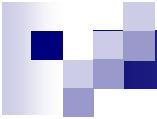
Participants

- **AbstractFactory**: declares an interface for operations that create abstract product objects.
 - **ConcreteFactory**: implements the operations to create concrete product objects.
 - **AbstractProduct**: declares an interface for a type of product object.
 - **ConcreteProduct**: defines a product object to be created by the corresponding concrete factory. implements the **AbstractProduct** interface.
 - **Client**: uses only interfaces declared by **AbstractFactory** and **AbstractProduct** classes.
-



Consequences

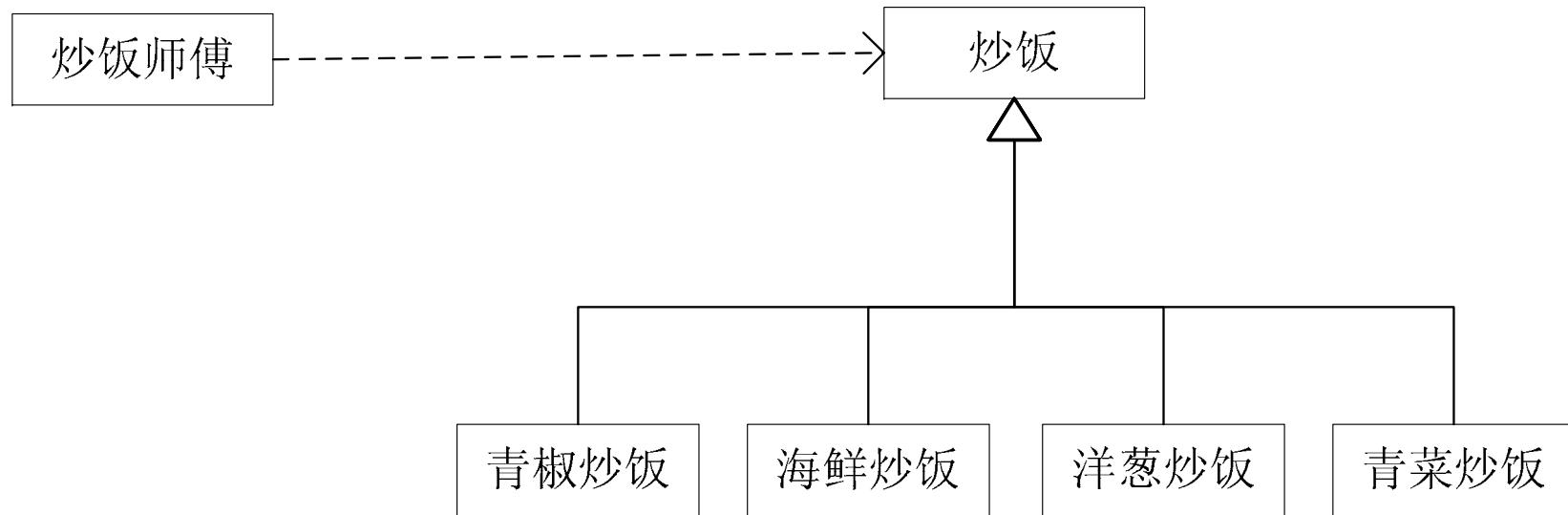
- It isolates concrete classes (products);
- It makes exchanging product families easy (OCP).
- It promotes consistency among products. it's important that an application use objects from only one family at a time.
- Supporting new products is difficult.



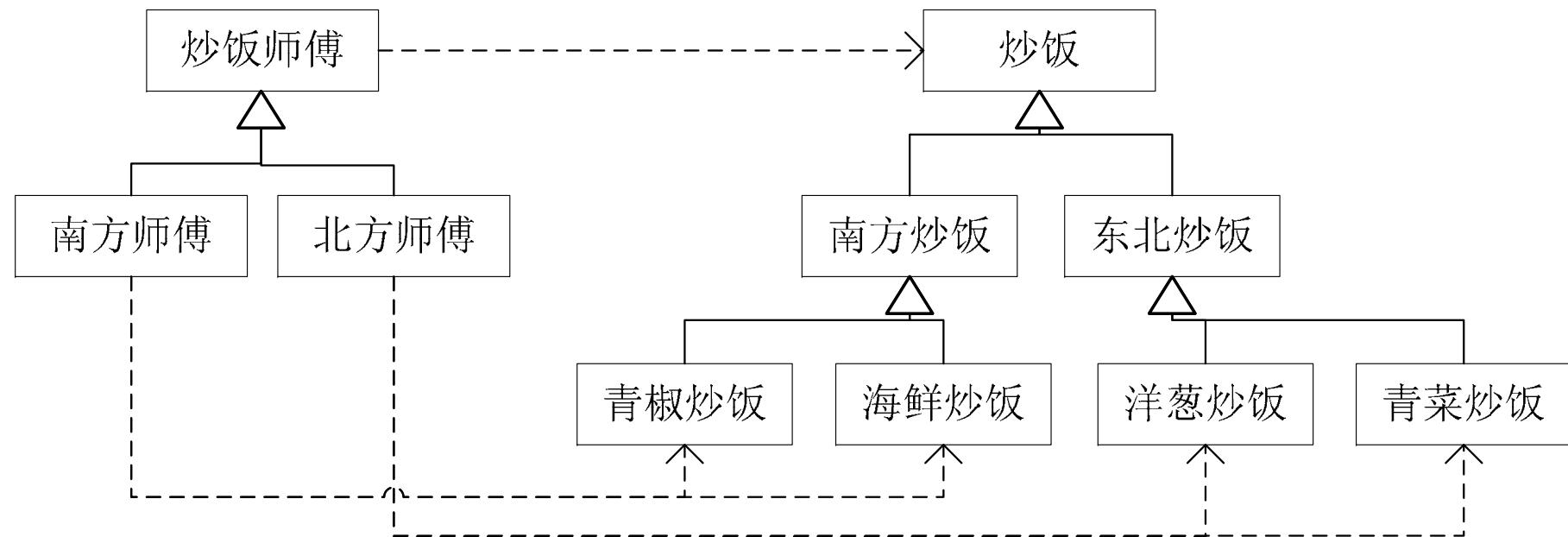
Applicability

- A system should be independent of how its products are created, composed, and represented.
 - A system should be configured with one of multiple families of products.
 - A family of related product objects is designed to be used together, and you need to enforce this constraint.
 - You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.
-

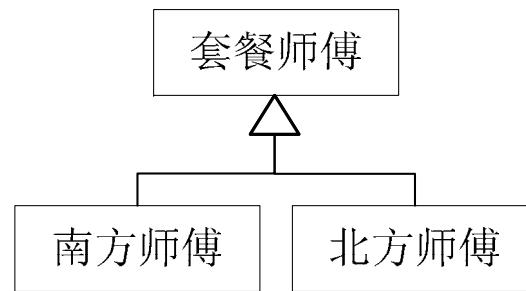
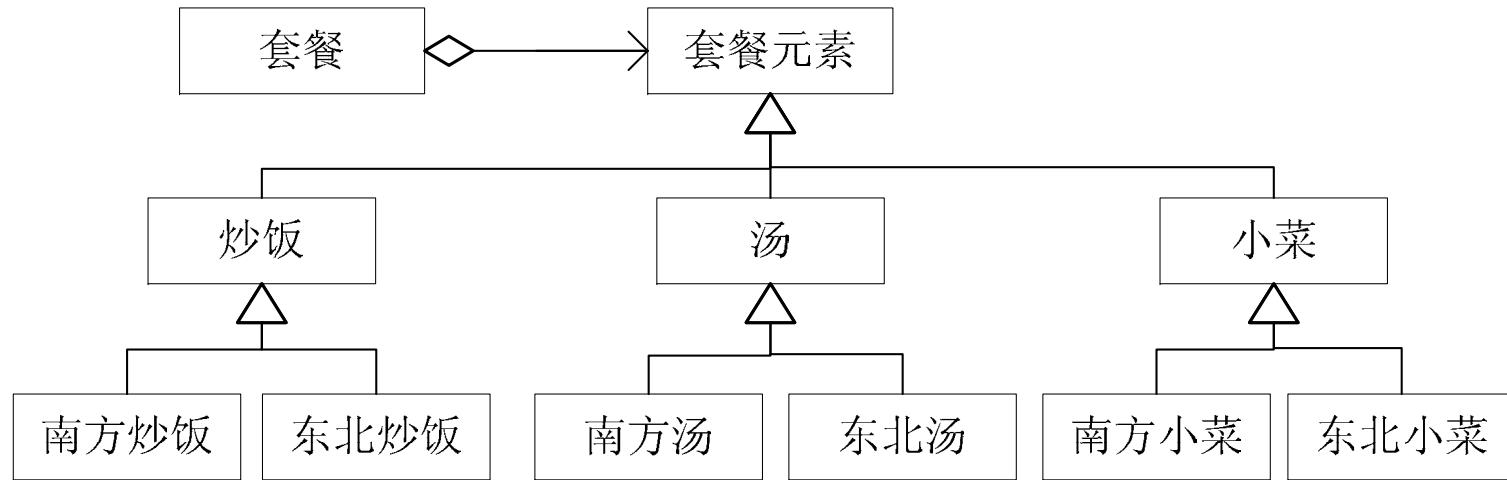
Examples Step 1

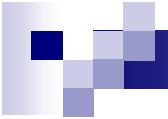


Examples Step 2



Examples Step 3





Related Patterns

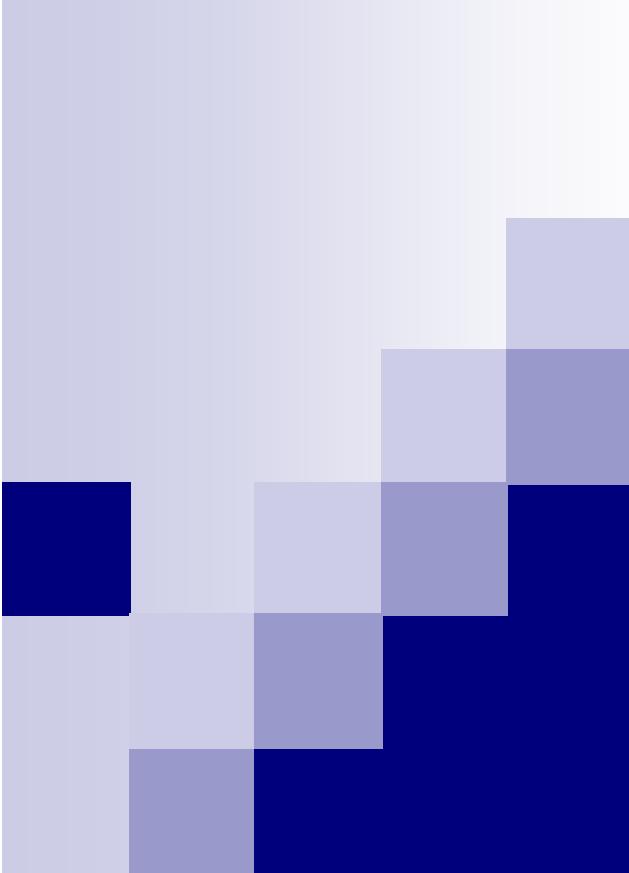
■ Factory Method

- Adding support to product family;
 - Concrete factory use Factory Method to implement the create logic.
-



Let's go to next...





Design Patterns

宋 杰

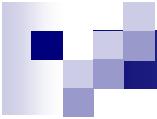
Song Jie

东北大学 软件学院

Software College, Northeastern
University

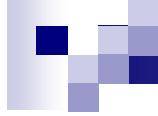


3. Singleton Pattern



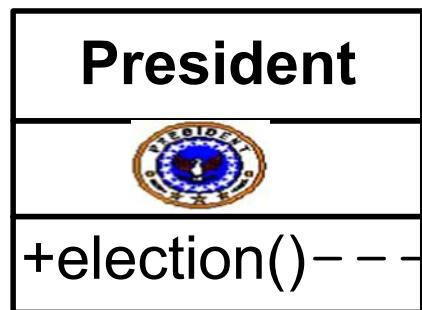
Intent

- Ensure a class only has **one instance**, and **provide a global point of access** to it.
 - Singleton should have one and only one instance;
 - Singleton should create the instance himself;
 - Singleton should provide an approach to access the instance.
-



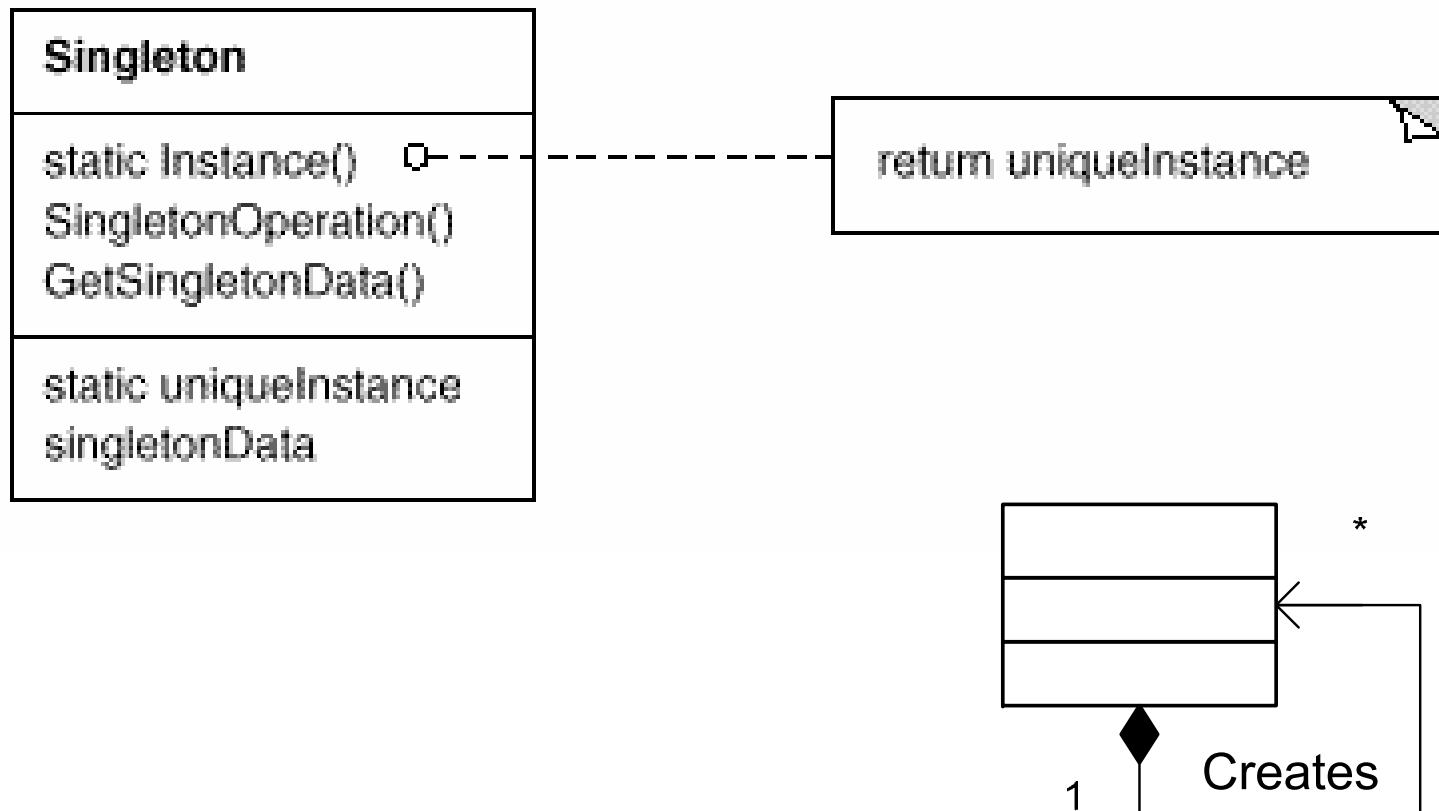
Intent

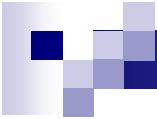
- A country should have one president;



Return a single instance

Structure

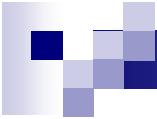




Participants

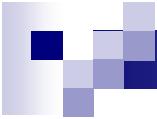
■ Singleton

- Be responsible for creating its own unique instance.
 - Defines an static instance method that lets clients access its unique instance.
-



Consequences

- Controlled access to sole instance;
 - Reduced name space;
 - Polluting the name space with global variables
 - More flexible than static class (class with all static property and method).
 - Static class must be stateless; Singleton is stateful.
-



Applicability

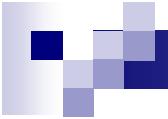
- There must be exactly **one instance of a class**, and it must be accessible to clients from a **well-known access point**.
 - On the contrary, if a system allow multiple instances, it is unnecessary to use singleton.
 - DO NOT reduce the number of instances for using singleton
 - Connection object of database
 - Printer in an system
 - Utilized (Tools) class
-

Implementation: Eager Singleton

```
public final class Singleton {  
    private static Singleton uniqueInstance = new Singleton();  
  
    private Singleton() {  
    }  
  
    public static Singleton getInstance() {  
        return uniqueInstance;  
    }  
}
```

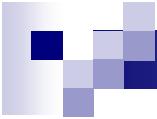
Implementation: Lazy Singleton

```
public final class Singleton {  
    private static Singleton uniqueInstance;  
  
    private Singleton() {  
    }  
  
    public static Singleton getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new Singleton();  
        }  
        return uniqueInstance;  
    }  
}
```



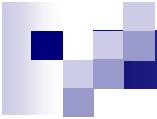
Examples

- President
 - Windows Recycle Bin
 - Java Runtime
- 



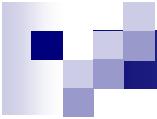
Extension 1: Different between eager and lazy singleton

- **Eager Singleton** initialized itself when class is loaded, it is statically loaded. **Lazy Singleton** initialized itself when instance is first required.
 - From resources utilizing: **Eager Singleton** is worse than **Lazy Singleton**;
 - From runtime efficient: **Eager Singleton** is better than **Lazy Singleton**.
 - **Lazy Singleton** have potential risk when in a multi-threads environment, because it is possible that several threads concurrently required the instance. It will cause that multiple instances are created.
 - **Eager Singleton** is satisfied by Java language. On the contrary, it is not suitable in C++ language because the order of static initialization is unfixed.
-



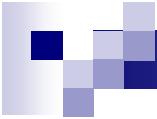
Extension 2: State of Singleton

- Stateful object
 - A stateful object contains and maintain a internal state that is retained across method calls and transactions.
 - Two stateful objects of one class are not same.
 - e.g. Constructor have arguments or class contains properties.
 - Stateless object
 - A stateless object does not have any state between calls to its methods.
 - Two stateless objects of one class are same.
 - e.g. The class do not defines any properties.
 - An singleton instance could be stateful, or stateless.
 - Stateful singleton shares the states among all clients;
 - Stateless singleton is suggested.
-



Extension 3: Singleton in distributed system

- Multiple JVM in a distributed system, OR
 - Multiple class loader
 - If the singleton is stateless, it is no problem;
 - If the singleton is stateful, it will cause inconsistent.
 - Stateless singleton is safety and recommended.
 - For example, and singleton for counting.
-



Extension 4: Singleton and inheritance

- An singleton which are not final (or protected constructor) will allow the sub-class
 - Incomplete singleton if constructor of subclass is public;
 - It is useful in some special situation but not recommended.
-

Extension 5: Singleton and multiple threads

- In the multiple threads environment
 - Lazy Singleton is unsafe
 - Eager Singleton is safe

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

Extension 5: Singleton and multiple threads

```
public final class Singleton {  
    private static Singleton uniqueInstance;  
  
    private Singleton() {  
    }  
  
    public synchronized static Singleton getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new Singleton();  
        }  
        return uniqueInstance;  
    }  
}
```

Extension 5: Singleton and multiple threads

- Optimized as:

- Not really solo instance

```
public static Singleton getInstance() {  
    if (uniqueInstance == null) {  
        // multiple threads will stop here  
        synchronized (Singleton.class) {  
            uniqueInstance = new Singleton();  
        }  
    }  
    return uniqueInstance;  
}
```

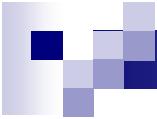
Extension 5: Singleton and multiple threads – Double Checked

```
public static Singleton getInstance() {  
    if (uniqueInstance == null) {  
        // multiple threads will stop here  
        synchronized (Singleton.class) {  
            if (uniqueInstance == null) {  
                uniqueInstance = new Singleton();  
            }  
        }  
    }  
    return uniqueInstance;  
}
```

Extension 5: Singleton and multiple threads – Double Checked

- *uniqueInstance* = new **Singleton()**;
 - locating memory;
 - Invoking the constructor **Singleton()**;
 - Constructing the members of class;
 - *uniqueInstance* = reference of located memory;

- locating memory;
- *uniqueInstance* = reference of located memory;
- Invoking the constructor **Singleton()**;
- Constructing the members of class;



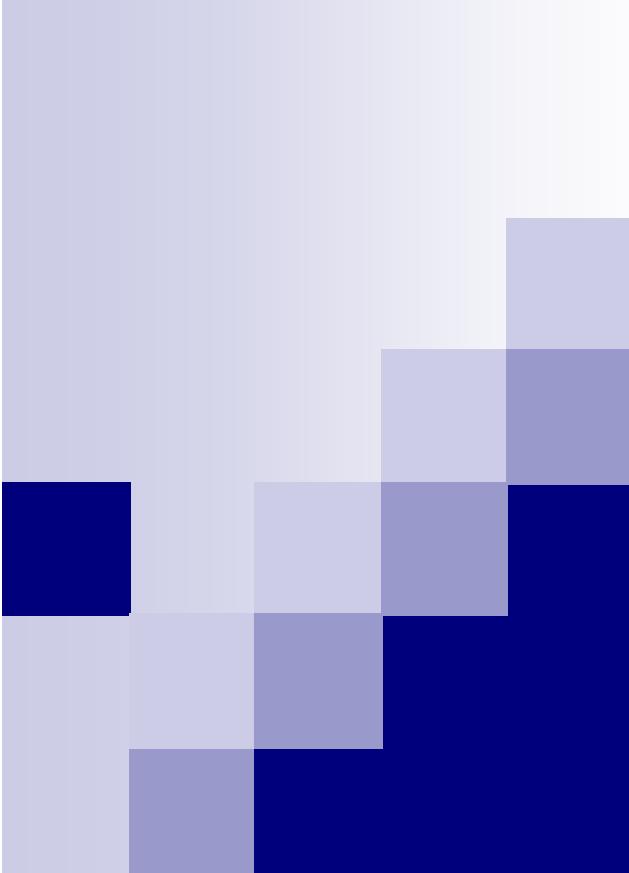
Extension 6: Multiton Pattern

- Multiton (多例)
 - An “singleton” class which create and manage multiple (numbered) instances, then provide global points to assess them .
 - Be treated as a kind of enumerations.
 - Currency, language, region, a group of config-file can adopt Multiton Pattern well.
-



Let's go to next...





Design Patterns

宋 杰

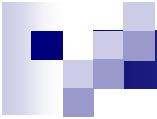
Song Jie

东北大学 软件学院

Software College, Northeastern
University



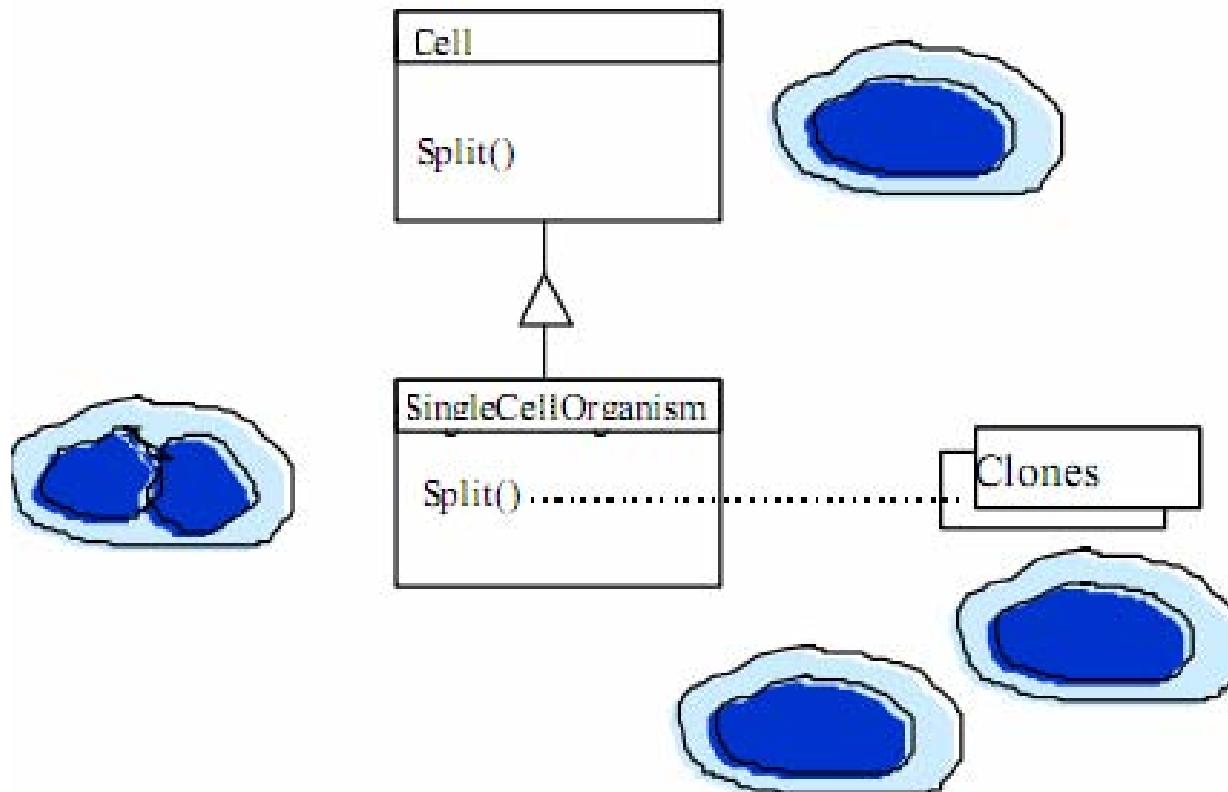
5. Prototype Pattern



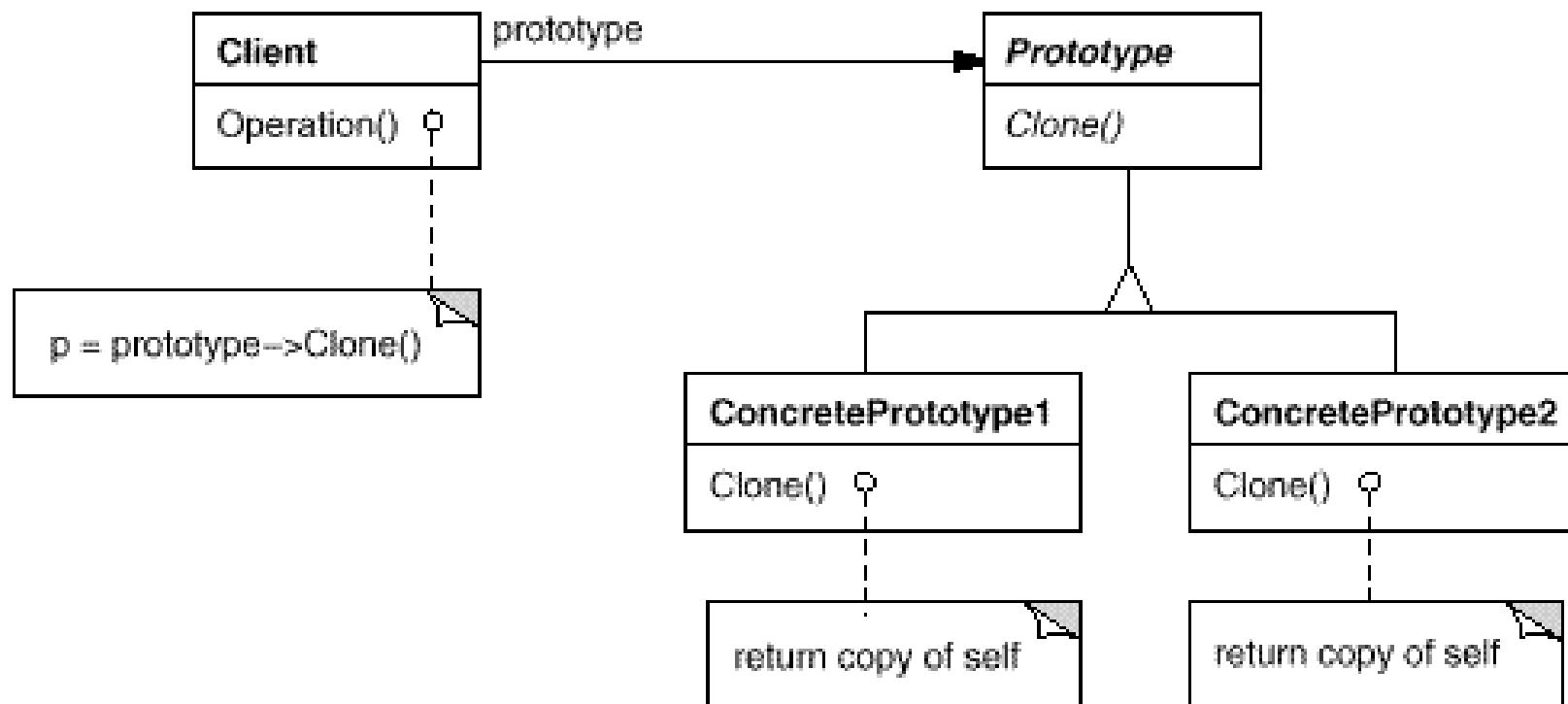
Intent

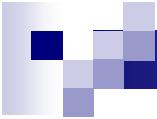
- Specify the kinds of objects to create using a **prototypical instance**, and create new objects by **copying** this prototype.
 - 通过给出一个原型对象来指明所要创建的对象的类型，然后用复制这个原型对象的办法创建出更多同类型的对象。
 - For some objects which are :
 - Complex in internal structure;
 - Difficult to create or unable to create;
 - Complex in initial state;can be created by clone the prototypical instance.
-

Example



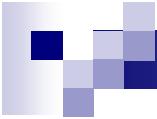
Structure





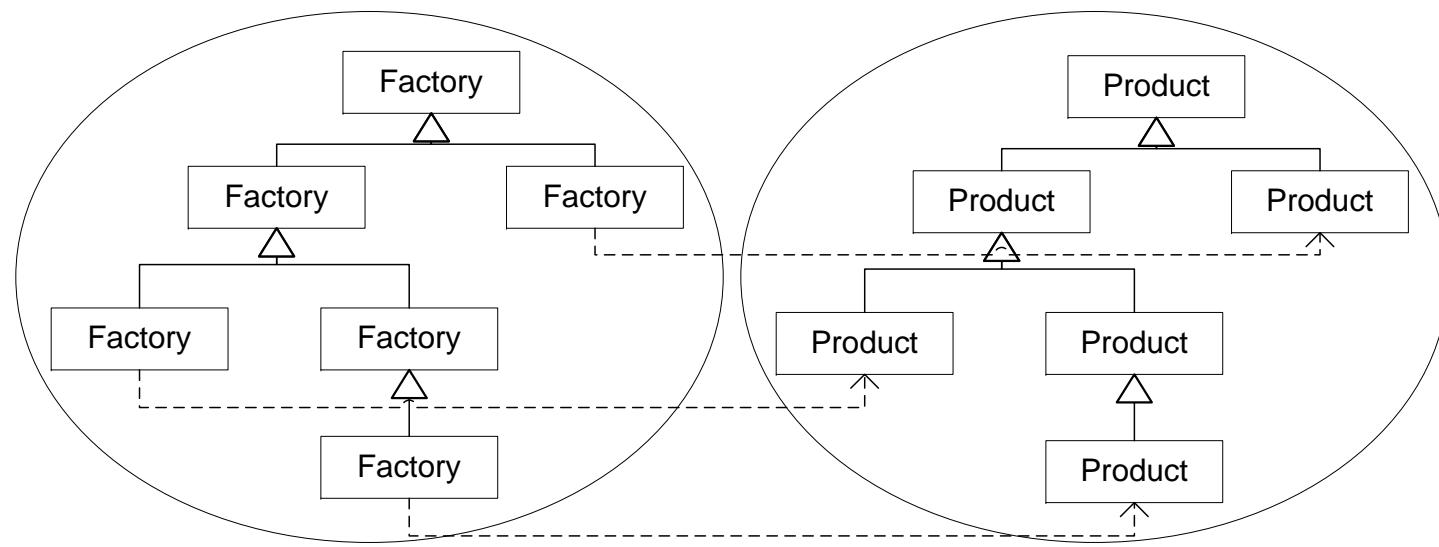
Participants

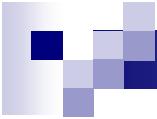
- **Prototype**: Declares an interface for cloning itself.
 - **ConcretePrototype**: Implements an operation for cloning itself.
 - **Client**: Creates a new object by asking a prototype to clone itself.
-



Consequences

- Same consequences that **Abstract Factory** and **Builder** have;
 - Adding and removing products at run-time;
 - Reducing the structure of creators.
 - Each subclass of **prototype** must implement the **Clone** operation, which may be difficult.
-



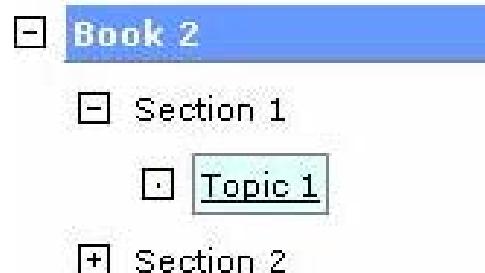
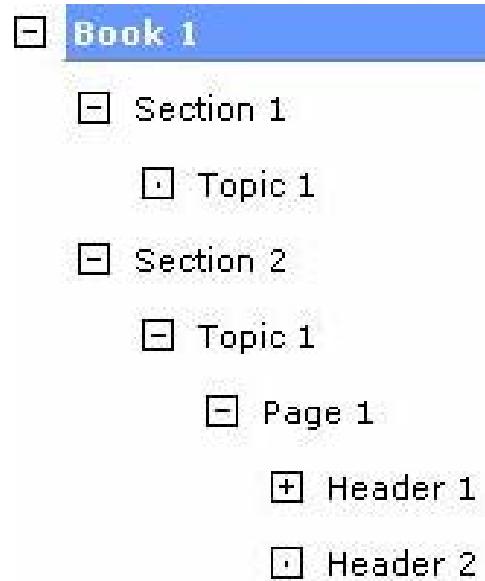


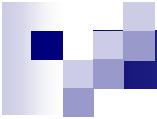
Applicability

- Use the **prototype pattern** when a system should be independent of how its products are created, composed, and represented; **AND**
 - When the classes to instantiate are specified at run-time, for example, by dynamic loading; **OR**
 - To avoid building a class hierarchy of factories that parallels the class hierarchy of products; **OR**
 - When instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.
-

Example

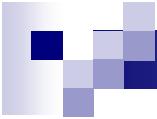
■ Tree-viewed system menu





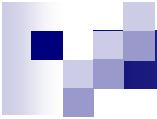
Extension 1: Clone method

- The class includes the `clone()` method for objects to make copies of themselves.
 - A copied object will be a new object instance, separate from the original.
 - The copied object may or may not contain exactly the same state (the same instance variable values) as the original. The state is controlled by the object being copied.
 - The decision as to whether the object allows itself to be cloned at all is up to the object.
-



Extension 2: Clone in Java

- **clone()** is a method in the Java programming language for object duplication.
 - In Java, objects are manipulated through reference variables, and there is no operator for copying an object.
 - The assignment operator duplicates the reference, not the object.
 - The **clone()** method provides this functionality.
clone() acts like a constructor.
-



Extension 2: Clone in `java.lang.Object`

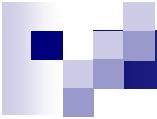
- The mechanisms provided by the Java `Object` class is used to make a simply copy (**shallow copy**) of an object including all of its state;
 - By default, this capability is turned off. In order for an object to be considered cloneable, an object must implement the `java.lang.Cloneable` interface.
 - Flag interfaces that indicates the object wants to cooperate in being cloned. The `Cloneable` interface **does not** actually contain any methods. (**Why??**)
 - If the object isn't `Cloneable`, the `clone()` method throws a `CloneNotSupportedException` exception.
-

Extension 2: Clone in `java.lang.Object`

- The `clone()` method is declared as "protected"
 - By default it can be called only by an object on itself, an object in the same package, or another object of the same type or a subtype.
- By convention, classes that implement this interface should override `Object.clone()` with a public method.
(Why not be abstract??)

```
protected native Object clone()
throws CloneNotSupportedException;
```

The Java Native Interface (JNI) is a programming framework that allows Java code running in a Java Virtual Machine (JVM) to call and to be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages, such as C, C++ and assembly.



Extension 3: Conditions of cloning

- For every object x , `Object.clone()` method satisfies:
 - $x.clone().getClass() == x.getClass();$
 - $x.clone() != x$ (reference equity);
 - $x.clone().equals(x)$ (if equals method is defined as valued equity)
-

Extension 4: Returned type of `clone()`

- The syntax for calling `clone` in Java is:

`Object copy = obj.clone();`

or commonly

`MyClass copy = (MyClass) obj.clone();`

which provides the typecasting needed to assign the generic `Object` reference returned from `clone` to a reference to a `MyClass` object.

- One disadvantage with the design of the `clone()` method is that the return type of `clone()` is `Object`, and needs to be explicitly cast back into the appropriate type.
- However, overriding `clone()` to return the appropriate type is preferable and eliminates the need for casting in the client (since J2SE 5.0).

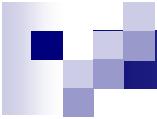
- Covariance (协变) of the return type refers to a situation where the return type of the overriding method is changed to a type related to (but different from) the return type of the original overridden method, following the Liskov substitution principle.

Extension 5: `clone()` in interface

- Cannot access the `clone()` method on an abstract type (**interface** but not **abstract class**).

```
List list = new ArrayList();
list.clone(); //?????
```

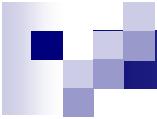
- The only way to use the `clone()` method is if you know the actual class of an object; which is contrary to the abstraction principle of using the most generic type possible (DIP) .



Extension 6: clone() and the Singleton pattern

- Override the `clone()` method in a Singleton

```
public Object clone() throws CloneNotSupportedException {  
    throw new CloneNotSupportedException();  
}
```



Extension 7: `clone()` and inherited hierarchy

- Every **type reference** that needs to support the `clone` function, The **type reference itself or one of its parents** must
 - have a publicly accessible `clone()` method
 - Implements `Cloneable`.
-

```
interface I extends Cloneable{
}

abstract class X implements I{
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

abstract class Y extends X {
}

class Z extends Y {
```

```
public class Prototype {
    public static void main(String[] args)
        throws CloneNotSupportedException {
        I i = new Z();
        X x = new Z();
        Y y = new Z();
        Z z = new Z();

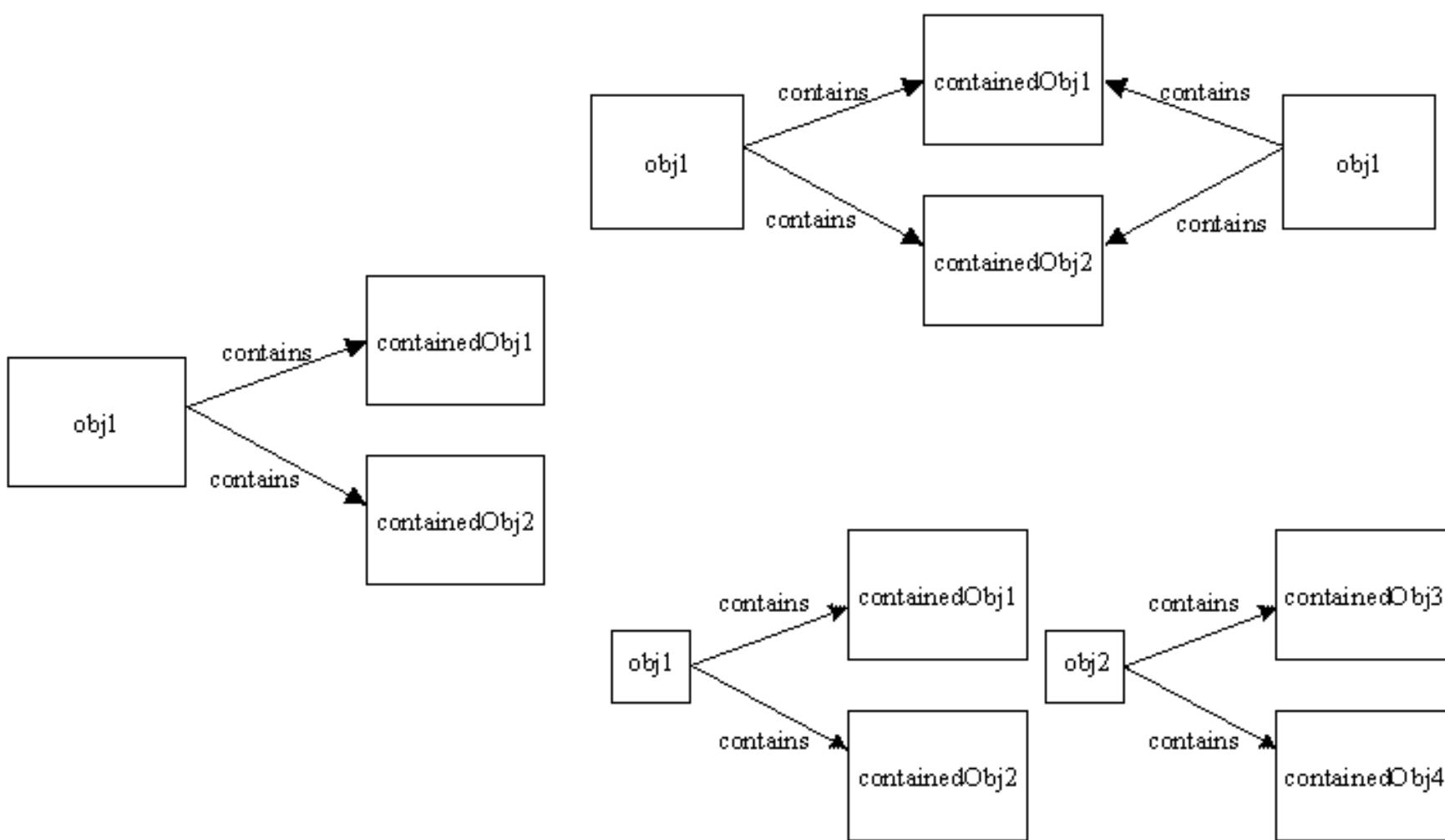
        X x0 = (X)i.clone();
        X x1 = (X)x.clone();
        X x2 = (X)y.clone();
        X x3 = (X)z.clone();

        Y y0 = (Y)i.clone();
        Y y1 = (Y)x.clone();
        Y y2 = (Y)y.clone();
        Y y3 = (Y)z.clone();

        Z z0 = (Z)i.clone();
        Z z1 = (Z)x.clone();
        Z z2 = (Z)y.clone();
        Z z3 = (Z)z.clone();
        System.out.println("OK");
    }
}
```

Extension 8: Shallow copy (clone) and Deep copy (clone)

- **Shallow copy**: A new object is created that has an exact copy of the values in the original object. If any of the fields of the object are references to other objects, just the references are copied. (`Object.clone()`)
- **Deep copy**: It is a complete duplicate copy of an object. A deep copy generates a copy not only of the primitive values of the original object, but copies of all sub-objects as well, all the way to the bottom, it is **field-for-field copy** .
- If you need a true, complete copy of the original object, then you will need to implement a full deep copy for the object.



Extension 8: Deep copy

- If the object refers to other complex objects, which in turn refer to others, then this task can be daunting indeed.
 - With complex object graphs deep copying can become problematic, with recursive references. Once one object is cloneable, others tend to follow until the entire graph attempts to implement **Cloneable**. Sooner or later you run into a class that you can't make **Cloneable**.
- Traditionally, each class in the object must be individually implement the **Cloneable** interface and override its **clone()** method, in order to make a deep copy of itself as well as its contained objects.

```
class A implements Cloneable{
    private B b = new B();
    private List<C> cList = new ArrayList<C>();

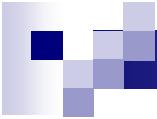
    public A clone() throws CloneNotSupportedException {
        A a = (A)super.clone();
        a.b = this.b.clone();
        //a.cList = this.cList.clone();
        List<C> clonedList = new ArrayList<C>();
        for (C object : this.cList) {
            clonedList.add(object.clone());
        }
        a.cList = clonedList;
        return a;
    }
}

class B implements Cloneable{
    public B clone() throws CloneNotSupportedException {
        return (B)super.clone();
    }
}

class C implements Cloneable{
    public C clone() throws CloneNotSupportedException {
        return (C)super.clone();
    }
}
```

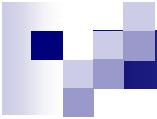
Extension 9: `clone()` and final fields

- Generally, deep `clone()` is incompatible with final fields.
 - `clone()` is essentially a default constructor (one that has no arguments), it is impossible to assign a final field within a `clone()` method;
 - A compiler error is the result.;
 - Changing the final field to immutable field.
- The only solution is to remove the final modifier from the field, giving up all the benefits it conferred.
- For this reason, many programmers prefer to clone the objects by Serialization.



Extension 10: Deep cloning through Serialization

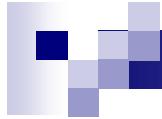
- **Serialization:** Saving the current state of an object to a stream,
 - **Deserialization:** Restoring an equivalent object from that stream.
 - The stream functions as a container for the object. Its contents include a partial representation of the object's internal structure, including variable types, names, and values.
 - The container may be transient (RAM-based) or persistent (disk-based). A transient container may be used to prepare an object for transmission from one computer to another.
-



Extension 10: Deep cloning through Serialization

- Serialization is used for reconstruct the object immediately.
 - Ensure that all classes in the object's graph (all fields) are **serializable** (implements **Serializable**).
 - Create **input stream** and **output stream**.
 - Use the **input stream** and **output streams** to create **object input stream** and **object output stream**.
 - Pass the object that you want to copy to the **object output stream**.
 - Read the new object from the **object input stream** and cast it back to the class of the object you sent.
-

```
public abstract class SerialCloneable implements Cloneable, Serializable {  
  
    private static final long serialVersionUID = SerialCloneable.class.hashCode();  
  
    public Object clone() {  
        try {  
            ByteArrayOutputStream bout = new ByteArrayOutputStream();  
            ObjectOutputStream out = new ObjectOutputStream(bout);  
            out.writeObject(this);  
            out.close();  
  
            ByteArrayInputStream bin = new ByteArrayInputStream(bout  
                    .toByteArray());  
            ObjectInputStream in = new ObjectInputStream(bin);  
            Object ret = in.readObject();  
            in.close();  
            return ret;  
        } catch (Exception e) {  
            e.printStackTrace();  
            return null;  
        }  
    }  
}
```

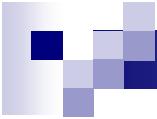


Extension 10: Problems of Serialization

- Serialization is **hugely expensive**. It could easily be a hundred times more expensive than the **clone()** method.
 - Not all objects are serializable.
 - Making a class serializable is tricky and not all classes can be relied on to get it right.
-

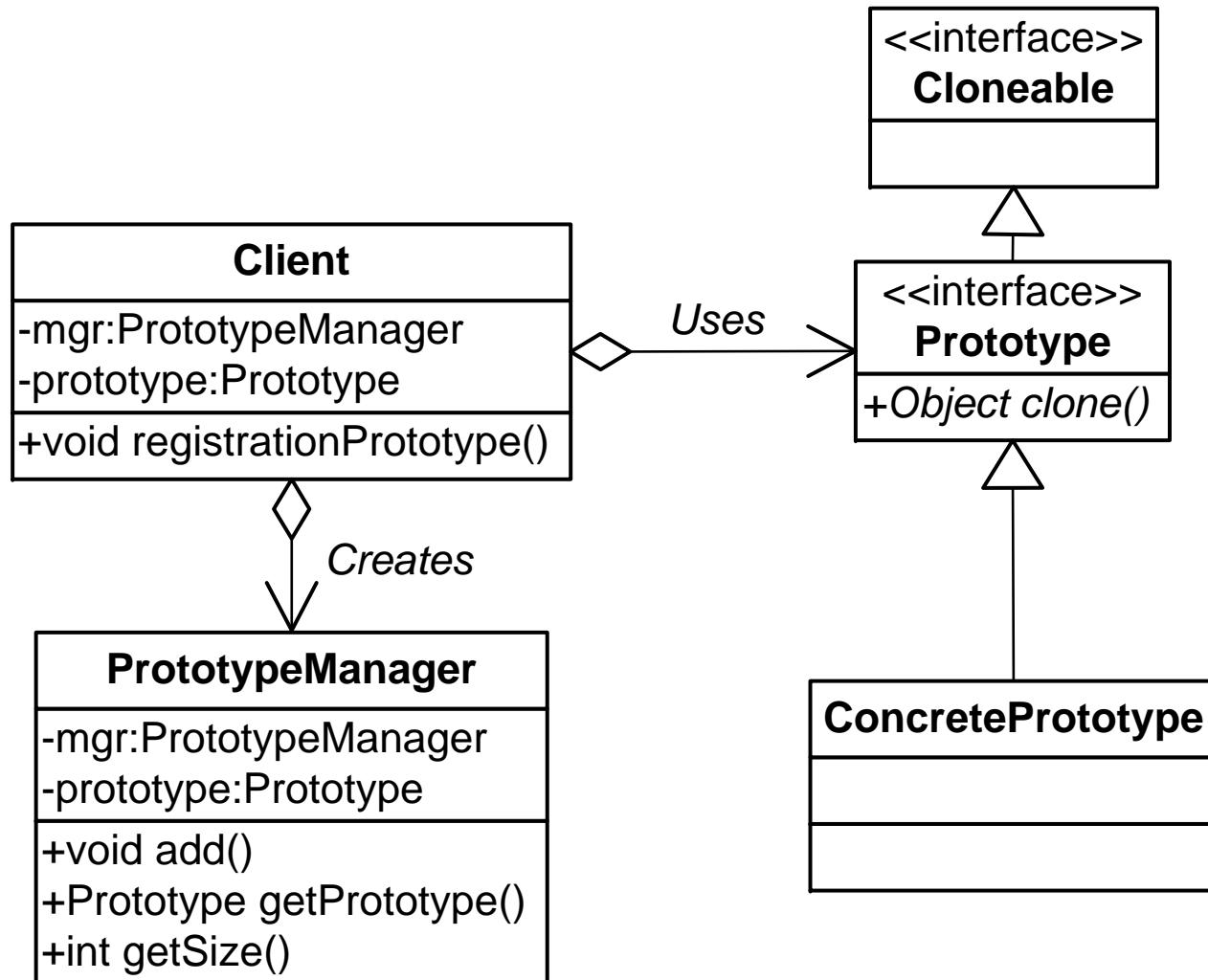
Extension 10: Conditions of Serialization

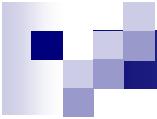
- Some classes are not suitable for serialization:
 - Related to the native code or unmanaged resources;
 - The internal state of an instance is relay on the runtime context or Java virtual machine, such as **Thread**, **InputStream**, **PrintJob**, **Connection**;
 - May bring the potential security problems;
 - Singletons;
 - The state of an instance is changed frequently;
 - An utilized class which mainly contains static methods;



Variation 1: Using a prototype manager

- When the number of prototypes in a system isn't fixed, keep a registry of available prototypes.
 - Clients won't manage prototypes themselves but will store and retrieve them from the registry.
 - A prototype manager is an associative store that returns the prototype matching a given key.
 - A prototype manager has operations for registering a prototype under a key and for unregistering it.
 - A client will ask the registry for a prototype before cloning it.
 - Clients can change or even browse through the registry at run-time. This lets clients extend and take inventory on the system without writing code.
-





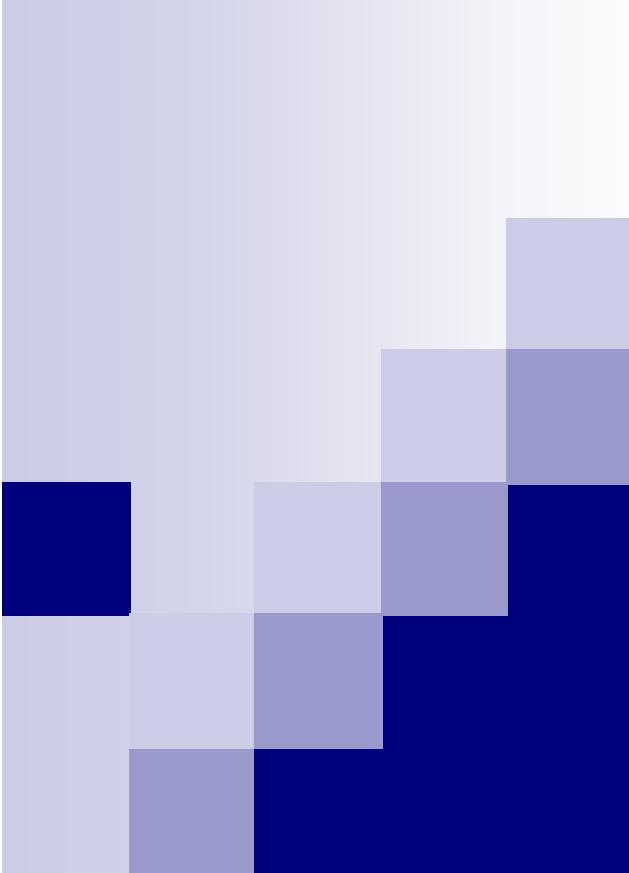
Variation 2: Initializing clones

- Some time the new cloned instance is required to have different states from the prototype.
 - Initialize or reset some or all of its internal state to values of their choosing.
 - Generally, you can't pass these values in the **clone** operation, because their number will vary between classes of prototypes.
 - Passing parameters in the **clone** operation precludes a uniform cloning interface.
 - If prototype classes already define operations for (re)setting states. Clients may use these operations immediately after cloning.
 - If not, then you may have to introduce an initialize operation that takes initialization parameters as arguments and sets the clone's internal state accordingly.
-



Let's go to next...





Design Patterns

宋 杰

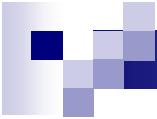
Song Jie

东北大学 软件学院

Software College, Northeastern
University



6. Adapter Pattern



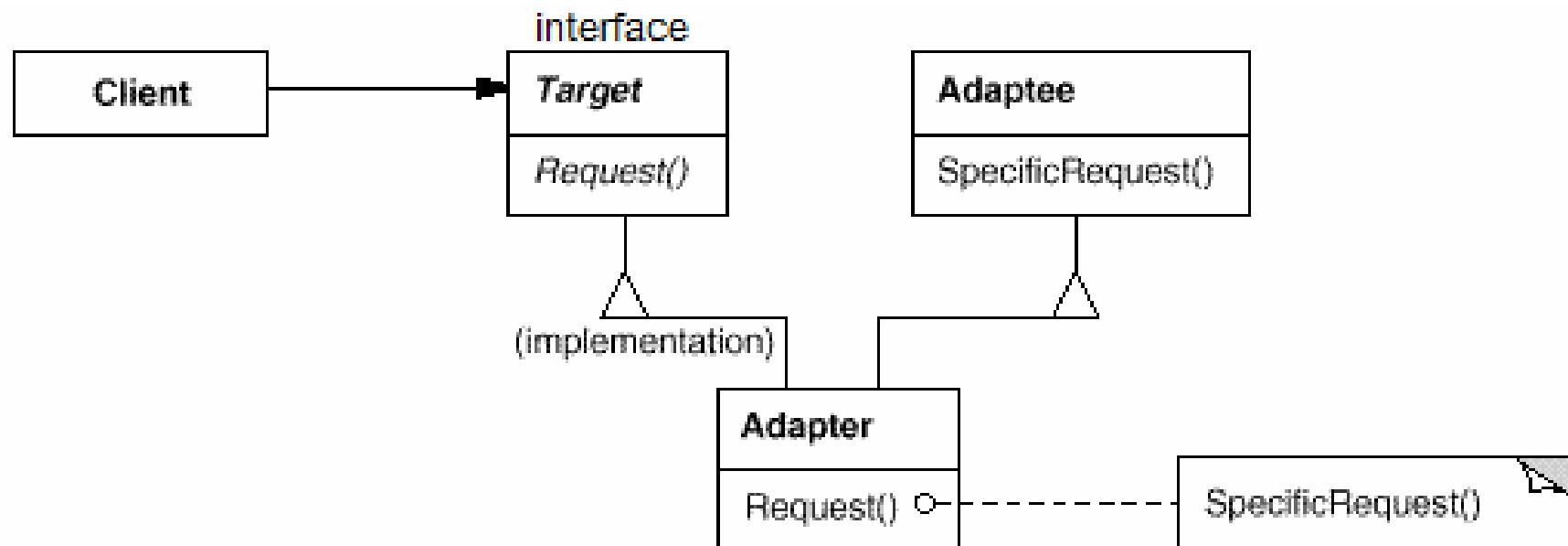
Intent

- Convert the interface of a class into another interface **clients expect**.
 - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
-

Example



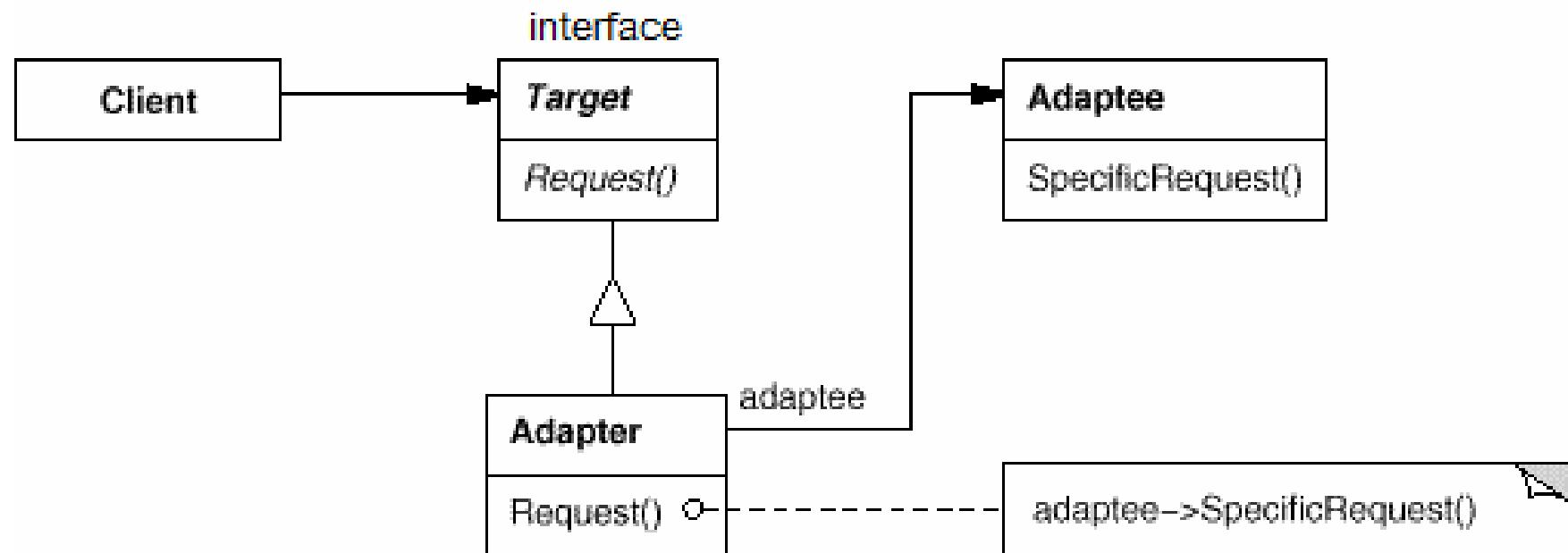
Structure - Class Adapter



Code Example

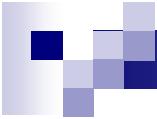
```
interface Target {  
    public void theMethod();  
}  
  
class Adaptee {  
    public void anotherMethod() {  
        // do something;  
        this.anotherMethod();  
        // do something;  
    }  
  
}  
  
class ClazzAdapter extends Adaptee implements Target {  
  
    public void theMethod() {  
        // do something;  
        this.anotherMethod();  
        // do something;  
    }  
  
    // override the super.anotherMethod() if necessary  
    // public void anotherMethod() {  
    //  
    //    }  
}
```

Structure - Object Adapter



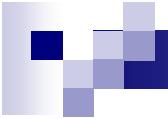
Code Example

```
class ObjectAdapter implements Target {  
    private Adaptee adaptee;  
  
    ObjectAdapter() {  
        adaptee = new Adaptee();  
    }  
    ObjectAdapter(Adaptee adaptee) {  
        this.adaptee = adaptee;  
    }  
    public void theMethod() {  
        // do something;  
        adaptee.anotherMethod();  
        // do something;  
    }  
}
```



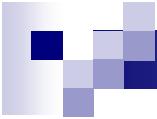
Participants

- **Target:** Defines the domain-specific interface that **Client** uses. It should be an **interface**;
 - **Client:** Collaborates with objects conforming to the **Target** interface;
 - **Adaptee:** Defines an existing interface that needs adapting, could be an **interface**, or abstract class, or class;
 - **Adapter:** Adapts the interface of **Adaptee** to the **Target** interface.
-



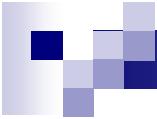
Consequences - Class Adapter

- Adapting **Adaptee** to **Target** by committing to a concrete **Adapter** class;
 - **Adapter** could override some of **Adaptee**'s behavior;
 - A class adapter won't work when we want to adapt a class and all its subclasses;
 - Introducing only one **concrete Adapter** class, there is only one way making client access the **Adaptee**.
-



Consequences - Object Adapter

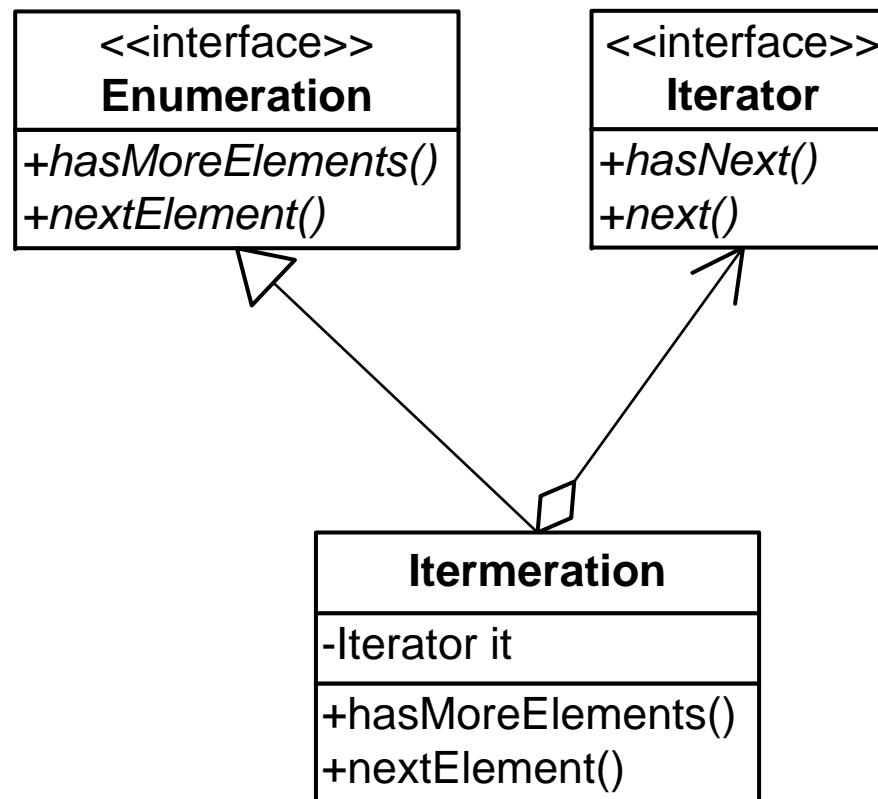
- A single **Adapter** work with many **Adaptees**—that is, the **Adaptee** itself and all of its subclasses (if any).
 - The **Adapter** can also add functionality to all **Adaptees** at once.
 - It is harder to override **Adaptee** behavior.
 - It will require subclassing **Adaptee**, then
 - Making **Adapter** aggregated the subclass rather than the **Adaptee** itself.
 - It is easy to add any new methods, what's more, the added method is suitable for all **Adaptee**.
-



Applicability

- You want to use an existing class, and its interface does not match the one you need.
 - You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
 - **(object adapter only)** You need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.
-

Example: Iterator and Enumeration



```
class Itermeration implements Enumeration<Object> {
    Iterator<Object> it;

    public Itermeration(Iterator<Object> it) {
        this.it = it;
    }

    public boolean hasMoreElements() {
        return it.hasNext();
    }

    public Object nextElement() throws NoSuchElementException {
        return it.next();
    }
}
```

```
class ItermerationTest {  
    public static void main(String args[]) {  
        List<Object> list = new ArrayList<Object>();  
        Iterator<Object> it = list.iterator();  
        Enumeration<Object> em = new Itermeration(it);  
        while (em.hasMoreElements()) {  
            System.out.println(em.nextElement());  
        }  
    }  
}
```

Example: Java I/O

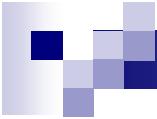
- **ByteArrayInputStream** inherited **InputStream** (abstract class), and contains an **byte array** . It adapter an byte array to **InputStream**
 - **ByteArrayOutputStream** and byte array.
 - **FileInputStream** and **FileDescriptor**
 - **FileOutputStream** and **FileDescriptor**

```
public class ByteArrayInputStream extends InputStream {  
  
    protected byte buf[];
```

Example: WINE

- Wine lets you run Windows software on other operating systems. With Wine, you can install and run these applications just like you would in Windows.
- Which is Target which is Adaptee?





Variation 1: Default Adapter

- In some situations, a class should implement an interface but it does not want to implement every methods that are defined in the interface;
 - A solution is let the unimplemented methods be empty;
 - An **default adapter** implements the interface, but let all the implemented methods be empty methods, or default implementations.
 - The concrete class extends **default adapter** for implementing the interface, overrides the special methods it wants to implement
 - Generally, the **default** adapter is an abstract class.
-

WindowListener

```
public interface WindowListener extends EventListener {  
    public void windowOpened(WindowEvent e);  
    public void windowClosing(WindowEvent e);  
    public void windowClosed(WindowEvent e);  
    public void windowIconified(WindowEvent e);  
    public void windowDeiconified(WindowEvent e);  
    public void windowActivated(WindowEvent e);  
    public void windowDeactivated(WindowEvent e);  
}
```

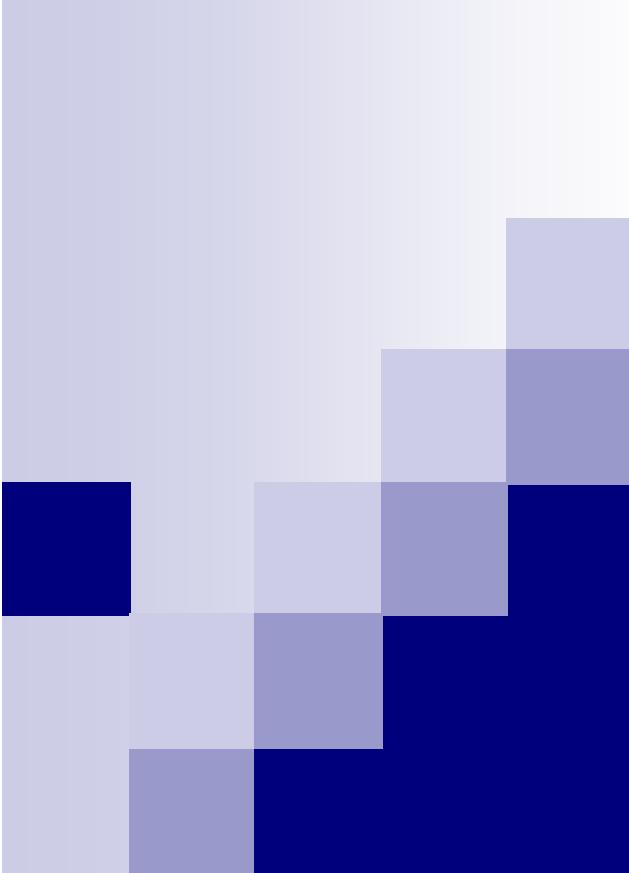
WindowAdapter

```
public abstract class WindowAdapter
    implements WindowListener, WindowStateListener, WindowFocusListener
{
    public void windowOpened(WindowEvent e) {}
    public void windowClosing(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
    public void windowStateChanged(WindowEvent e) {}
    public void windowGainedFocus(WindowEvent e) {}
    public void windowLostFocus(WindowEvent e) {}
}
```



Let's go to next...





Design Patterns

宋 杰

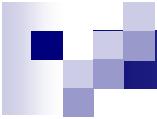
Song Jie

东北大学 软件学院

Software College, Northeastern
University



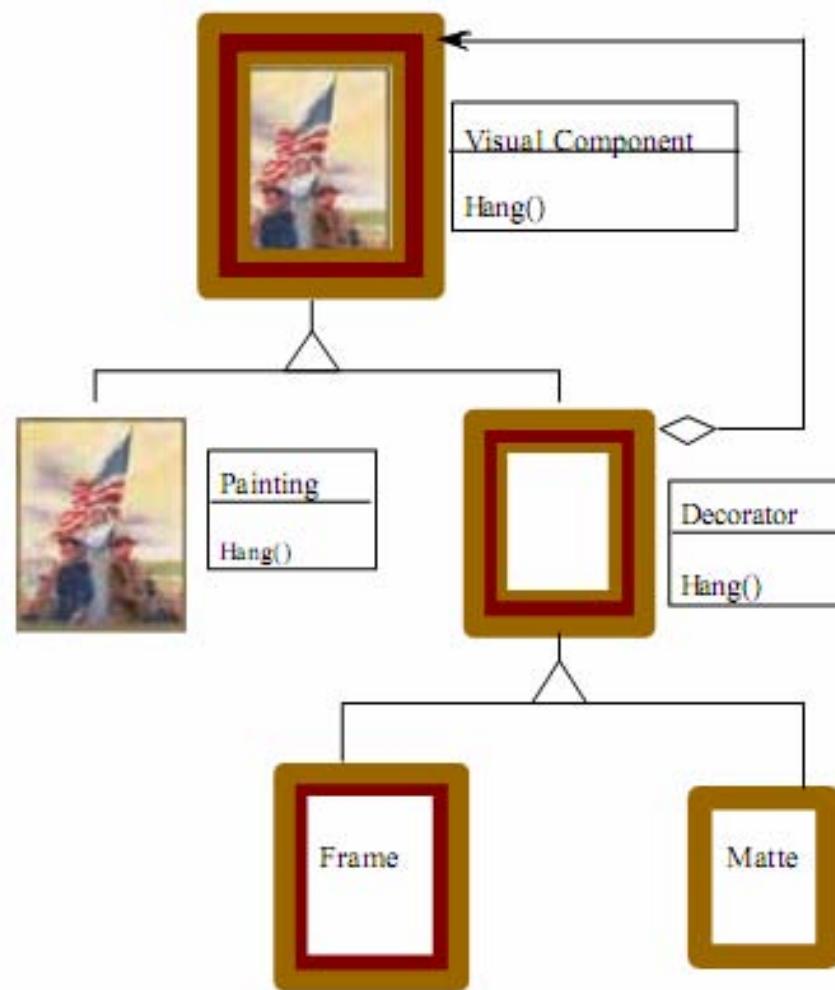
7. Decorator Pattern



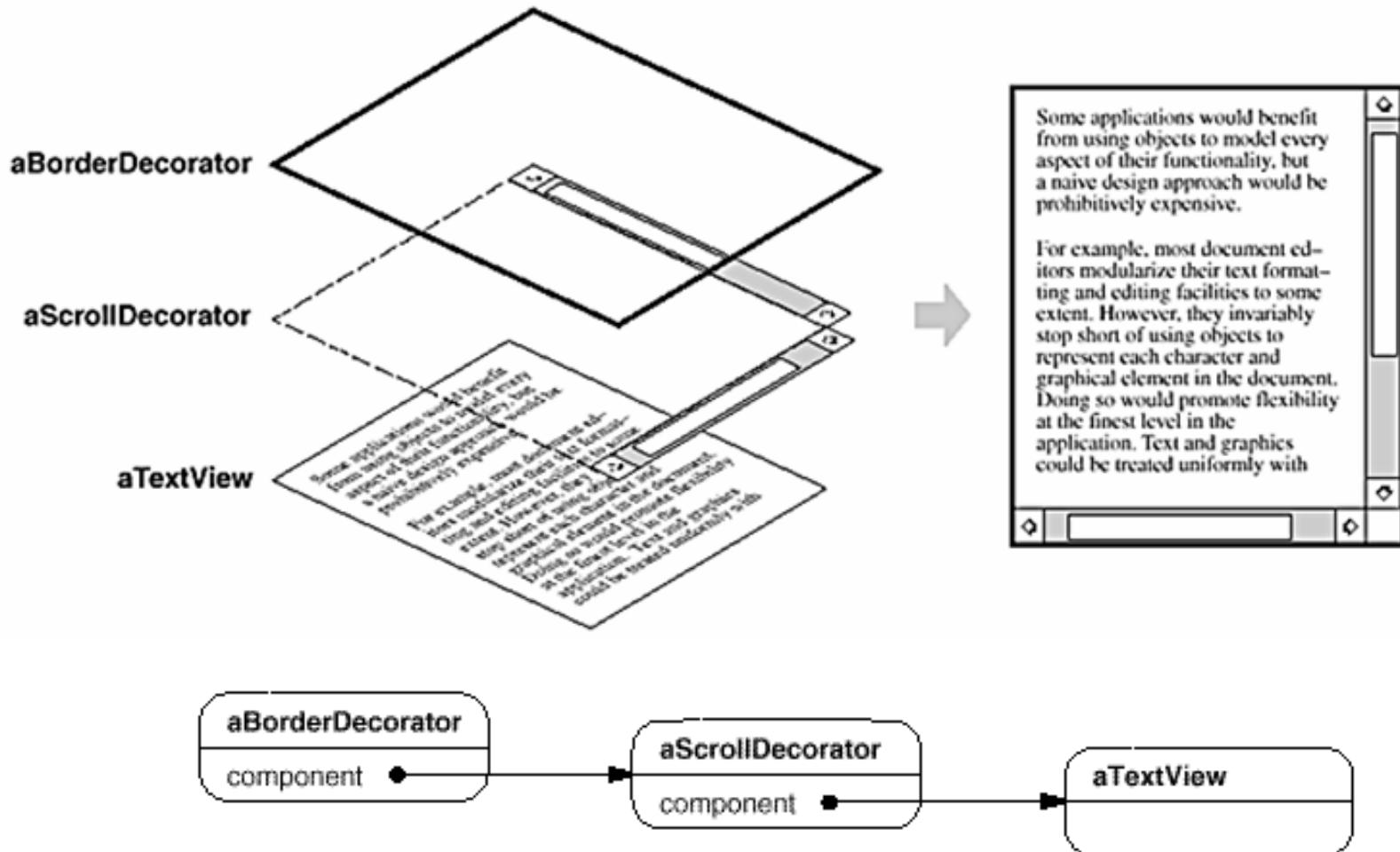
Intent

- Attach additional **responsibilities** to an object **dynamically**.
 - Decorators provide a flexible alternative to **subclassing** for extending functionality.
 - Dynamically extension;
 - Better than inheritance;
 - Wrapper;
-

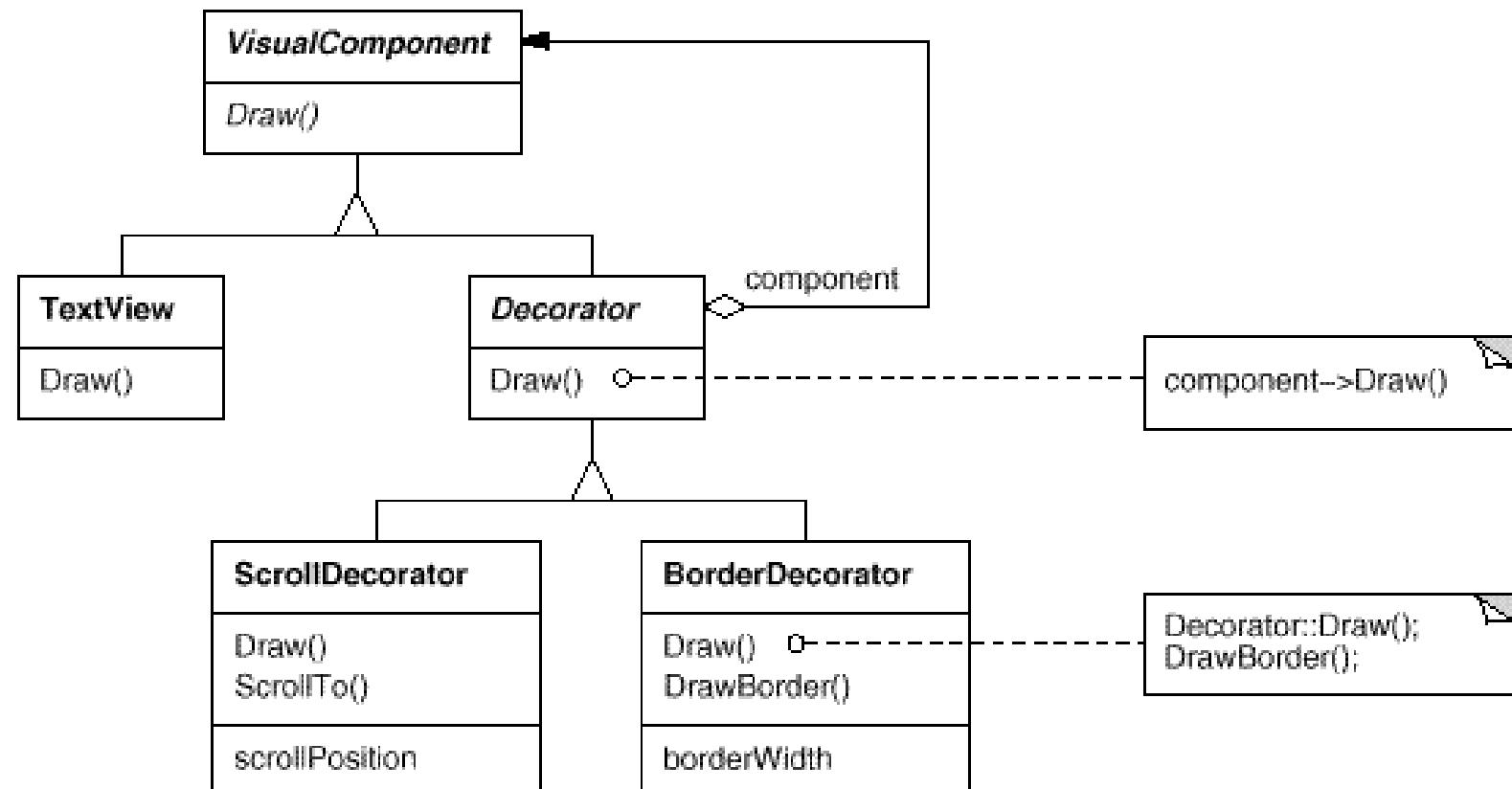
Example



Example



Example

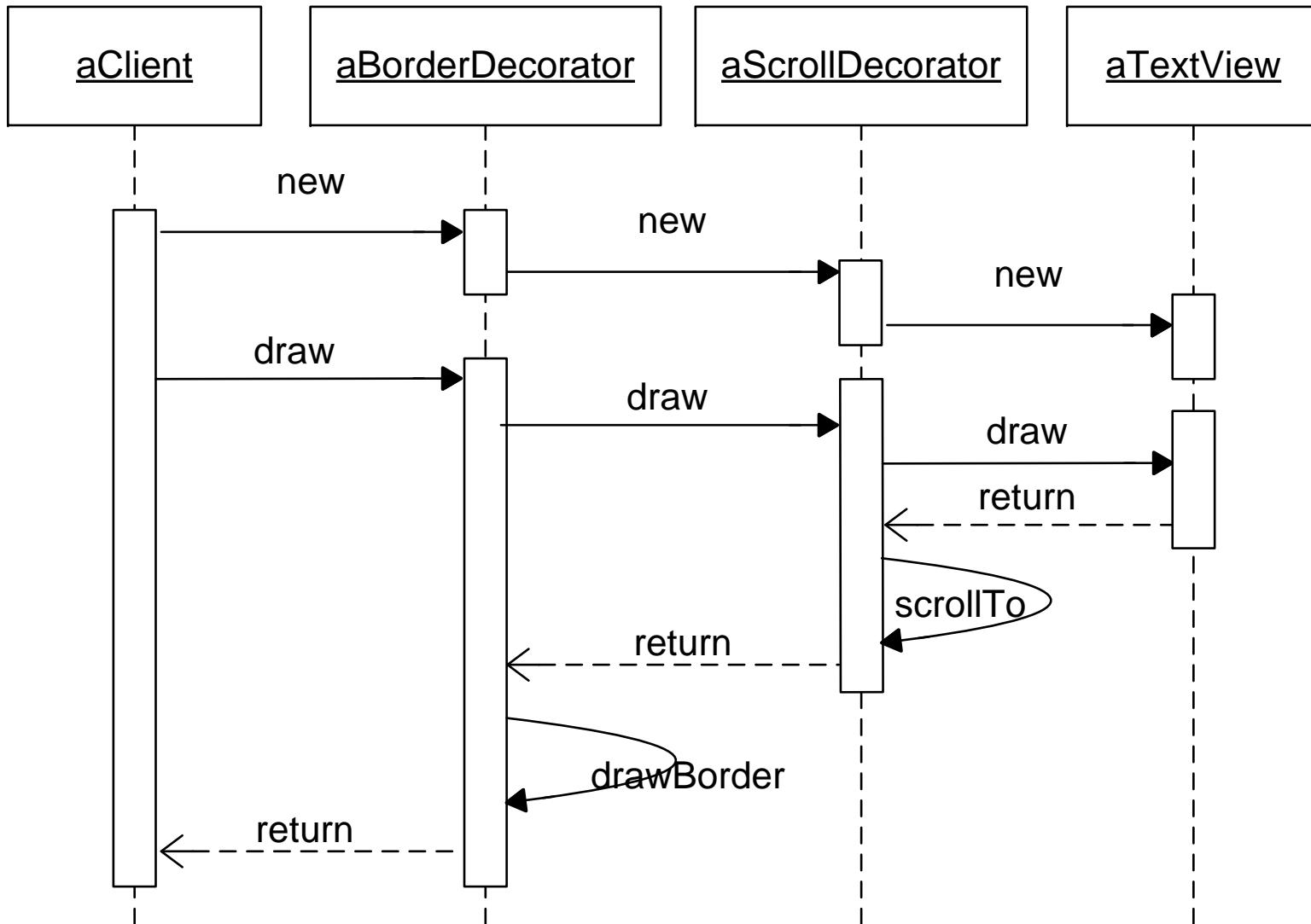


```
interface VisualComponent {  
    public void draw();  
}  
  
class TextView implements VisualComponent {  
    public void draw() {  
        // do something  
    }  
}  
  
abstract class Decorator implements VisualComponent {  
    protected VisualComponent component;  
    public Decorator(VisualComponent component) {  
        this.component = component;  
    }  
    public abstract void draw();  
}
```

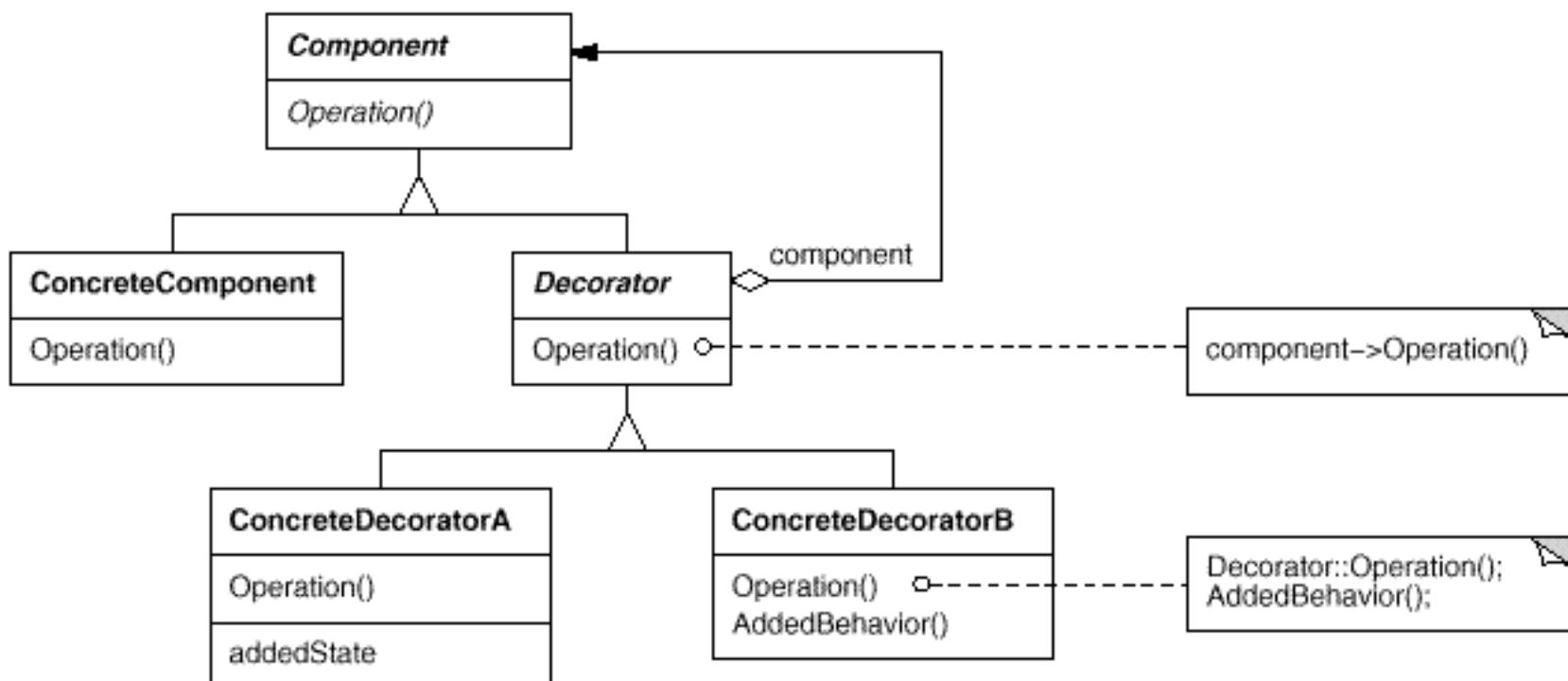
```
class ScrollDecorator extends Decorator {  
    public ScrollDecorator(VisualComponent component) {  
        super(component);  
    }  
    public void draw() {  
        component.draw();  
        scrollTo();  
    }  
    public void scrollTo() {  
        // do something  
    }  
}
```

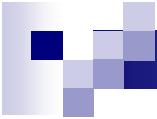
```
class BorderDecorator extends Decorator {  
    public BorderDecorator(VisualComponent component) {  
        super(component);  
    }  
    public void draw() {  
        component.draw();  
        drawBorder();  
    }  
    public void drawBorder() {  
        // do something  
    }  
}
```

```
class DecoratorClient {  
    public void decoratorClient() {  
        VisualComponent component =  
            new BorderDecorator(  
                new ScrollDecorator(  
                    new TextView()));  
  
        // VisualComponent border = new BorderDecorator();  
        // VisualComponent scroll = new ScrollDecorator();  
        // VisualComponent text = new TextView();  
        // border.setComponent(scroll);  
        // scroll.setComponent(text);  
  
        component.draw();  
    }  
}
```



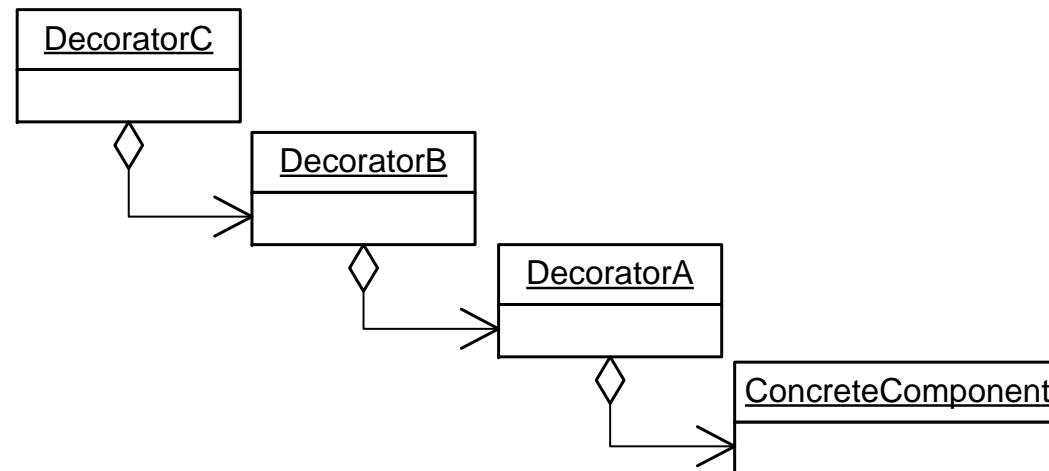
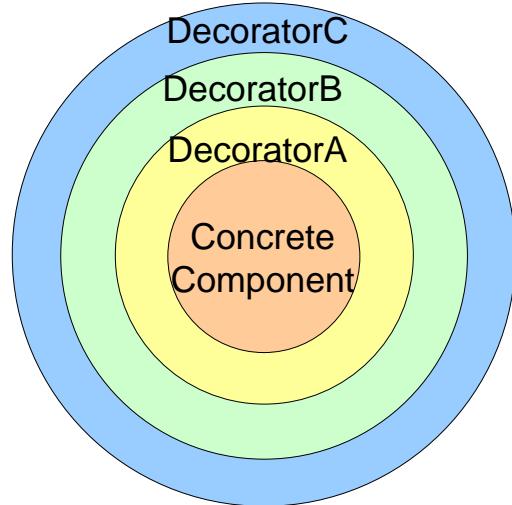
Structure



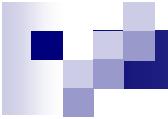


Participants

- **Component**: defines the interface for objects that can have responsibilities added to them dynamically.
 - **ConcreteComponent**: defines an object to which additional responsibilities can be attached.
 - **Decorator**: maintains a reference to a Component object and defines an interface that conforms to Component's interface.
 - **ConcreteDecorator**: adds responsibilities to the component.
-

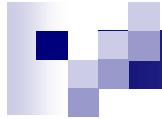


```
Component component =  
    new DecoratorC(  
        new DecoratorB(  
            new DecoratorA(  
                new ConcreteComponent() ) ) );
```



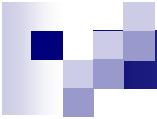
Collaborations

- **Decorator** forwards requests to its **Component** object. It may optionally perform additional operations before and after forwarding the request.
-



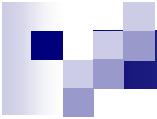
Consequences

- More flexibility than static inheritance.
 - With **Decorators**, responsibilities can be added and removed at run-time simply by attaching and detaching them. **Decorators** also make it easy to add a property twice. For example, to give a TextView a double border, simply attach two BorderDecorators.
 - Avoids feature-laden(过多特性的) classes high up in the hierarchy.
 - Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with **Decorator** objects. Functionality can be composed from simple pieces.
 - By permutation and combination, lots of behavioral combinations can be created.
-



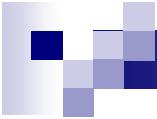
Consequences

- A **decorator** and its **component** aren't identical. shouldn't rely on object identity or true type when you use decorators.
 - A design that uses **Decorator** often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected.
 - Remove the **Decorator** from **component** is very difficult rather than recreation a new one.
-



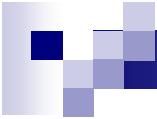
Applicability

- To add responsibilities to individual objects dynamically and transparently.
 - For responsibilities that can be withdrawn. (difficult to implement)
 - When extension by subclassing is impractical.
 - Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or
 - A class definition may be hidden or otherwise unavailable for subclassing.
-



Implementation

- Interface conformance.
 - A decorator object's interface must conform to the interface of the component it decorates
 - Keeping **Component** classes lightweight.
 - To ensure a conforming interface, components and decorators must inherit from a common **Component** class.
 - The complexity of the **Component** class might make the decorators too heavyweight to used.
 - Putting a lot of functionality into **Component** also increases the probability that concrete subclasses will pay for features they don't need.
-



Think about it

- **Changing the skin** of an object versus **changing its guts**. We can think of a decorator as a skin over an object that changes its behavior. An alternative is to change the object's guts.
 - The **Strategy pattern** is a good example of a pattern for changing the guts.
-

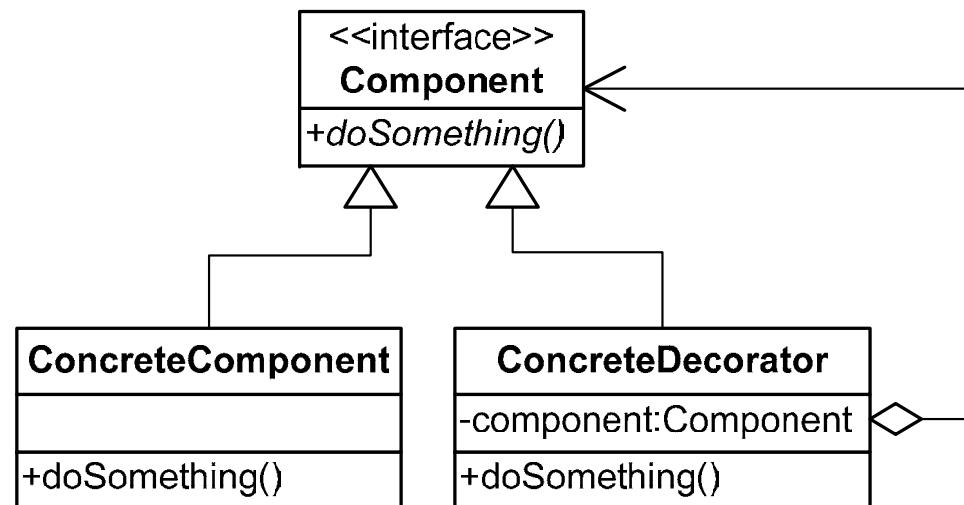
Example

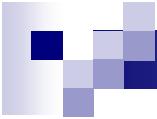
■ Ticket of an supermarket



Variation: Decorator is omitted

- It is unnecessary to providing a **Decorator** if there is only one **ConcreteDecorator** to decorate the **Component**.





Extension: Semi-transportation of Decorator

- Transparent decorator pattern: the transparency of decorator pattern requires the **ConcreteDecorator** do not contains the public methods which are not defined in **Component**, or clients do not required such methods. (DIP, 针对接口编程)
 - Decorator pattern could **semi-transportation**. The intent of decorator pattern is adding behaviors dynamically without modifying the interface and introducing the subclasses. But sometimes the public method is defined in **ConcreteDecorator** when new behaviors are introduced.
-

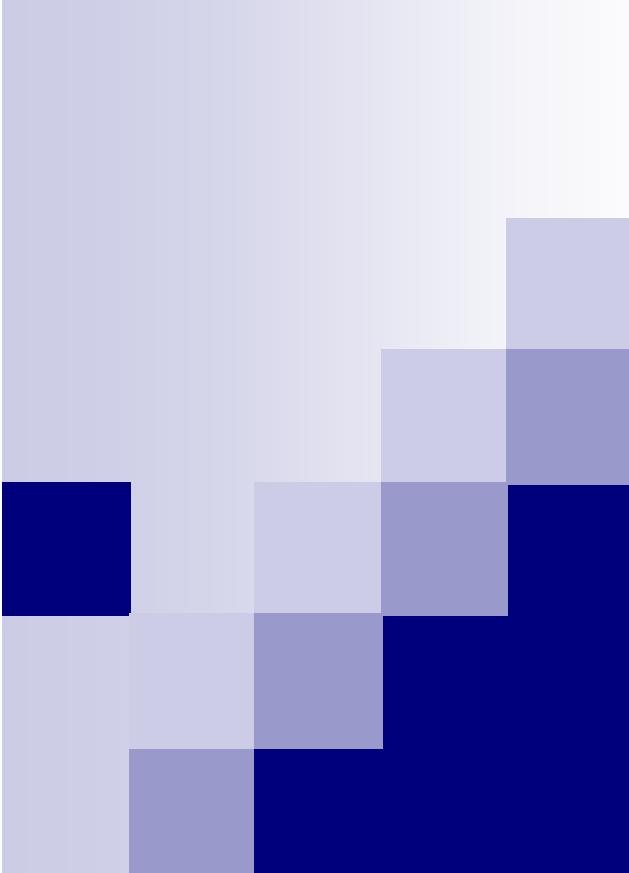
Extension: Semi-transportation of Decorator

```
class DecoratorClient {  
    public void decoratorClient() {  
        VisualComponent component =  
            new BorderDecorator(  
                new ScrollDecorator(  
                    new TextView())));  
  
        component.draw();  
        BorderDecorator borderComponent  
            = (BorderDecorator) component;  
        borderComponent.drawBorder();  
    }  
}
```



Let's go to next...





Design Patterns

宋 杰

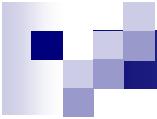
Song Jie

东北大学 软件学院

Software College, Northeastern
University



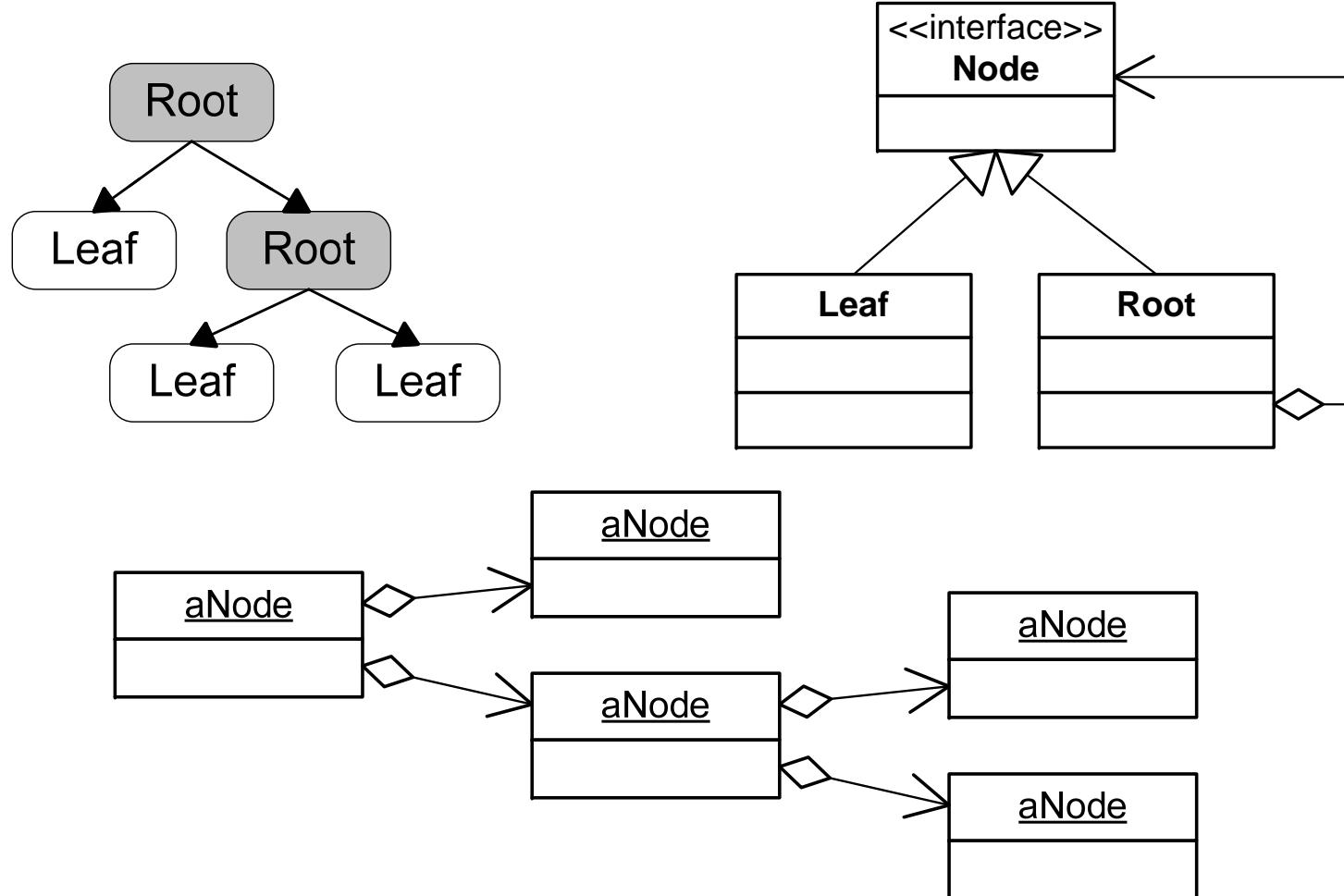
8. Composite Pattern



Intent

- Compose objects into tree structures to represent **part-whole** hierarchies. Composite lets clients **treat individual** objects and **compositions of objects uniformly**.
- 将对象组合成树形结构以表示“**部分-整体**”的层次结构。合成模式使得用户对**单个对象**(叶子节点, 单纯元素)和**组合对象**(根节点, 复合元素)的使用具有一致性。

Example



Version 1-1

```
interface Node {  
    public void operation();  
}  
  
class Leaf implements Node {  
  
    public void operation() {  
        // do Leaf's operation  
    }  
}  
  
class NodeClient {  
    public Node createTree() {  
        Root rootA = new Root();  
        Root rootB = new Root();  
        Leaf leafA = new Leaf();  
        Leaf leafB = new Leaf();  
        Leaf leafC = new Leaf();  
        rootA.addSubNode(rootB);  
        rootA.addSubNode(leafA);  
        rootB.addSubNode(leafB);  
        rootB.addSubNode(leafC);  
        return rootA;  
    }  
}
```

```
class Root implements Node {  
    private List<Node> nodeList;  
    public Root() {  
        nodeList = new ArrayList<Node>();  
    }  
    public void operation() {  
        // pre-operations here  
        for (Node node : nodeList) {  
            node.operation();  
        }  
        // post-operations here  
    }  
    public void addSubNode(Node node) {  
        nodeList.add(node);  
    }  
    public Node removeSubNodeByIndex(int index) {  
        return nodeList.remove(index);  
    }  
    public void clearSubNodes() {  
        nodeList.clear();  
    }  
    public int getSubNodesSize() {  
        return nodeList.size();  
    }  
    public Node getSubNodeByIndex(int index) {  
        return nodeList.get(index);  
    }  
}
```

Version 1-2

```
class NodeClient {  
    public Node createTree() {  
        Root rootA = new Root();  
        Root rootB = new Root();  
        Leaf leafA = new Leaf();  
        Leaf leafB = new Leaf();  
        Leaf leafC = new Leaf();  
        rootA.addSubNode(rootB)  
            .addSubNode(leafA);  
        rootB.addSubNode(leafB)  
            .addSubNode(leafC);  
        return rootA;  
    }  
}
```

```
class Root implements Node, Iterable<Node> {  
    private List<Node> nodeList;  
    public Root() {  
        nodeList = new ArrayList<Node>();  
    }  
    public void operation() {  
        // pre-operations here  
        for (Node node : this) {  
            node.operation();  
        }  
        // post-operations here  
    }  
    public Iterator<Node> iterator() {  
        return nodeList.iterator();  
    }  
    public Root addSubNode(Node node) {  
        nodeList.add(node);  
        return this;  
    }  
    public Node removeSubNodeByIndex(int index) {  
        return nodeList.remove(index);  
    }  
    public void clearSubNodes() {  
        nodeList.clear();  
    }  
}
```

Version 2

```
interface Node extends Iterable<Node> {
    public void operation();
    public Node addSubNode(Node node);
    public Node removeSubNodeByIndex(int index);
    public void clearSubNodes();
}

class Leaf implements Node {
    public void operation() {
        // do Leaf's operation
    }
    public Iterator<Node> iterator() {
        return new ArrayList<Node>().iterator();
    }
    public Node addSubNode(Node node) {
        return null;
    }
    public void clearSubNodes() {
    }
    public Node removeSubNodeByIndex(int index) {
        return null;
    }
}
```

Version 2

```
class NodeClient {  
    public Node createTree() {  
        Node rootA = new Root();  
        Node rootB = new Root();  
        Node leafA = new Leaf();  
        Node leafB = new Leaf();  
        Node leafC = new Leaf();  
        rootA.addSubNode(rootB)  
            .addSubNode(leafA);  
        rootB.addSubNode(leafB)  
            .addSubNode(leafC);  
        return rootA;  
    }  
}
```

```
class Root implements Node {  
    private List<Node> nodeList;  
    public Root() {  
        nodeList = new ArrayList<Node>();  
    }  
    public void operation() {  
        // pre-operations here  
        for (Node node : this) {  
            node.operation();  
        }  
        // post-operations here  
    }  
    public Iterator<Node> iterator() {  
        return nodeList.iterator();  
    }  
    public Node addSubNode(Node node) {  
        nodeList.add(node);  
        return this;  
    }  
    public Node removeSubNodeByIndex(int index) {  
        return nodeList.remove(index);  
    }  
    public void clearSubNodes() {  
        nodeList.clear();  
    }  
}
```

Version 3

```
abstract class Node implements Iterable<Node> {
    protected Node parentNode;
    public void addParentNode(Node parentNode) {
        this.parentNode = parentNode;
    }
    public Node getParentNode() {
        return parentNode;
    }
    public void removeParentNode() {
        parentNode = null;
    }
    public abstract Node addSubNode(Node node);
    public abstract Node removeSubNodeByIndex(int index);
    public abstract void clearSubNodes();
    public abstract Iterator<Node> iterator();
    public abstract void operation();
}
```

Version 3

```
class Leaf extends Node {  
    public Node addSubNode(Node node) {  
        return null;  
    }  
    public void clearSubNodes() {  
    }  
    public Node removeSubNodeByIndex(int index) {  
        return null;  
    }  
    public Iterator<Node> iterator() {  
        return new ArrayList<Node>().iterator();  
    }  
    public void operation() {  
        // do Leaf's operation  
    }  
}
```

Version 3

```
class Root extends Node {  
    private List<Node> nodeList;  
    public Root() {  
        this.nodeList = new ArrayList<Node>();  
    }  
    public void operation() {  
        // do Root's operation  
    }  
    public Iterator<Node> iterator() {  
        return nodeList.iterator();  
    }  
    public Root addSubNode(Node node) {  
        nodeList.add(node);  
        return this;  
    }  
    public Node removeSubNodeByIndex(int index) {  
        return nodeList.remove(index);  
    }  
    public void clearSubNodes() {  
        nodeList.clear();  
    }  
}
```

Version 3

```
class NodeClient {
    public Node createTree() {
        Node rootA = new Root();
        Node rootB = new Root();
        Node leafA = new Leaf();
        Node leafB = new Leaf();
        Node leafC = new Leaf();

        rootB.addParentNode(rootA);
        leafA.addParentNode(rootA);
        leafB.addParentNode(rootB);
        leafC.addParentNode(rootB);

        rootA.addSubNode(rootB).addSubNode(leafA);
        rootB.addSubNode(leafB).addSubNode(leafC);

        return rootA;
    }

    public void ModifyTree() {
        Node tree = createTree();
        Node leafA = tree.removeSubNodeByIndex(0);
        leafA.removeParentNode();
    }
}
```

Version 4

```
abstract class Node implements Iterable<Node> {
    protected Node parentNode;
    protected void addparentNode(Node parentNode) {
        this.parentNode = parentNode;
    }
    protected void removeparentNode() {
        parentNode = null;
    }
    public Node getParentNode() {
        return parentNode;
    }

    public abstract Node addSubNode(Node node);
    public abstract Node removeSubNodeByIndex(int index);
    public abstract void clearSubNodes();
    public abstract Iterator<Node> iterator();
    public abstract void operation();
}
```

Version 4

```
class Leaf extends Node {  
    public Node addSubNode(Node node) {  
        return null;  
    }  
    public void clearSubNodes() {  
    }  
    public Node removeSubNodeByIndex(int index) {  
        return null;  
    }  
    public Iterator<Node> iterator() {  
        return new ArrayList<Node>().iterator();  
    }  
    public void operation() {  
        // do Leaf's operation  
    }  
}
```

Version 4

```
class Root extends Node {  
    private List<Node> nodeList;  
    public Root() {  
        this.nodeList = new ArrayList<Node>();  
    }  
    public Node addSubNode(Node node) {  
        nodeList.add(node);  
        node.addParentNode(this);  
        return this;  
    }  
    public Node removeSubNodeByIndex(int index) {  
        Node node = nodeList.remove(index);  
        node.removeParentNode();  
        return node;  
    }  
    public void clearSubNodes() {  
        for (Node node : this) {  
            node.removeParentNode();  
        }  
        nodeList.clear();  
    }  
}
```

```
public void operation() {  
    // do Root's operation  
}  
public Iterator<Node> iterator() {  
    return nodeList.iterator();  
}
```

Version 4

```
class NodeClient {  
    public Node createTree() {  
        Node rootA = new Root();  
        Node rootB = new Root();  
        Node leafA = new Leaf();  
        Node leafB = new Leaf();  
        Node leafC = new Leaf();  
          
        rootA.addSubNode(rootB).addSubNode(leafA);  
        rootB.addSubNode(leafB).addSubNode(leafC);  
  
        return rootA;  
    }  
  
    public void ModifyTree() {  
        Node tree = createTree();  
        tree.removeSubNodeByIndex(0);  
          
    }  
}
```

Version 5

```
interface Node extends Iterable<Node> {  
    public boolean isLeaf();  
  
    public void addParentNode(Node parentNode);  
    public void removeParentNode();  
  
    public Node getParentNode();  
    public Node addSubNode(Node node);  
    public Node removeSubNodeByIndex(int index);  
    public void clearSubNodes();  
  
    public Iterator<Node> iterator();  
    public void operation();  
}
```

Version 5

```
class NodeImpl implements Node {  
    private List<Node> nodeList;  
    private Node parentNode;  
    private boolean leaf;  
  
    public NodeImpl() {  
        this.nodeList = new ArrayList<Node>();  
    }  
    public NodeImpl(boolean isLeaf) {  
        this();  
        this.leaf = isLeaf;  
    }  
  
    public boolean isLeaf() {  
        return leaf;  
    }  
    public void operation() {  
        if (leaf) {  
            // do Leaf's operation  
        } else {  
            // do Root's operation  
        }  
    }  
    public Iterator<Node> iterator() {  
        return nodeList.iterator();  
    }  
}
```

```
public void addParentNode(Node parentNode) {  
    this.parentNode = parentNode;  
}  
public void removeParentNode() {  
    parentNode = null;  
}  
public Node getParentNode() {  
    return parentNode;  
}  
public Node addSubNode(Node node) {  
    if(leaf) {return null;}  
    nodeList.add(node);  
    node.addParentNode(this);  
    return this;  
}  
public Node removeSubNodeByIndex(int index) {  
    if(leaf) {return null;}  
    Node node = nodeList.remove(index);  
    node.removeParentNode();  
    return node;  
}  
public void clearSubNodes() {  
    if(leaf) {return;}  
    for (Node node : this) {  
        node.removeParentNode();  
    }  
    nodeList.clear();  
}
```

Version 5

```
class NodeClient {
    public Node createTree() {
        Node rootA = new NodeImpl(false);
        Node rootB = new NodeImpl(false);
        Node leafA = new NodeImpl(true);
        Node leafB = new NodeImpl(true);
        Node leafC = new NodeImpl(true);

        rootA.addSubNode(rootB)
            .addSubNode(leafA);
        rootB.addSubNode(leafB)
            .addSubNode(leafC);

        return rootA;
    }

    public void ModifyTree() {
        Node tree = createTree();
        tree.removeSubNodeByIndex(0);
    }
}
```

Version 6

```
interface Node extends Iterable<Node> {  
    public boolean isRoot();  
    public void addParentNode(Node parentNode);  
    public void removeParentNode();  
    public Node getParentNode();  
    public Node addSubNode(Node node);  
    public Node removeSubNodeByIndex(int index);  
    public void clearSubNodes();  
    public Iterator<Node> iterator();  
    public void operation();  
}
```

Version 6

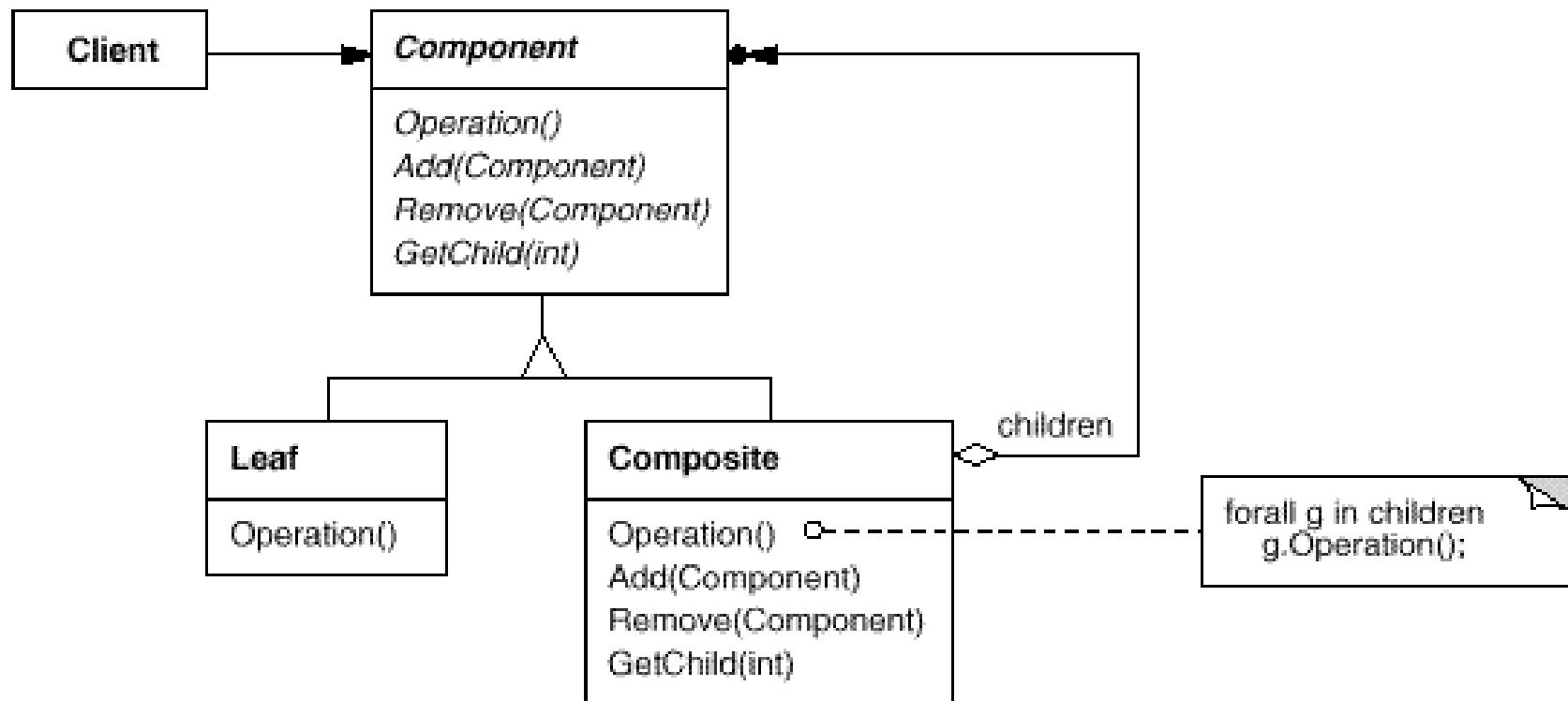
```
class NodeImpl implements Node {  
    private List<Node> nodeList;  
    private Node parentNode;  
    private boolean root;  
    public NodeImpl() {  
        this.nodeList = new ArrayList<Node>();  
    }  
    public boolean isRoot() {  
        return root;  
    }  
    public void operation() {  
        if (root) {  
            // do Root's operation  
        } else {  
            // do Root's operation  
        }  
    }  
    public Iterator<Node> iterator() {  
        return nodeList.iterator();  
    }  
    public void addParentNode(Node parentNode) {  
        this.parentNode = parentNode;  
    }
```

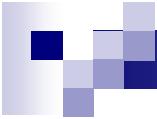
```
    public void removeParentNode() {  
        parentNode = null;  
    }  
    public Node getParentNode() {  
        return parentNode;  
    }  
    public Node addSubNode(Node node) {  
        root = true;  
        nodeList.add(node);  
        node.addParentNode(this);  
        return this;  
    }  
    public Node removeSubNodeByIndex(int index) {  
        root = true;  
        Node node = nodeList.remove(index);  
        node.removeParentNode();  
        return node;  
    }  
    public void clearSubNodes() {  
        root = true;  
        for (Node node : this) {  
            node.removeParentNode();  
        }  
        nodeList.clear();  
    }
```

Version 6

```
class NodeClient {  
    public Node createTree() {  
        Node rootA = new NodeImpl();  
        Node rootB = new NodeImpl();  
        Node leafA = new NodeImpl();  
        Node leafB = new NodeImpl();  
        Node leafC = new NodeImpl();  
  
        rootA.addSubNode(rootB).addSubNode(leafA);  
        rootB.addSubNode(leafB).addSubNode(leafC);  
  
        return rootA;  
    }  
  
    public void ModifyTree() {  
        Node tree = createTree();  
        tree.removeSubNodeByIndex(0);  
    }  
}
```

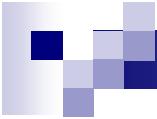
Structure





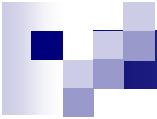
Participants

- **Component**
 - Declares the interface for objects in the composition.
 - Implements default behavior for the interface common to all classes, as appropriate.
 - Declares an interface for accessing and managing its child components.
 - (*Optional*) Defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
 - **Leaf**
 - Represents leaf objects in the composition. A leaf has no children.
 - Defines behavior for primitive objects in the composition.
 - **Composite**
 - Defines behavior for components having children.
 - Stores child components.
 - Implements child-related operations in the **Component** interface.
 - **Client**
 - Manipulates objects in the composition through the **Component** interface.
-



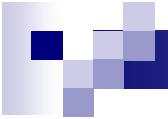
Collaborations

- Clients use the **Component** to interact with objects in the composite structure.
 - If the recipient is a **Leaf**, then the request is handled directly.
 - If the recipient is a **Composite**, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.
-



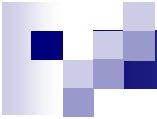
Consequences

- Defines class hierarchies consisting of **primitive objects** and **composite objects**.
- Makes the client simple.
 - Clients can treat composite structures and individual objects uniformly.
 - Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite component.
- Makes it easier to add new kinds of components.
 - Newly defined **Composite** or **Leaf** subclasses work automatically with existing structures and client code.
 - Clients don't have to be changed for new **Component** classes.



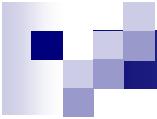
Consequences

- Can make your design overly general.
 - The disadvantage that it is harder to restrict the components of a composite.
 - Sometimes you want a composite to have only certain components. With **Composite**, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.
-



Applicability

- You want to represent part-whole hierarchies of objects.
 - You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.
-

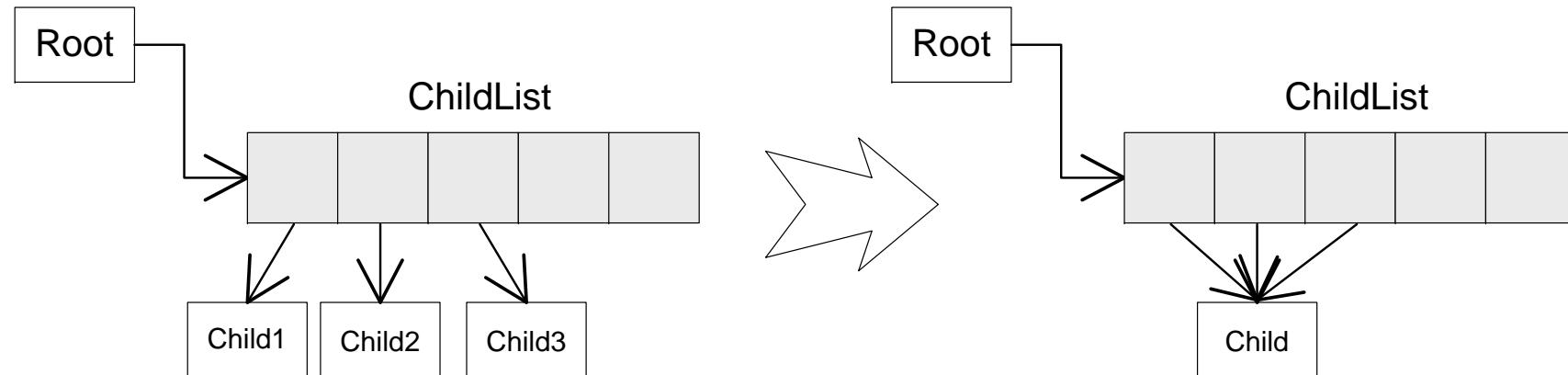


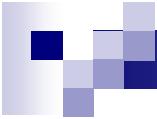
Implementation 1: Explicit parent references (bi-direction reference)

- Maintaining references from child components to their parent can simplify the traversal and management of a composite structure.
 - The usual place to define the parent reference is in the **Component** class. **Leaf** and **Composite** classes can inherit the reference and the operations that manage it.
- It is unnecessary to let clients maintain bi-directions. Usually parent-to-children references are maintained by clients, child-to-parent reference are maintained inside **composite pattern** automatically
 - The easiest way to ensure this is to change a component's parent only when it's being added or removed from a composite. If this can be implemented once in the **Add** and **Remove** operations of the **Composite** class.

Implementation 2: Sharing components

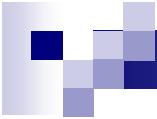
- It's often useful to share components, for example, to reduce storage requirements.
 - The component must be stateless or sharable state.





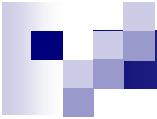
Implementation 3: Maximizing the Component interface

- Composite pattern makes clients unaware of the specific **Leaf** or **Composite** classes they're using.
 - **Component** class should define as many common operations for **Composite** and **Leaf** classes as possible.
 - There are many operations that **Component** supports that don't seem to make sense for **Leaf** classes. So that it conflict with Interface Segregation Principle (ISP)
 - How can **Leaf** provide a default implementation for them?
 - Make the useless operations, do nothing, or return null, or return mock object, or throws exception
 - The child management operations are more troublesome and are discussed in the next item.
-



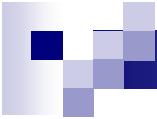
Implementation 4: Declaring the child management operations

- Should we declare “*child management operations*” in the **Component** and make them meaningful for **Leaf** classes, or should we declare and define them only in **Composite** and its subclasses?
- The decision involves a trade-off between safety and transparency:
 - **Transparency**: Defining the child management interface at the root of the class hierarchy gives you transparency, because you can treat all components uniformly. It costs you safety, however, because clients may try to do meaningless things like add and remove objects from leaves.
 - **Safety** : Defining child management in the **Composite** class gives you safety, because any attempt to add or remove objects from leaves will be caught at compile-time in a statically typed language. But you lose transparency, because leaves and composites have different interfaces.



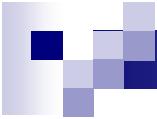
Implementation 5: Heavy component or light component operations

- Transparency solution
 - The heavy **Component** is suggested because it can stand for the operations of both **Leaf** and **Composite**
 - Safety solution
 - The light **Component** is suggested to let it only stand for the common operations of both **Leaf** and **Composite**
-



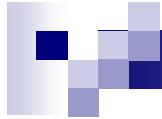
Implementation 6: Child ordering

- Many designs specify an ordering on the children of **Composite**.
 - When child ordering is an issue, you must design child access and management interfaces carefully to manage the sequence of children.
-



Implementation 7: Caching to improve performance

- If you need to traverse or search compositions frequently, the **Composite** class can cache traversal or search information about its children.
 - Changes to a component will require invalidating the caches of its parents.
 - This works best when components know their parents.
 - So if you're using caching, you need to define an interface for telling composites that their caches are invalid.
-

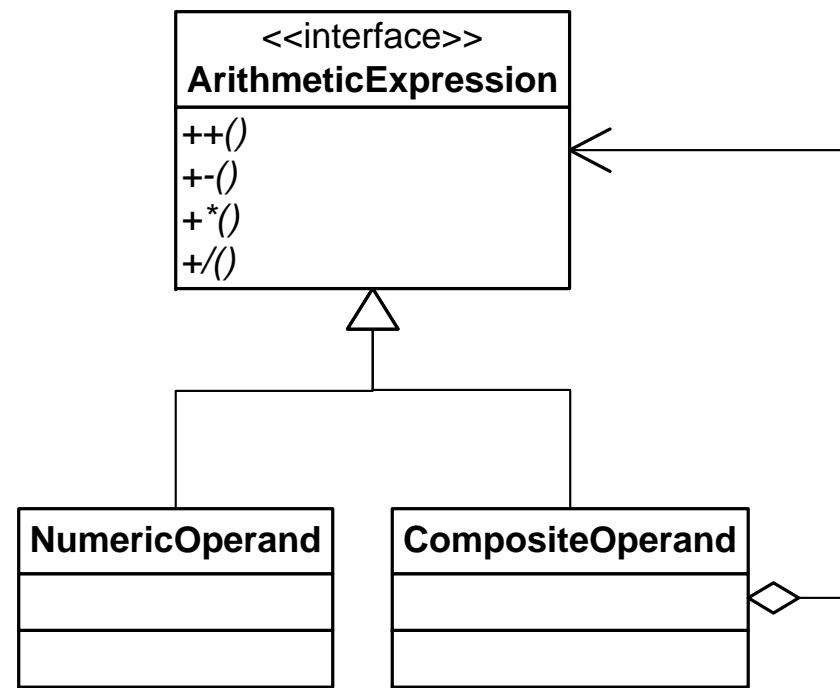
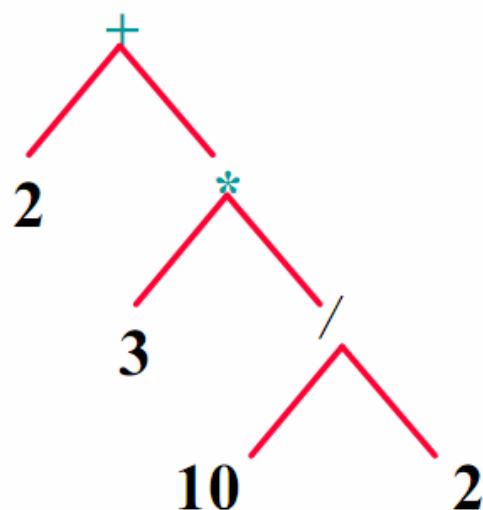


Implementation 8: What's the best data structure for storing components?

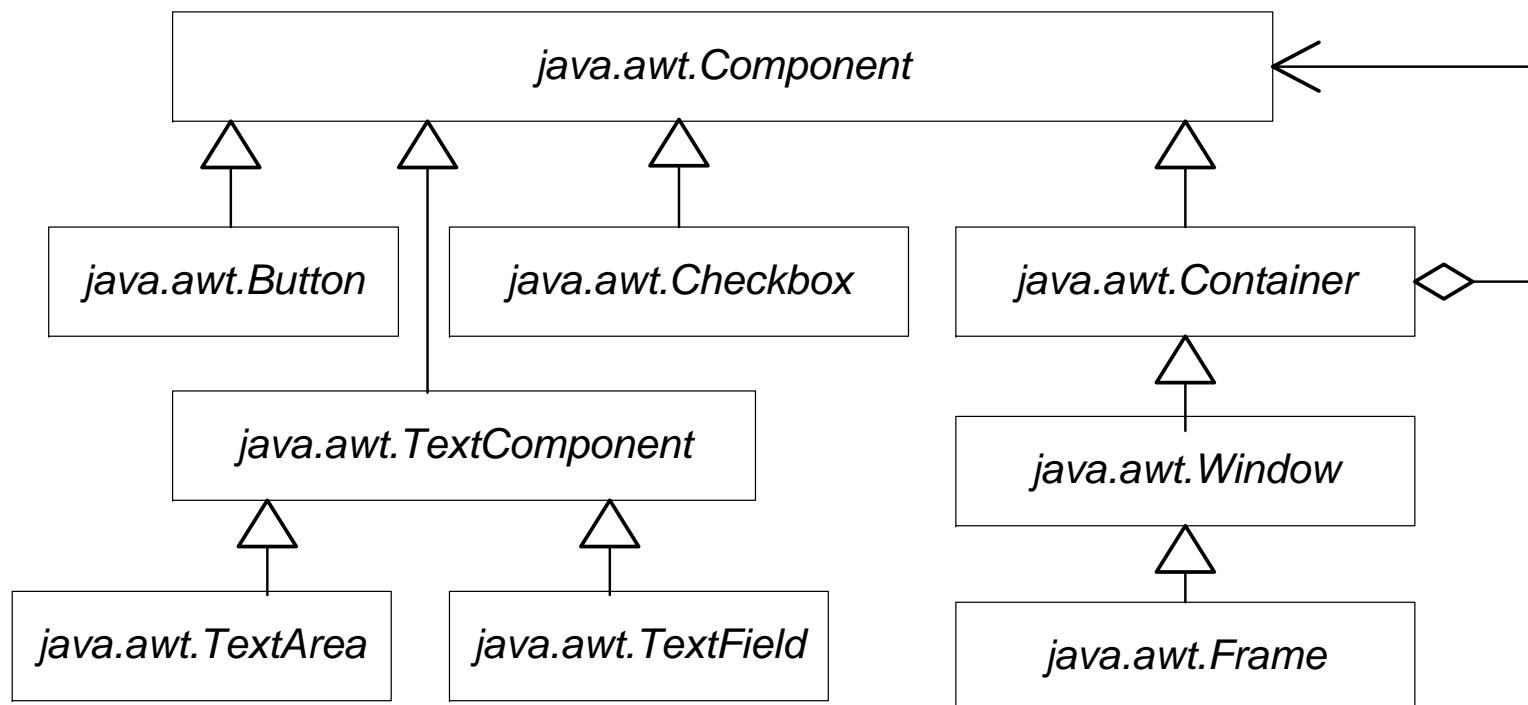
- Composites may use a variety of data structures to store their children,
 - Arrays, List, Set, HashMap
 - The choice of data structure depends (as always) on efficiency.
 - Sometimes composites have a variable for each child (limited quantity).
 - binary tree: left and right
-

Example 1: Arithmetic expressions

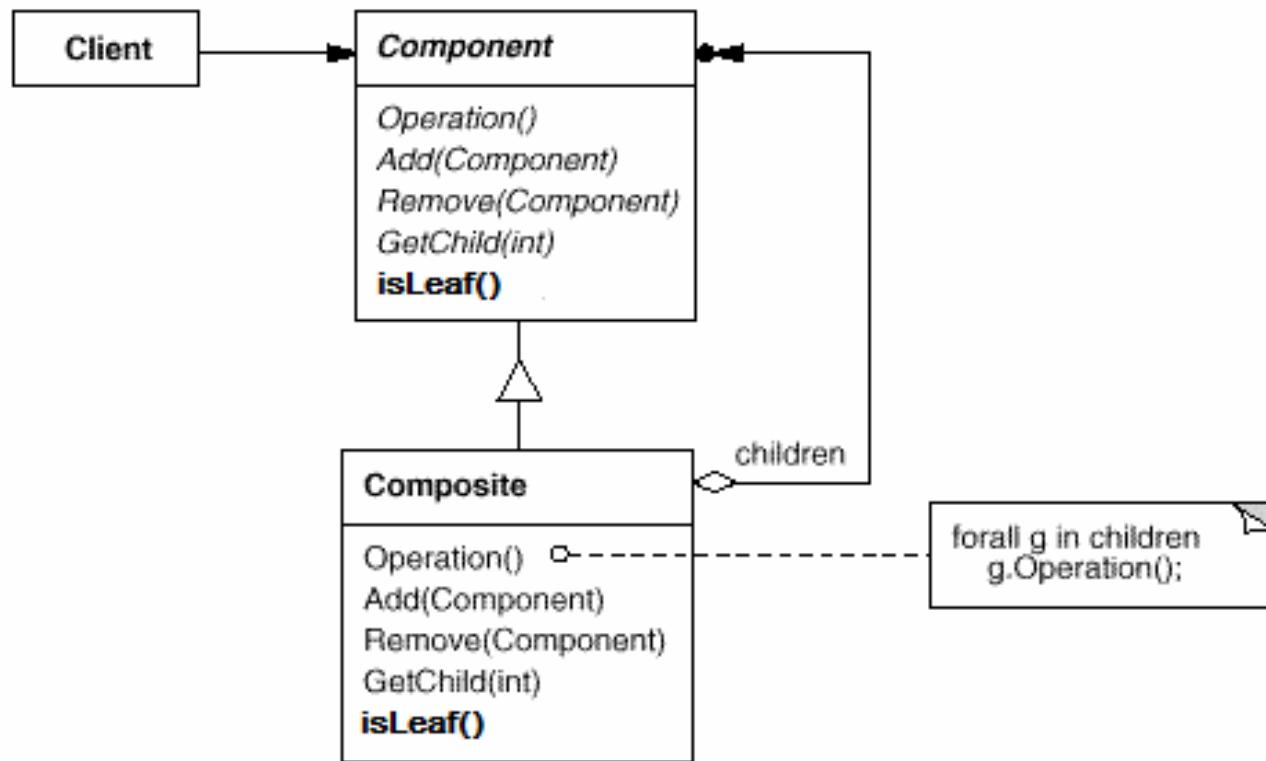
- Arithmetic expressions can be expressed as trees where an operand can be a number or an arithmetic expression.



Example 2: Java AWT

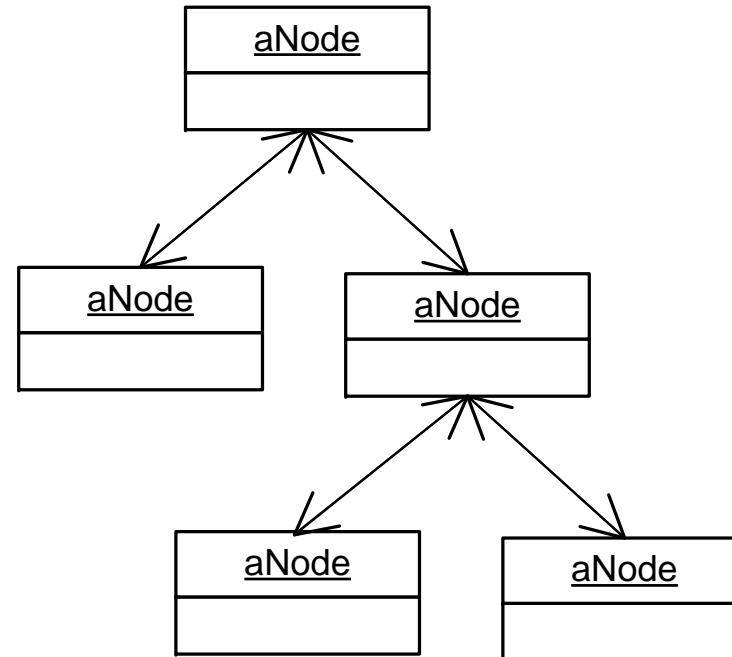


Variation: Leaf and composite be one class



Extension: Directions of the tree structure

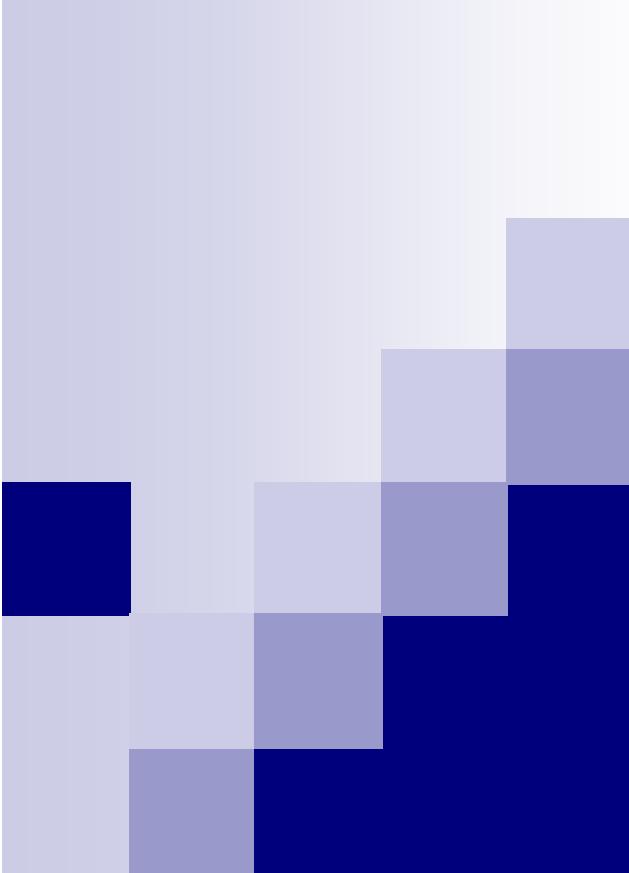
- Top-down tree
- Bottom-up tree
- Bi-directions tree





Let's go to next...





Design Patterns

宋 杰

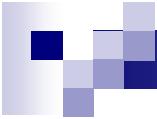
Song Jie

东北大学 软件学院

Software College, Northeastern
University



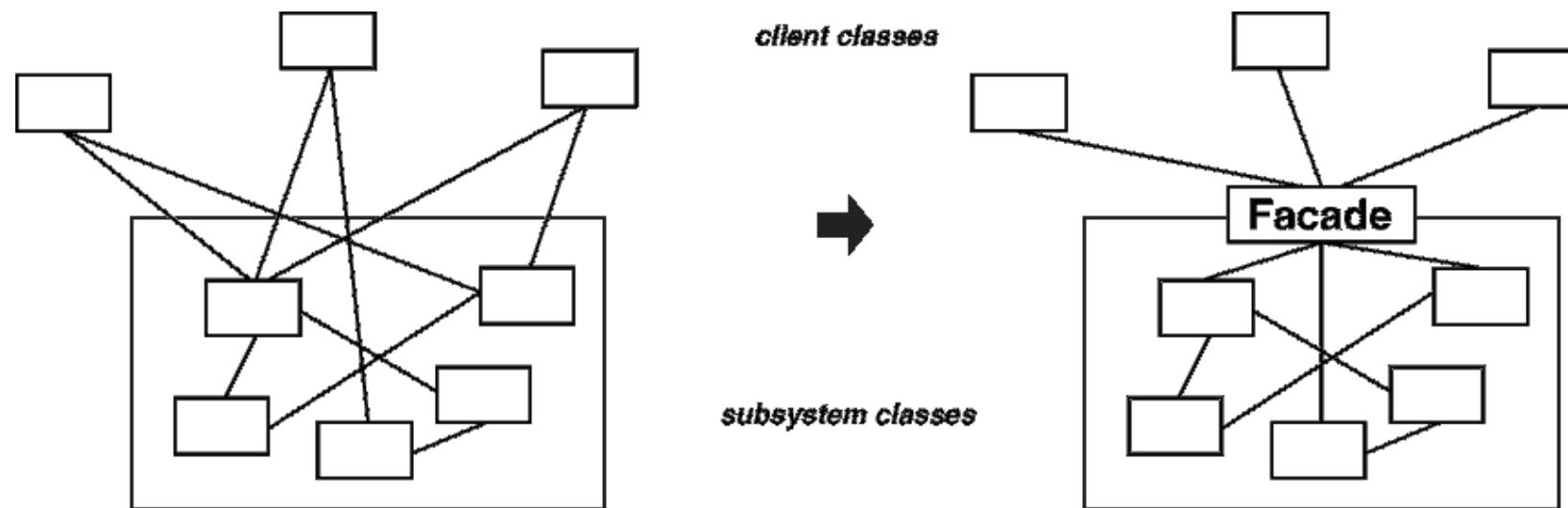
9. Facade Pattern



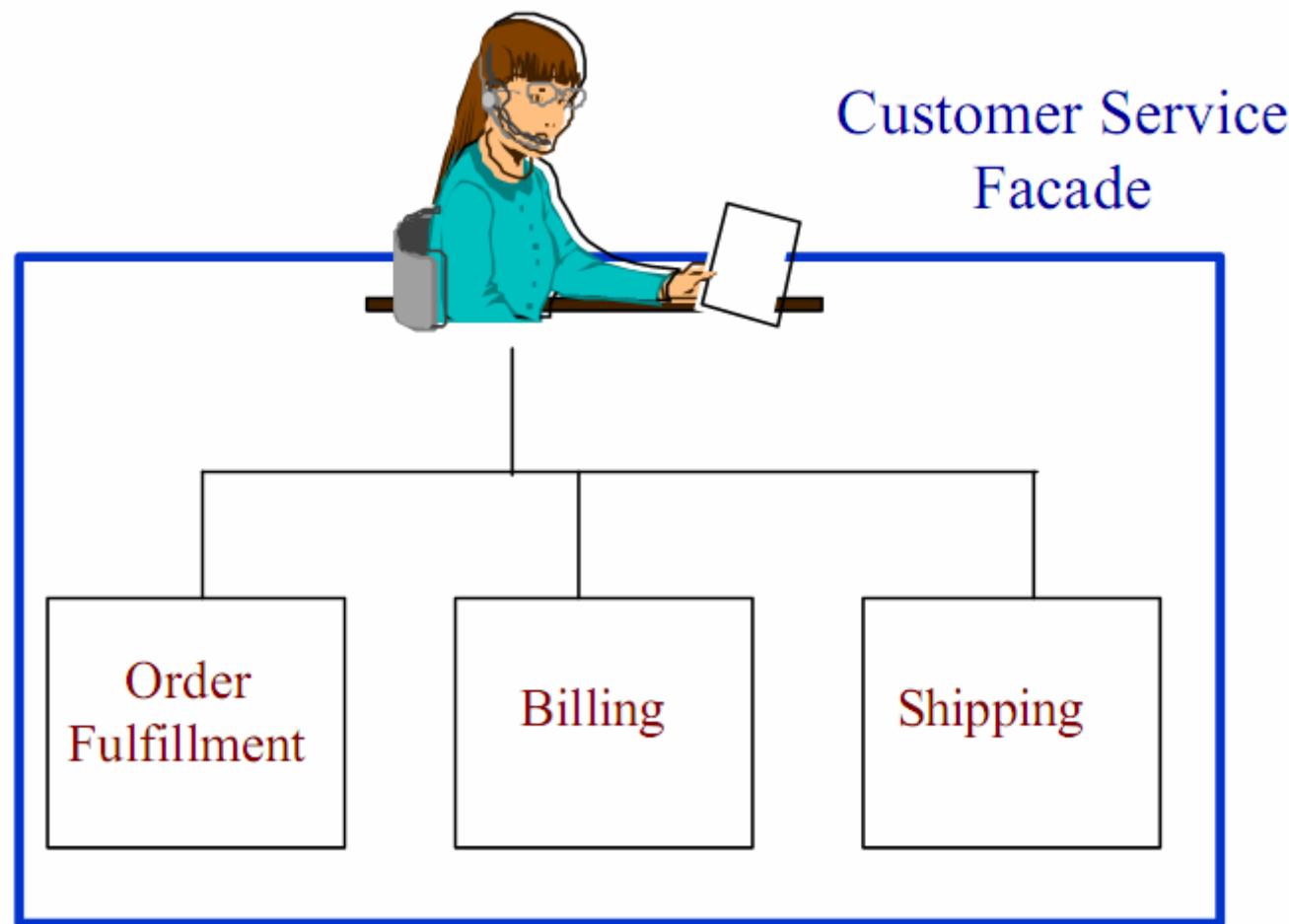
Intent

- Provide a **unified interface** to a set of interfaces in a **subsystem**. Facade defines a higher-level interface that makes the subsystem easier to use.
 - 门面模式或外观模式
-

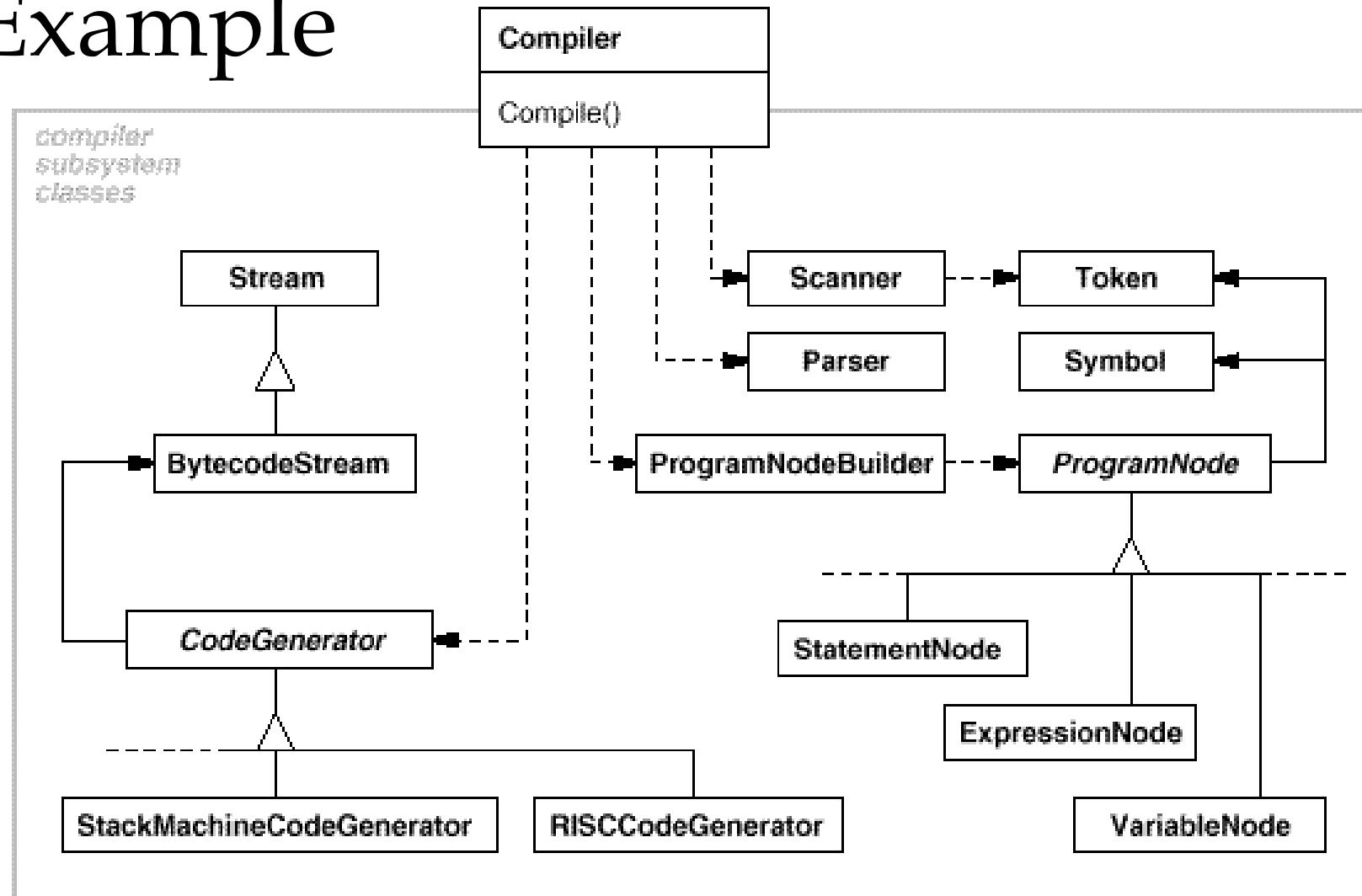
- Structuring a system into subsystems helps reduce complexity.
- A common design goal is to minimize the communication and dependencies between subsystems.
- One way to achieve this goal is to introduce a **Facade object** that provides a single, simplified interface to the more general facilities of a subsystem.



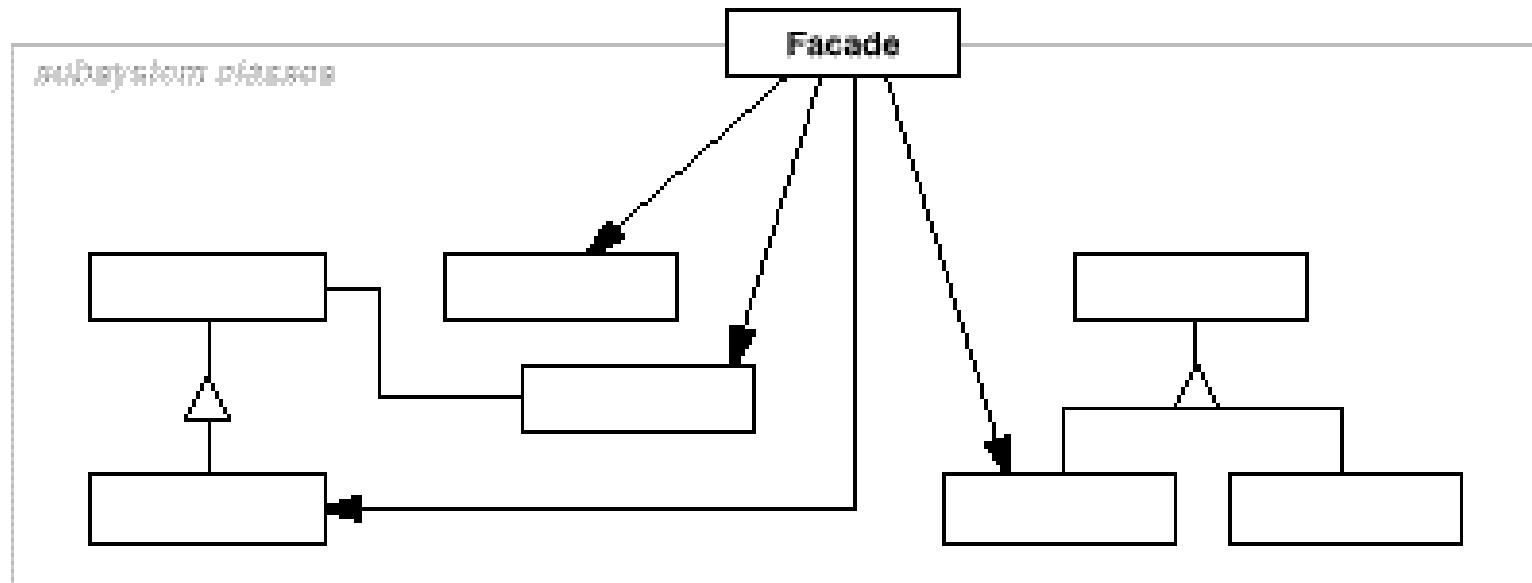
Example

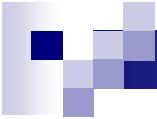


Example



Structure





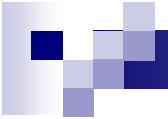
Participants

■ Facade

- Knows which subsystem classes are responsible for a request.
- Delegates client requests to appropriate subsystem objects.

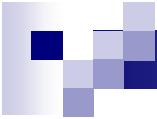
■ Subsystem classes

- Implement subsystem functionality.
 - Handle work assigned by the **Facade** object.
 - Have no knowledge of the facade; that is, they keep no references to it.
-



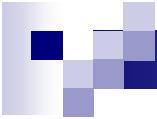
Consequences

- It shields clients from subsystem components.
 - It promotes weak coupling between the subsystem and its clients.
 - It **doesn't** prevent applications from using subsystem classes if they need to.
-



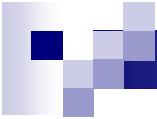
Applicability

- You want to provide a simple interface to a complex subsystem.
 - Subsystems often get more complex as they evolve(演化).
 - Most patterns, when applied, result in more and smaller classes.
 - This makes the subsystem more reusable and easier to customize,
 - But it also becomes harder to use for clients that don't need to customize it.
 - A facade can provide a simple default view of the subsystem that is good enough for most clients.
 - Only clients needing more customizability will need to look beyond the facade.



Applicability

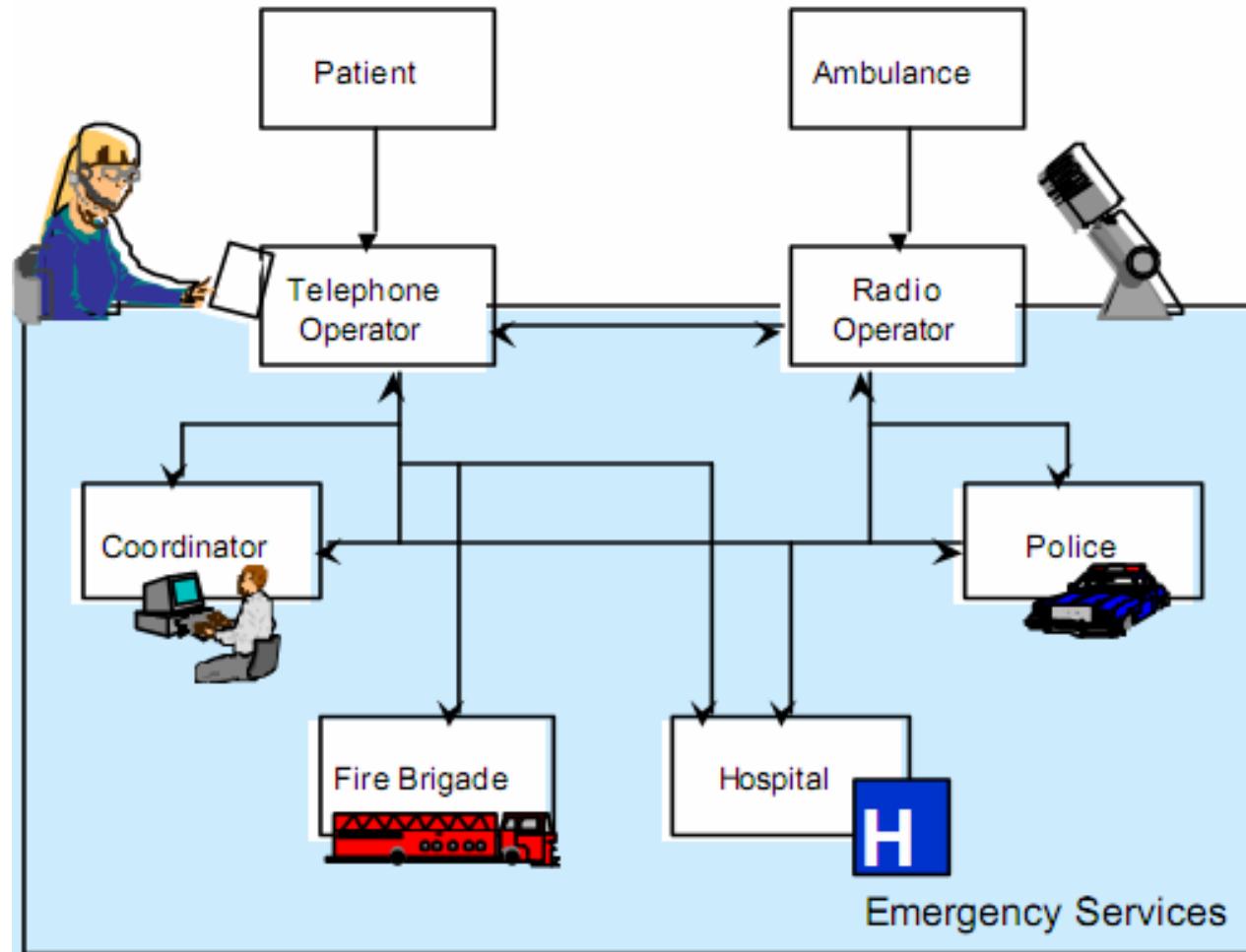
- There are many dependencies between **clients** and **implementation**.
 - Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.
 - You want to layer your subsystems.
 - Use a facade to define an entry point to each subsystem level, simplify the dependencies between them by making them communicate with each other solely through their facades.
-

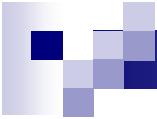


Implementation 1 Multiple Facades

- In general, a subsystem only need one facade.
 - But in some situation, the multiple facades is meaningful, it should also be considered.
-

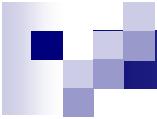
Example





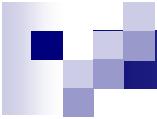
Implementation 2 There is not new behaviors defined in Facades

- **DO NOT** introduce new behaviors to the facade.
 - If an facade can not providing the required behaviors:
 - If such behaviors is implemented by subsystem, then extend the facade.
 - If such behaviors is not implemented by subsystem yet, the extend the subsystem and facade both.
-



Variation: Abstract Facade

- Making **Facade** an abstract class with concrete subclasses for different implementations of a subsystem. This abstract coupling keeps clients from knowing which implementation of a subsystem is used.
 - An alternative to subclassing is to configure a **Facade** object with different **subsystem objects**. To customize the facade, simply replace one or more of its **subsystem objects**.
-



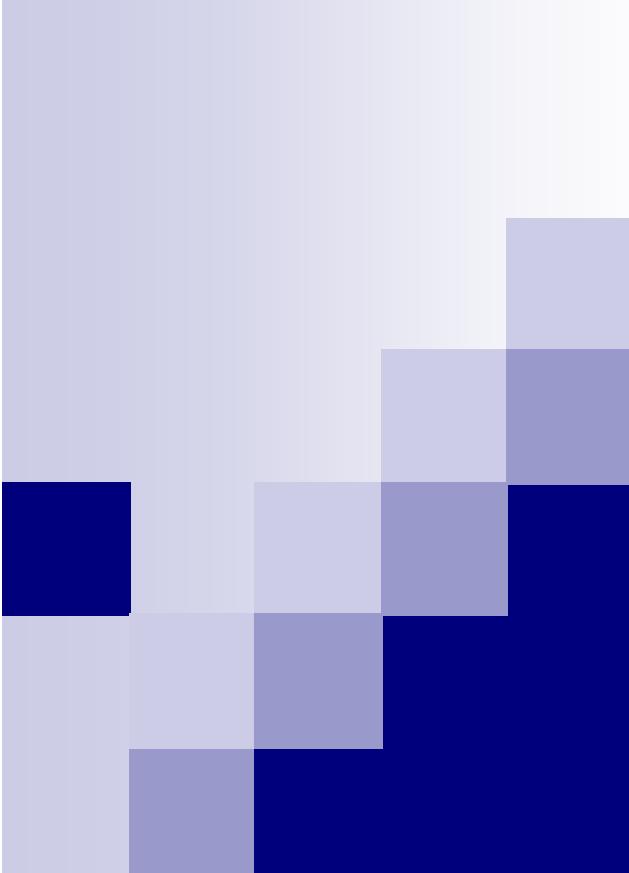
Extension: public subsystem and private subsystem

- A subsystem is analogous to a class in that both have interfaces (**NOT Java interface here**), and both encapsulate something
 - A class encapsulates state and operation
 - A subsystem encapsulates classes.
 - Think about the public and private interface of a class, **AND**
 - Think about the public and private interface of a subsystem.
 - The public interface to a subsystem consists of classes that all clients can access;
 - The private interface is just for subsystem extenders.
 - The Facade class is part of the public interface, of course, but it's not the only part. Other subsystem classes are usually public as well.
-



Let's go to next...





Design Patterns

宋 杰

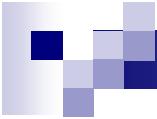
Song Jie

东北大学 软件学院

Software College, Northeastern
University

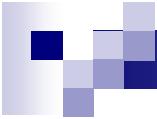


10. Flyweight Pattern



Intent

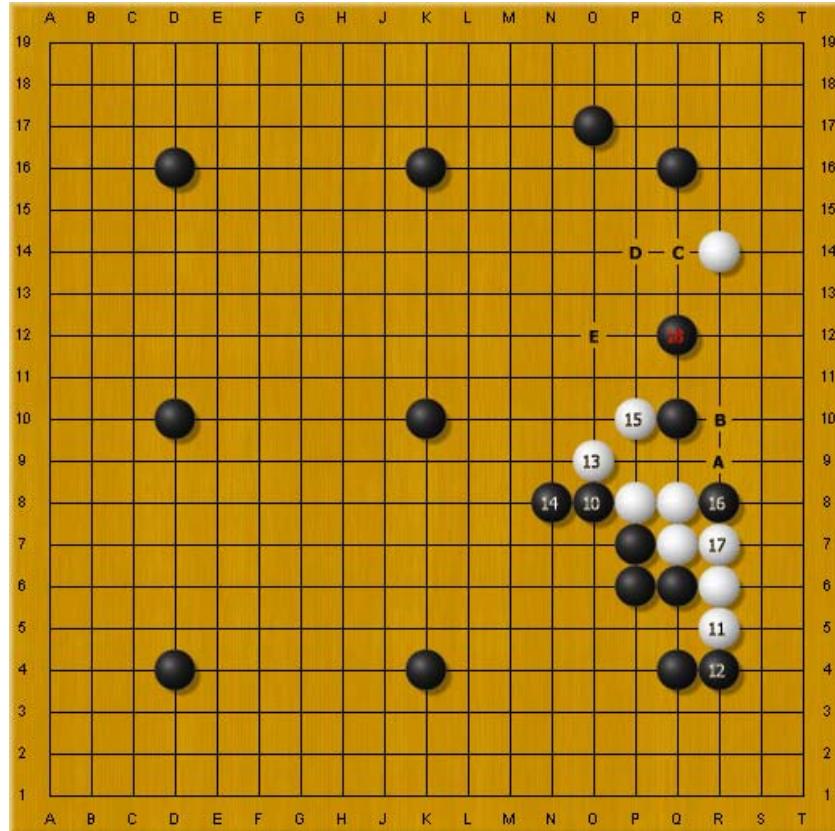
- Use sharing to support large numbers of **fine-grained** objects efficiently.
 - 享元模式以共享的方式高效地支持大量的细粒度对象。
-



Intent

- The shared flyweight is separated the extrinsic state from intrinsic state:
 - Intrinsic state is stored in the flyweight, sharable, not changed according to the context.
 - Extrinsic state is stored outside the flyweight, unsharable, changed according to the context.
 - For sharing the instances, **extrinsic state** of the **flyweight** (object) is stored by the clients, outside the **flyweight**. **Extrinsic state** is passed when **flyweight** need it.
 - In one word, **extrinsic state** of **flyweight** is independent from **flyweight**.
-

Example: Go Game



- Shared flyweight
 - Stone
(game piece)
- Intrinsic state
 - Black and white.
- Extrinsic state
 - Point
(position of Stone)

```
interface Stone {
    public boolean isBlack();
    public boolean isWhite();
    public void place(int x, int y);
}

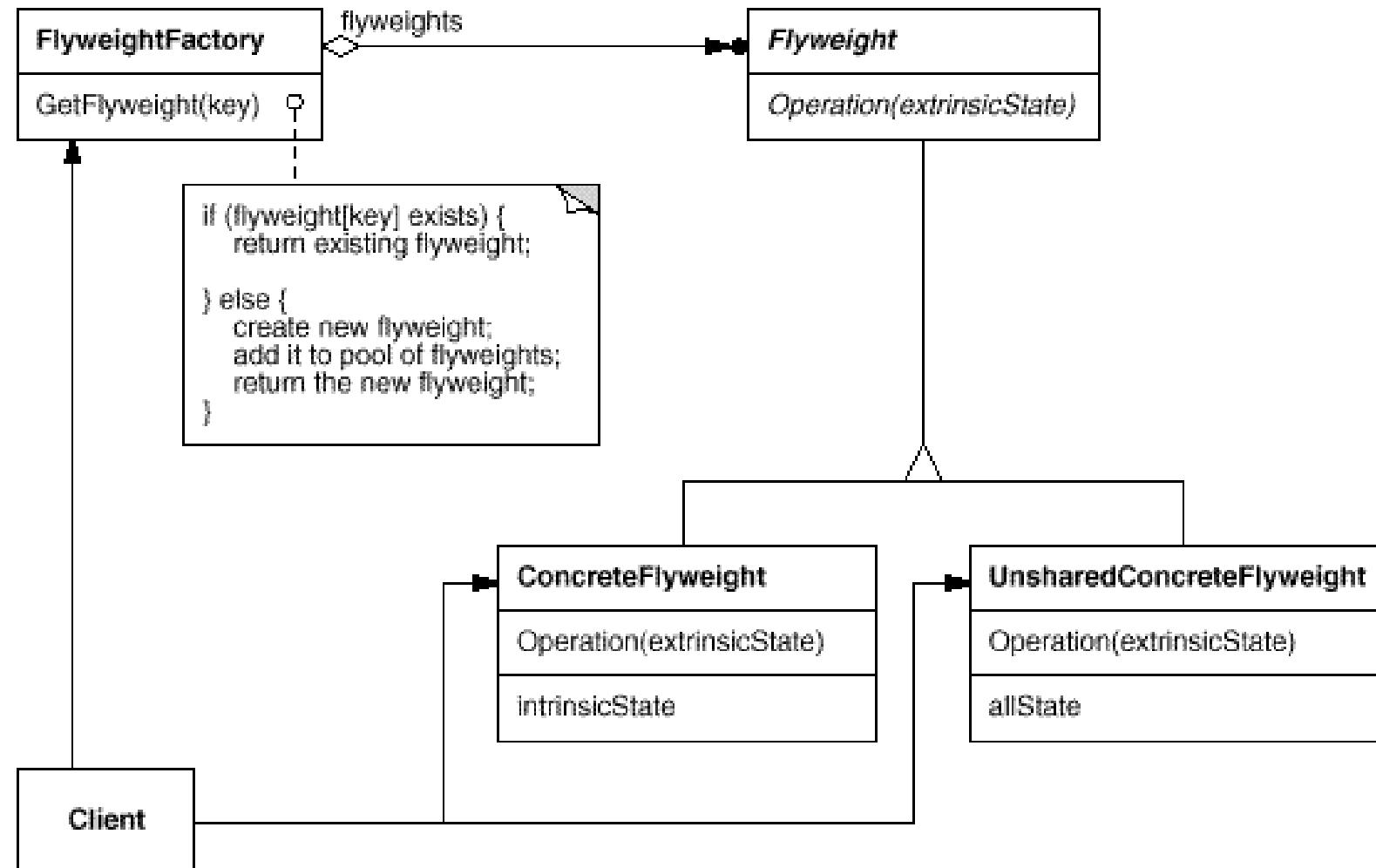
class GoStone implements Stone {
    private boolean black;
    public GoStone(boolean isBlack) {
        this.black = isBlack;
    }
    public boolean isBlack() {
        return black;
    }
    public boolean isWhite() {
        return !black;
    }
    public void place(int x, int y) {
    }
}
```

```
class StoneFactory{
    private static Stone white = new GoStone(false);
    private static Stone black = new GoStone(true);

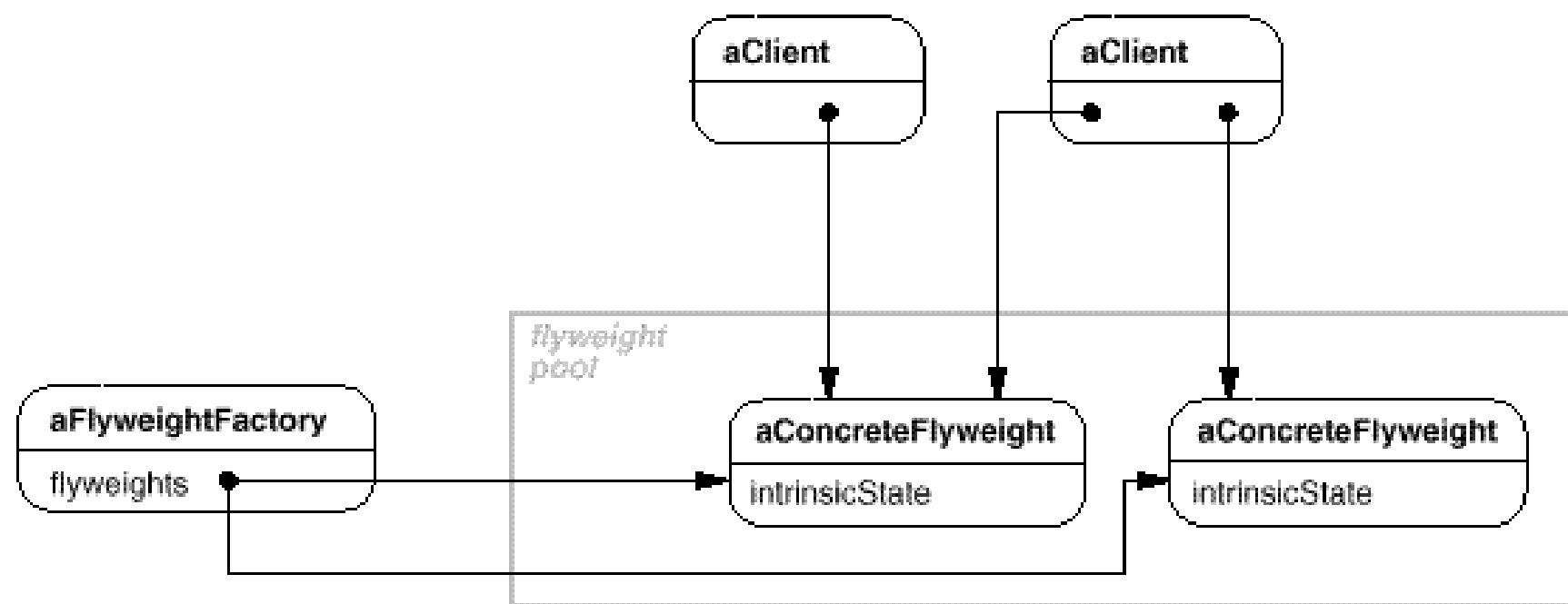
    public static Stone getWhite() {
        return white;
    }
    public static Stone getBlack() {
        return black;
    }
}
class FlyweightClient {
    public static void main(String[] args) {
        StoneFactory.getBlack().place(9, 9);
        StoneFactory.getWhite().place(9, 10);
        StoneFactory.getBlack().place(1, 1);
        StoneFactory.getWhite().place(9, 8);

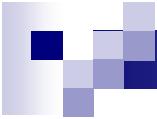
    }
}
```

Structure



Structure





Participants

- **Flyweight:**

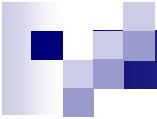
- Declares an interface through which flyweights can receive and act on extrinsic state.

- **ConcreteFlyweight**

- Implements the **Flyweight** interface and adds storage for intrinsic state, if any.
 - A **ConcreteFlyweight** object must be sharable.
 - Any state it stores must be intrinsic; that is, it must be independent of the **ConcreteFlyweight** object's context.

- **UnsharedConcreteFlyweight**

- Not all **Flyweight** subclasses need to be shared.
 - The **Flyweight** interface enables sharing; it doesn't enforce it.



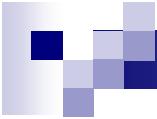
Participants

- **FlyweightFactory**

- Creates and manages **flyweight** objects, and ensures that **flyweights** are shared properly.
 - When a client requests a **flyweight**, the **FlyweightFactory** object supplies an existing instance or creates one, if none exists.

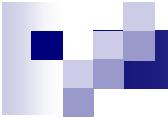
- **Client**

- Maintains a reference to **flyweight**(s).
 - Computes or stores the extrinsic state of **flyweight**(s).



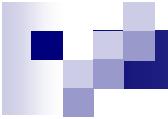
Collaborations

- State that a flyweight needs to function must be characterized as either intrinsic or extrinsic.
 - Intrinsic state is stored in the **ConcreteFlyweight** object;
 - Extrinsic state is stored or computed by **Client** objects. Clients pass this state to the flyweight when they invoke its operations.
 - Clients should not instantiate **ConcreteFlyweights** directly. Clients must obtain **ConcreteFlyweight** objects exclusively from the **FlyweightFactory** object to ensure they are shared properly.
-



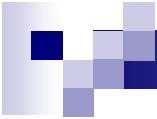
Consequences

- Flyweights may introduce run-time costs associated with **transferring, finding, and/or computing extrinsic state**. However, such costs are offset by space savings, which increase as more flyweights are shared.
-



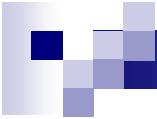
Consequences -Storage savings

- Storage savings are a function of several factors:
 - The reduction in the total number of instances that comes from sharing
 - The amount of intrinsic state per object
 - Whether extrinsic state is computed or stored.
 - The more flyweights are shared, the greater the storage savings.
 - The greatest savings occur when the extrinsic state can be computed rather than stored. Then you save on storage in two ways: Sharing reduces the cost of intrinsic state, and you trade extrinsic state for computation time.
-



Applicability

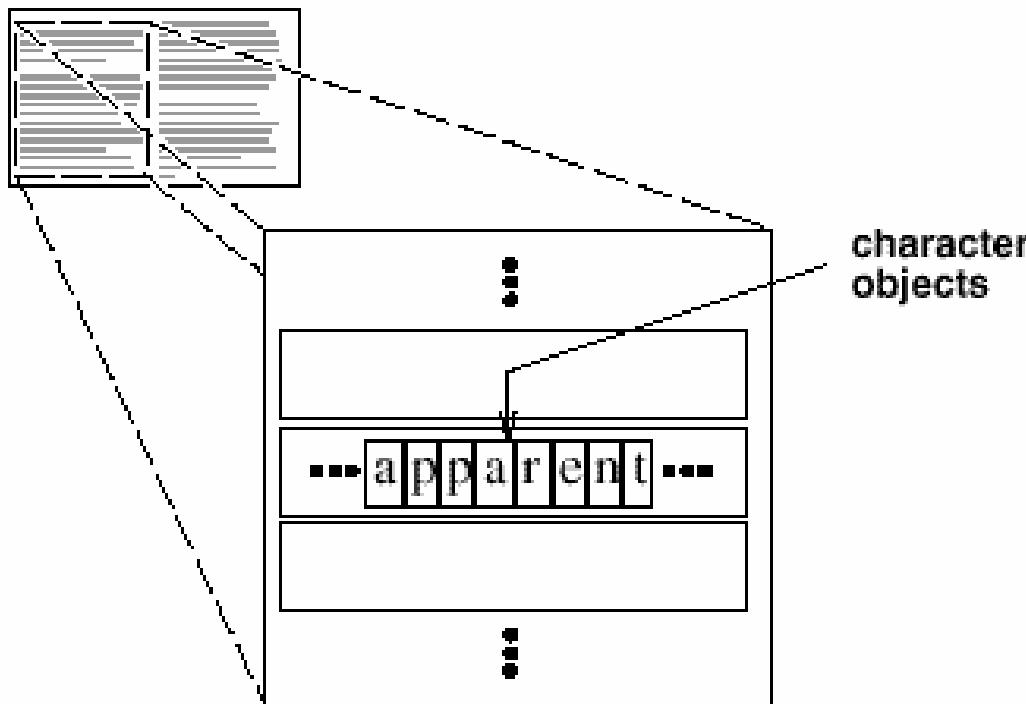
- An application uses a large number of objects.
 - Storage costs are high because of the larger quantity of objects.
 - Most object state can be made extrinsic.
 - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
 - The application doesn't depend on object identity.
Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.
-



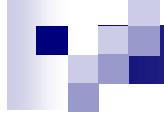
Implementation 1: Removing extrinsic state

- The pattern's applicability is determined largely by how easy it is to identify extrinsic state and remove it from shared objects.
-

Example

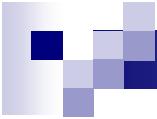


- Characters
- Font
 - Color
 - Size
 - Space



Implementation 2: How to removing extrinsic state

- Remove the extrinsic state and associated behaviors (codes) from the flyweight to clients together; OR
 - The clients store and pass the extrinsic state to the flyweight by parameters; OR
 - Encapsulate the extrinsic state and corresponding behaviors to an independent structure, that is, building new classes.
-



Implementation 3: Managing shared objects

- Because objects are shared, clients shouldn't instantiate them directly. **FlyweightFactory** lets clients locate a particular flyweight.
 - **FlyweightFactory** objects often use an associative store to let clients look up flyweights of interest. The manager returns the proper flyweight given its code, creating the flyweight if it does not already exist.
 - Sharability also implies some form of **reference counting** or **garbage collection** to reclaim a flyweight's storage when it's no longer needed. However, neither is necessary if the number of flyweights is fixed and small.
-

Example 1: An FlyweightFactory with registered pool and unshared Flyweight

```
abstract class Flyweight {
    protected String idenity;
    public String getIdentity() {
        return idenity;
    }
}

class UnsharedFlyweight extends Flyweight {
    public UnsharedFlyweight(String idenity) {
        this.idenity = idenity;
    }
}

class SharedFlyweight extends Flyweight {
    public SharedFlyweight(String idenity) {
        this.idenity = idenity;
    }
}
```

```
class FlyweightFactory {
    private static Map<String, Flyweight> flyweightPool
        = new HashMap<String, Flyweight>();

    static {
        // can initialize the flyweightPool with default flyweights here
    }

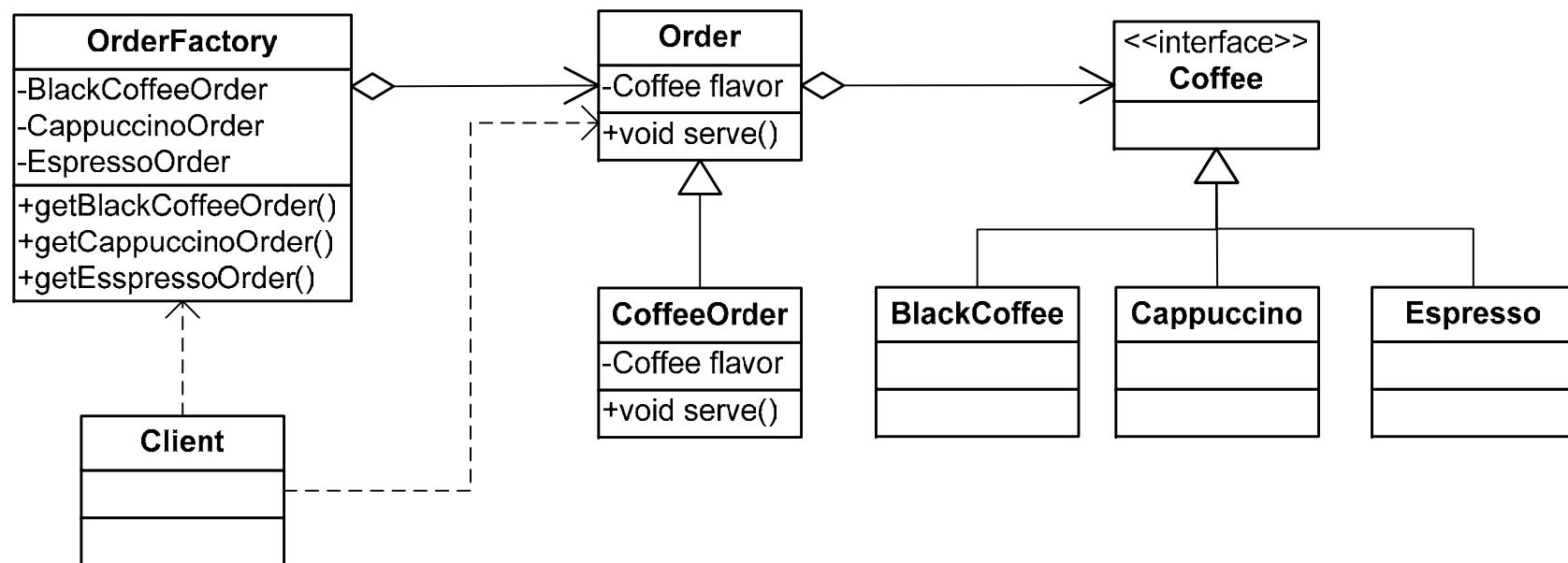
    public static Flyweight getUnsharedFlyweight(String idenity) {
        return new UnsharedFlyweight(idenity);
    }

    public static Flyweight getSharedFlyweight(String idenity) {
        if (!flyweightPool.containsKey(idenity)) {
            flyweightPool.put(idenity, new SharedFlyweight(idenity));
        }
        return flyweightPool.get(idenity);
    }

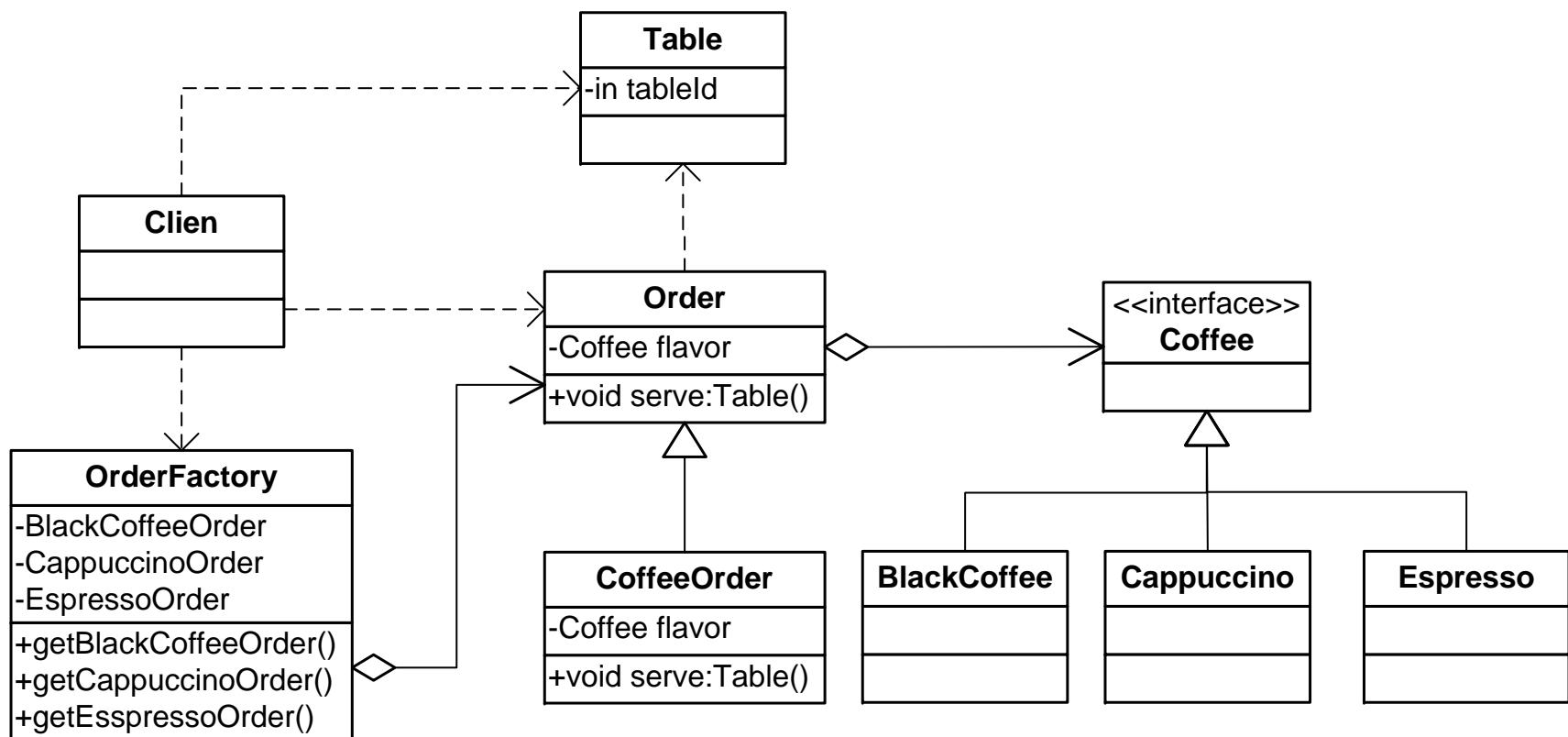
    public static void clearPool() {
        flyweightPool.clear();
    }
}
```

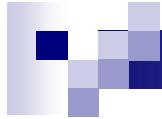
Example 2: Coffee Booth

Coffee : Black coffee, Cappuccino, Espresso
Order : serve()



Example 2: Coffee Bar

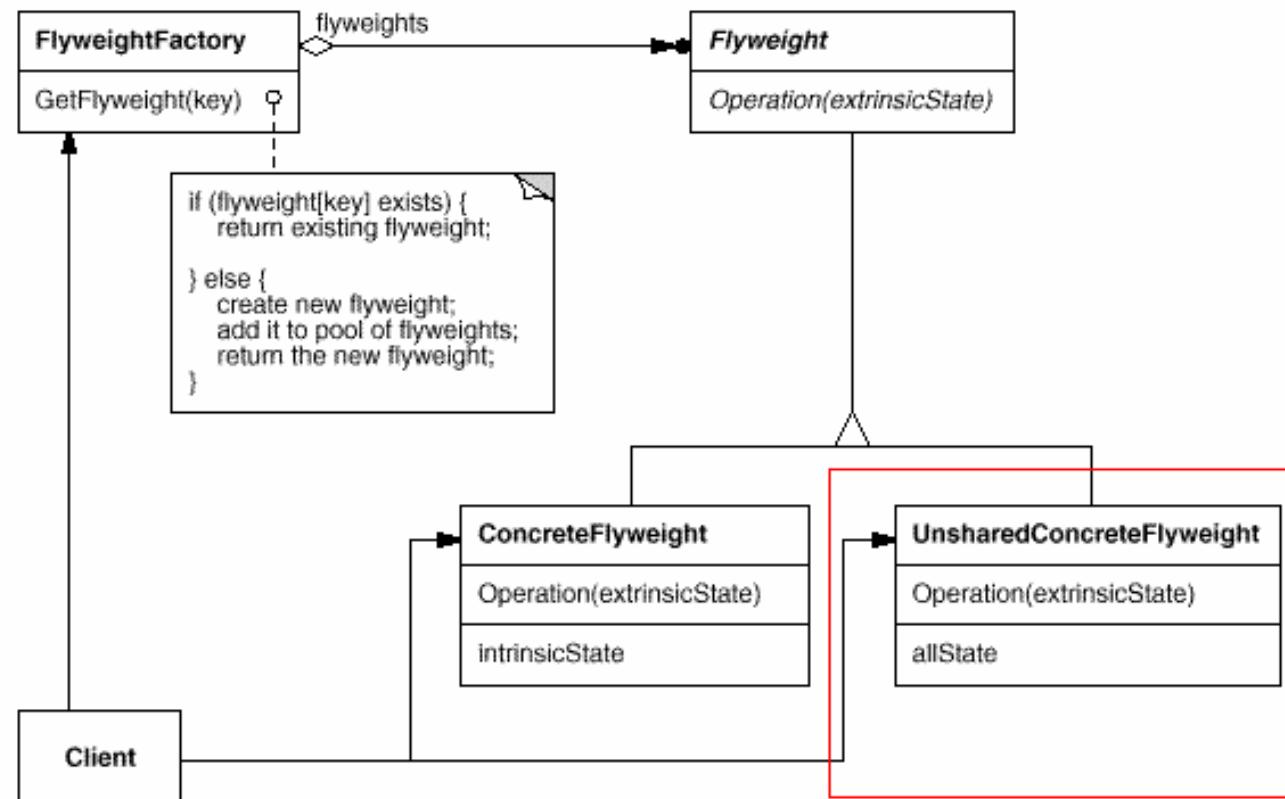




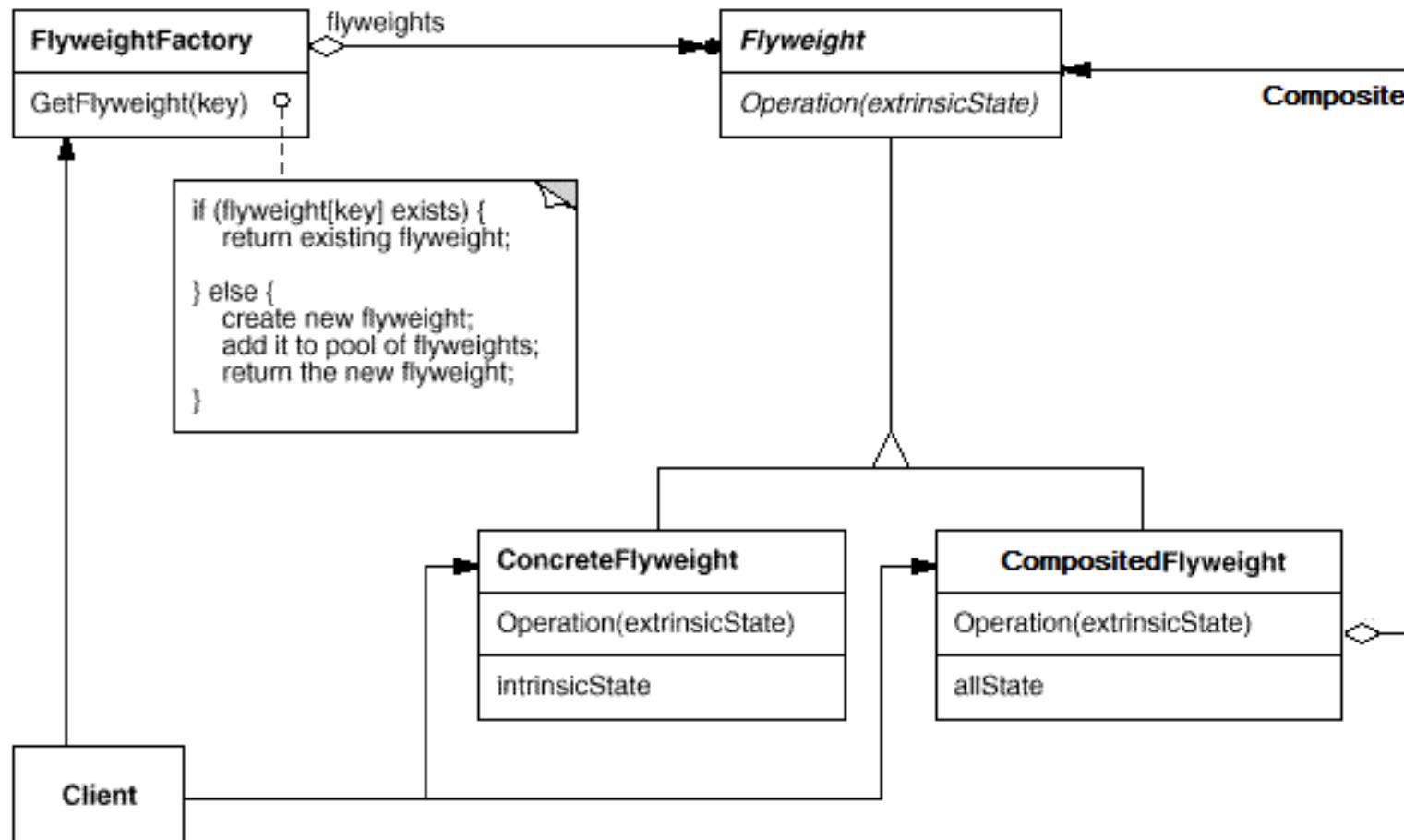
Extension: Shared Object Pattern

- The intent of flyweight pattern is saving storage by sharing larger quantity of **fine granularity** objects;
 - Sometimes, we share object but not adopt the flyweight pattern because the shared object is **coarse granularity**.
 - Saving storage by sharing the bigger, heavy and less quantity objects.
-

Variation 1: Constrainedly shared flyweight



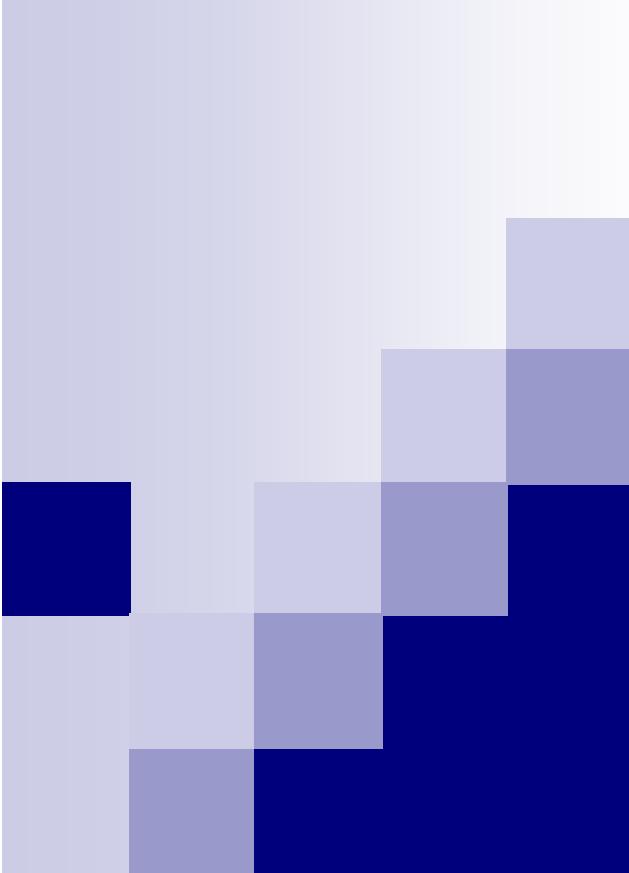
Variation 2: Composited flyweight





Let's go to next...





Design Patterns

宋 杰

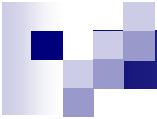
Song Jie

东北大学 软件学院

Software College, Northeastern
University



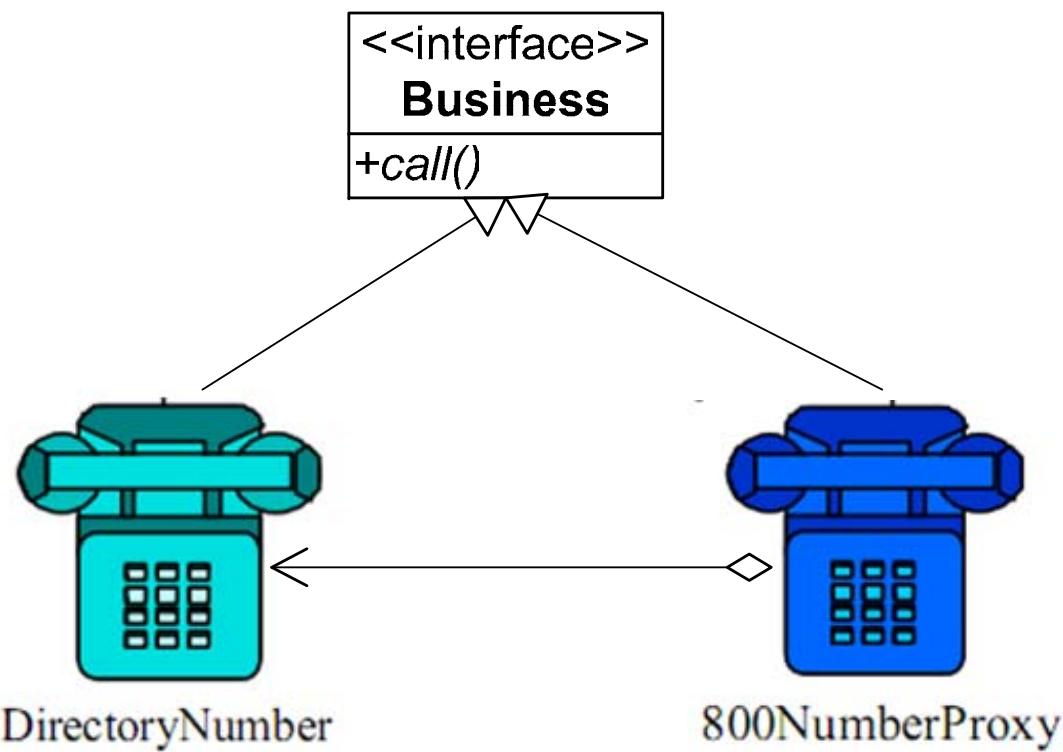
11. Proxy Pattern



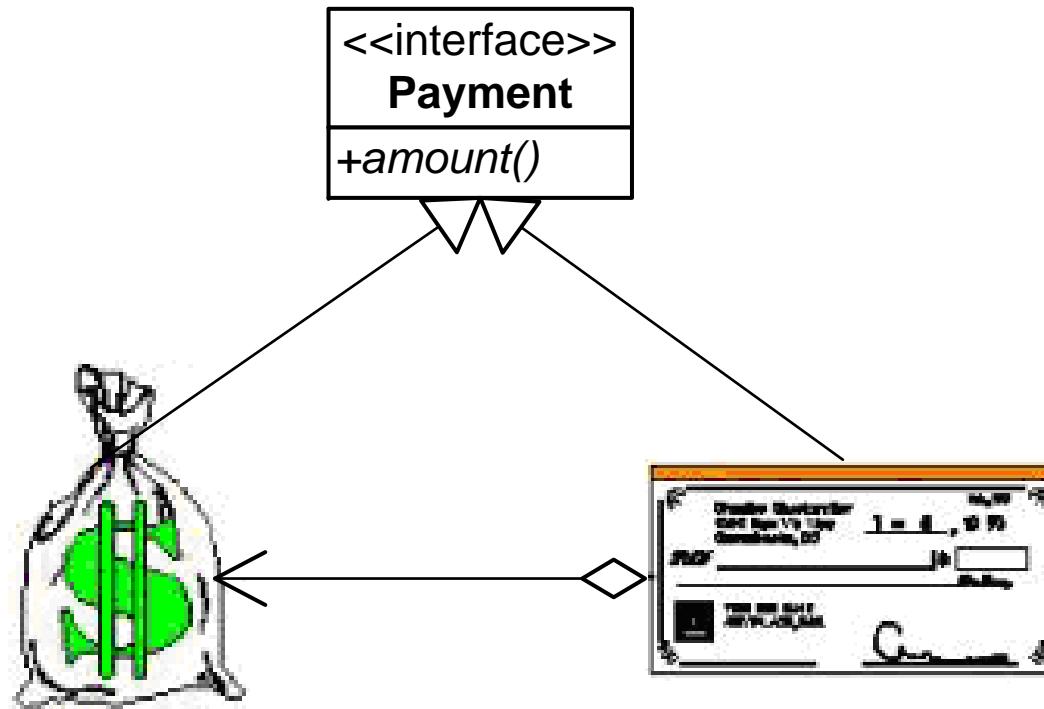
Intent

- Provide a surrogate or placeholder for another object to control access to it.
 - 代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。
-

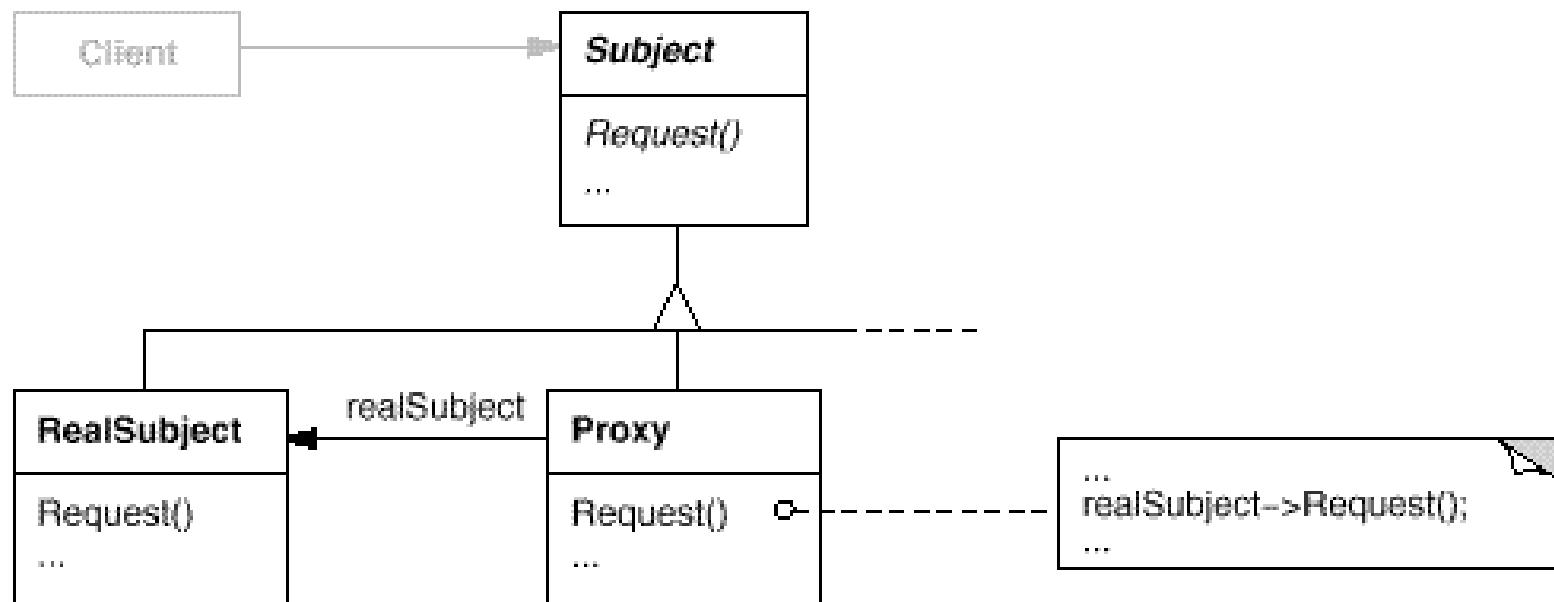
Example

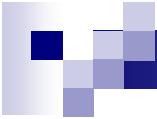


Example



Structure





Participants

■ Proxy

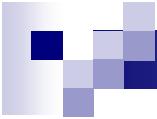
- Maintains a reference that lets the proxy access the real subject. **Proxy** may refer to a **Subject** if the **RealSubject** and **Subject** interfaces are the same.
- Provides an interface identical to **Subject**'s so that a proxy can be substituted for the real subject.
- Controls access to the real subject and may be responsible for creating and deleting it.

■ Subject

- Defines the common interface for **RealSubject** and **Proxy** so that a **Proxy** can be used anywhere a **RealSubject** is expected.

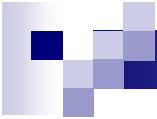
■ RealSubject

- Defines the real object that the proxy represents.
-



Collaborations

- aProxy contains the reference of aRealSubject, thus it can manipulate the aRealSubject in anytime;
 - aProxy provides an interface which is in according with the interface of aRealSubject, thus it can replacement the aRealSubject in anytime;
 - aProxy controls the reference of aRealSubject, that is, it create and destroy the aRealSubject;
 - aProxy forwards requests to aRealSubject when appropriate, depending on the kind of proxy.
 - aProxy always add some pre- or post- behaviors when the invocation of clients is delegated to aRealSubject.
-

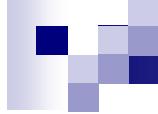


Consequences

- The **Proxy** pattern introduces a level of indirection when accessing an object. The additional indirection has many uses, depending on the kind of proxy:
-

Proxy Cases (Applicability)

- **Remote Proxy**: A remote proxy can hide the fact that an object resides in a different address space.
- **Virtual Proxy**: A virtual proxy can perform optimizations such as creating an object on demand (maybe the creating the real subject is expensive).
- **Smart Reference Proxy**: It add some additional functionalities when the real subject is invoked, for examples:
 - Recording the consumed time the real object being invoked
 - Counting the number of references to the real object
 - Loading a persistent object into memory when it's first referenced.
 - Checking that the real object is locked before it's accessed to ensure that no other object can change it.
- **Protection Proxy (Access Proxy)**: Protection proxy control the access of real subject (access controlling).
- **Copy-on-Write**: it is for clone object, it really clone the real subject for client on demand, that is, the status of object is going to be changed.
- **Cache Proxy**: it provides the cached space for the real subject, thus the clients can store and share more results.
- **Firewall Proxy**: it filter the hostile requests and hostile data.
- **Synchronization Proxy**: it provides the functionalities that multiple clients (threads) can access the real subject without conflicts.



Example: Achilles

- Achilles是一个用来测试网站的安全性能的工具软件。
 - 当Achilles处于截取状态时，它会向客户端假装是服务器，同时向真正的服务器假装是浏览器，在两端商议SSL通信。
 - Achilles可以破解加密的数据，给Achilles的用户显示已经解密的内容，并且允许用户更改处于通信过程中的数据。
-

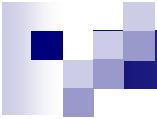
Example: Shortcut of Windows

- Alias in Macintosh
- Link in UNIX
- Shortcut in Windows



Example: 替考

- 有一位同学（考生）请求另一位同学（枪手）代替他参加一个英文水准考试。
- Who is Proxy and Who is RealSubject?
 - 枪手 is RealSubject;
 - 考生 is Proxy;



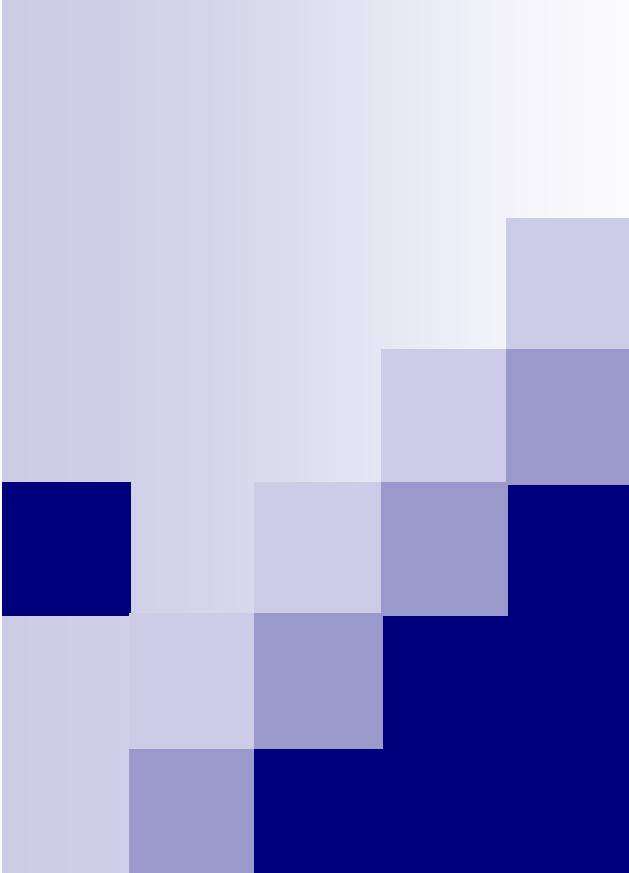
Extension: Transparent proxy

- Proxy doesn't always have to know the type of real subject.
 - If a Proxy class can deal with its subject solely through an abstract interface, then there's no need to make a Proxy class for each RealSubject class; the proxy can deal with all RealSubject classes uniformly.
 - But if Proxy is going to instantiate, then they have to know the concrete class.
-



Let's go to next...





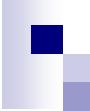
Design Patterns

宋 杰

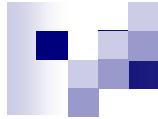
Song Jie

东北大学 软件学院

Software College, Northeastern
University

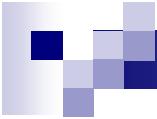


12. Bridge Pattern



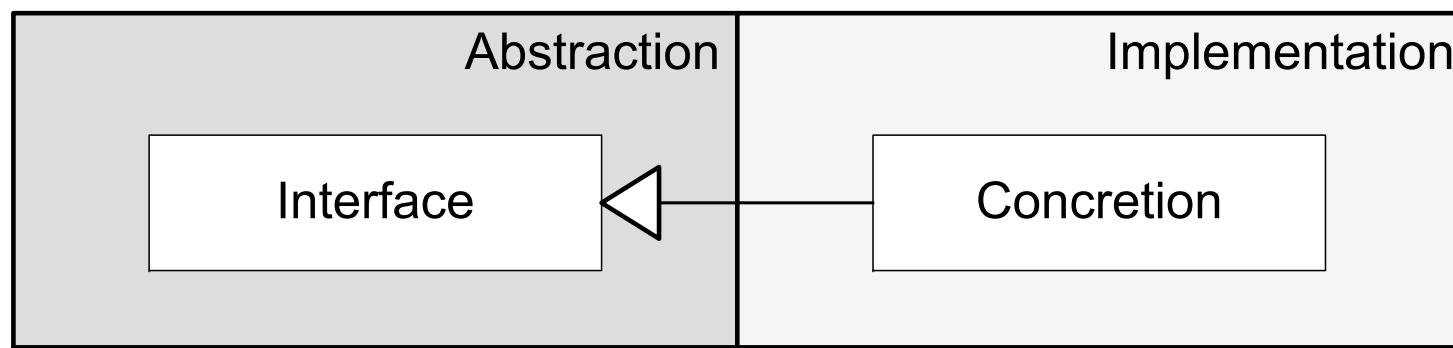
Intent

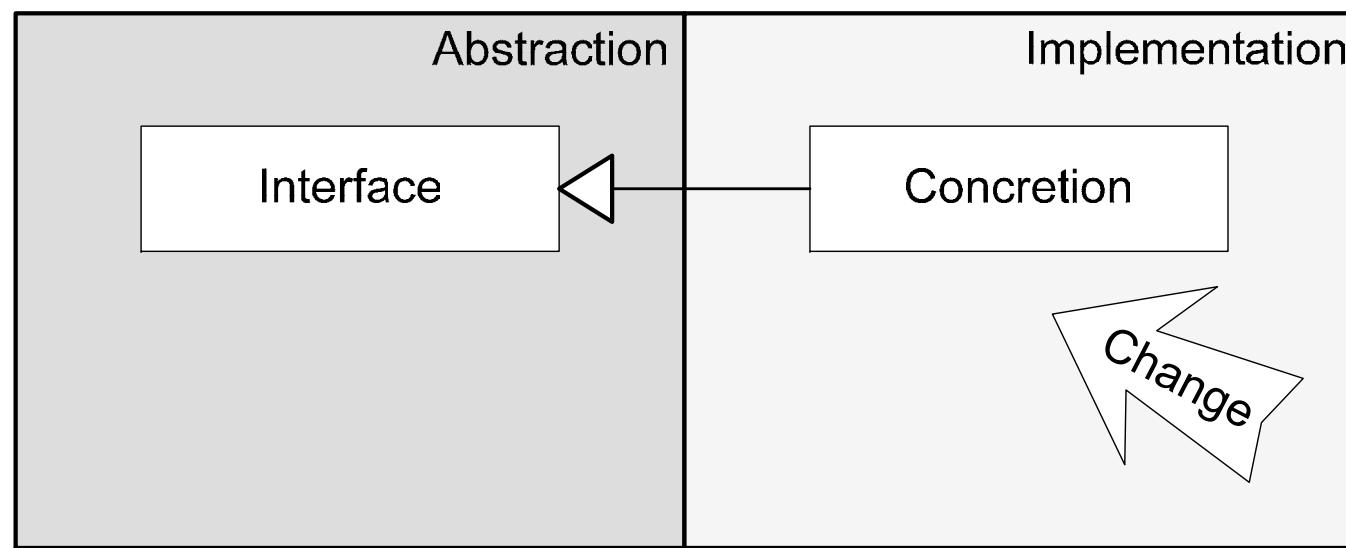
- Decouple an **abstraction** from its **implementation** so that the two can **vary (change)** independently.
- 桥梁模式的用意是将**抽象化(Abstraction)**与**实现化(Implementation)**解耦，使得二者可以独立地变化。
 - 抽象化(Abstraction)不等于接口(Interface)，存在于多个实体中的共同的概念性联系，就是抽象化。接口是一种抽象化的方式。
 - 所谓强耦和，就是在编译时期已经确定的，无法在运行时期动态改变的关联；所谓弱耦和，就是可以动态地确定并且可以在运行时期动态地改变的关联。

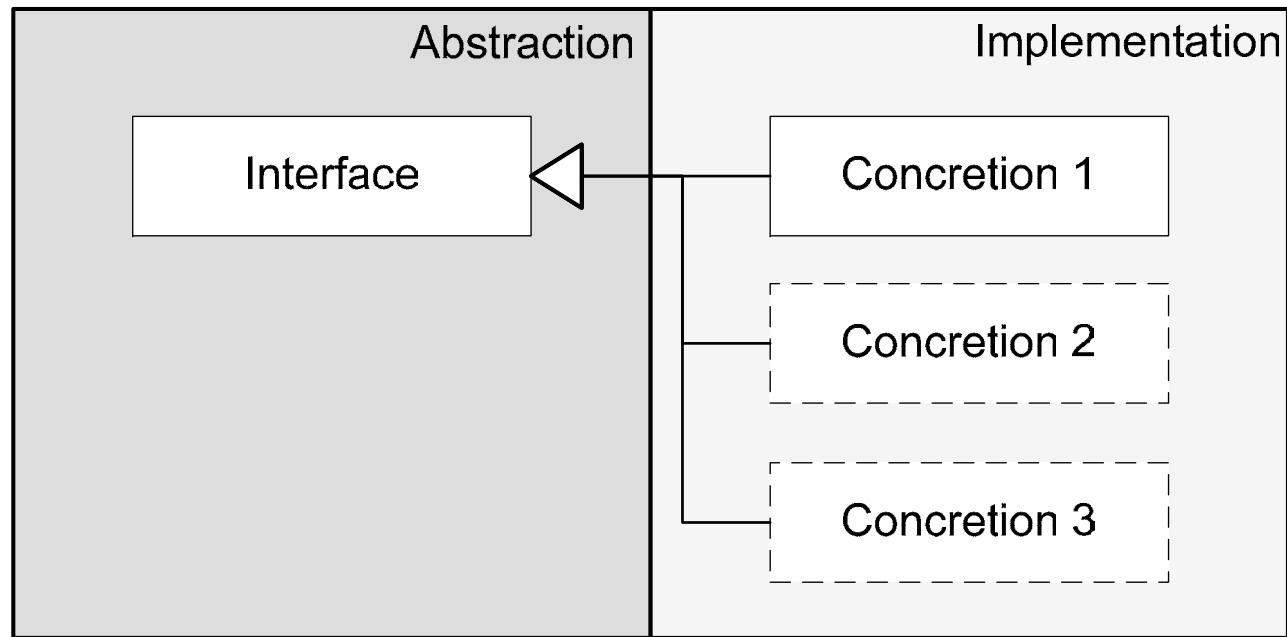


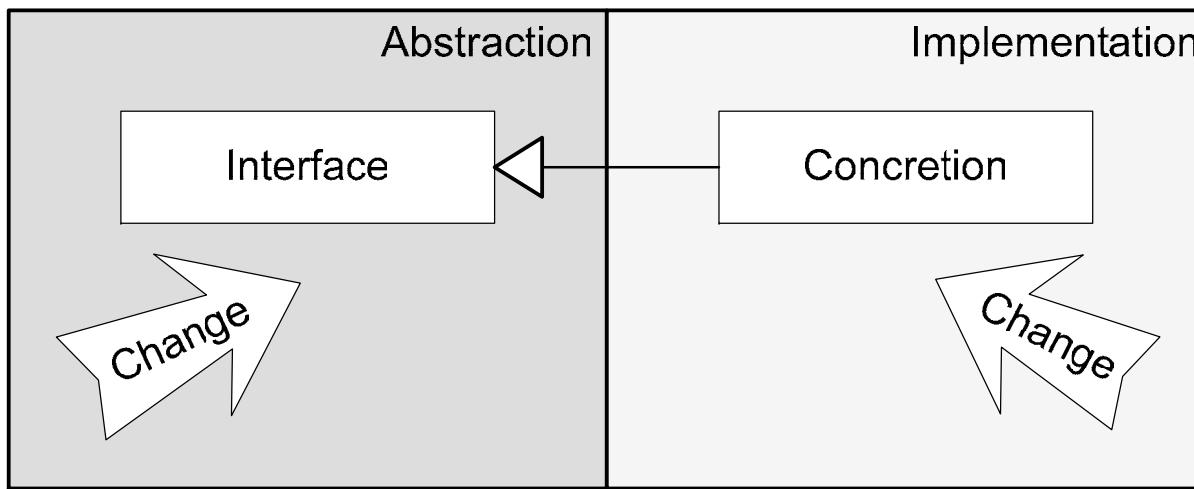
Intent

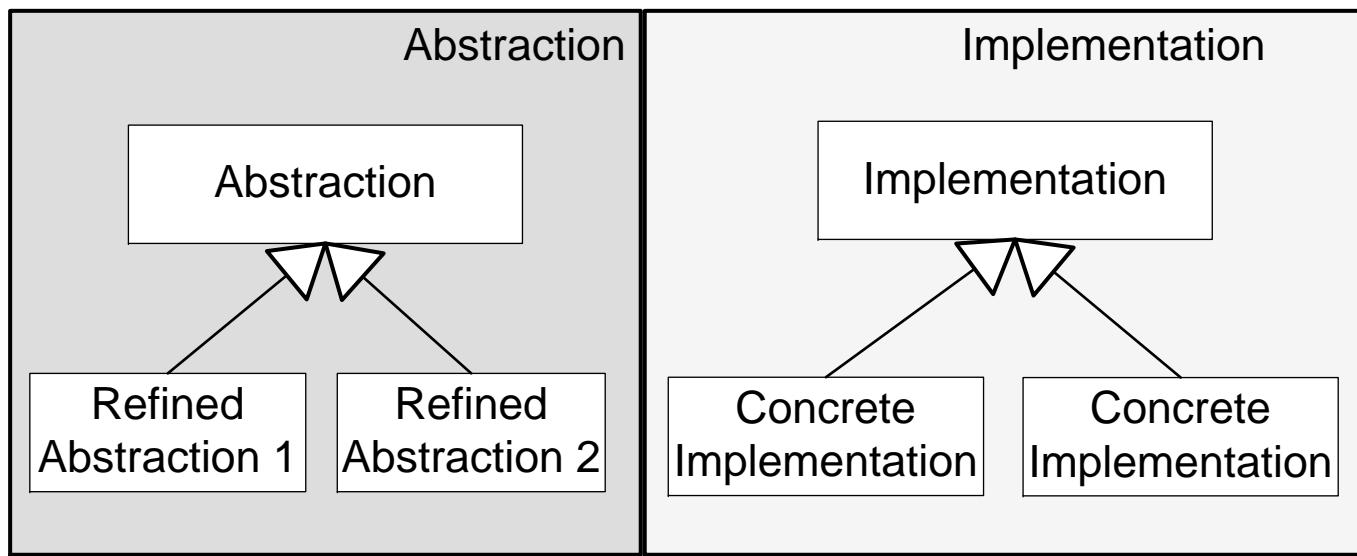
- When an **abstraction** can have one of several possible **implementations**, the usual way to accommodate them is to use inheritance.
 - An abstract class defines the **interface** to the abstraction, and concrete subclasses implement it in different ways.
 - But this approach isn't always flexible enough. Inheritance binds an implementation to the abstraction **permanently**, which makes it difficult to modify, extend, and reuse abstractions and implementations independently.
-

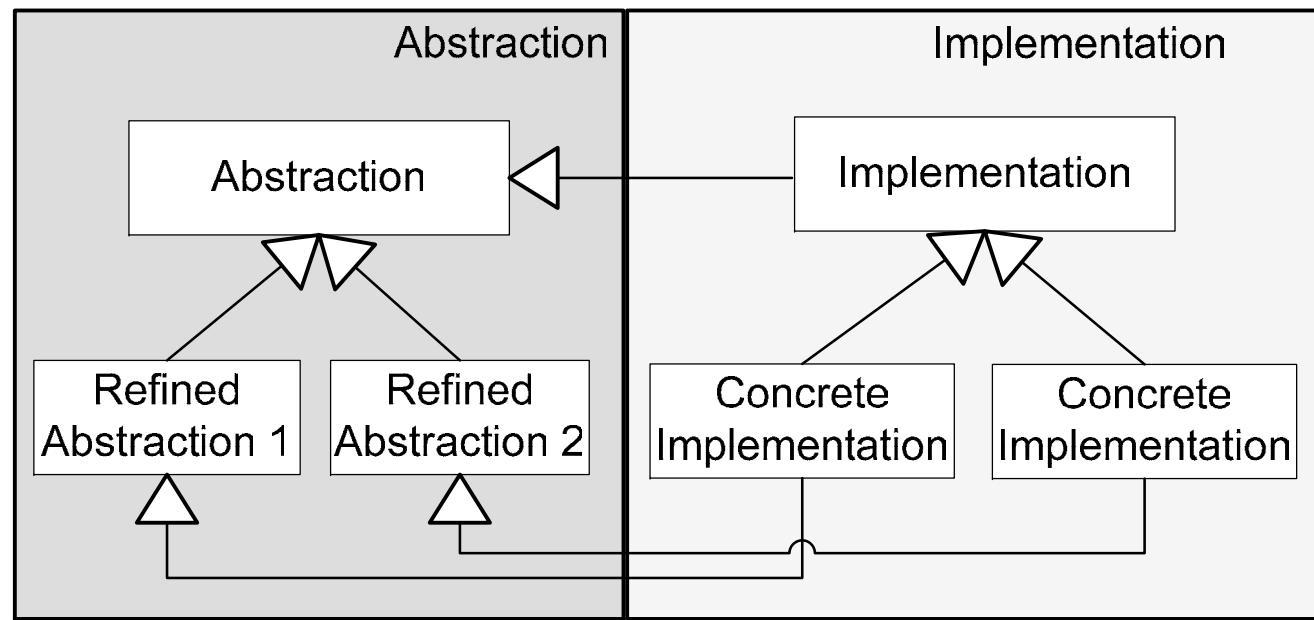


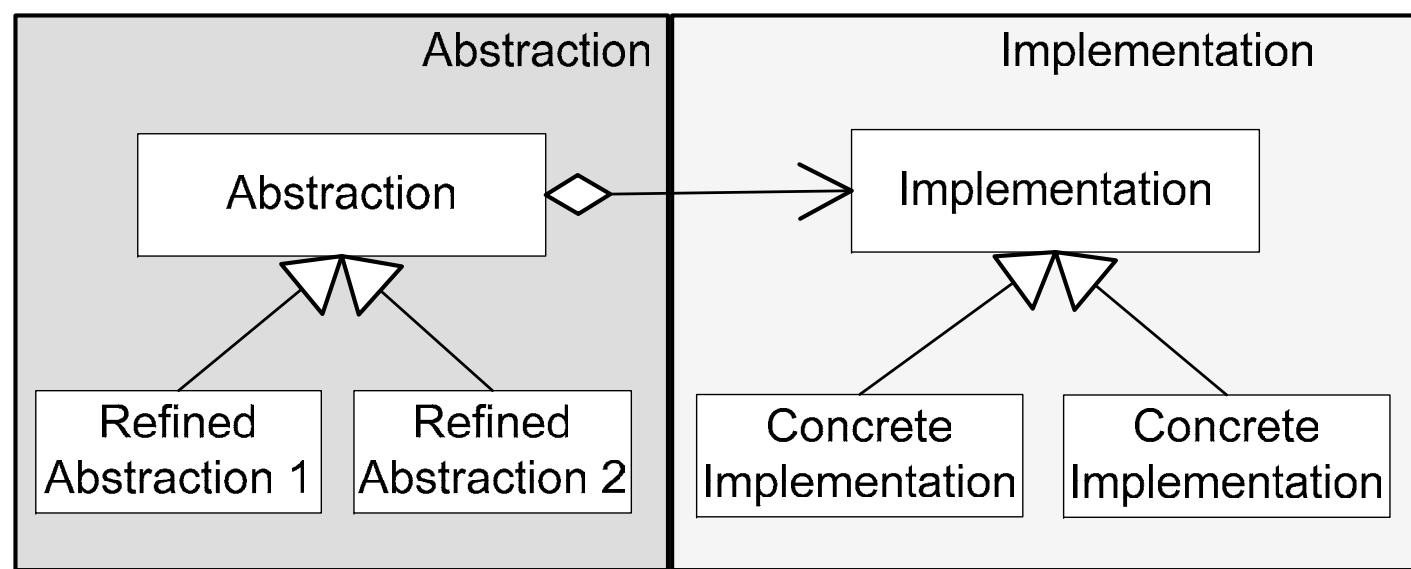




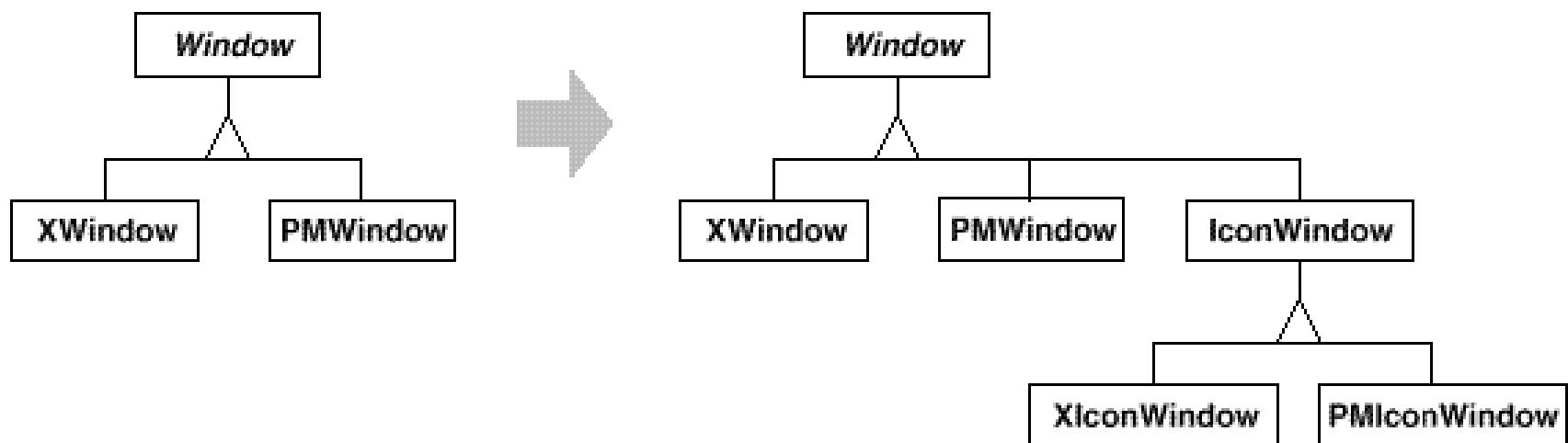




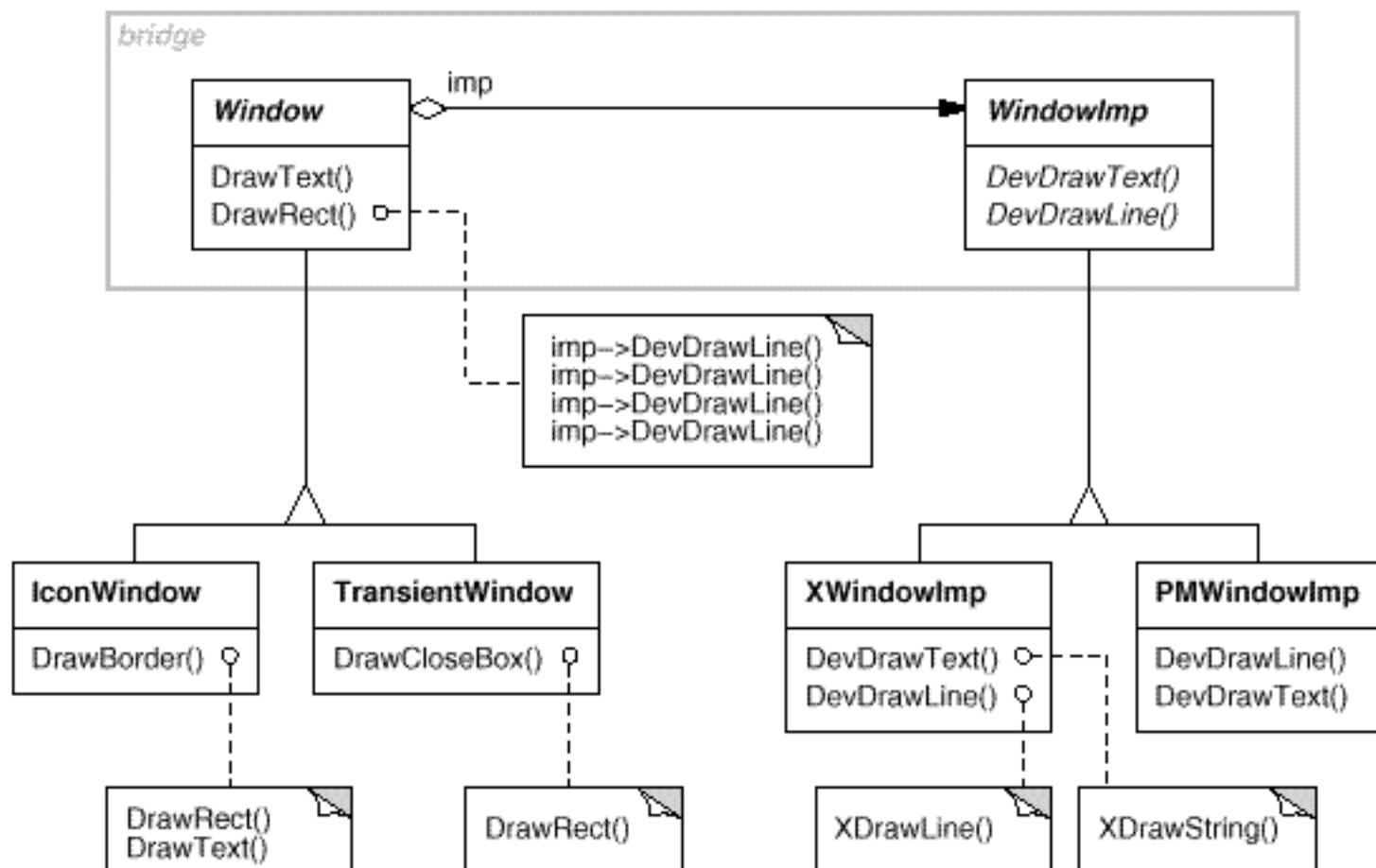




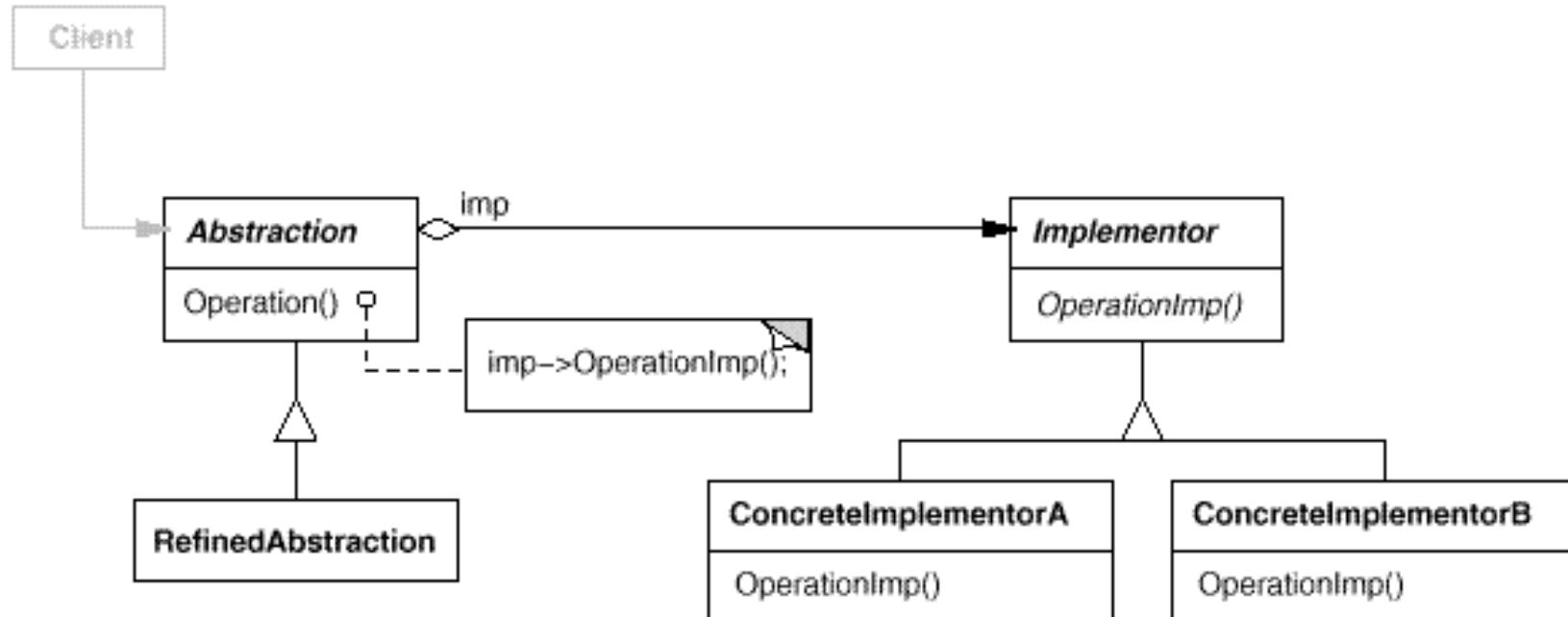
Example

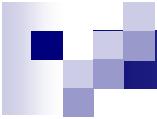


Example



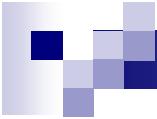
Structure





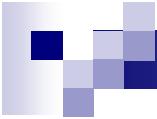
Participants

- **Abstraction**
 - Defines the abstraction's interface.
 - Maintains a reference to an object of type **Implementor**.
 - **RefinedAbstraction**
 - Extends the interface defined by **Abstraction**.
 - **Implementor**
 - Defines the interface for implementation classes. This interface doesn't have to correspond exactly to **Abstraction**'s interface; in fact the two interfaces can be quite different. Typically the **Implementor** interface provides only primitive operations, and **Abstraction** defines higher-level operations based on these primitives.
 - **ConcreteImplementor**
 - implements the **Implementor** interface and defines its concrete implementation.
-



Consequences

- Decoupling interface and implementation.
 - An implementation is not bound permanently to an interface.
 - The implementation of an abstraction can be configured at run-time. It's possible for an object to change its implementation at run-time.
 - Eliminates compile-time dependencies on the implementation. Changing an **implementation** class doesn't require recompiling the **Abstraction** class and its clients.
 - Encourage layering that can lead to a better-structured system. The high-level part of a system only has to know about **Abstraction** and **Implementor**.
 - Improved extensibility.
 - Hiding implementation details from clients.
-

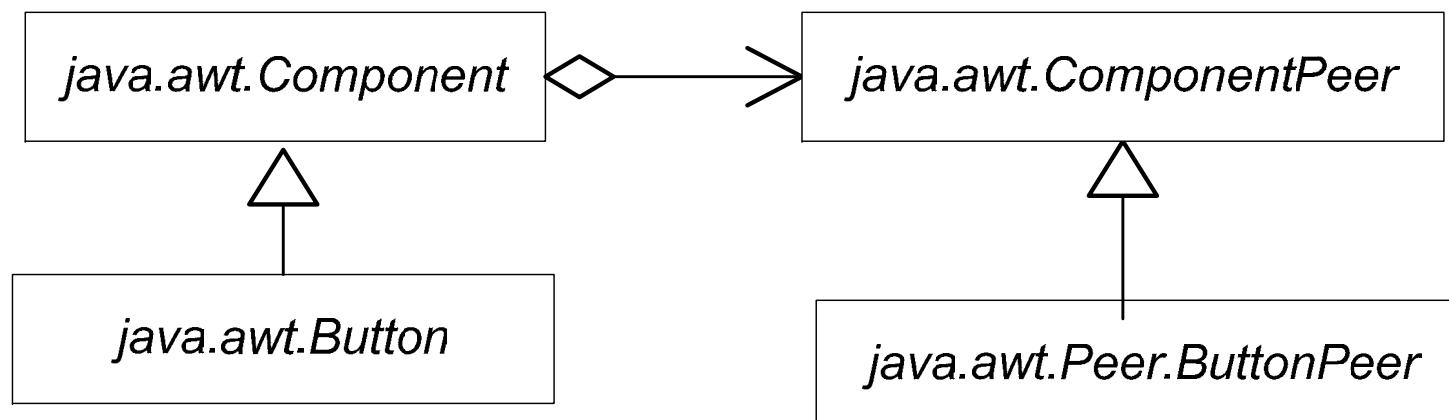


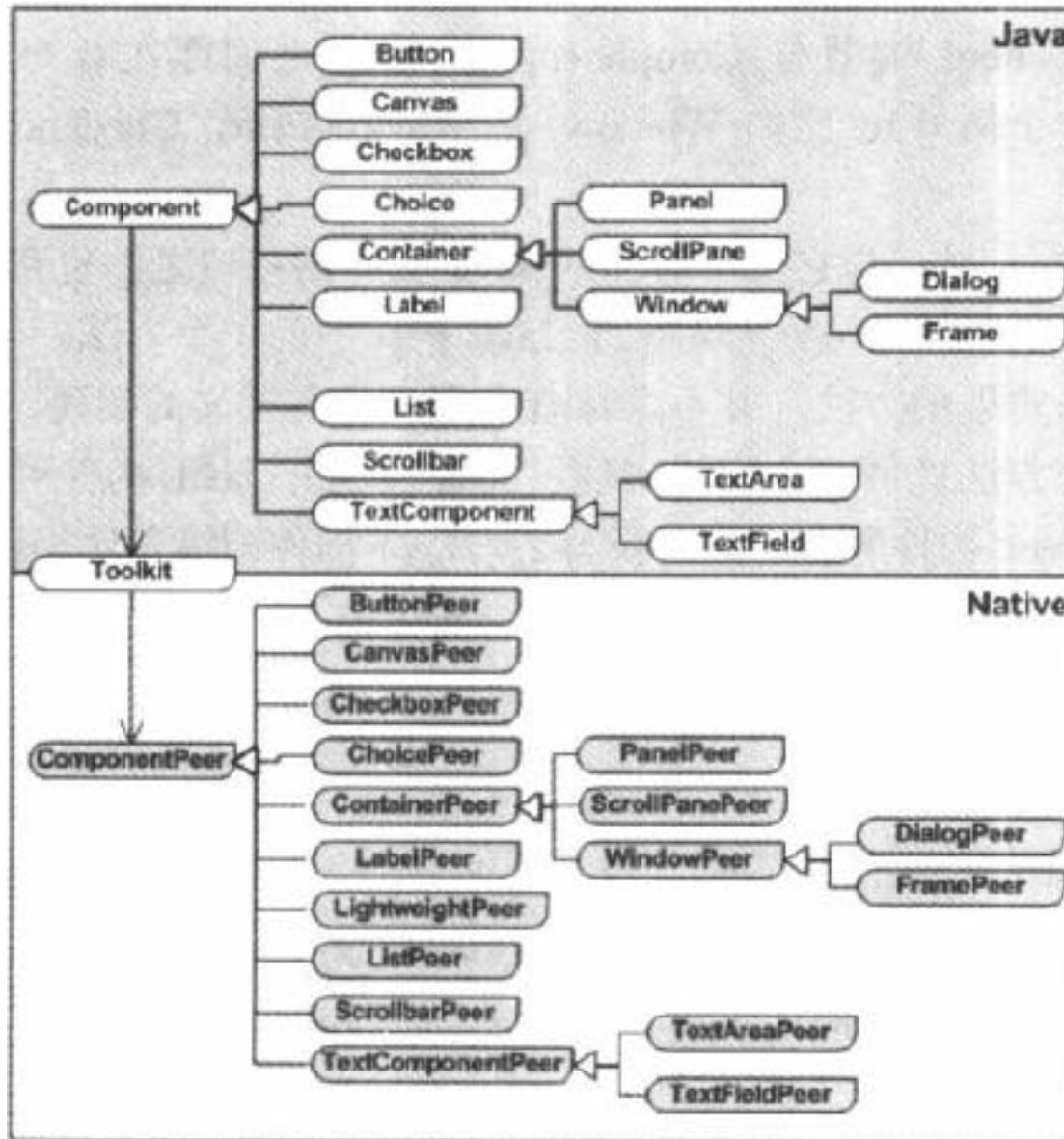
Applicability

- You want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.
 - Both the abstractions and their implementations should be extensible by subclassing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
 - Changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
-

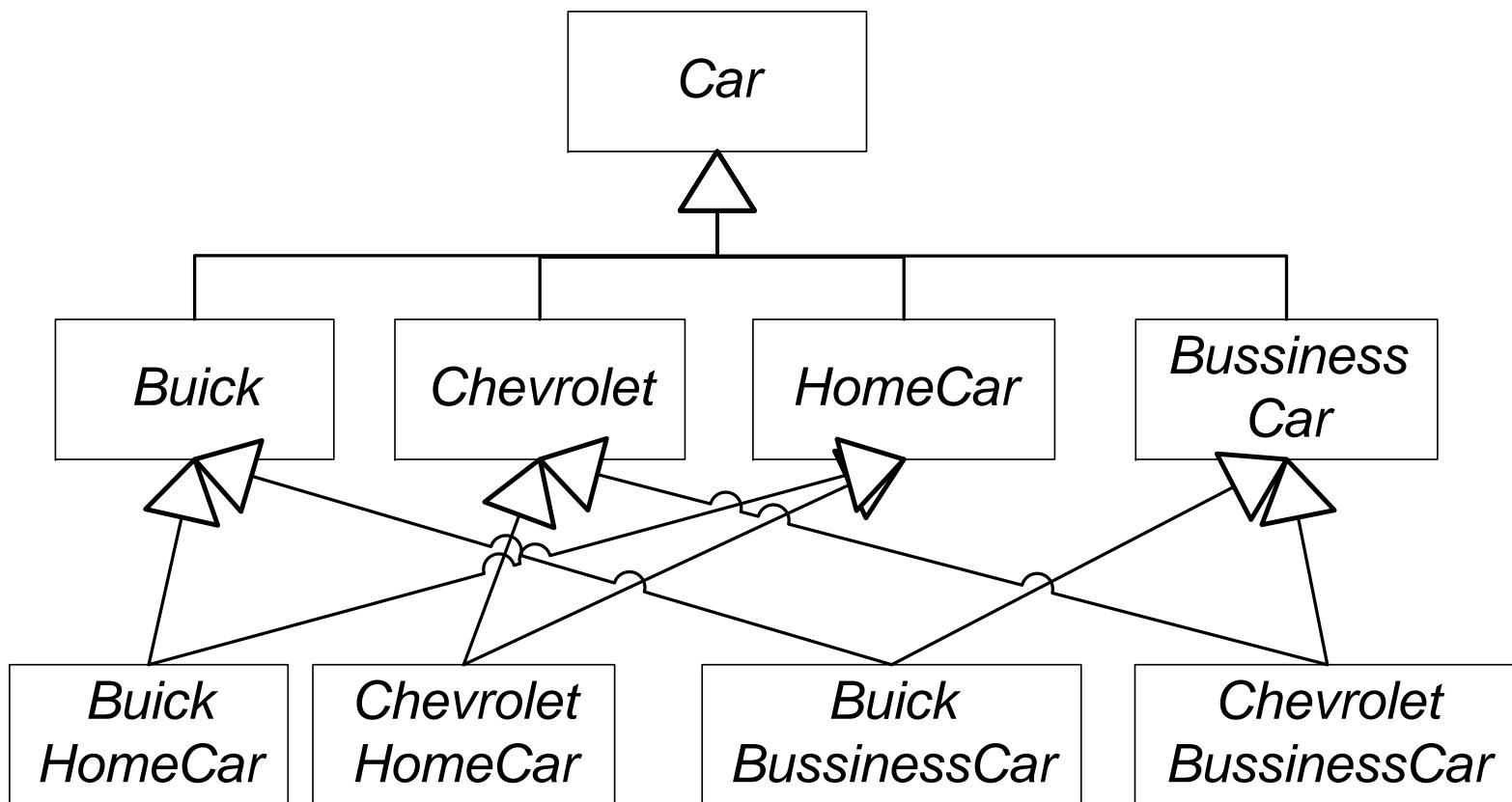
Example: Peer

- The interfaces in the `java.awt.peer` package define the native GUI capabilities that are required by the heavyweight AWT components of the `java.awt` package.

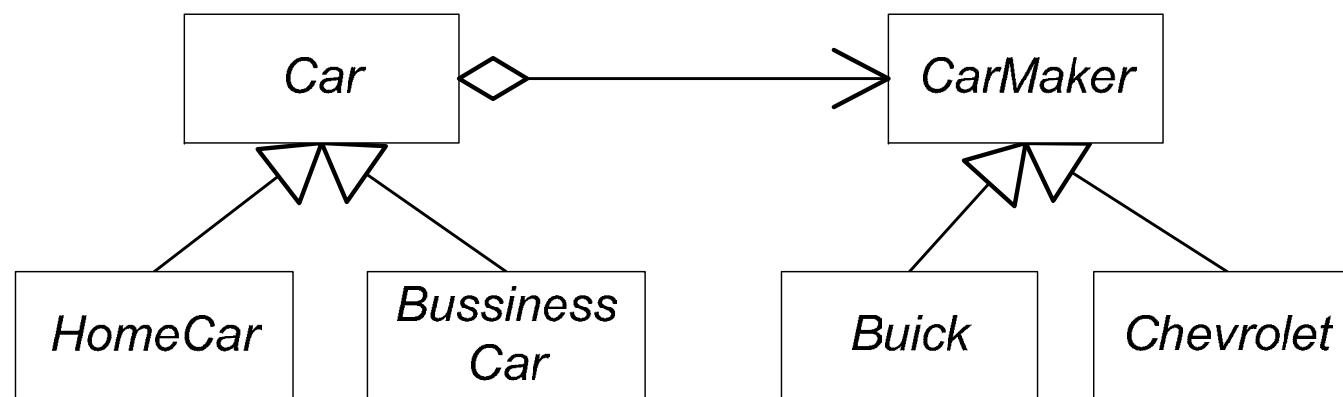




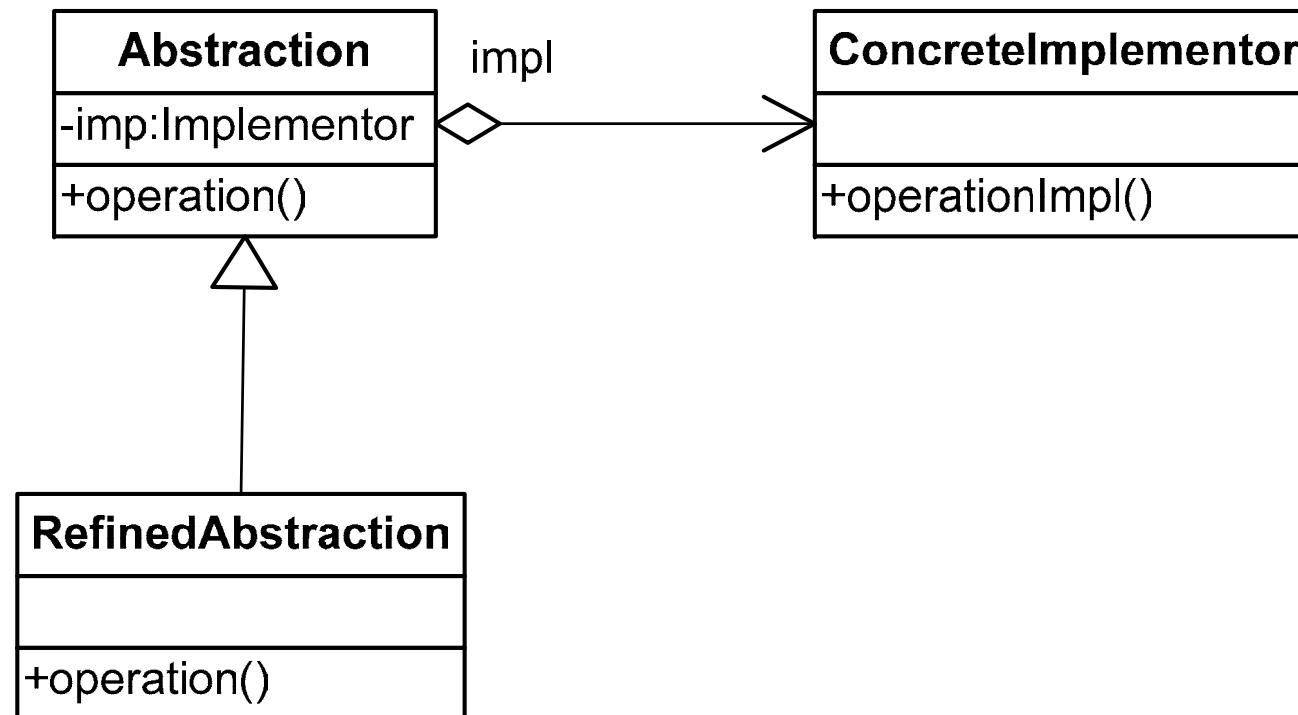
Example: Car Factory



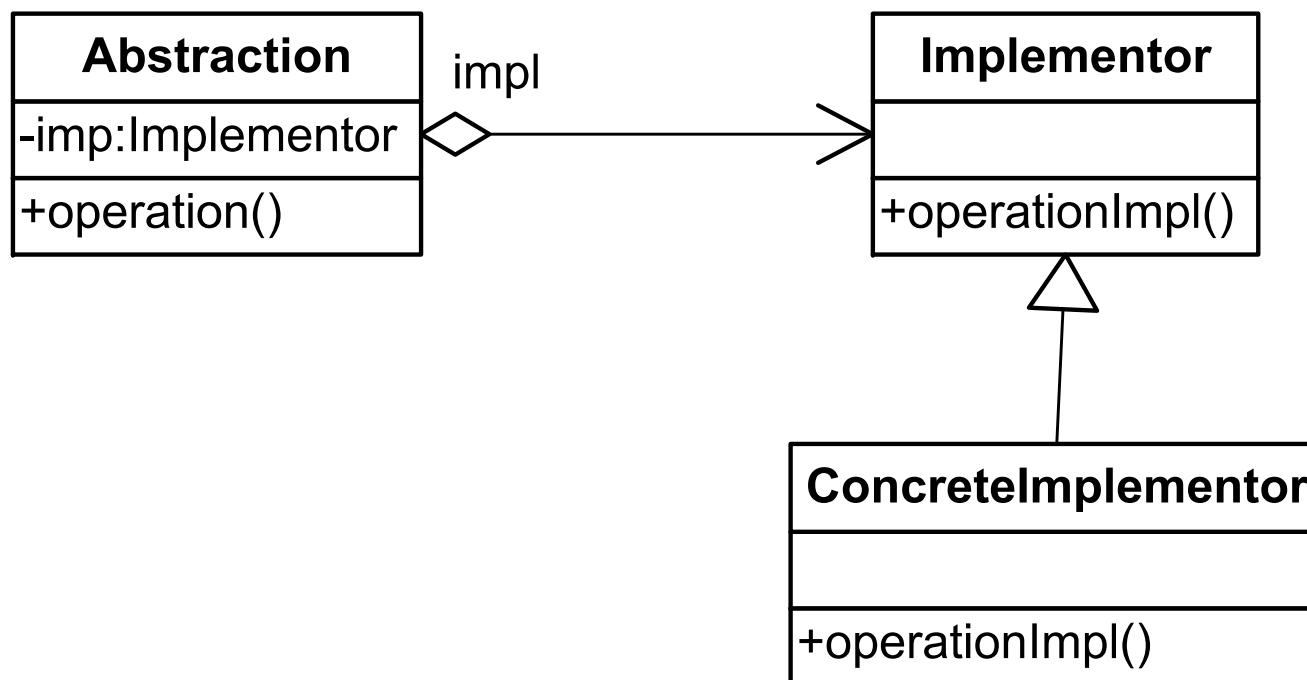
Example: Car Factory



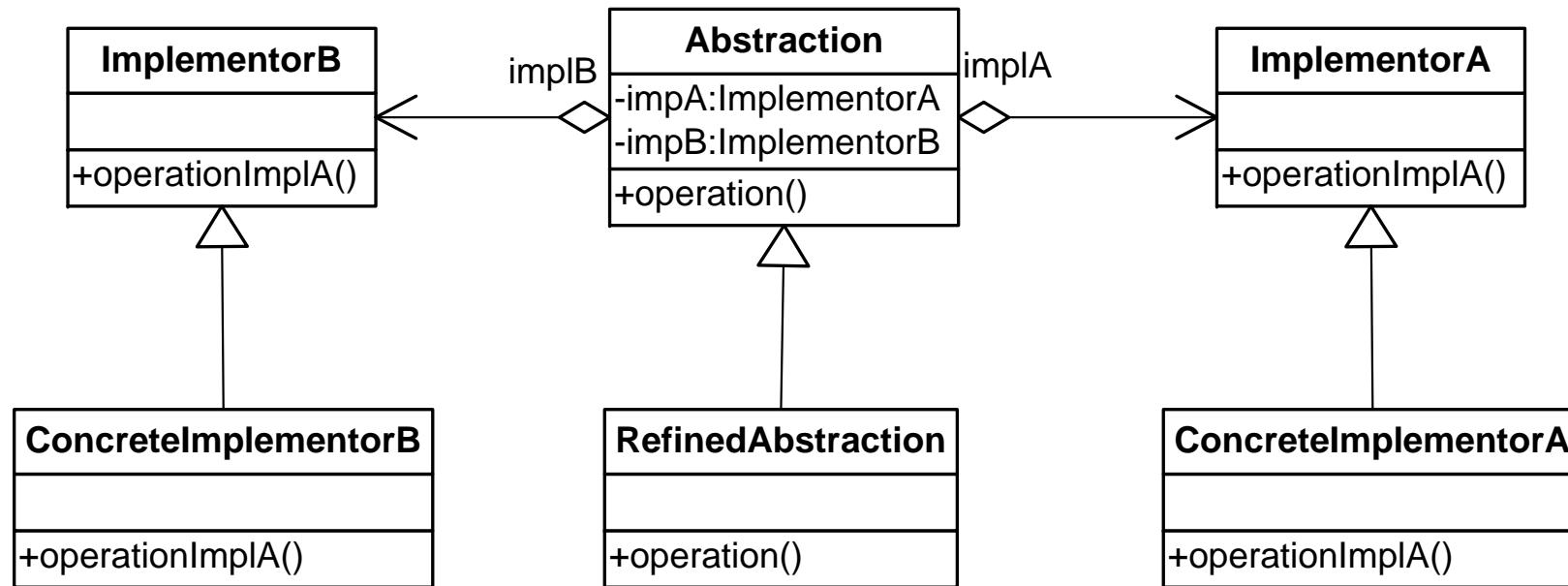
Variation 1: Implementor is omitted



Variation 2: Refined Abstraction is omitted



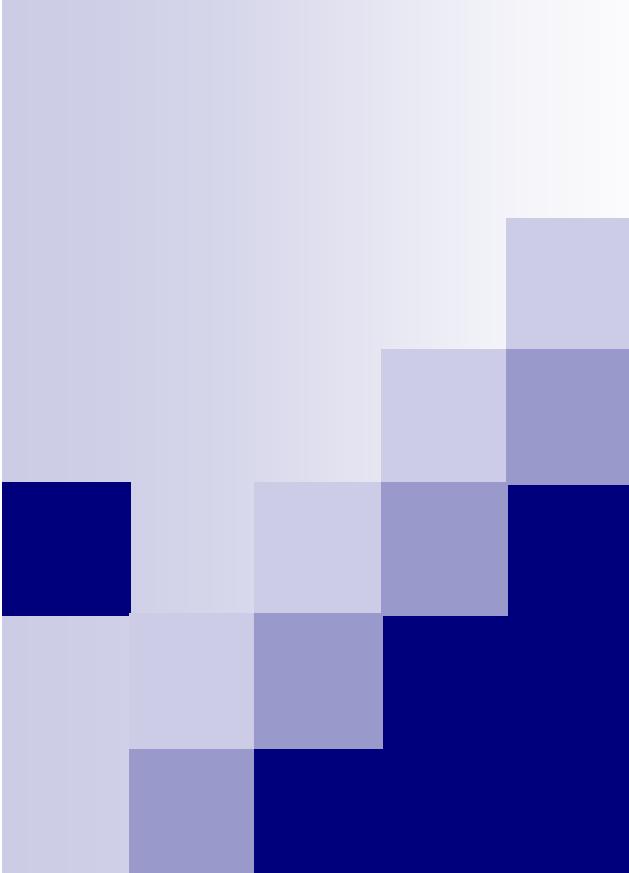
Variation 3: Sharing Implementors





Let's go to next...





Design Patterns

宋 杰

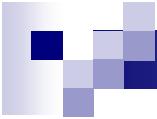
Song Jie

东北大学 软件学院

Software College, Northeastern
University



1. Strategy Pattern

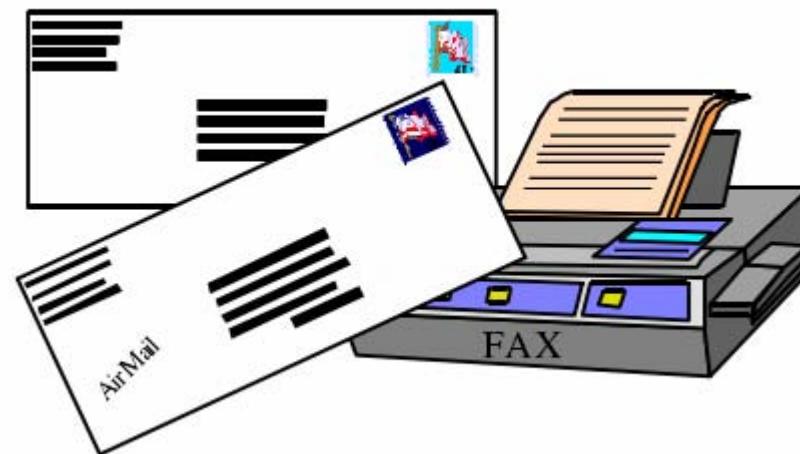


Intent

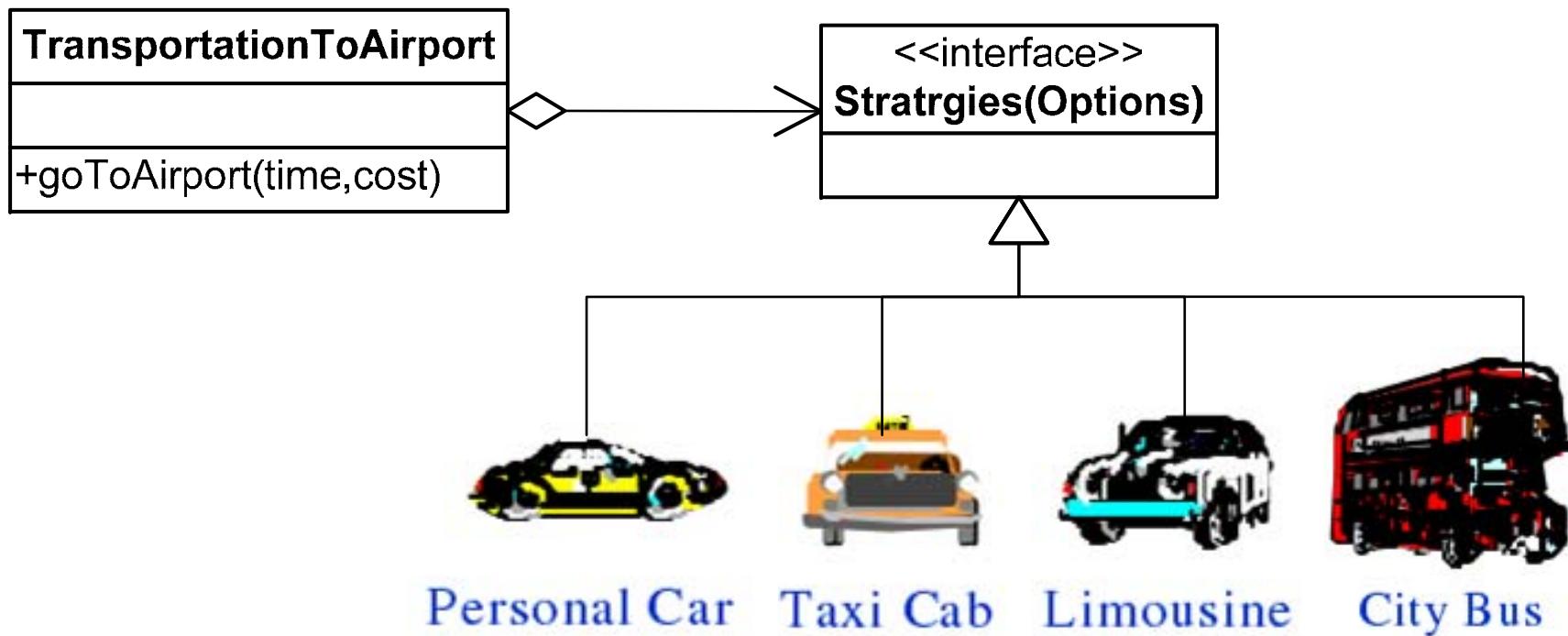
- Define a **family of algorithms**, encapsulate each one, and make them **interchangeable**. Strategy lets the **algorithm vary independently from clients that use it**.
 - 针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。
 - **Pluggable Algorithms**
-

Example

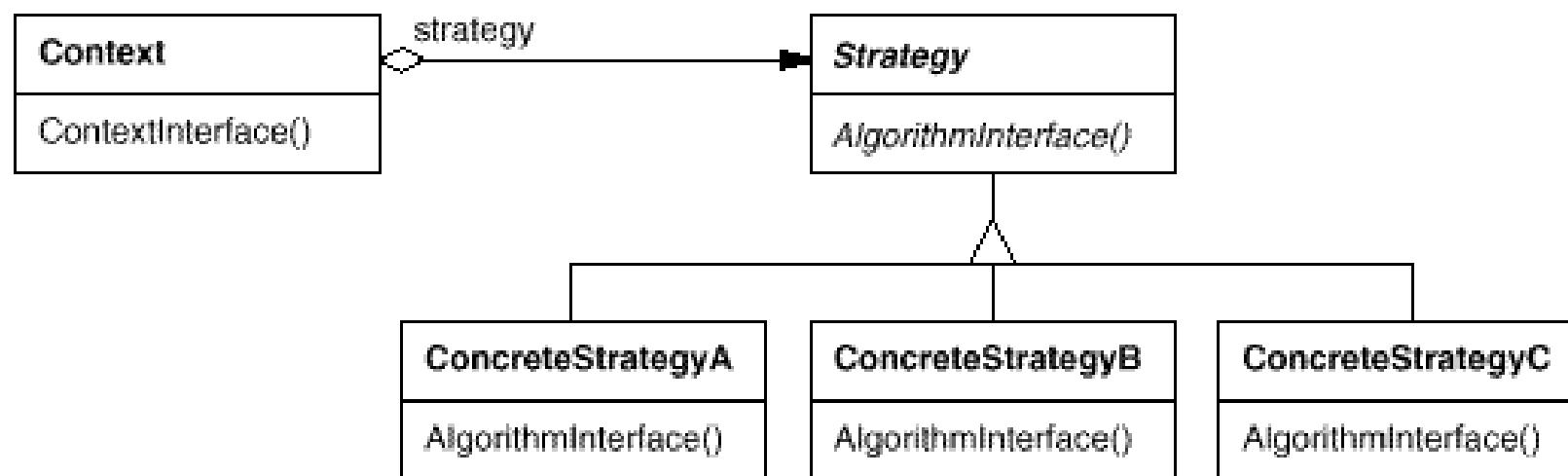
- Faxing, common mail, air-mail, and surface-mail all get a document from one place to another, but in different ways.

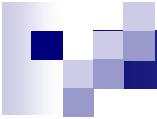


Example



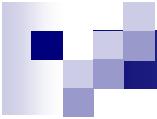
Structure





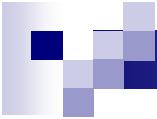
Participants

- **Strategy:** Declares an interface common to all supported algorithms. **Context** uses this interface to call the algorithm defined by a **ConcreteStrategy**.
 - **ConcreteStrategy:** Implements the algorithm using the **Strategy** interface.
 - **Context** is configured with a **ConcreteStrategy** object. maintains a reference to a **Strategy** object.
 - May define an interface that lets **Strategy** access its data.
-



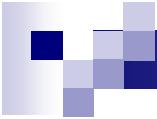
Collaborations

- **Strategy** and **Context** interact to implement the chosen algorithm.
 - A **context** may pass all data required by the algorithm to the **strategy** when the algorithm is called.
 - Alternatively, the **context** can pass itself as an argument to **strategy** operations. That lets the **strategy** call back on the **context** as required.
 - A **Context** forwards requests from its clients to its **strategy**.
 - There is often a family of **ConcreteStrategy** classes for a client to choose from.
 - Clients usually create and pass a **ConcreteStrategy** object to the **context**;
 - Clients interact with the **context** exclusively.
-



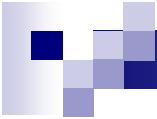
Consequences -benefits

- Families of related algorithms
 - Hierarchies of **Strategy** classes define a family of algorithms or behaviors for **contexts** to reuse.
 - An alternative to subclassing.
 - Encapsulating the algorithm in separate **Strategy** classes lets you vary the algorithm independently of its **context**, making it easier to switch, understand, and extend. (CRP)
 - Strategies eliminate conditional statements.
 - The **Strategy** pattern offers an alternative to conditional statements for selecting desired behavior.
 - A choice of implementations.
 - Strategies can provide different implementations of the same behavior. The client can choose among strategies with different time and space trade-offs.
-



Consequences-drawbacks

- Clients must be aware of different Strategies.
 - Client must understand how Strategies differ before it can select the appropriate one.
 - Communication overhead between Strategy and Context.
 - The Strategy is shared by all ConcreteStrategy classes whether they are simple or complex. Hence it's likely that some ConcreteStrategies won't use all the information passed by Context;
 - Increased number of objects.
 - Shared strategies which not maintain state across invocations can reduce the number of objects. (Flyweight)
-



Applicability

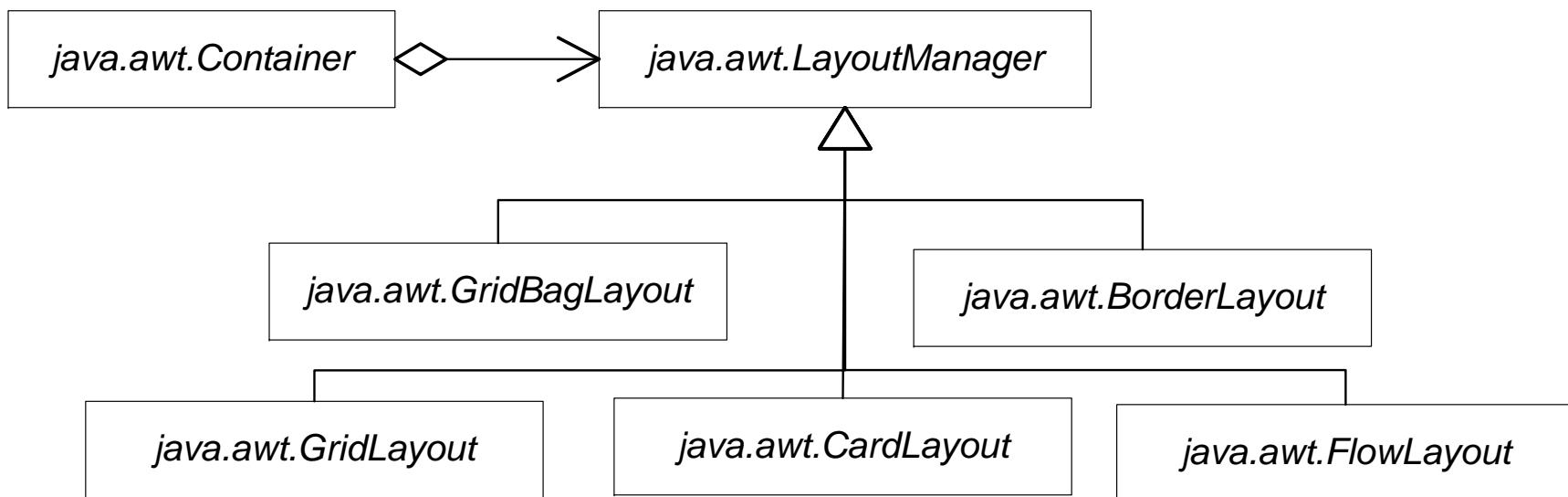
- Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
 - You need different variants of an algorithm. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.
 - For example, you might define algorithms reflecting different space/time trade-offs.
 - An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
 - A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.
-

Example 1: Promotion

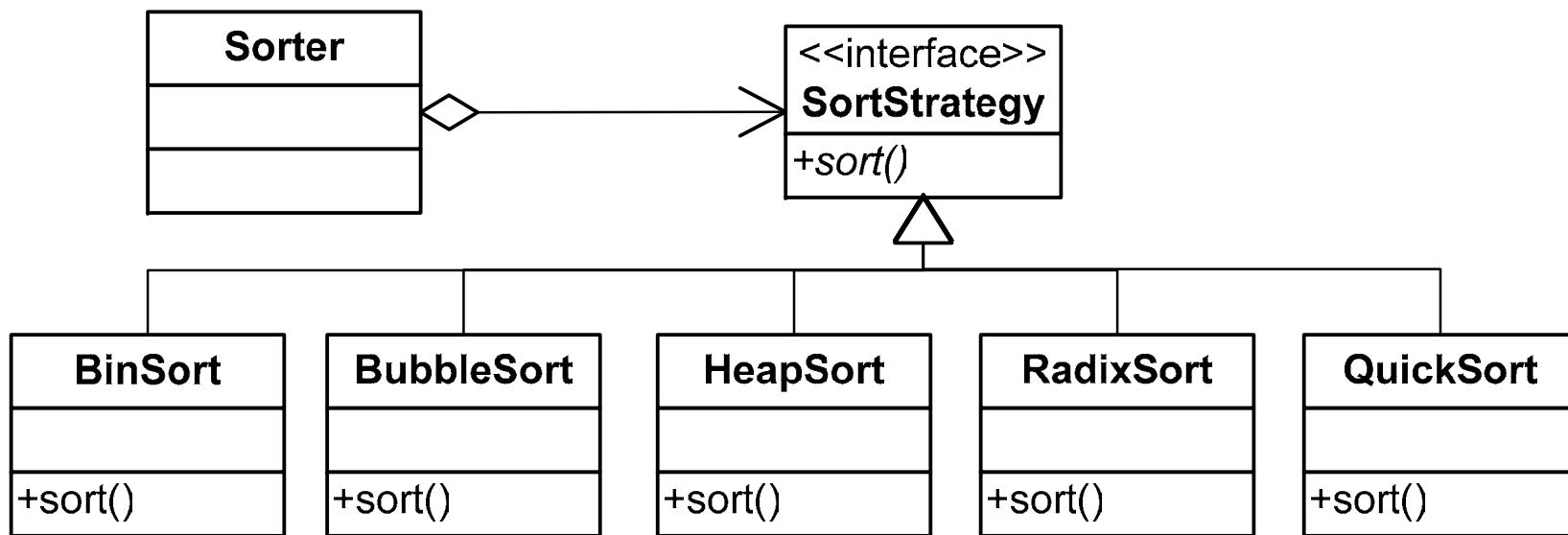
- Different commodities ,for example, books, have different discount
 - Computer books, 20%;
 - English books, 30%;
 - Children's books, 40%
 - Economic books, 0%;
 - Special book, 80%
- New discount approach may be introduced.



Example 2: LayoutManager in AWT

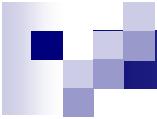


Example 3: Sorter System



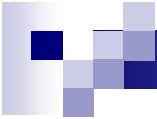
Extension 1: Passing data between Context and Strategy

- The **Strategy** and **Context** must give a **ConcreteStrategy** efficient access to any data it needs from a **context**, and vice versa.
 - One approach is to have **Context** pass data in parameters to **Strategy** operations.
 - Simple approach
 - This keeps **Strategy** and **Context** decoupled.
 - **Context** might pass data the **Strategy** doesn't need.
 - A **context** pass itself as an argument, and the **strategy** requests data from the **context** explicitly.
 - **Strategy** can store a reference to its **context**, eliminating the need to pass anything at all.
 - **Context** must define a more elaborate interface to its data, which couples **Strategy** and **Context** more closely.



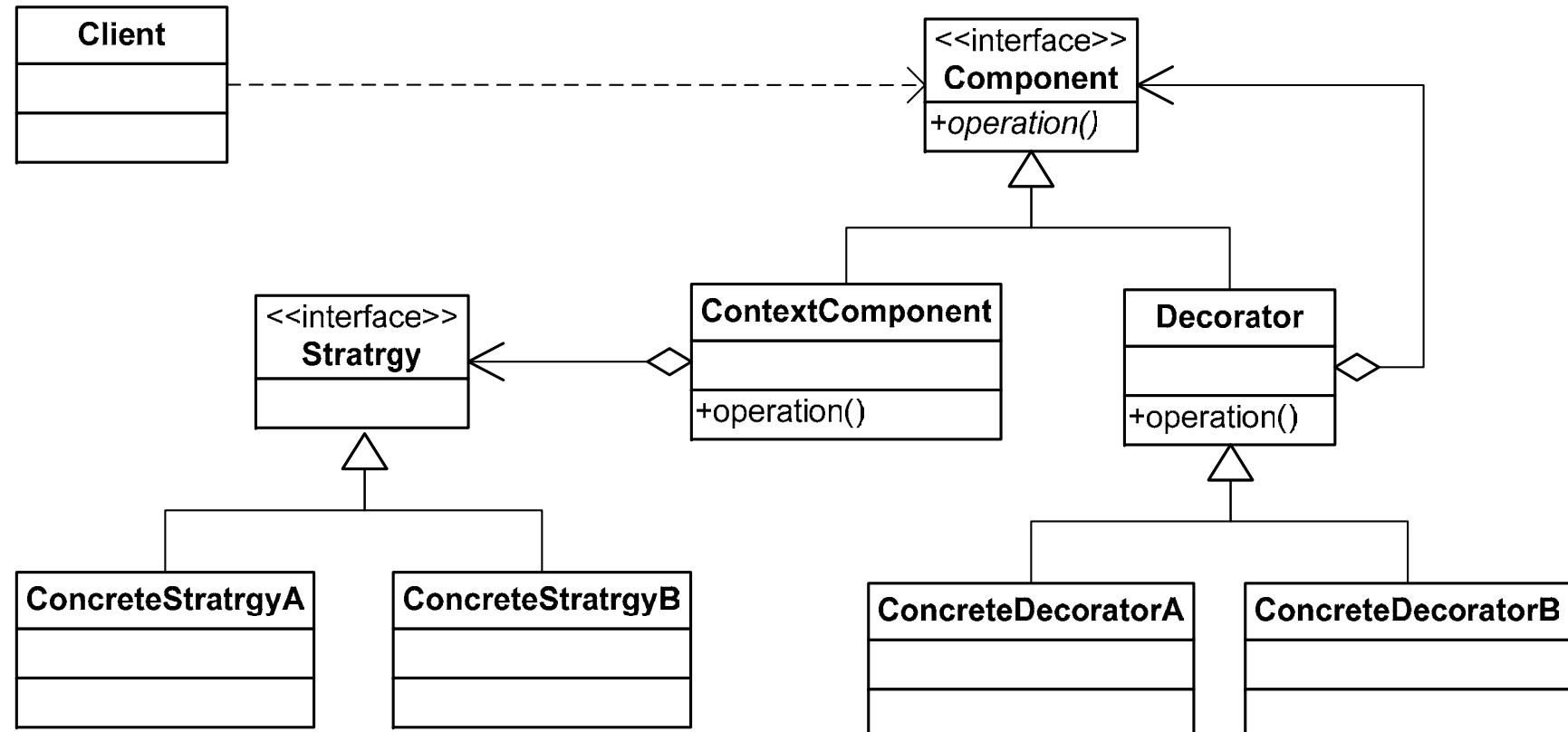
Extension 2: Making **strategy** objects optional

- The **Context** class may be simplified if it's meaningful not to have a **Strategy** object.
 - **Context** checks to see if it has a **Strategy** object before accessing it.
 - If there is one, then **Context** uses it normally.
 - If there isn't a strategy, then **Context** carries out default behavior.
 - Or it can be treated as an default **Strategy** is set up to the **Context** .
-



Think about it

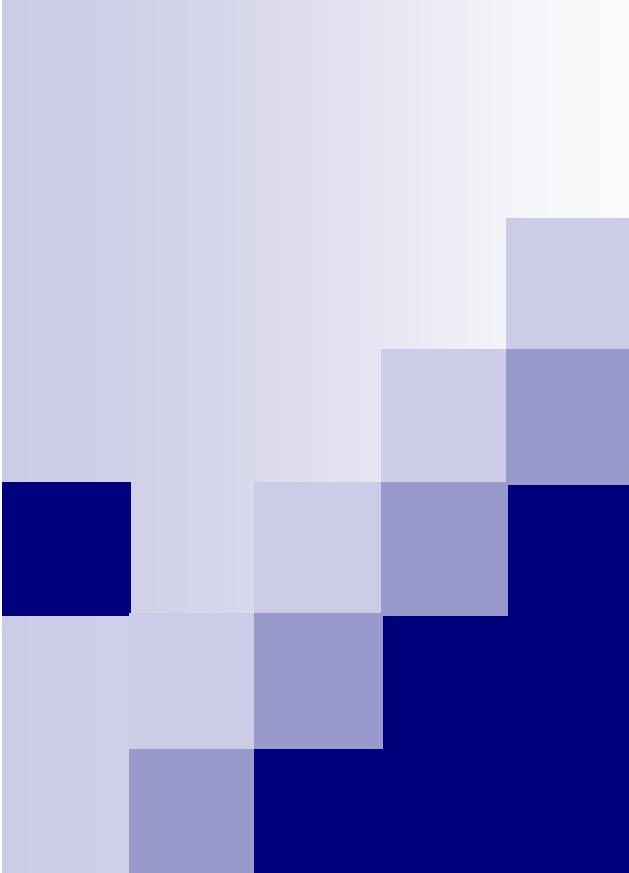
- Changing the guts of an object versus changing its skin.
 - The **Strategy pattern** is a good example of a pattern for changing the guts.
 - The **Decorator pattern** is a good example of a pattern for changing the skin.
 - Can we use **Strategy** and **Decorator** together ?
-





Let's go to next...





Design Patterns

宋 杰

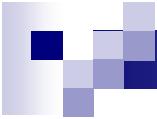
Song Jie

东北大学 软件学院

Software College, Northeastern
University



14. Template Method Pattern

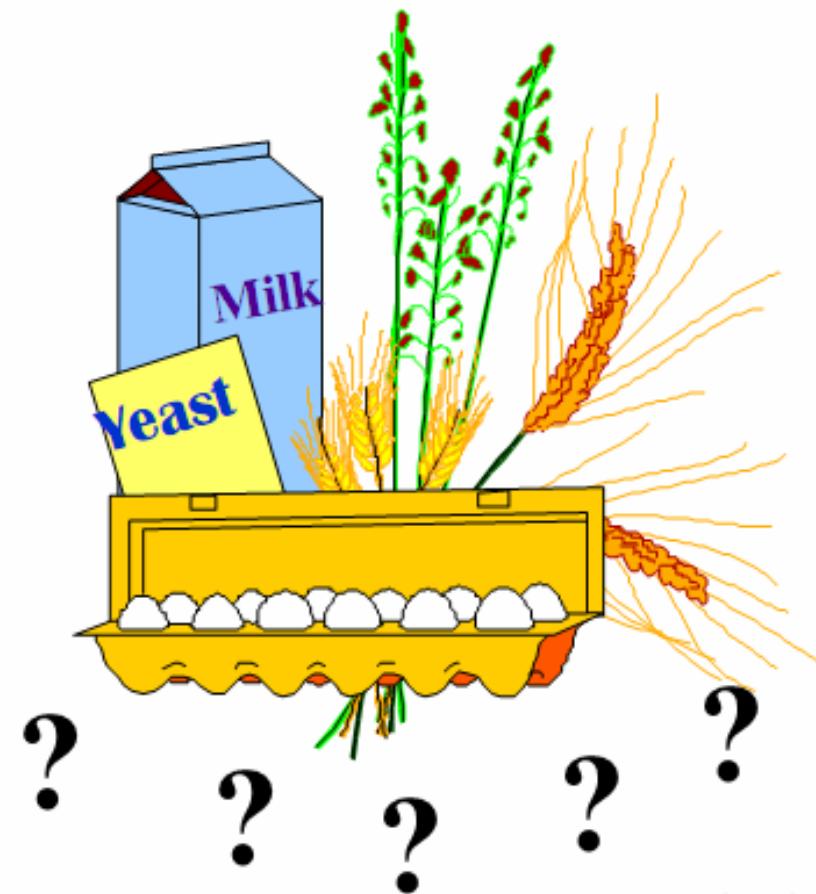


Intent

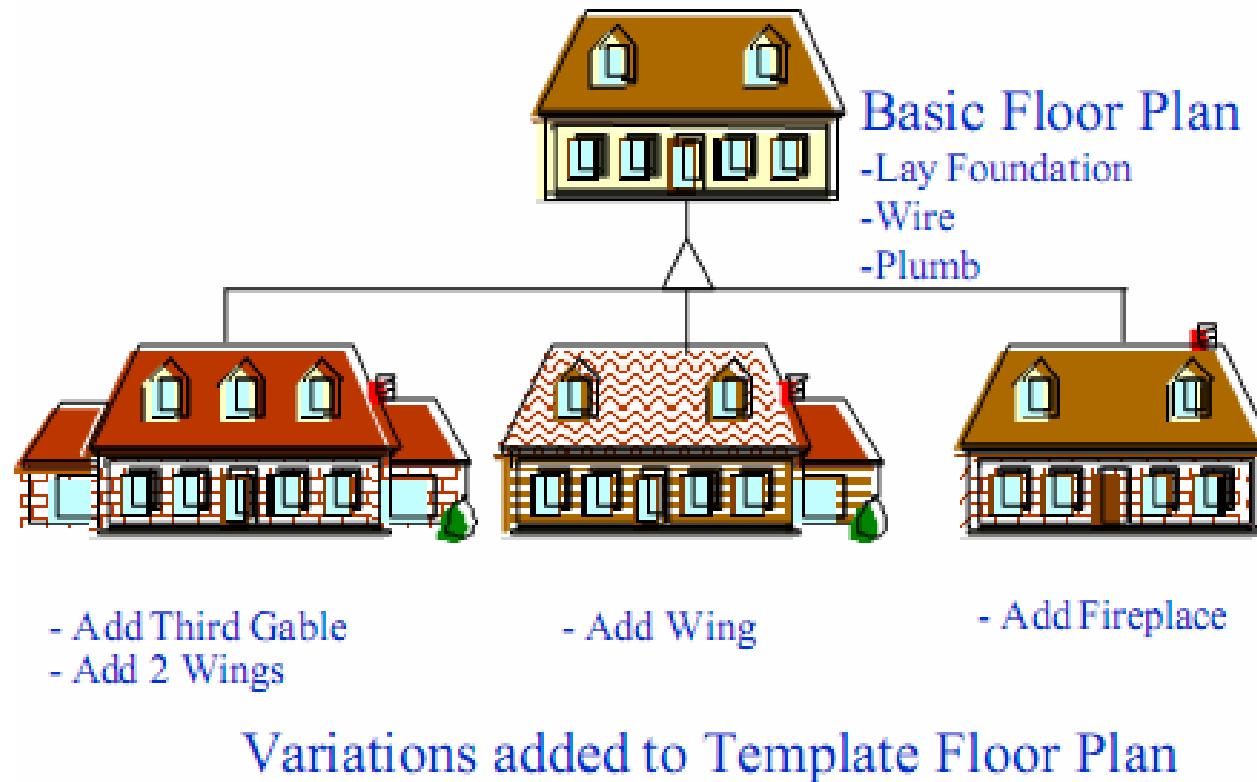
- Define the **skeleton** of an algorithm in an operation, deferring **some steps to subclasses**. Template Method lets subclasses **redefine certain steps of an algorithm without changing the algorithm's structure**.
- **Java 实现:** 准备一个抽象类，将部分逻辑以具体方法以及具体构造子的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。

Example

- Once a basic bread recipe (烹饪法) is developed, additional steps, such as adding cinnamon(肉桂), raisins(葡萄干), nuts, peppers(胡椒粉), cheese, etc. can be used to create different types of bread.



Example



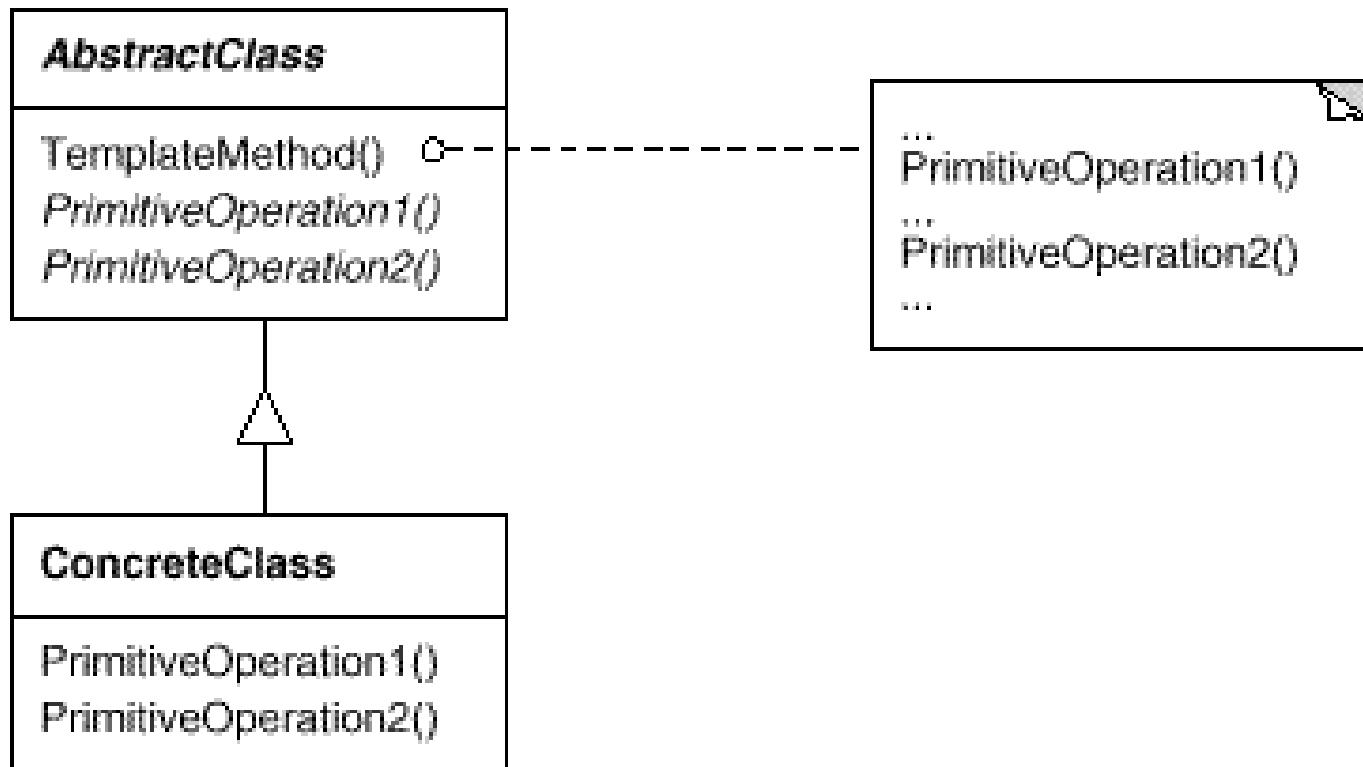
Example

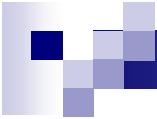
```
abstract class Sorter {  
    // template method  
    public final void sort(List<Object> target) {  
        int length = target.size();  
        for (int i = 0; i < length; i++) {  
            Object former = target.get(i);  
            for (int j = i + 1; j < length; j++) {  
                Object later = target.get(j);  
                // call the primitive operations  
                if (isIncreasedSort() != isFormerLessThanLater(former, later)) {  
                    // switch elements  
                    target.set(i, later);  
                    target.set(j, former);  
                    former = later;  
                }  
            }  
        }  
    }  
  
    protected abstract boolean isIncreasedSort();  
    protected abstract boolean isFormerLessThanLater(Object former, Object later);  
}
```

Example

```
class TextSorter extends Sorter {  
    protected boolean isIncreasedSort() {  
        return true;  
    }  
  
    protected boolean isFormerLessThanLater(Object former, Object later) {  
        return former.toString().compareTo(later.toString()) < 0;  
    }  
}
```

Structure





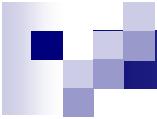
Participants

■ **AbstractClass**

- Defines abstract **primitive operations** that concrete subclasses define to implement steps of an algorithm.
- Implements a **template method** defining the skeleton of an algorithm. The **template method** calls primitive operations as well as operations defined in **AbstractClass**.

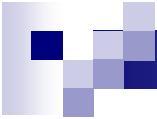
■ **ConcreteClass**

- Implements the primitive operations to carry out subclass-specific steps of the algorithm.
-



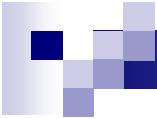
Consequences

- Template methods are a fundamental technique for **code reuse**.
 - Template methods are particularly important in class libraries, because they are the means for factoring out (分解) common behavior in library classes.
-



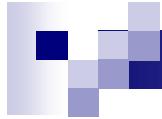
Hollywood principle

- Template methods lead to an inverted control structure (DIP) that's sometimes referred to as "**the Hollywood principle**," that is
 - "Don't call us, we'll call you".
 - This refers to how a parent class calls the operations of a subclass and not the other way around.
-



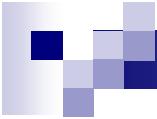
Applicability

- To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
 - Refactoring to generalize.
 - When common behavior among subclasses should be factored and localized in a common class to avoid code duplication.
 - Control subclasses extensions.
 - You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points.
- 



Implementation 1: Naming conventions

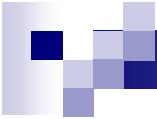
- Identify the operations that should be overridden by adding a prefix to their names.
 - For example, the prefixes template method names with "do-": "doCreateDocument()", "doRead()"
-



Implementation 2: Using access control

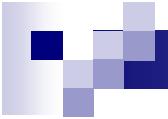
■ In Java

- The primitive operations can be declared as **protected** and **abstract** method;
 - The template method can be declared as **final** method.
-



Implementation 3: Minimizing primitive operations

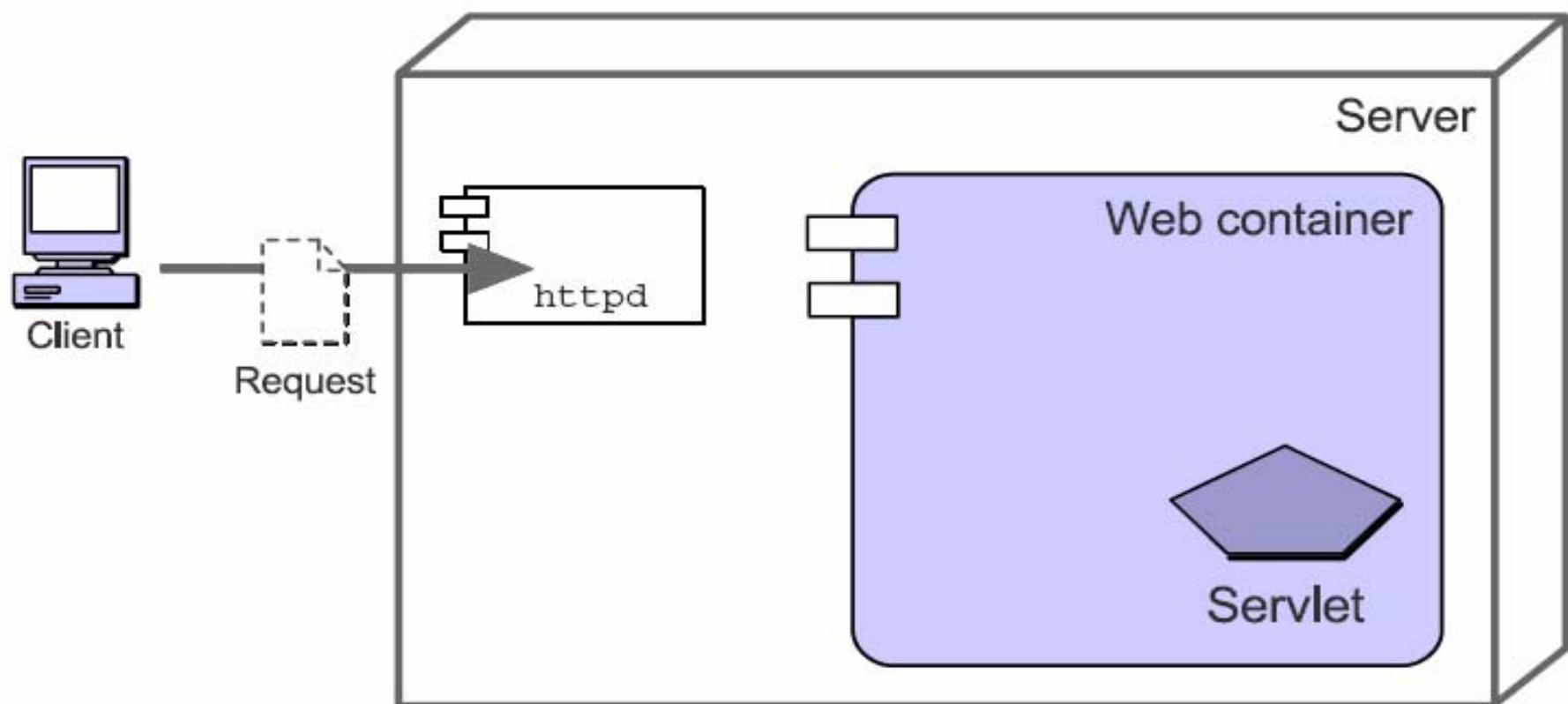
- An important goal in designing template methods is **to minimize the number of primitive operations** that a subclass must **override** to flesh out the algorithm.
 - The more operations that need overriding, the more tedious things get for clients.
-



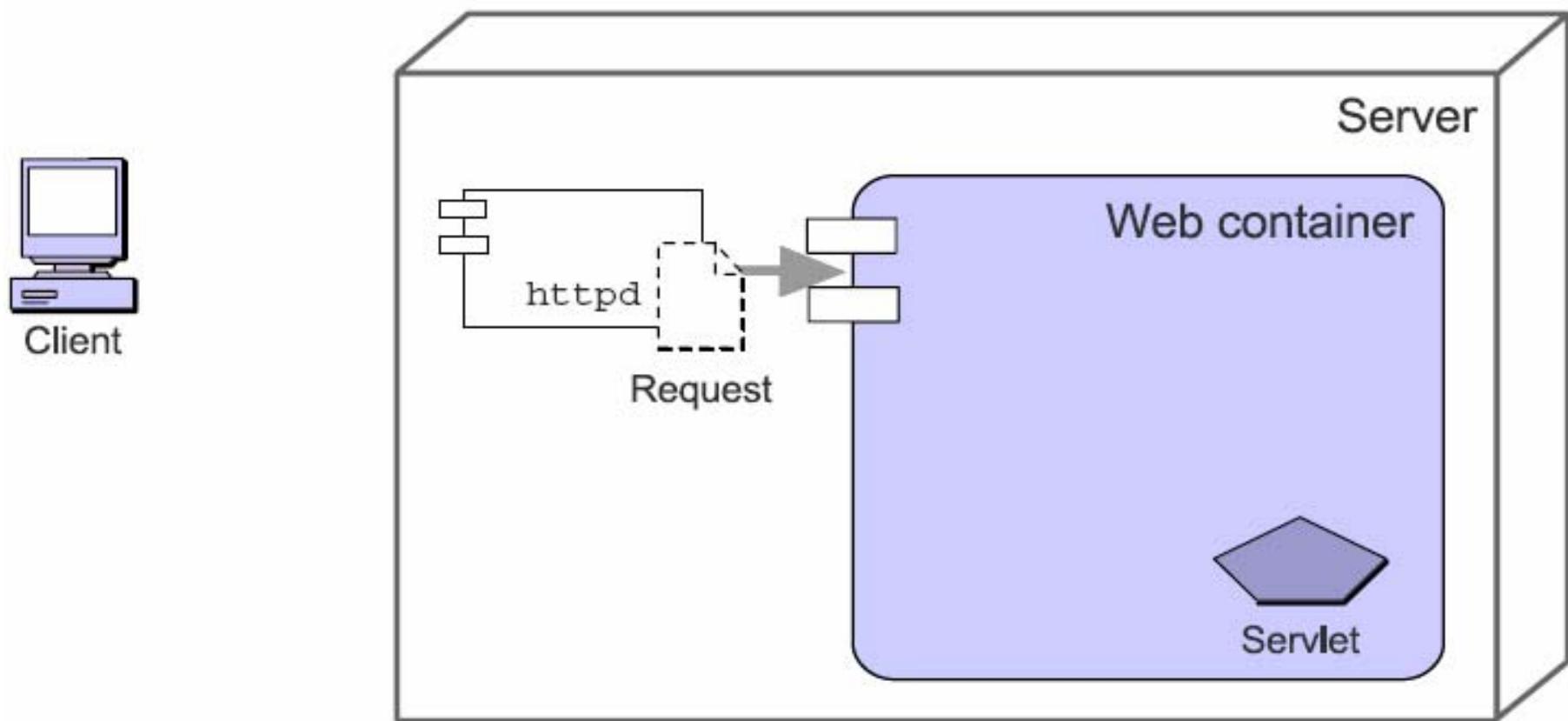
Example: HttpServlet in JavaEE

- **HttpServlet** use **Template Method Pattern** a lot;
 - The subclass of **HttpServlet** is used to process the http request in different according to its request method (type).
-

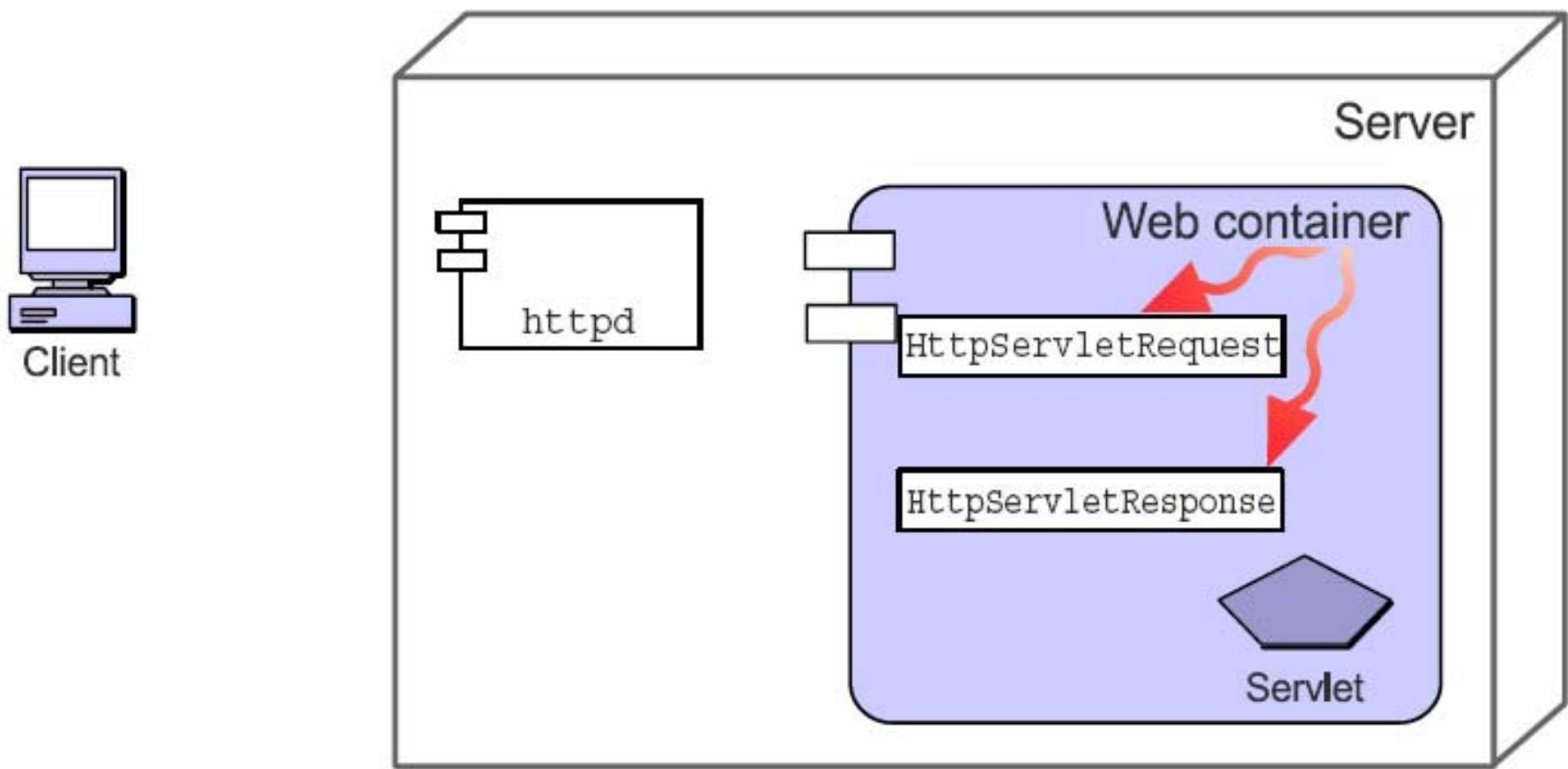
Step 1: Client send the request



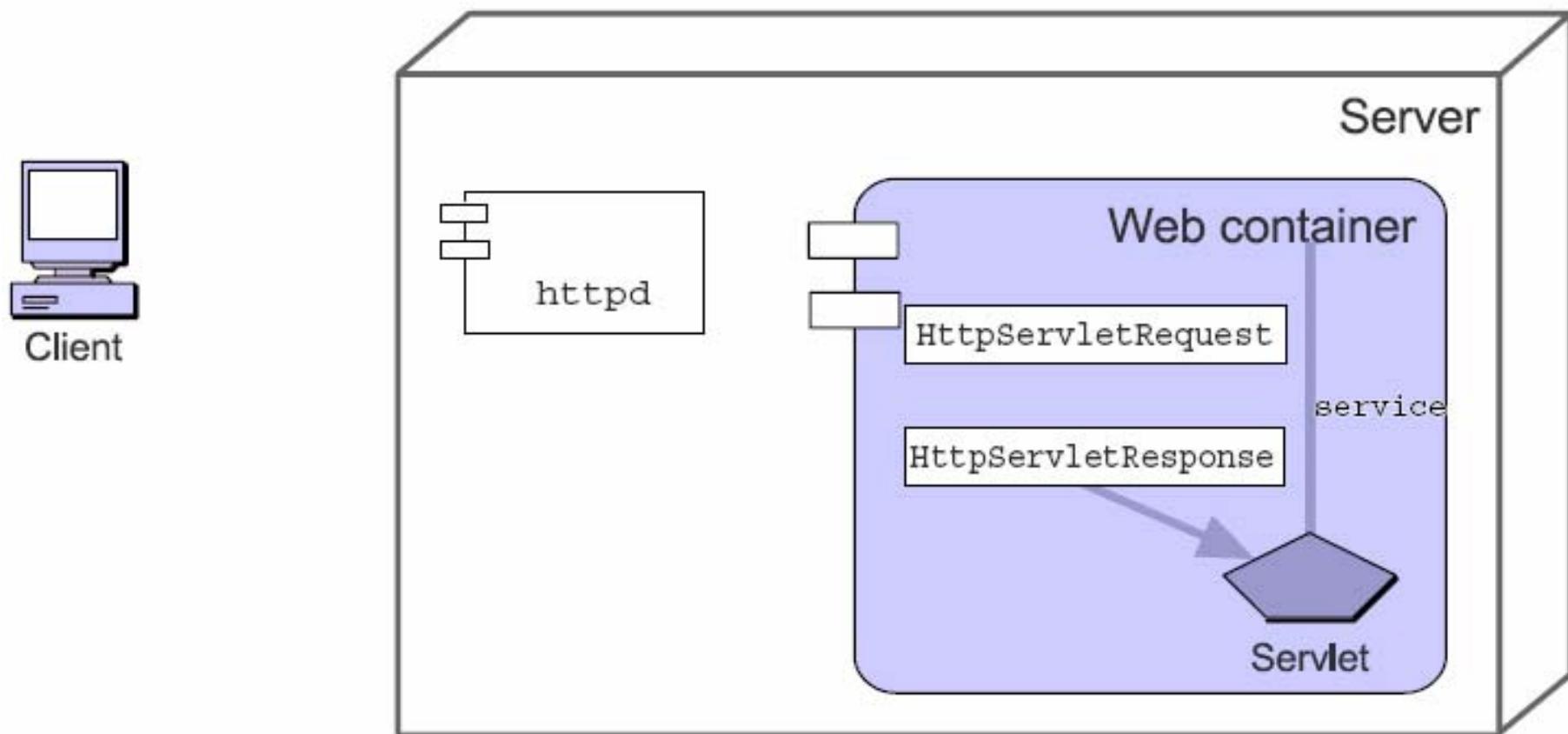
Step 2: Web server send the request to the Web container



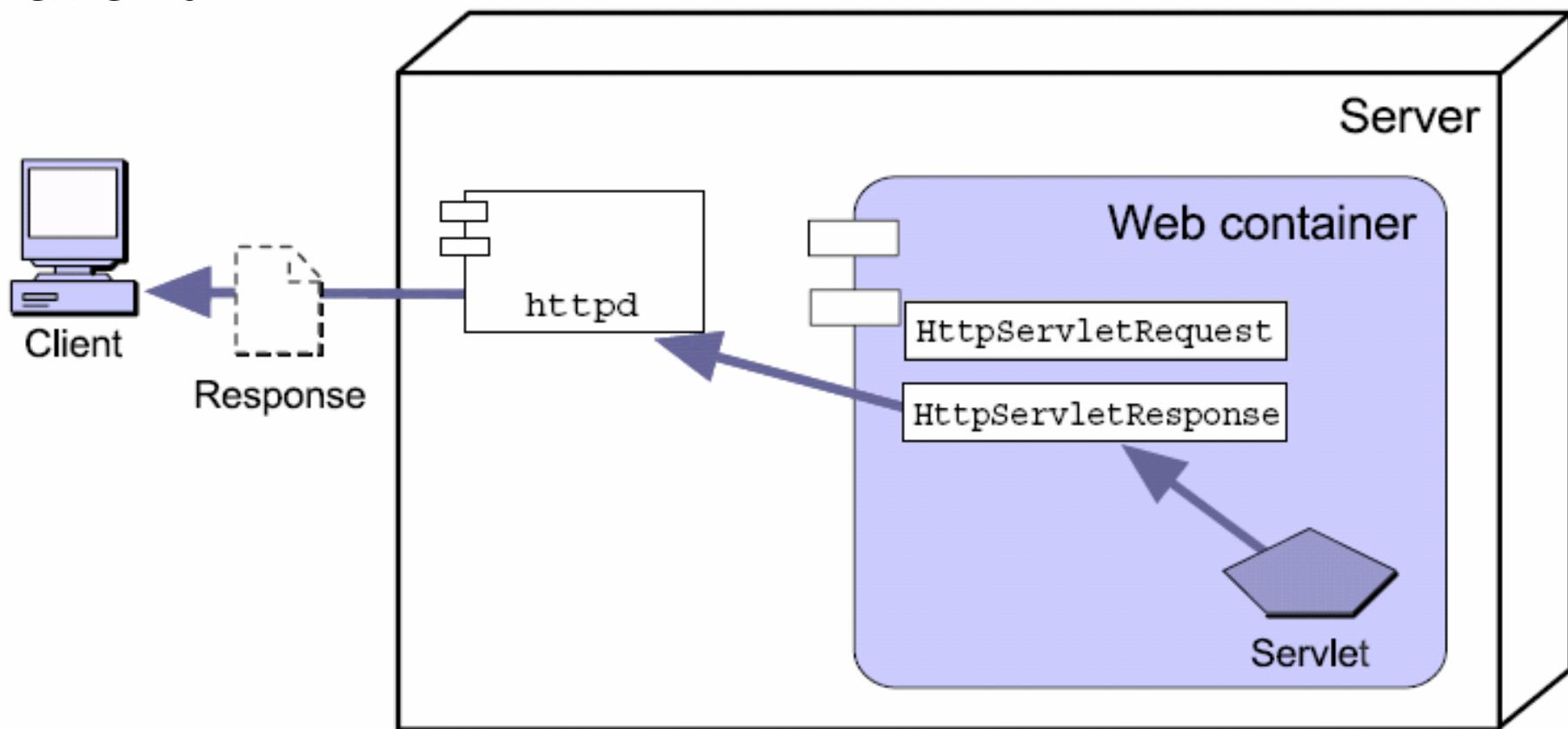
Step 3: Web container initializes the request and response object



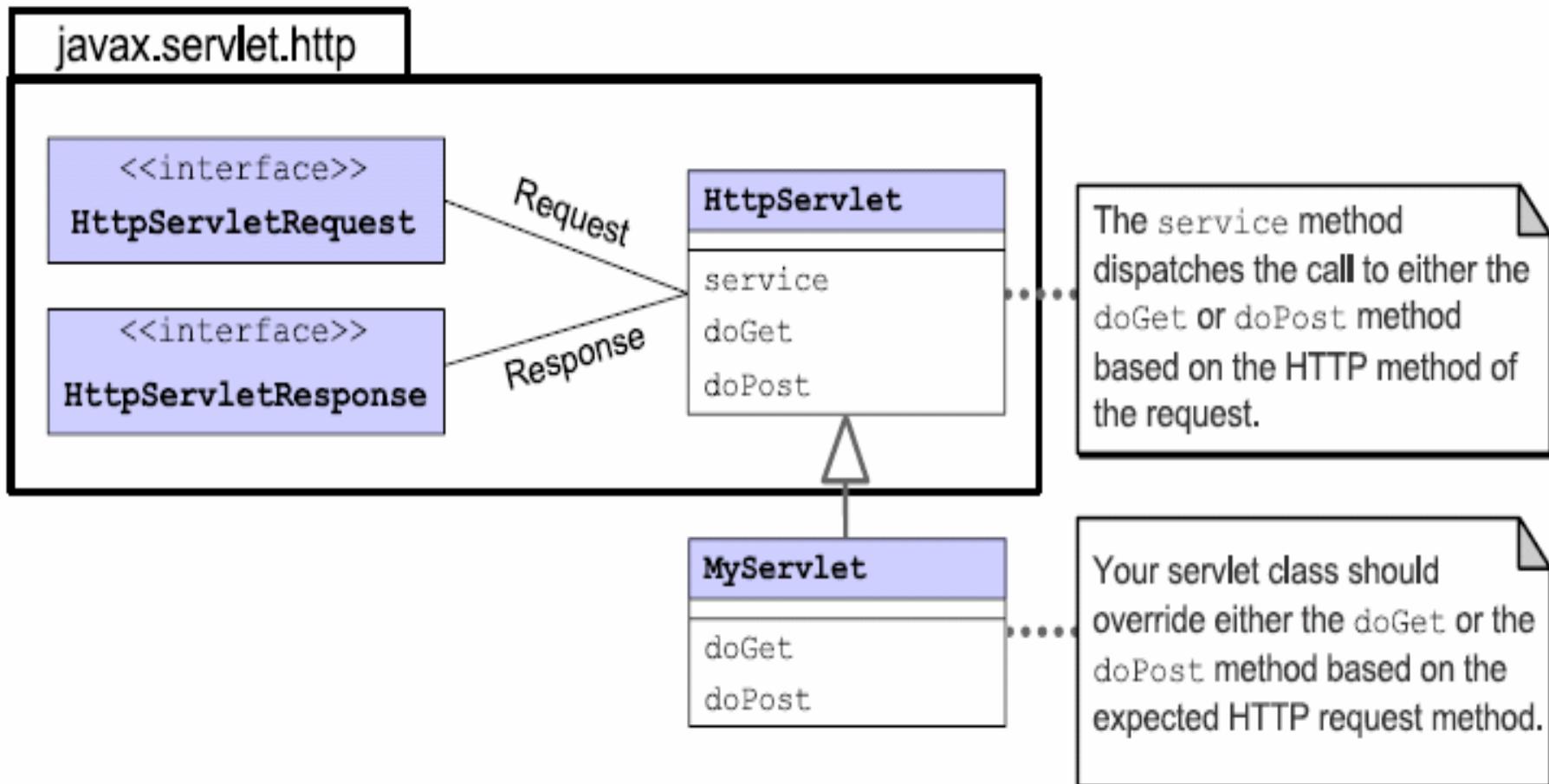
Step 4: Web container invokes the Servlet according to the request URL

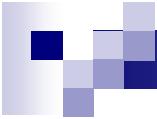


Step 5: Web container return the response, which is modified by the target Servlet, to the client



HttpServlet

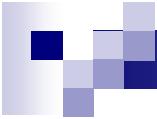




doXxx method in HttpServlet

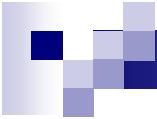
- There are many types of request.
 - In service method, the types of request is determined, and corresponding *doXxx* method is invoked
 - `doGet`
 - `doPost`
 - `doPut`
 - `doDelete`
 - `doOptions`
 - `doTrace`
-

```
        if (method.equals(METHOD_GET)) {
            long lastModified = getLastModified(req);
            if (lastModified == -1) {
                // servlet doesn't support if-modified-since, no reason
                // to go through further expensive logic
                doGet(req, resp);
            } else {
                long ifModifiedSince = req.getDateHeader(HEADER_IFMODSINCE);
                if (ifModifiedSince < (lastModified / 1000 * 1000)) {
                    // If the servlet mod time is later, call doGet()
                    // Round down to the nearest second for a proper compare
                    // A ifModifiedSince of -1 will always be less
                    maybeSetLastModified(resp, lastModified);
                    doGet(req, resp);
                } else {
                    resp.setStatus(HttpServletResponse.SC_NOT_MODIFIED);
                }
            }
        } else if (method.equals(METHOD_HEAD)) {
            long lastModified = getLastModified(req);
            maybeSetLastModified(resp, lastModified);
            doHead(req, resp);
        } else if (method.equals(METHOD_POST)) {
            doPost(req, resp);
        } else if (method.equals(METHOD_PUT)) {
            doPut(req, resp);
        } else if (method.equals(METHOD_DELETE)) {
            doDelete(req, resp);
        } else if (method.equals(METHOD_OPTIONS)) {
            doOptions(req, resp);
        } else if (method.equals(METHOD_TRACE)) {
            doTrace(req, resp);
        } else {
```



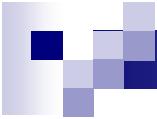
Extension 1: Method in template method pattern

- Template methods
- Primitive methods
 - Concrete methods;
 - Abstract methods;
 - Hook methods
 - Which provide default behavior that subclasses can extend if necessary.
 - A hook operation often does nothing by default.



Extension 2: Refactor in template method pattern

- Defining a class according to its **behaviors**, not **states**. That is, the implementation of a class should firstly base on its behaviors instead of its states
 - An exception is the value object;
 - Using **abstract** state instead of **concrete state** for implementations.
 - Using **indirect reference** instead of **direct reference**. That is, if a behavior involves a state of an object. Using access method instead of using property directly.
-



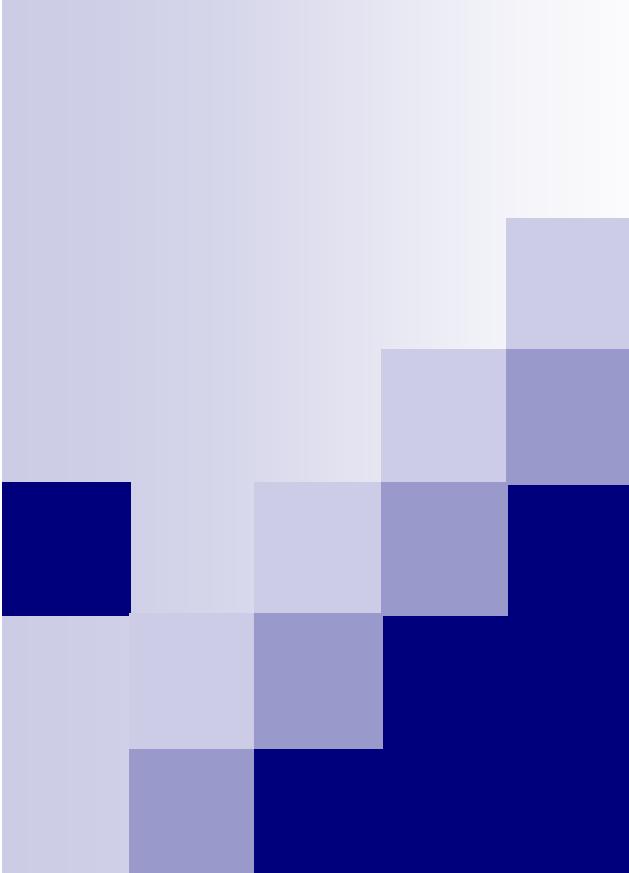
Extension 2: Refactor in template method pattern

- **Layering the operations.** The behaviors of an class should be distributed in a group of kernel methods (primitive methods) , thus they can be replaced easily in the subclass.
 - **Postponing the implementation of the state in the subclass.** Do not declare the properties in the abstraction (try to using interface and abstract class to present the abstraction, instead of concrete class).
 - In the abstraction, if the states are defined, **using abstract accessed methods** to access the states, and let subclass implements them.
-



Let's go to next...





Design Patterns

宋 杰

Song Jie

东北大学 软件学院

Software College, Northeastern
University

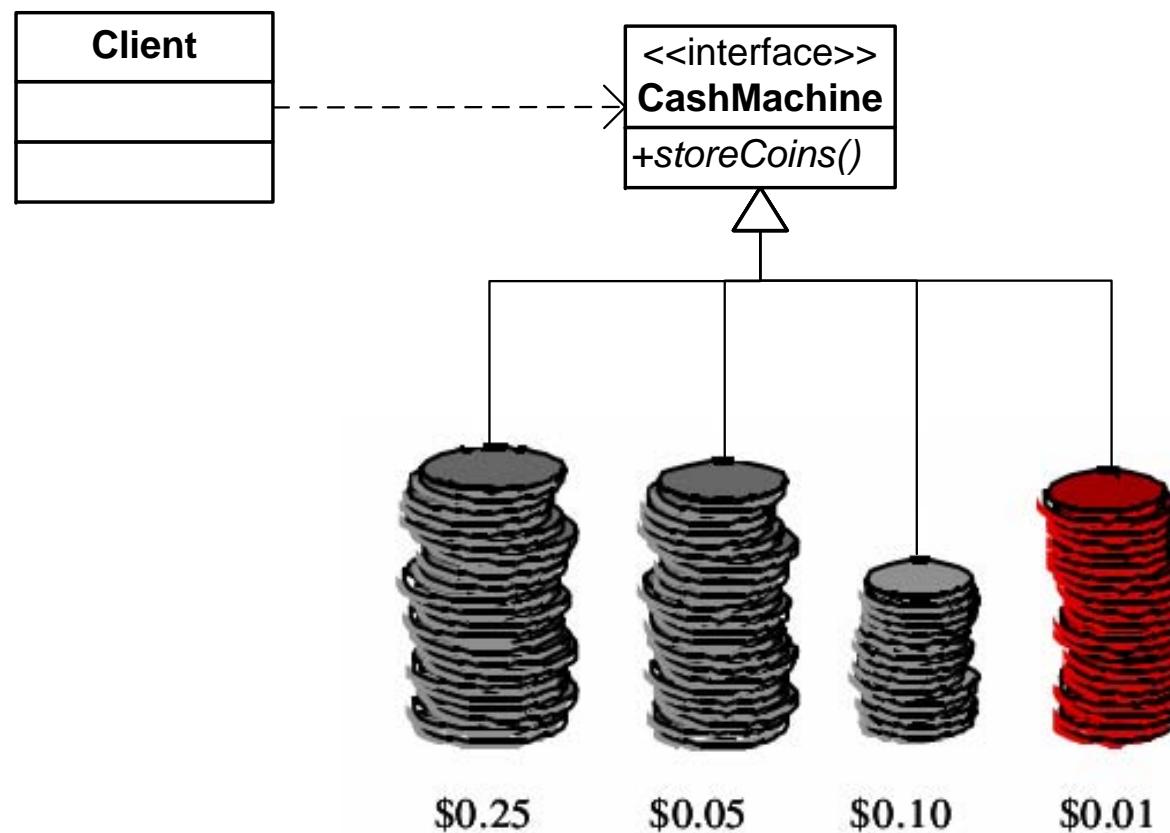


15. Chain Of Responsibility Pattern

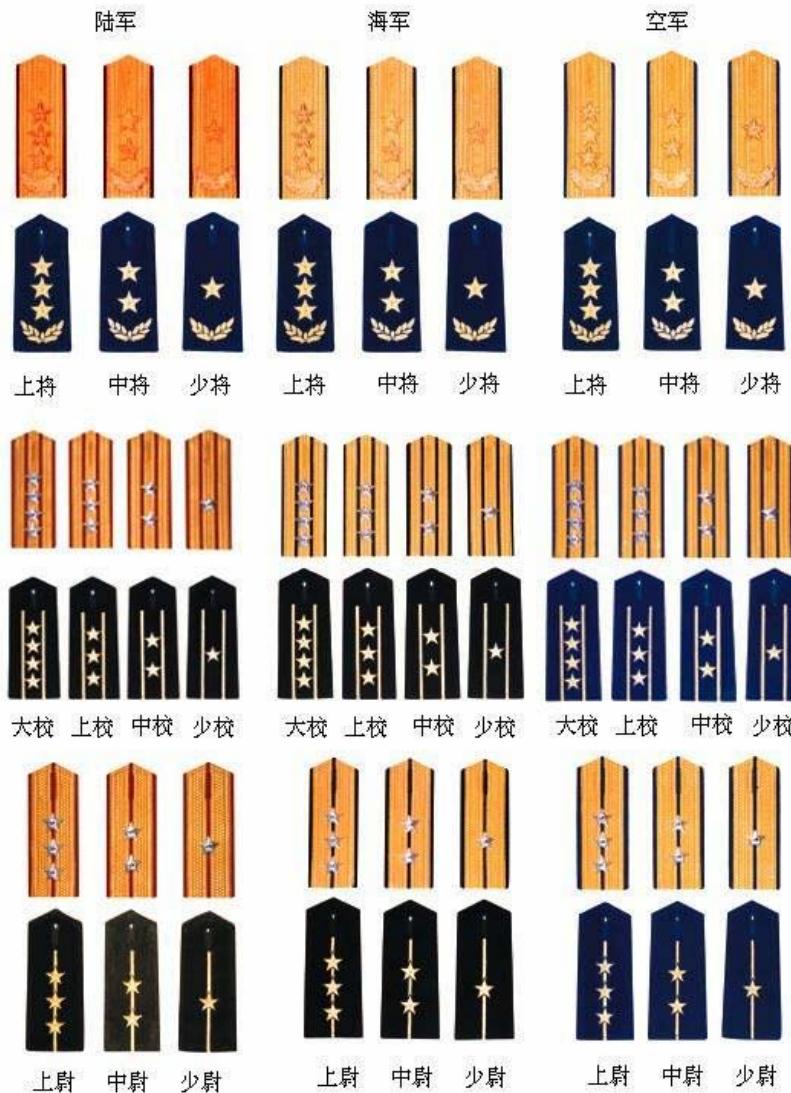
Intent

- Avoid coupling the **sender** of a request to its **receiver** by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- The chain of responsibility could be a line, a ring, a hierarchy or a graph.
- 在责任链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织链和分配责任。

Example: Cash Machine



Example: Rank in military



```
interface Coin {  
    public int getParValue();  
    public void setParValue(int parValue);  
}  
  
class RMBCoin implements Coin {  
    private int parValue;  
    public RMBCoin() {  
    }  
    public RMBCoin(int parValue) {  
        this.parValue = parValue;  
    }  
    public int getParValue() {  
        return parValue;  
    }  
    public void setParValue(int parValue) {  
        this.parValue = parValue;  
    }  
}
```

```
abstract class CoinCollector {
    private CoinCollector collector;
    public CoinCollector(CoinCollector next) {
        this.collector = next;
    }
    protected final CoinCollector next() {
        return collector;
    }
    public final void collect(Coin coin) {
        if (matched(coin)) {
            store(coin);
        } else {
            next().collect(coin);
        }
    }
    protected abstract boolean matched(Coin coin);
    protected abstract void store(Coin coin);
}
```

```
class OneCoinCollector extends CoinCollector {  
    private static final int PAR_VALUE = 1;  
    public OneCoinCollector(CoinCollector next) {  
        super(next);  
    }  
    protected boolean matched(Coin coin) {  
        return coin.getParValue() == PAR_VALUE;  
    }  
    protected void store(Coin coin) {  
        // store 1-valued Coin  
    }  
}
```

```
class FiveCoinCollector extends CoinCollector {  
    private static final int PAR_VALUE = 5;  
    public FiveCoinCollector(CoinCollector next) {  
        super(next);  
    }  
    protected boolean matched(Coin coin) {  
        return coin.getParValue() == PAR_VALUE;  
    }  
    protected void store(Coin coin) {  
        // store 5-valued Coin  
    }  
}
```

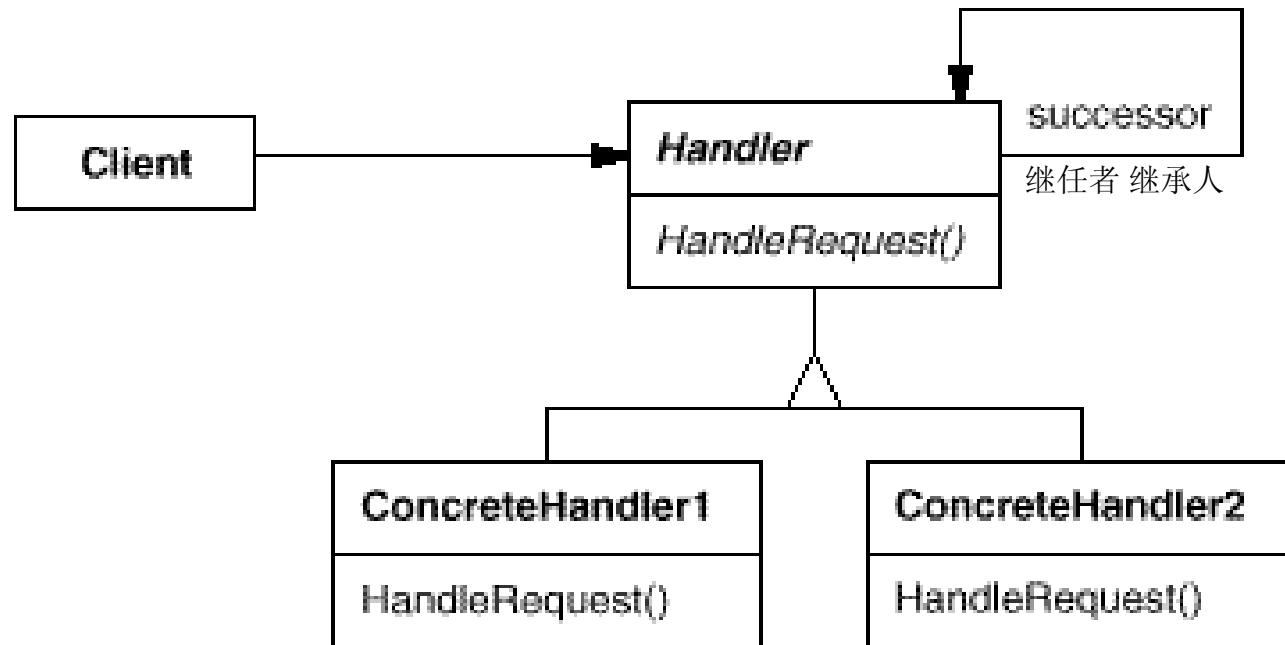
```
class TenCoinCollector extends CoinCollector {  
    private static final int PAR_VALUE = 10;  
    public TenCoinCollector(CoinCollector next) {  
        super(next);  
    }  
    protected boolean matched(Coin coin) {  
        return coin.getParValue() == PAR_VALUE;  
    }  
    protected void store(Coin coin) {  
        // store 10-valued Coin  
    }  
}
```

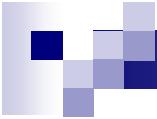
```
class UnmatchedCoinCollector extends CoinCollector {  
    public UnmatchedCoinCollector(CoinCollector next) {  
        super(next);  
    }  
    protected boolean matched(Coin coin) {  
        return true;  
    }  
    protected void store(Coin coin) {  
        // reject unmatched Coin  
    }  
}
```

```
class CoinClient {
    public void testCoinCollector() {
        CoinCollector coinCollector =
            new OneCoinCollector(
                new FiveCoinCollector(
                    new TenCoinCollector(
                        new UnmatchedCoinCollector(null))));

        Coin coin = new RMBCoin(7);
        coinCollector.collect(coin);
    }
}
```

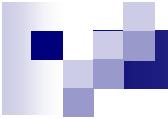
Structure





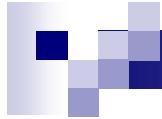
Participants

- **Handler**
 - Defines an interface for handling requests.
 - (Optional) Implements the successor link.
 - **ConcreteHandler**
 - Handles requests it is responsible for.
 - If the **ConcreteHandler** can handle the request, it does so; otherwise it forwards the request to its successor.
 - **Client**
 - Initiates the request to a **ConcreteHandler** object on the chain.
-



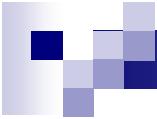
Collaborations

- When a client issues a request, the request propagates along the chain until a **ConcreteHandler** object takes responsibility for handling it.
-



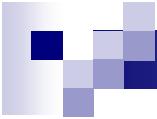
Consequences -benefits

- Reduced coupling, (nothing need to know).
 - The client is free from knowing which object handles a request.
 - The client only has to know that a request will be handled "appropriately."
 - Both the receiver and the sender have no explicit knowledge of each other.
 - An handler in the chain doesn't have to know about the chain's structure.
- Added flexibility in assigning responsibilities to objects.
 - You can combine this with subclassing to specialize handlers statically.
 - You can add or change responsibilities for handling a request by adding to or otherwise changing the chain at run-time.



Consequences-drawbacks

- Receipt isn't guaranteed. Since a request has no explicit receiver, there's no guarantee it'll be handled
 - The request can fall off the end of the chain without ever being handled.
 - A request can also go unhandled when the chain is not configured properly.
 - The returned value is possible but not straightforward;
 - All handler must confirm the same interface.
-



Applicability

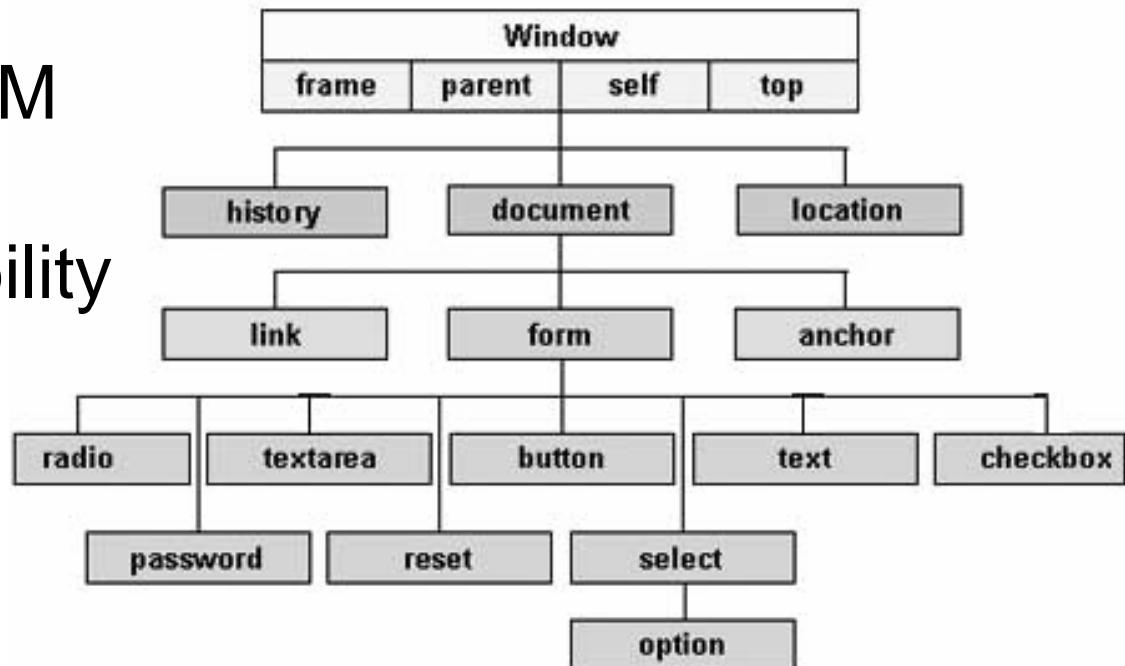
- More than one object may handle a request, and the handler isn't known *a priori*. The handler should be ascertained automatically.
 - You want to issue a request to one of several objects without specifying the receiver explicitly.
 - The set of objects that can handle a request should be specified dynamically.
-

Example: Event handler model in DOM (Document Object Model)

- DOM (Document Object Model)

- The event handle mechanism of DOM in Brower adopt chain of responsibility pattern

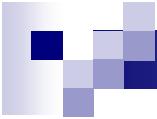
- Netscape
- Internet Explorer



DOM

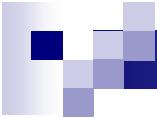
The screenshot shows a web browser displaying the Google homepage. The browser interface includes a navigation bar with links like Web, Images, Groups, News, Froogle, Local, and more. Below the navigation bar is a toolbar with various icons. The main content area shows the Google logo and search bar. On the left side, there is a sidebar containing a tree view of the Document Object Model (DOM) structure. The tree view lists elements such as <HTML>, <HEAD>, <BODY>, <FORM>, <SCRIPT>, and <A>. To the right of the DOM tree is a code editor window showing the rendered HTML source code. The code includes standard HTML tags and some CSS inline styles. A status bar at the bottom provides navigation controls.

```
1 <HTML><HEAD><TITLE>Google</TITLE>
2 <META http-equiv=content-type content="tex
3 <STYLE>BODY (
4     FONT-FAMILY: arial,sans-serif
5 )
6 TD (
7     FONT-FAMILY: arial,sans-serif
8 )
9 A (
10    FONT-FAMILY: arial,sans-serif
```



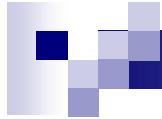
Example: Netscape

- Event handle mechanism in Netscape is named **Event Capturing** by invoking methods from top to bottom.
 - `captureEvent()`
 - `releaseEvents()`
- Event handle mechanism in InternetExplorer is named **Event Bubbling** by invoking method from bottom to top.
 - `onXxx()`
 - `cancelBubble = true`



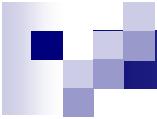
Extension 1: Representing requests

- Invoking an operation directly
 - In the simplest form, the request is a hard-coded operation invocation.
 - This is convenient and safe;
 - But you can forward only the fixed set of requests that the **Handler** class defines.
-



Extension 1: Representing requests

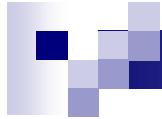
- Using the encoded request code
 - Use a single handler function that takes a request code (e.g., an integer constant or a string) as parameter.
 - The sender and receiver agree on how the request should be encoded.
 - This approach is more flexible, but it requires conditional statements for dispatching the request based on its code.
 - Moreover, there's no type-safe way to pass parameters, so they must be packed and unpacked manually.
 - Obviously this is less safe than invoking an operation directly.
-



Extension 1: Representing requests

■ Separate request objects

- Using separate request objects that bundle request parameters.
 - A **Request** class can represent requests explicitly, and new kinds of requests can be defined by subclassing. Subclasses can define different parameters.
 - Handlers must know the kind of request (that is, which **Request** subclass they're using) to access these parameters.
-



Extension 2: Complete and incomplete CoR (chain of responsibility)

■ Complete CoR

- A ConcreteHandler can only choice to take the responsibility or forward the responsibility to the successor;
- The request must be handled by one and only one handler.

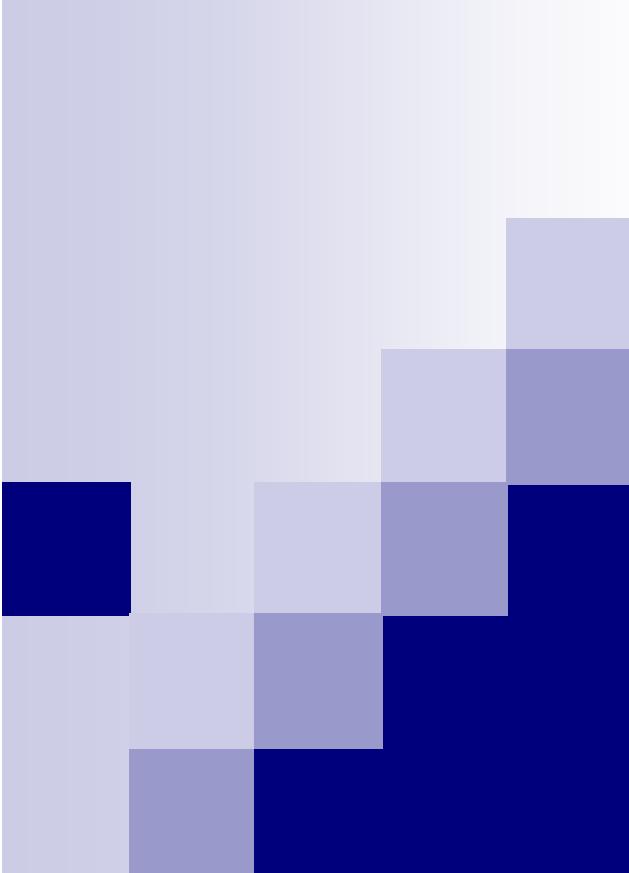
■ Incomplete CoR

- A ConcreteHandle can only choice to take the responsibility partly, then forward the responsibility to the successor;
 - The request must be handled by many or nor handler.
-



Let's go to next...





Design Patterns

宋 杰

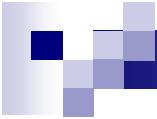
Song Jie

东北大学 软件学院

Software College, Northeastern
University



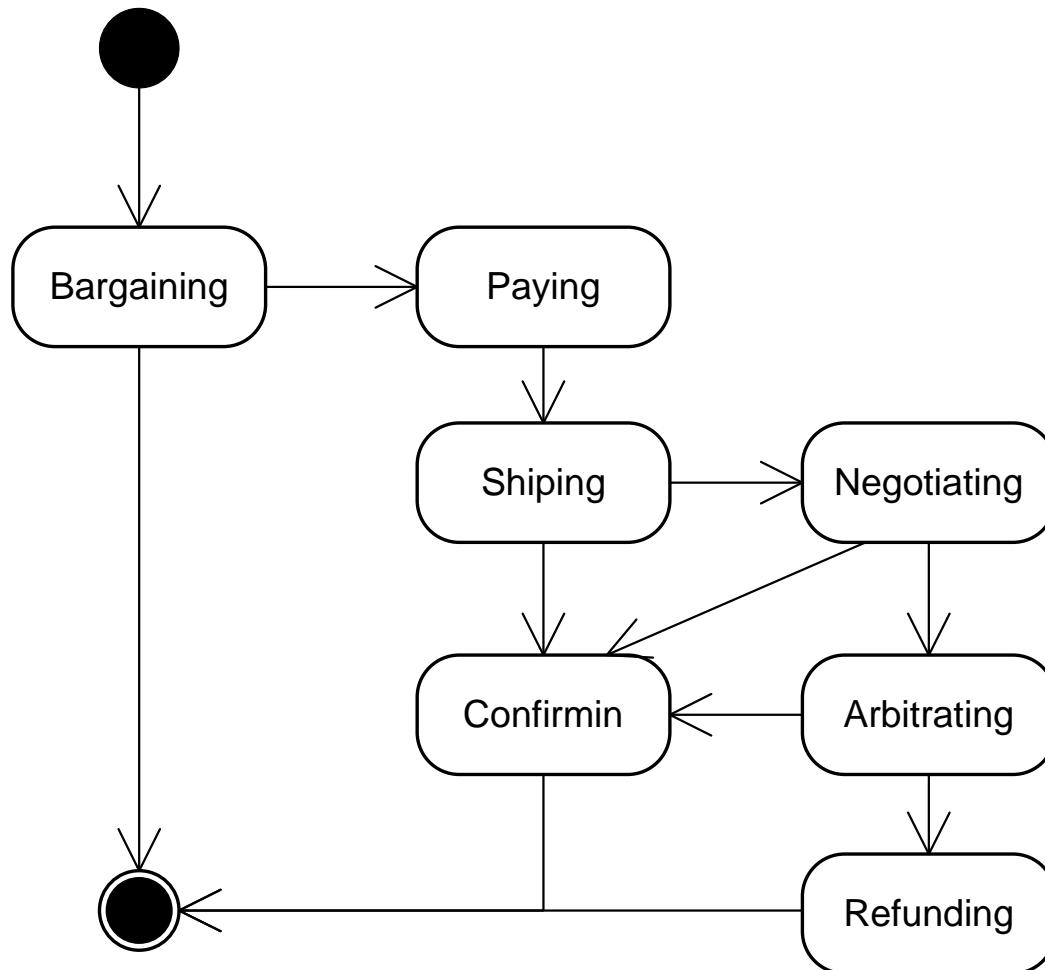
16. State Pattern



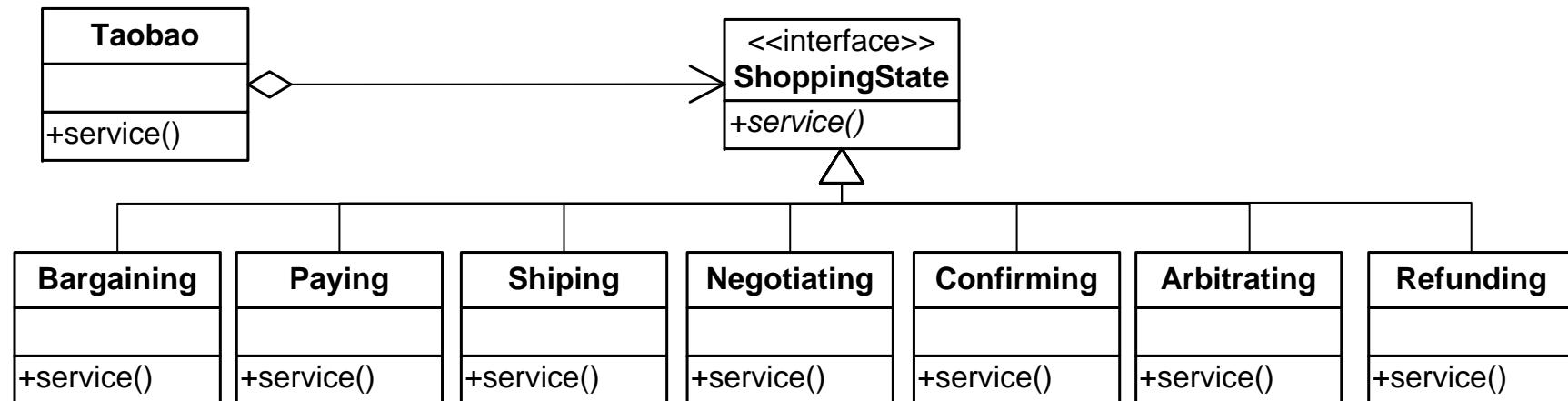
Intent

- Allow an object to alter its **behavior** when its **internal state** changes. The object will appear to change its class.
 - 状态模式允许一个对象在其内部状态改变的时候改变其行为。这个对象看上去就像改变了它的类一样。
-

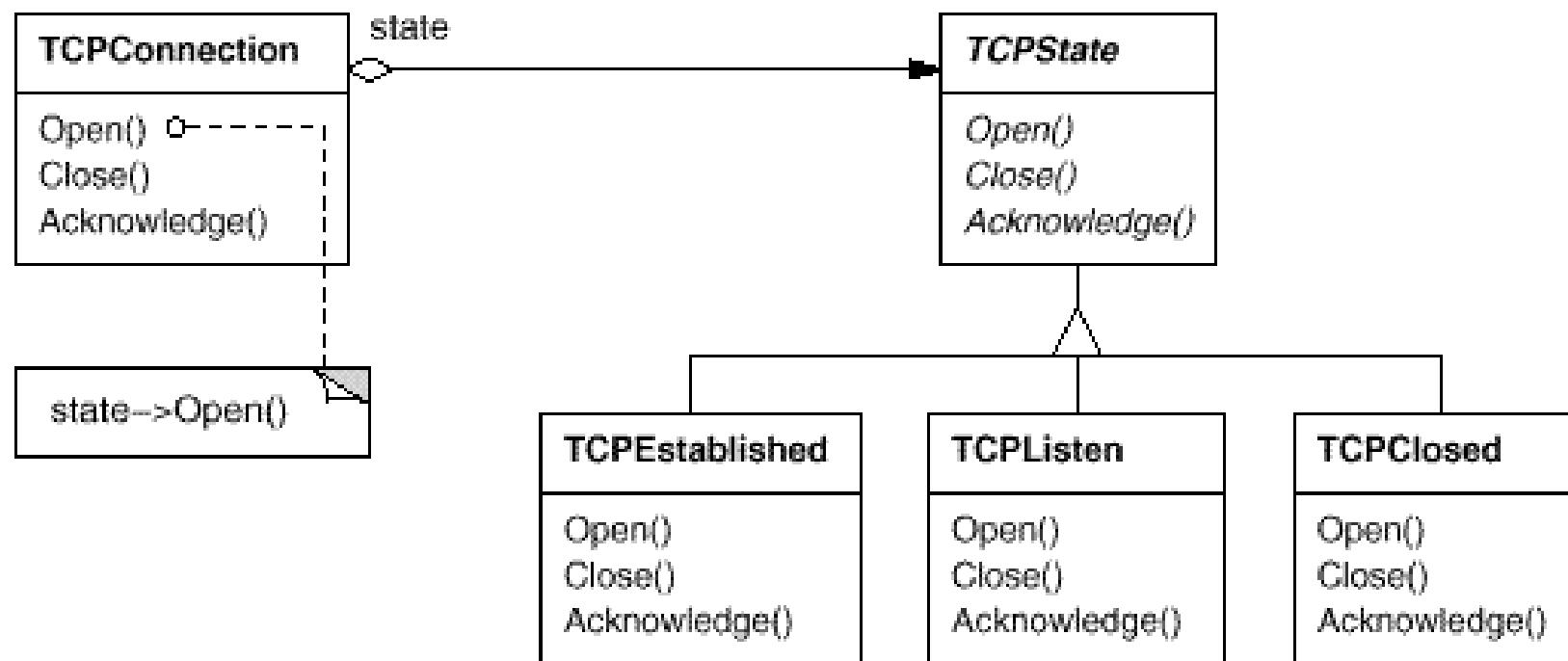
Example: Shopping in Taobao



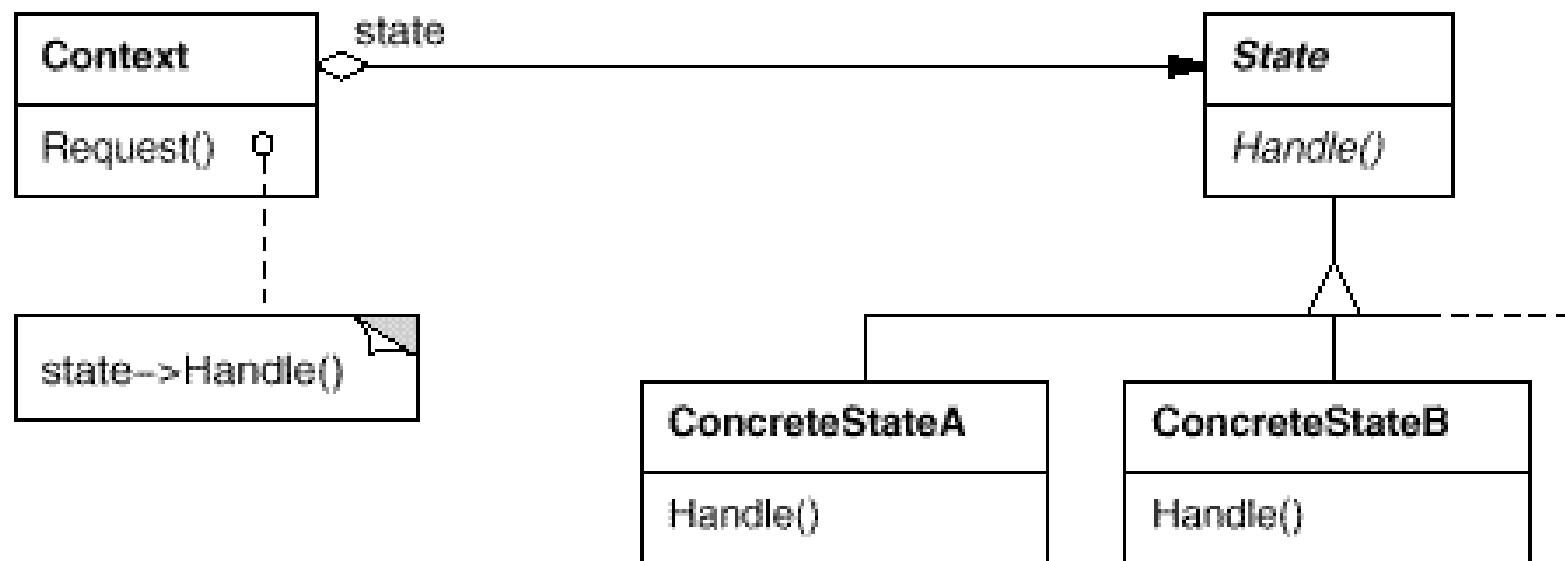
Example: Shopping in Taobao

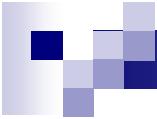


Example: TCP Connection



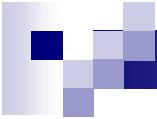
Structure





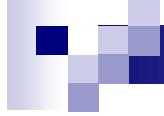
Participants

- **Context:** Defines the interface of interest to clients; maintains an instance of a **ConcreteState** that defines the current state.
 - **State:** Defines an interface for encapsulating the behavior associated with a particular state of the **Context**.
 - **ConcreteState:** each implements a behavior associated with a state of the **Context**.
-



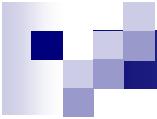
Collaborations

- **Context** delegates state-specific requests to the current **ConcreteState** object.
 - A **context** may pass itself as an argument to the **State** object handling the request. This lets the **State** object access the **context** if necessary.
 - **Context** is the primary interface for clients. **State** objects can be configured to **context**. Once a **context** is configured, its clients don't have to deal with the **State** objects directly.
 - Either **Context** or the **ConcreteState** can decide which state succeeds another and under what circumstances.
-



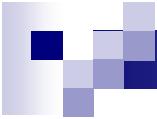
Consequences

- It localizes state-specific behavior and partitions behavior for different states.
 - New states and transitions can be added easily by defining new subclasses;
 - Avoiding large conditional statements which are undesirable.
 - It increases the number of classes and is less compact than a single class. But such distribution is actually good if there are many states.
-



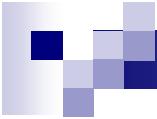
Consequences

- It makes state transitions explicit.
 - When an object defines its current state by internal data values, its state transitions have no explicit representation; they only show up as assignments to some variables.
 - State objects can be shared (Flyweight).
-



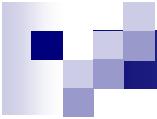
Applicability

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
 - Operations have large, multipart conditional statements that depend on the object's state.
-



Implementation 1: Who defines the state transitions?

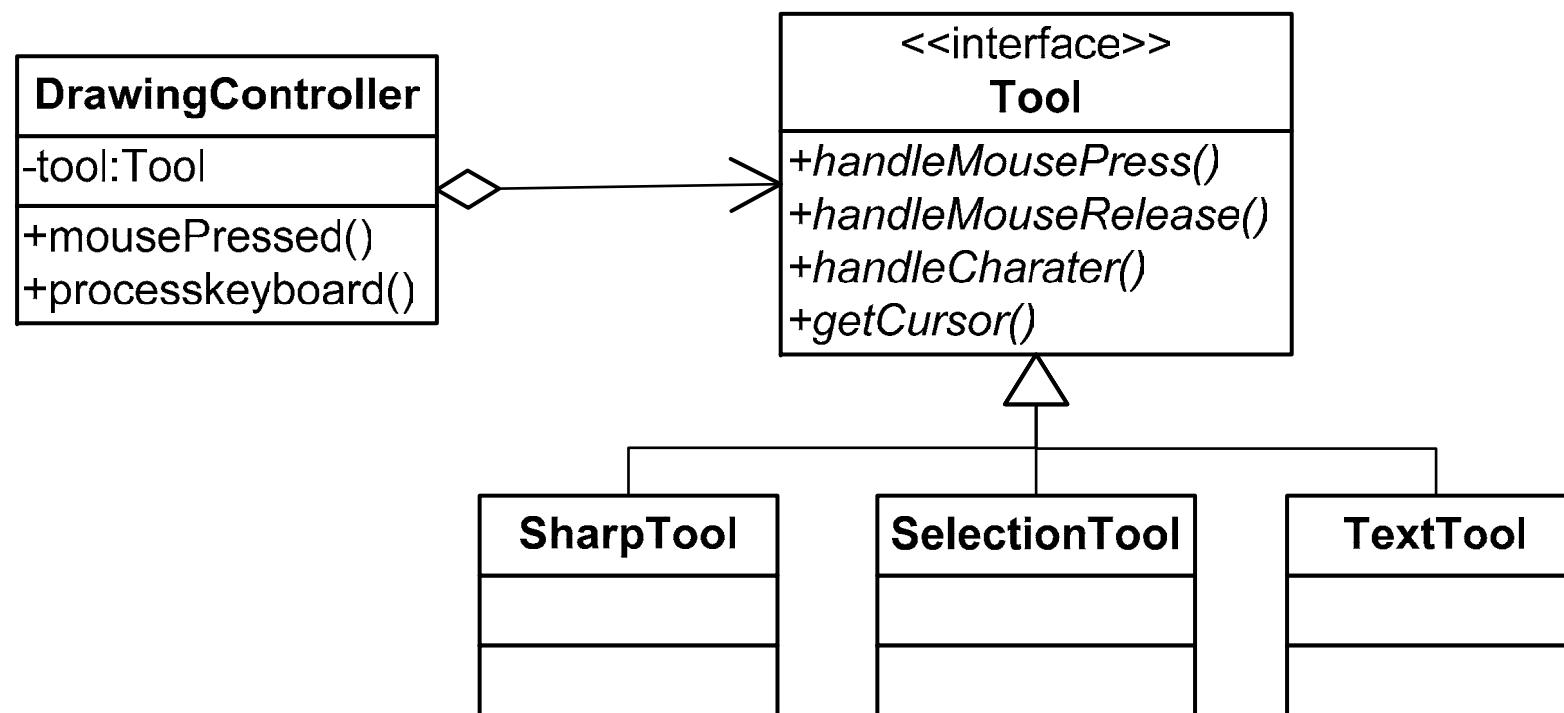
- The **State pattern** does not specify which participant defines the criteria for state transitions.
 - If the criteria are fixed, then they can be implemented entirely in the **Context**.
 - It is generally more flexible and appropriate to let the **State** subclasses themselves specify their successor state and when to make the transition.
 - It is easy to modify or extend the logic by defining new **State** subclasses.
 - A disadvantage is **State** subclass will have knowledge of at least one other, which introduces implementation dependencies between subclasses.
-

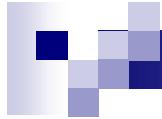


Implementation 2: Creating and destroying **State** objects.

- A common implementation trade-off worth considering is whether:
 - **Lazy**: to create **State** objects only when they are needed and destroy them thereafter.
 - When the **states** that will be entered aren't known at run-time, and contexts change state infrequently.
 - **Eager**: creating them ahead of time and never destroying them.
 - When state changes occur rapidly

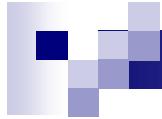
Examples





Extension: Table-driven approach

- Using **tables** to map inputs to state transitions.
For each state, a **table** maps every possible input to a succeeding state.
 - This approach converts conditional code into a **table** look-up.
 - The main advantage of tables is their regularity:
You can change the transition criteria by modifying data instead of changing program code.
-



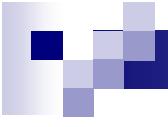
Extension: Table-driven approach

■ Disadvantages

- A table look-up is often less efficient than a function call.
- Less explicit and harder to understand.
- It's usually difficult to add actions to accompany the state transitions.

■ The key difference between table-driven and the State pattern

- The State pattern models state-specific behavior
 - the table-driven approach focuses on defining state transitions.
-



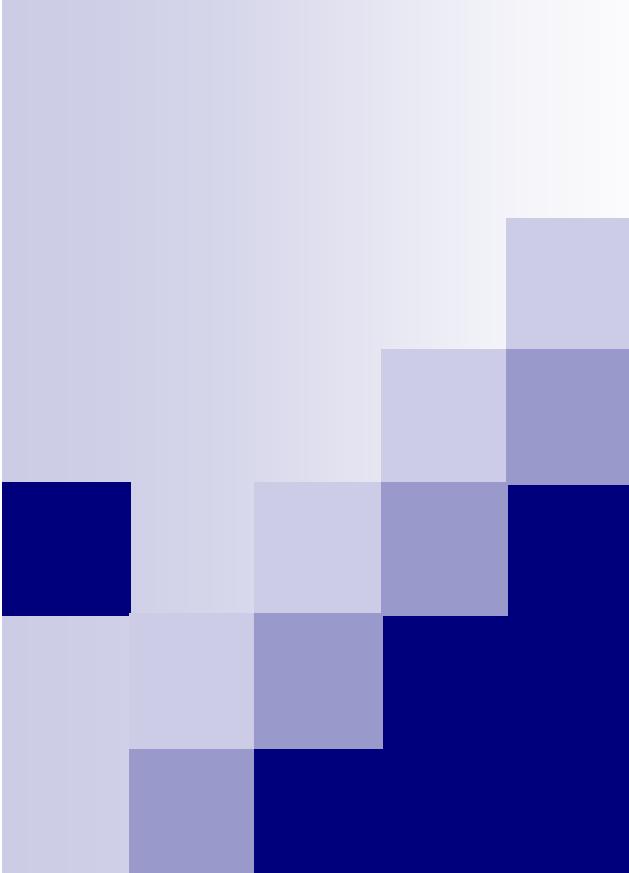
Related Patterns

- Strategy: One state with many algorithms;
 - State: many States with different behaviors.
-



Let's go to next...





Design Patterns

宋 杰

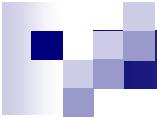
Song Jie

东北大学 软件学院

Software College, Northeastern
University



17. Iterator Pattern

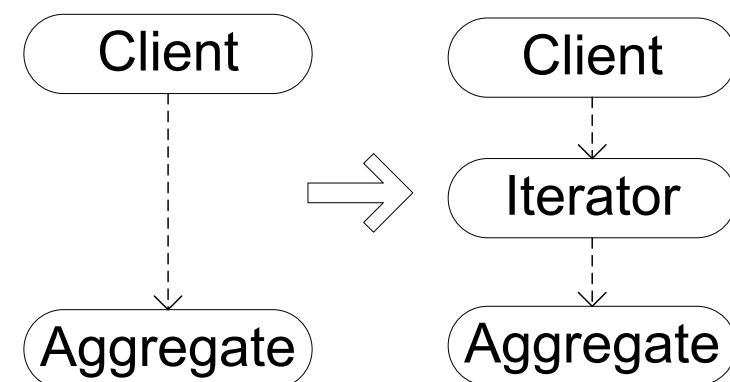


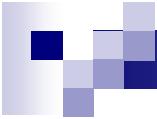
Intent

- Provide a way to **access** the elements of an aggregate object **sequentially** without exposing its underlying **representation**.
 - Cursor
 - 迭代子模式可以顺序地访问一个聚集中的元素而不必暴露聚集的内部表象。
-

Intent for OCP

- Traversal mechanism is unchanged, but the traversed aggregate is changed. The code in client side should be modified because different aggregates have different traversal interface.
- Aggregate is unchanged, but traversal mechanism is changed. For example, add filtering algorithm. The interface of aggregate should be modified to introduced the new traversal approaches.

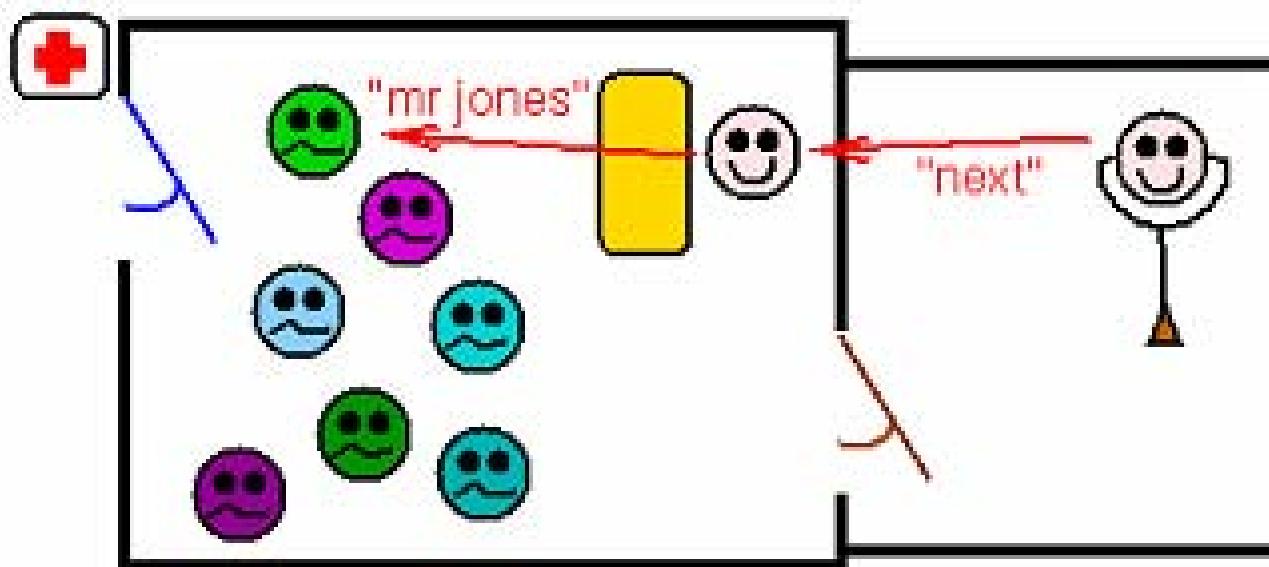




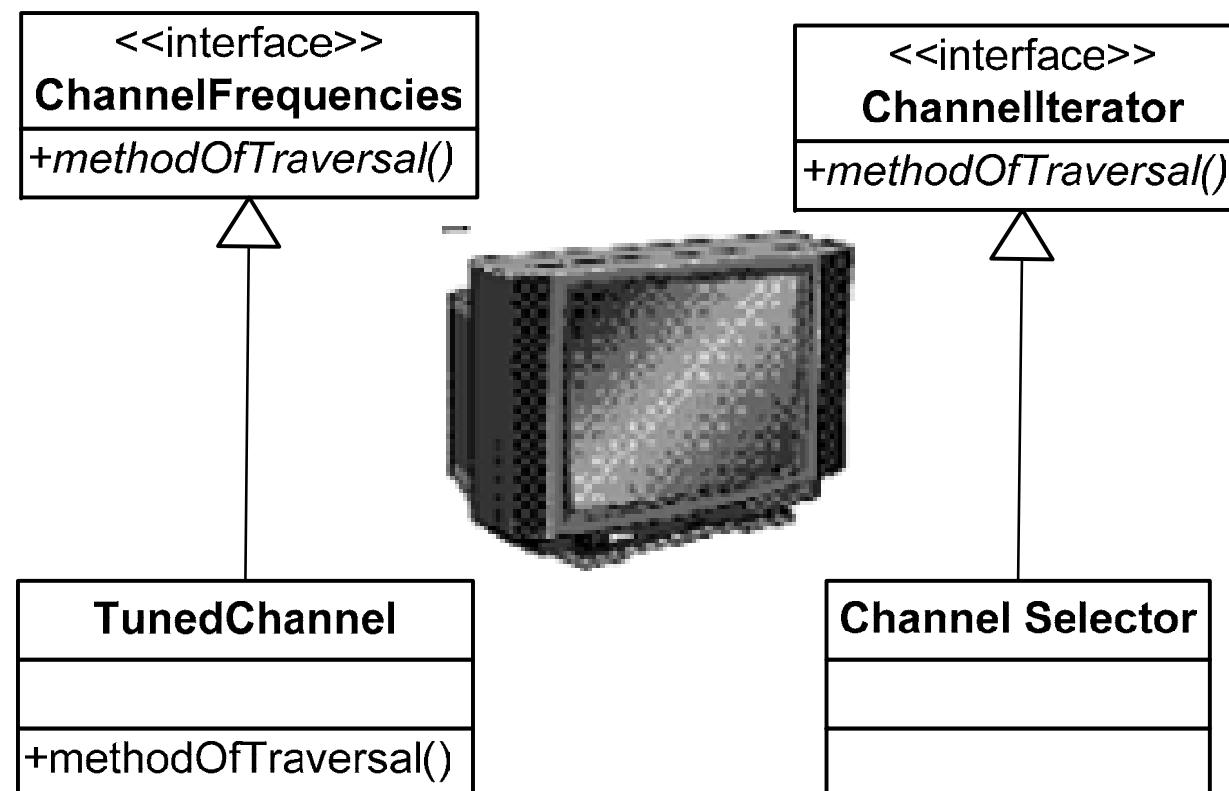
Traversal mechanism

- Forward: The directions that the elements increased;
 - Forward Iteration
 - Backward: The directions that the elements decreased
 - Backward Iteration
-

Example



Example



Example

```
interface Channel extends Comparable<Channel> {
    public String getName();
    public void setName(String name);
}

class ChannelImpl implements Channel {
    private String name;
    public ChannelImpl() {
    }
    public ChannelImpl(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int compareTo(Channel channel) {
        return this.getName().compareTo(channel.getName());
    }
    public String toString() {
        return this.name;
    }
}
```

1 - Public Iterator

```
interface ChannelSet<E extends Channel> {  
    public void addChannel(E channel);  
    public Iterator<E> iterator();  
    public E getChannel(int index);  
    public int size();  
}  
  
interface Iterator<E extends Channel> {  
    public boolean hasNext();  
    public E next();  
}
```

1 - Public Iterator

```
class ChannelSetImpl implements ChannelSet<Channel>{
    private List<Channel> channelList;
    public ChannelSetImpl() {
        channelList = new ArrayList<Channel>();
    }
    public void addChannel(Channel channel) {
        channelList.add(channel);
    }
    public Channel getChannel(int index) {
        return channelList.get(index);
    }
    public int size() {
        return channelList.size();
    }
    public Iterator<Channel> iterator() {
        return new ChannelItr(this);
    }
}
```

1 – Public Iterator

```
class ChannelItr implements Iterator<Channel> {
    private ChannelSet<Channel> channelSet;
    private int current = 0;
    public ChannelItr(ChannelSet<Channel> channelSet) {
        this.channelSet = channelSet;
    }
    public boolean hasNext() {
        return current < channelSet.size();
    }
    public Channel next() {
        return channelSet.getChannel(current++);
    }
}
```

1 – Public Iterator

```
class Client {  
    public void testChannelIterator() {  
        ChannelSet<Channel> channelSet = new ChannelSetImpl();  
        channelSet.addChannel(new ChannelImpl("CCTV-1"));  
        channelSet.addChannel(new ChannelImpl("CCTV-2"));  
        channelSet.addChannel(new ChannelImpl("CCTV-3"));  
        Iterator<Channel> it = channelSet.iterator();  
        while (it.hasNext()) {  
            System.out.println(it.next());  
        }  
    }  
}
```

2 - Private Passive (Internal) Iterator

```
interface ChannelSet<E extends Channel> {  
    public void addChannel(E channel);  
    public Iterator<E> iterator();  
}  
interface Iterator<E extends Channel> {  
    public boolean hasNext();  
    public E next();  
}
```

2 - Private Passive (Internal) Iterator

```
class ChannelSetImpl implements ChannelSet<Channel>{
    private List<Channel> channelList;
    public ChannelSetImpl() {
        channelList = new ArrayList<Channel>();
    }
    public void addChannel(Channel channel) {
        channelList.add(channel);
    }
    public Iterator<Channel> iterator() {
        return new Itr();
    }
    private class Itr implements Iterator<Channel> {
        private int current = 0;
        private Itr() {
            Collections.sort(channelList);
        }
        public boolean hasNext() {
            return current < channelList.size();
        }
        public Channel next() {
            return channelList.get(current++);
        }
    }
}
```

3 - Private Active (External) Iterator

```
interface ChannelSet<E extends Channel> {
    public void addChannel(E channel);
    public Iterator<E> iterator();
}

interface Iterator<E extends Channel> {
    public int size();
    public E getChannel(int index);
}
```

3 - Private Active (External) Iterator

```
class ChannelSetImpl implements ChannelSet<Channel> {
    private List<Channel> channelList;
    public ChannelSetImpl() {
        channelList = new ArrayList<Channel>();
    }
    public void addChannel(Channel channel) {
        channelList.add(channel);
    }
    public Iterator<Channel> iterator() {
        return new Itr();
    }
    private class Itr implements Iterator<Channel> {
        private Itr() {
            Collections.sort(channelList);
        }
        public int size() {
            return channelList.size();
        }
        public Channel getChannel(int index) {
            return channelList.get(index);
        }
    }
}
```

3 - Private Active (External) Iterator

```
class Client {
    public void testChannelIterator() {
        ChannelSet<Channel> channelSet = new ChannelSetImpl();
        channelSet.addChannel(new ChannelImpl("CCTV-1"));
        channelSet.addChannel(new ChannelImpl("CCTV-2"));
        channelSet.addChannel(new ChannelImpl("CCTV-3"));
        Iterator<Channel> it = channelSet.iterator();
        for (int i = 0; i < it.size(); i++) {
            System.out.println(it.getChannel(i));
        }
    }
}
```

4 - Private Passive (Internal) Static (Copied) Iterator

```
private class Itr implements Iterator<Channel> {
    private int current = 0;
    private List<Channel> copy;
    private Itr() {
        copy = new ArrayList<Channel>();
        copy.addAll(channelList);
        Collections.sort(channelList);
    }
    public boolean hasNext() {
        return current < channelList.size();
    }
    public Channel next() {
        return channelList.get(current++);
    }
}
```

5 - Private Passive (Internal) Fastfail Iterator

```
private class Itr implements Iterator<Channel> {
    private int originalSize;
    private int current = 0;
    private Itr() {
        originalSize = channelList.size();
        Collections.sort(channelList);
    }
    public boolean hasNext() {
        checkModify();
        return current < channelList.size();
    }
    public Channel next() {
        checkModify();
        return channelList.get(current++);
    }
    public void checkModify() {
        if (originalSize != channelList.size()) {
            throw new RuntimeException(
                "Iterator is invalid, the aggregate is modified!");
        }
    }
}
```

6 - Private Passive (Internal) Robust Iterator

```
interface ChannelSet<E extends Channel> {
    public void addChannel(E channel);
    public void removechannel(int index);
    public Iterator<E> iterator();
}

interface Iterator<E extends Channel> {
    public boolean hasNext();
    public E next();
}
```

6 - Private Passive (Internal) Robust Iterator

```
class ChannelSetImpl implements ChannelSet<Channel> {
    private List<Channel> channelList;
    private List<Itr> iterators;
    public ChannelSetImpl() {
        channelList = new ArrayList<Channel>();
        iterators = new ArrayList<Itr>();
    }
    public void addChannel(Channel channel) {
        channelList.add(channel);
    }
    public void removechannel(int index) {
        if (index >= channelList.size()) {
            return;
        }
        channelList.remove(index);
        for (Itr it : iterators) {
            if (index <= it.current) {
                it.current--;
            }
        }
    }
```

6 - Private Passive (Internal) Robust Iterator

```
private class Itr implements Iterator<Channel> {
    private int current = 0;
    private Itr() {
        Collections.sort(channelList);
    }
    public boolean hasNext() {
        boolean hasNext = current < channelList.size();
        if (!hasNext) {
            iterators.remove(this);
        }
        return hasNext;
    }
    public Channel next() {
        return channelList.get(current++);
    }
}
```

7 - Private Passive (Internal) Editable Iterator

```
interface ChannelSet<E extends Channel> {
    public Iterator<E> iterator();
}

interface Iterator<E extends Channel> {
    public boolean hasNext();
    public E next();
    //add to the last
    public void add(E channel);
    //remove the element which returned by next()
    public void remove();
    //replace the element which returned by next()
    public void replace(E channel);
}
```

7 - Private Passive (Internal) Editable Iterator

```
class ChannelSetImpl implements ChannelSet<Channel> {
    private List<Channel> channelList;
    public ChannelSetImpl() {
        channelList = new ArrayList<Channel>();
    }
    public Iterator<Channel> iterator() {
        return new Itr();
    }
    private class Itr implements Iterator<Channel> {
        private int current = 0;
        public boolean hasNext() {
            return current < channelList.size();
        }
        public Channel next() {
            return channelList.get(current++);
        }
        public void add(Channel channel) {
            channelList.add(channel);
        }
        public void remove() {
            channelList.remove(--current);
        }
        public void replace(Channel channel) {
            channelList.set(current - 1, channel);
        }
    }
}
```

8 - Private Passive (Internal) Iterator with additional operations

```
interface ChannelSet<E extends Channel> {
    public void addChannel(E channel);
    public Iterator<E> iterator();
}

interface Iterator<E extends Channel> {
    public boolean hasNext();
    public boolean hasPrevious();
    public int currentIndex();
    // All return current item
    public E next();
    public E previous();
    // return the first or last item
    public E first();
    public E last();
    public E moveTo(int index);
}
```

8 - Private Passive (Internal) Iterator with additional operations

```
private class Itr implements Iterator<Channel> {
    private int current = 0;
    private Itr() {
        Collections.sort(channelList);
    }
    public boolean hasNext() {
        return current < channelList.size();
    }
    public boolean hasPrevious() {
        return current >= 0;
    }
    public int currentIndex() {
        return current;
    }
    public Channel next() {
        return channelList.get(current++);
    }
    public Channel previous() {
        return channelList.get(current--);
    }
    public Channel first() {
        current = 0;
        return channelList.get(current);
    }
    public Channel last() {
        current = channelList.size() - 1;
        return channelList.get(current);
    }
    public Channel moveTo(int index) {
        if (index >= 0 && index < channelList.size()) {
            current = index;
            return channelList.get(current);
        }
        throw new RuntimeException("input index is out of range");
    }
}
```

9 - Self Iterator

```
interface Iterator<E extends Channel> {
    public boolean hasNext();
    public E next();
}

interface ChannelSet<E extends Channel> extends Iterator<E> {
    public void addChannel(E channel);
}

class ChannelSetImpl implements ChannelSet<Channel>{
    private List<Channel> channelList;
    private int current = 0;
    public ChannelSetImpl() {
        channelList = new ArrayList<Channel>();
    }
    public void addChannel(Channel channel) {
        channelList.add(channel);
    }
    public boolean hasNext() {
        return current < channelList.size();
    }
    public Channel next() {
        return channelList.get(current++);
    }
}
```

9 - Self Iterator

```
class Client {  
    public void testChannelIterator() {  
        ChannelSet<Channel> channelSet = new ChannelSetImpl();  
        channelSet.addChannel(new ChannelImpl("CCTV-1"));  
        channelSet.addChannel(new ChannelImpl("CCTV-2"));  
        channelSet.addChannel(new ChannelImpl("CCTV-3"));  
        Iterator<Channel> it = channelSet;  
        while (it.hasNext()) {  
            System.out.println(it.next());  
        }  
    }  
}
```

10 – Iterator without Aggregate

```
interface ChannelSet<E extends Channel> {
    public void addChannel(E channel);
    public Iterator<E> iterator();
}

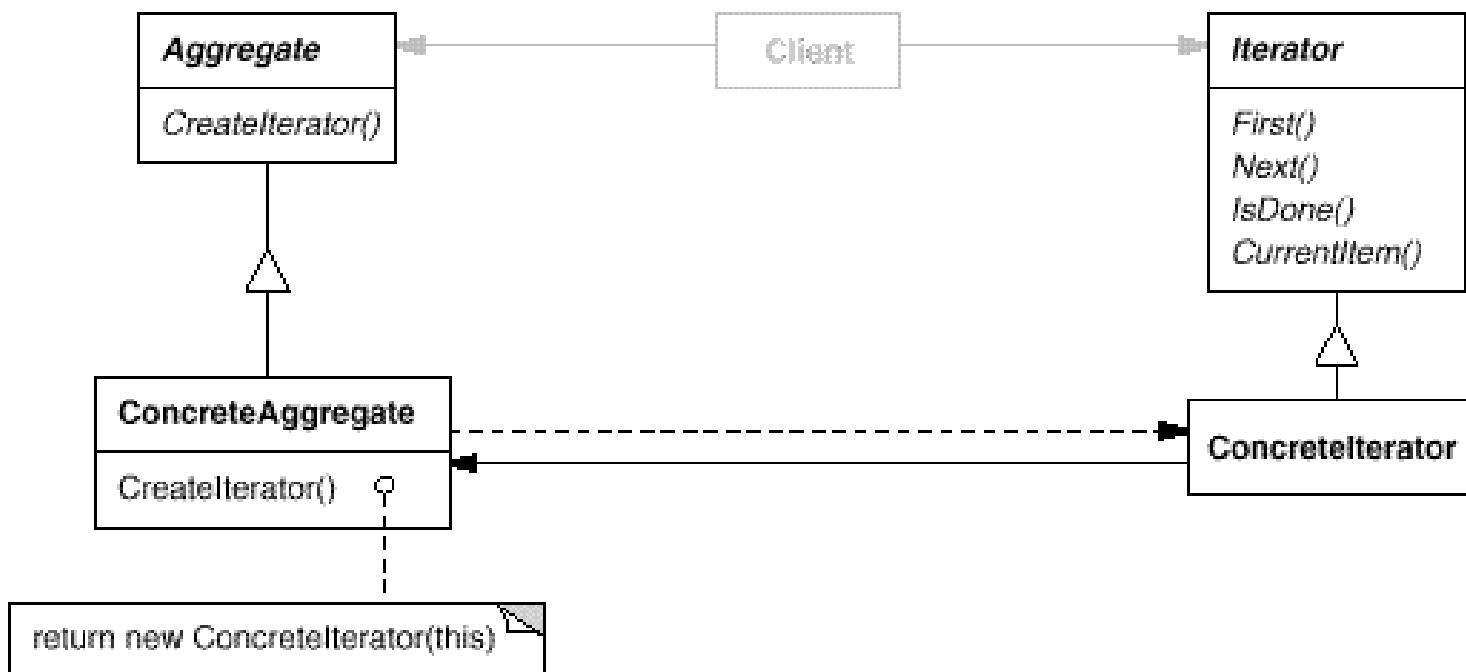
interface Iterator<E extends Channel> {
    public boolean hasNext();
    public E next();
}

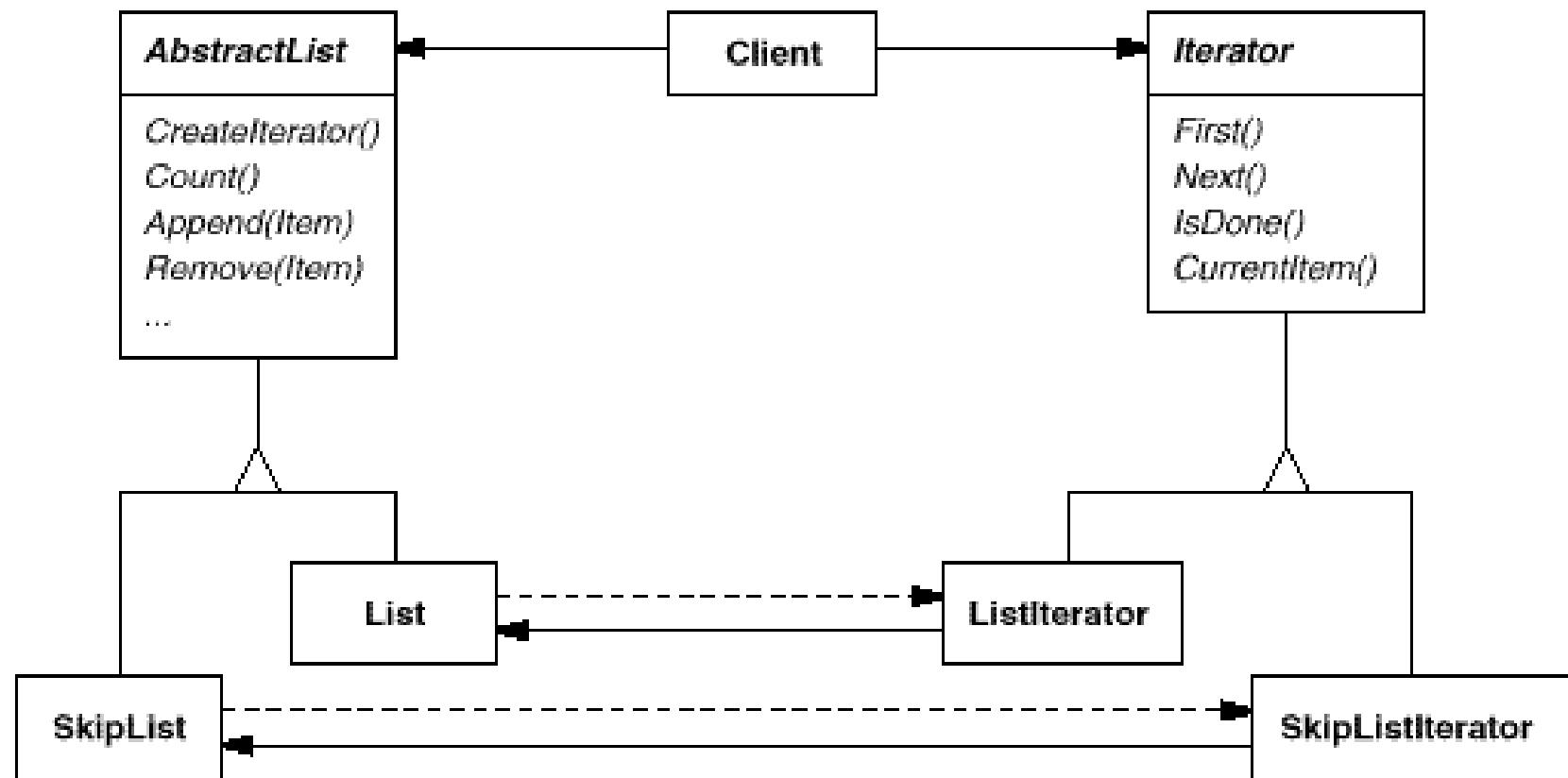
class CCTVChannelSet implements ChannelSet<Channel> {
    private Channel CCTV1 = new ChannelImpl("CCTV-1");
    private Channel CCTV2 = new ChannelImpl("CCTV-2");
    private Channel CCTV3 = new ChannelImpl("CCTV-3");
    private Channel CCTV4 = new ChannelImpl("CCTV-4");
    private Channel CCTV5 = new ChannelImpl("CCTV-5");
    private final int CCTVChannelSize = 5;
    public Iterator<Channel> iterator() {
        return new Itr();
    }
    public void addChannel(Channel channel) {
    }
}
```

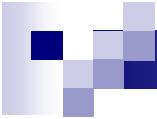
10 - Iterator without Aggregate

```
private class Itr implements Iterator<Channel> {  
  
    private int current = 0;  
    public boolean hasNext() {  
        return current < CCTVChannelSize;  
    }  
    public Channel next() {  
        switch (current++) {  
        case 0:  
            return CCTV1;  
        case 1:  
            return CCTV2;  
        case 2:  
            return CCTV3;  
        case 3:  
            return CCTV4;  
        case 4:  
            return CCTV5;  
        default:  
            return null;  
        }  
    }  
}
```

Structure







Participants

- **Iterator**

- Defines an interface for accessing and traversing elements.

- **Concreteliterator**

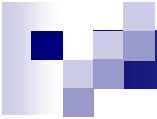
- Implements the **Iterator** interface.
 - Keeps track of the current position in the traversal of the aggregate.

- **Aggregate**

- Defines an interface for creating an **Iterator** object.

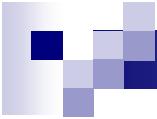
- **ConcreteAggregate**

- Implements the **Iterator** creation interface to return an instance of the proper **Concreteliterator**.



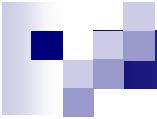
Consequences – advantages

- It supports variations in the traversal of an aggregate. Complex aggregates may be traversed in many ways.
 - It supports multiple traversals of an aggregate. Many **iterators** can traverse the aggregate together.
 - **Iterators** simplify the **Aggregate** interface.
 - More than one traversal can be pending on an aggregate.
-



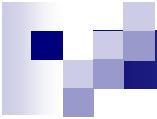
Consequences – drawbacks

- Because **Iterators** rely on linear traversal, the traversed aggregate seems to be an ordered sequence, on the contrary, it is not true in most cases.
 - The elements returned by **Iterators** is implicit type (object or general type) . So that client must know the true type of the elements explicitly.
-



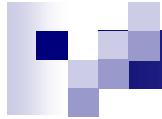
Applicability

- To access an aggregate object's contents without exposing its internal representation.
 - To support multiple traversals of aggregate objects.
 - To provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).
-



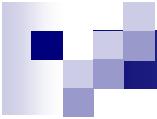
Implementation 1: Where the **concrete iterator** is defined?

- **Public Iterator:** **Concrete iterator** is defined as a class independent from aggregate.
 - More straightforward;
 - Polymorphic iteration;
 - Can store multiple cursor for different clients.
 - Need the aggregate expose the details, thus break the encapsulation.
 - **Private Iterator:** **Concrete iterator** is defined as an inner class in the aggregate.
 - Less straightforward;
 - Protect the encapsulation of aggregate;
 - Suggested in most cases.
-



Implementation 2: Who controls the iteration?

- **Active Iterator (External Iterator)**: The client controls the iteration;
 - Clients that use an **active iterator** must advance the traversal and request the next element explicitly from the **iterator**.
 - more flexible than **passive iterators**;
 - **Passive Iterator (Internal Iterator)**: The **iterator** controls the iteration;
 - The client hands an **passive iterator** an operation to perform, and the **iterator** applies that operation to every element in the aggregate.
 - Easier to use, because it define the iteration logic for you.
-

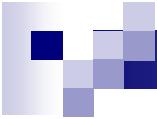


Implementation 3: Who defines the traversal algorithm?

- The aggregate might define the traversal algorithm and use the **iterator** to store just the state of the iteration (**cursor**), it points to the current position in the aggregate.
 - The **iterator** is responsible for the traversal algorithm, then it's easy to use different iteration algorithms on the same aggregate, and it can also be easier to reuse the same algorithm on different aggregates.
 - Defining the **iterator** in aggregate's the inner class if traversal algorithm might need to access the private variables of the aggregate.
-

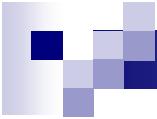
Implementation 3: How robust is the iterator?

- It can be dangerous to modify an aggregate while you're traversing it.
 - **Copied Iterator:** A simple solution is to copy the aggregate and traverse the copy, but that's too expensive to do in general.
 - **Robust iterator:** Ensures that insertions and removals won't affect traversal, and it does it without copying the aggregate.
- Robust iterator rely on registering the iterator with the aggregate. On insertion or removal, the aggregate either adjusts the internal state of iterators it has produced, or it maintains information internally to ensure proper traversal.



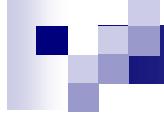
Static Iterator and Dynamic Iterator

- **Static Iterator:** A **copied iterator** which contains a snapshot of the aggregate when **iterator** is created. New changes are invisible to the traversal approach.
 - **Dynamic Iterator:** **Dynamic Iterator** is opposed to the static one. Any changes to the aggregate are allowed and available when traversing the aggregate.
 - Completely **Dynamic Iterator** is not easy to be implemented.
-



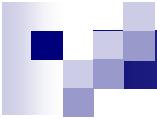
Fail Fast

- **Fail-fast** is a property of a system or module with respect to its response to failures.
 - A fail-fast system is designed to immediately report at its interface any failure or condition that is likely to lead to failure.
 - Fail-fast systems are usually designed to stop normal operation rather than attempt to continue a possibly-flawed process.
 - **FailFast Iterator** throws an exception when the aggregate is changed during iteration.
-



Implementation 4: Additional **Iterator** operations.

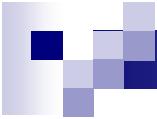
- The minimal interface to **Iterator** consists of the operations **First**, **Next**, **IsDone**, and **CurrentItem**.
 - Some additional operations might prove useful.
 - **Frist**
 - **Last**
 - **Previous**
 - **SkipTo**
-



Implementation 4: Additional **Iterator** operations

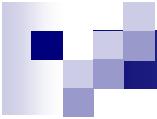
■ Filter Iterator

- A common iterator traverse each element of an aggregate.
 - A **filter iterator** compute the element of the aggregate and return the elements which match a certain condition.
-



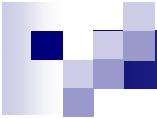
Implementation 5: polymorphic iterators

- **Polymorphic iterators** allows the traversal algorithm operate on different aggregates which is assigned dynamically.
 - **Polymorphic iterators** have a drawback that client is responsible for deleting them.
-



Implementation 7: Iterators for composites.

- External **iterators** can be difficult to implement over recursive aggregate structures like those in the Composite pattern, because a position in the structure may span many levels of nested aggregates.
 - An **active (external) iterator** has to store a path through the Composite to keep track of the current object.
 - It's easier just to use an **passive (internal) iterator**. It can record the current position simply by calling itself recursively, thereby storing the path implicitly in the call stack.
 - Composites often need to be traversed in more than one way.
 - Pre-order, post-order, in-order, and breadth-first traversals are common.
-



Implementation 8: Null iterators.

- A **Null Iterator** is a degenerate **iterator** that's helpful for handling boundary conditions.
 - By definition, a **Null Iterator** is always done with traversal; that is, its **IsDone** operation always evaluates to true.
 - **NullIterator** can make traversing tree-structured aggregates (like Composites) easier.
-

Example: Java Collection

■ Interface Collection<E>

- Vector, ArrayList<E>, LinkedList<E>, Stack<E>, Queue<E>, HashSet<E>, TreeSet<E>, HashMap<K,V>, TreeMap<K,V>, Hashtable<K,V>.

java.util.concurrent.LinkedBlockingQueue
java.util.concurrent.BlockingQueue
java.util.concurrent.ArrayBlockingQueue
java.util.concurrent.LinkedBlockingQueue
java.util.concurrent.PriorityBlockingQueue
...

■ Interface Iterator<E>

■ Interface ListIterator<E> **extends** Iterator<E>

■ Interface Iterable<T>

```
/** Implementing this interface allows an object to be the target of
 * the "foreach" statement.
 * @since 1.5
 */
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}

public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove();
    void set(E e);
    void add(E e);
}
```

ListIterator<E>

- **void add(E o)**
 - Inserts the specified element into the list.
 - The element is inserted immediately before the next element that would be returned by `next`, if any, and after the next element that would be returned by `previous`, if any.
 - The new element is inserted before the implicit cursor: a subsequent call to `next` would be unaffected, and a subsequent call to `previous` would return the new element.
- **void remove()**
 - Removes from the list the last element that was returned by `next` or `previous`
 - This call can only be made once per call to `next` or `previous`.
 - It can be made only if `ListIterator.add` has not been called after the last call to `next` or `previous`.

java.util.AbstractList

java.util.AbstractList.Itr

java.util.AbstractList.ListItr

```
package java.util;

public abstract class AbstractList<E> extends AbstractCollection<E> implements
    List<E> {
    public Iterator<E> iterator() {
        return new Itr();
    }

    public ListIterator<E> listIterator() {
        return listIterator(0);
    }

    public ListIterator<E> listIterator(final int index) {
        if (index < 0 || index > size())
            throw new IndexOutOfBoundsException("Index: " + index);

        return new ListItr(index);
    }
}
```

```
private class Itr implements Iterator<E> {
    int cursor = 0;
    int lastRet = -1;
    int expectedModCount = modCount;

    public boolean hasNext() {
        return cursor != size();
    }

    public E next() {
        checkForComodification();
        try {
            E next = get(cursor);
            lastRet = cursor++;
            return next;
        } catch (IndexOutOfBoundsException e) {
            checkForComodification();
            throw new NoSuchElementException();
        }
    }
}
```

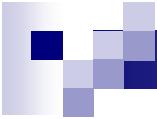
```
public void remove() {
    if (lastRet == -1)
        throw new IllegalStateException();
    checkForComodification();

    try {
        AbstractList.this.remove(lastRet);
        if (lastRet < cursor)
            cursor--;
        lastRet = -1;
        expectedModCount = modCount;
    } catch (IndexOutOfBoundsException e) {
        throw new ConcurrentModificationException();
    }
}

final void checkForComodification() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
}
```

```
private class ListItr extends Itr implements ListIterator<E> {
    ListItr(int index) {
        cursor = index;
    }
    public boolean hasPrevious() {
        return cursor != 0;
    }
    public E previous() {
        checkForComodification();
        try {
            int i = cursor - 1;
            E previous = get(i);
            lastRet = cursor = i;
            return previous;
        } catch (IndexOutOfBoundsException e) {
            checkForComodification();
            throw new NoSuchElementException();
        }
    }
    public int nextIndex() {
        return cursor;
    }
    public void set(E e) {
        if (lastRet == -1)
            throw new IllegalStateException();
        checkForComodification();

        try {
            AbstractList.this.set(lastRet, e);
            expectedModCount = modCount;
        } catch (IndexOutOfBoundsException ex) {
            throw new ConcurrentModificationException();
        }
    }
    public void add(E e) {
        checkForComodification();
        try {
            AbstractList.this.add(cursor++, e);
            lastRet = -1;
            expectedModCount = modCount;
        } catch (IndexOutOfBoundsException ex) {
            throw new ConcurrentModificationException();
        }
    }
}
```

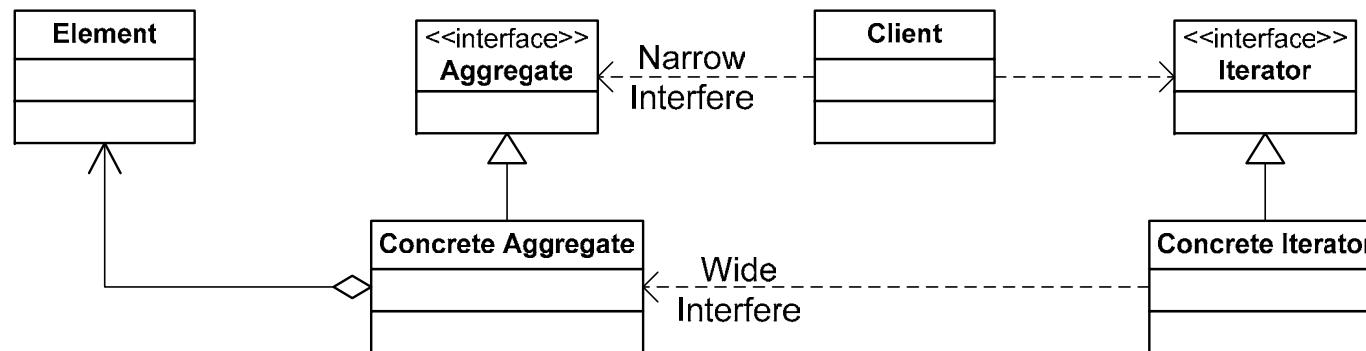


Extension 1: Wide and Narrow interface of an aggregate

- Wide interface: the interfaces which allow the client to modify the elements of aggregate.
 - White-box Aggregate
 - Narrow interface: the interfaces which only allow the client to traverse the elements of aggregate.
 - Black-box Aggregate
-

Extension 1

- An aggregate should provide a wide interface to its **iterator**, and narrow interface to its clients.
- Let **concrete iterator** be an inner class of an aggregate



Extension 2: Enumeration<E> vs Iterator<E>

<<interface>>
Iterator
+ <i>hasNext()</i> : boolean
+ <i>next()</i> : E
+ <i>remove()</i>

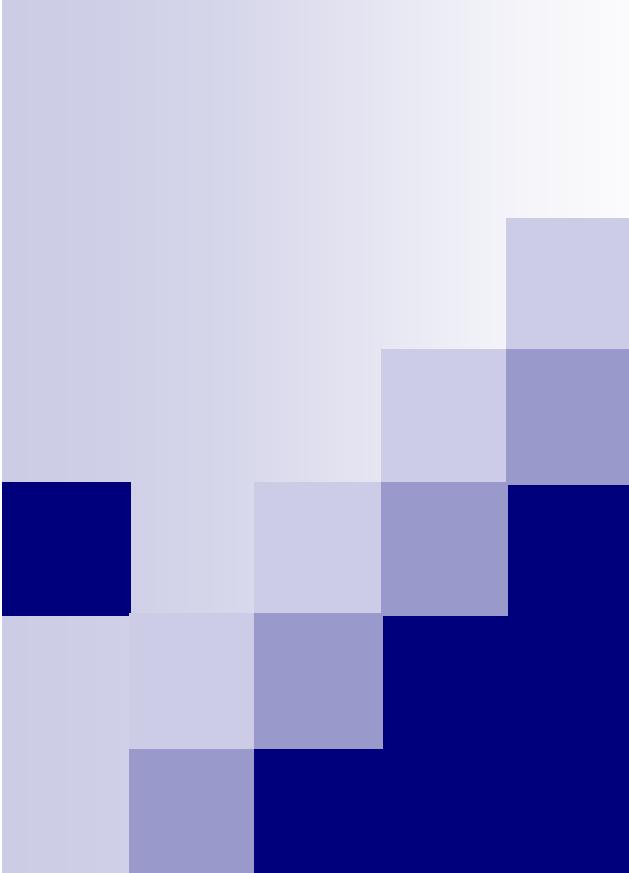
<<interface>>
Enumeration
+ <i>hasMoreElements()</i> : boolean
+ <i>nextElement()</i> E

- **Iterator** simplifies the interfaces;
- **Enumeration** does not supports Fail Fast;
- **Enumeration** is twice as fast as **Iterator** and uses very less memory;
- **Iterator** is much safer as compared to **Enumeration** because it always denies other threads to modify the collection object which is being iterated by it;



Let's go to next...





Design Patterns

宋 杰

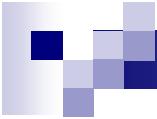
Song Jie

东北大学 软件学院

Software College, Northeastern
University

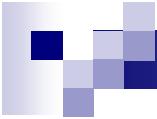


18. Command Pattern



Intent

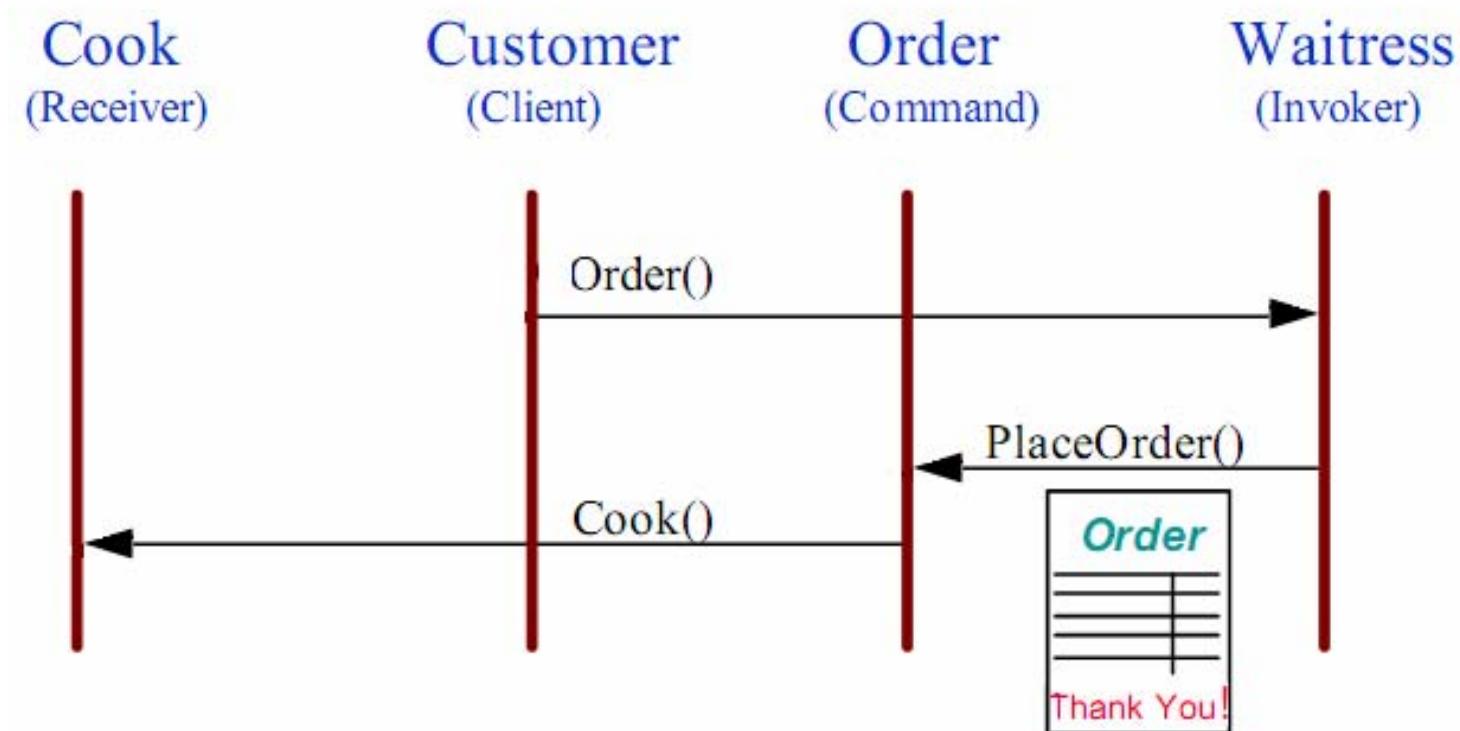
- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable (redoable) operations.
 - Action, Transaction
 - 命令模式把一个请求封装到一个对象中。命令模式允许系统使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。
-



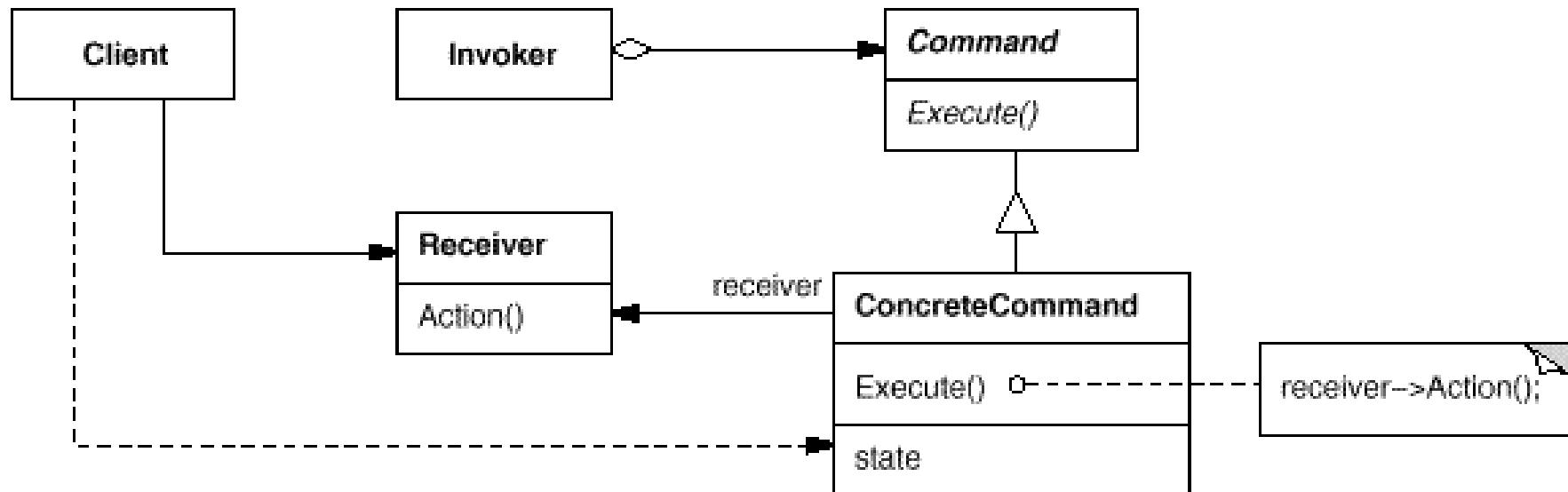
Intent

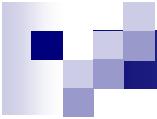
- Command pattern separate the responsibility of sending command and executing command, delegates command to different objects;
 - Each command is an operation;
 - Invoker send a command as an request of the operation;
 - Receiver take a command and execute the operation;
 - Invoker is separate from Receiver, and when, where, how the command is executed.
-

Example



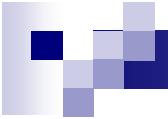
Structure





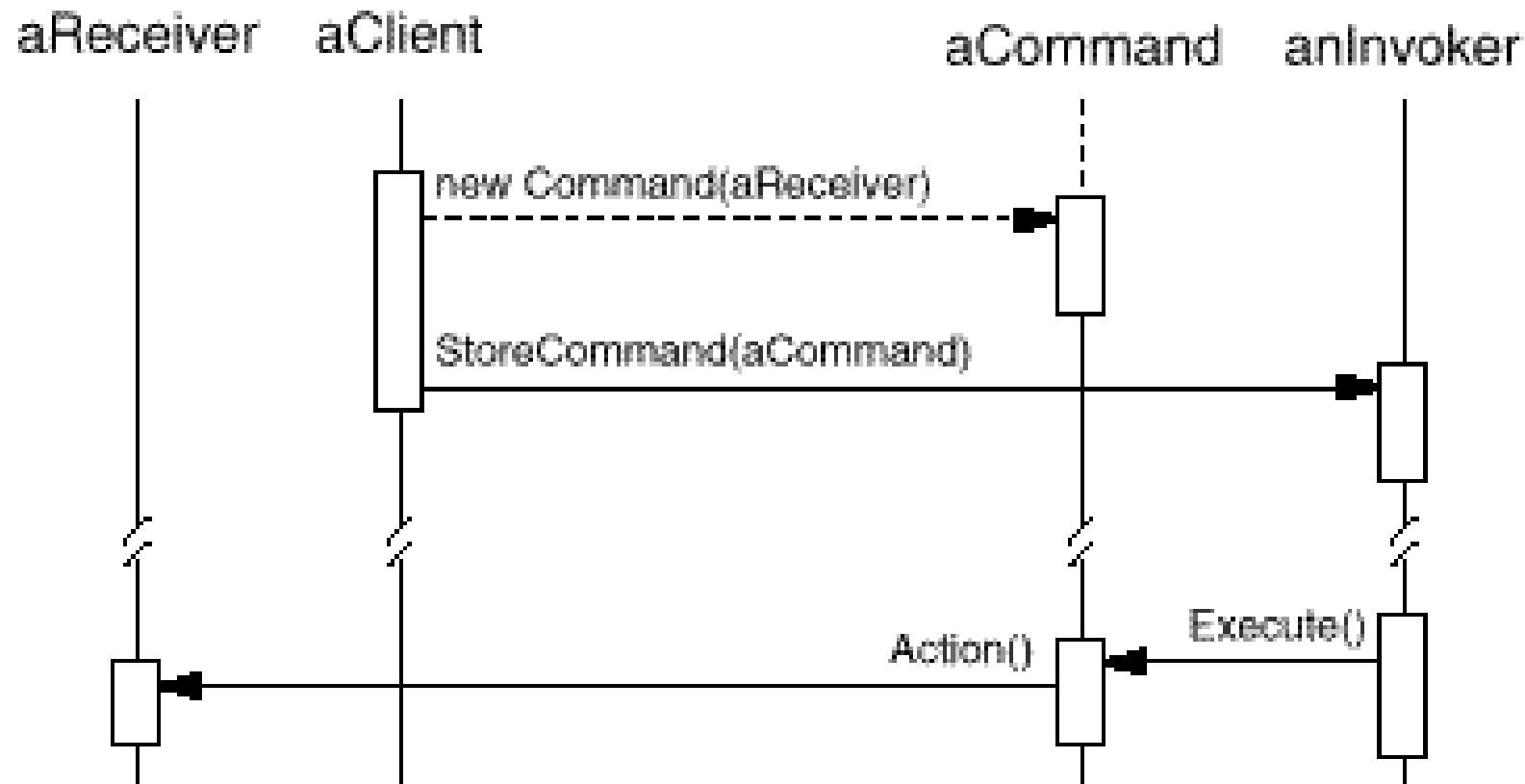
Participants

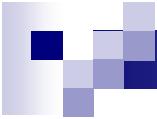
- **Command:** Declares an interface for executing an operation.
 - **ConcreteCommand:** Defines a binding between a **Receiver** object and an action. Implements *Execute()* by invoking the corresponding operation(s) on **Receiver**.
 - **Client:** Creates a **ConcreteCommand** object and sets its receiver.
 - **Invoker:** Asks the **command** to carry out the request.
 - **Receiver:** Knows how to perform the operations associated with carrying out a request. Any class may serve as a **Receiver**.
-



Collaborations

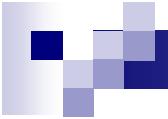
- The **client** creates a **ConcreteCommand** object and specifies its **receiver**.
 - An **Invoker** object stores the **ConcreteCommand** object.
 - The **invoker** issues a request by calling ***Execute*** on the command. When **commands** are undoable, **ConcreteCommand** stores state for undoing the command prior to invoking ***Execute***.
 - The **ConcreteCommand** object invokes operations on its **receiver** to carry out the request.
-





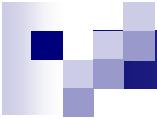
Consequences

- **Command** decouples the object that invokes the operation from the one that knows how to perform it.
 - **Commands** are first-class objects. They can be manipulated and extended like any other object.
 - You can assemble commands into a composite command. An example is the **MacroCommand** class. In general, composite commands are an instance of the Composite pattern.
-



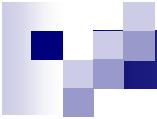
Consequences

- Allow the **receiver** veto the **command** (request);
 - It's easy to add new **Commands**, because you don't have to change existing classes.
 - It is easy to implement a command queue;
 - It is easy to implement Undo and Redo;
 - It is easy to implement Logging mechanisms;
 - Command pattern will introduce too many **command** classes and objects.
-



Applicability

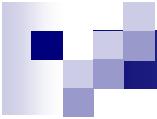
- Parameterize objects (clients) by an action to perform.
 - You can express such parameterization with a **callback** function, that is, a function that's registered somewhere to be called at a later point.
 - Commands are an object-oriented replacement for callbacks.
 - Specify, queue, and execute requests at different times.
 - A Command object can have a lifetime independent of the original request.
- 



Applicability

- Support undo.

- The **Command**'s *Execute* operation can store state for reversing its effects in the command itself.
 - The **Command** interface must have an added *Unexecute* operation that reverses the effects of a previous call to *Execute*.
 - Executed **commands** are stored in a history list.
 - Unlimited-level undo and redo is achieved by traversing this list backwards and forwards calling *Unexecute* and *Execute*, respectively.



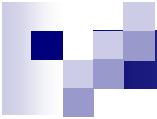
Applicability

- Support logging changes

- Adding the **Command** interface with *Load* and *Store* operations, you can keep a persistent log of changes.
 - They can be reapplied in case of a system crash.
 - Recovering from a crash involves *Load* logged commands from disk and re-executing them with the *Execute* operation.

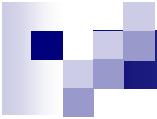
- Support transactions.

- Structure a system around high-level operations built on primitives operations.
 - A transaction encapsulates a set of changes to data.
 - Commands have a common interface, letting you invoke all transactions the same way.



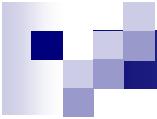
Implementation 1: How intelligent should a command be?

- A command can have a wide range of abilities.
 - At one extreme, it merely defines a binding between a receiver and the actions that carry out the request.
 - Sometime commands have enough knowledge to find their receiver dynamically.
 - At the other extreme, it implements everything itself without delegating to a receiver at all.
 - It is useful when you want to define commands that are independent of existing classes, when no suitable receiver exists, or when a command knows its receiver implicitly.
-



Implementation 2: Supporting undo and redo

- Commands can support undo and redo capabilities if they provide a way to reverse their execution (*Unexecute* or *Undo* operation).
 - A `ConcreteCommand` class might need to store additional state to do so.
 - The Receiver object
 - The arguments to the operation performed on the receiver
 - Any original values in the receiver that can change as a result of handling the request.
 - The receiver must provide operations that let the command return the receiver to its prior state.
-



Implementation 2: Supporting undo and redo

- To support one level of undo, an application needs to store only the command that was executed last.
 - For multiple-level undo and redo, the application needs a **history list** of commands that have been executed,
 - The maximum length of the list determines the number of undo/redo levels.
 - Traversing backward through the list and reverse-executing commands cancels their effect;
 - Traversing forward and executing commands re-executes them.
-

Implementation

Avoiding error accumulation in the undo process.

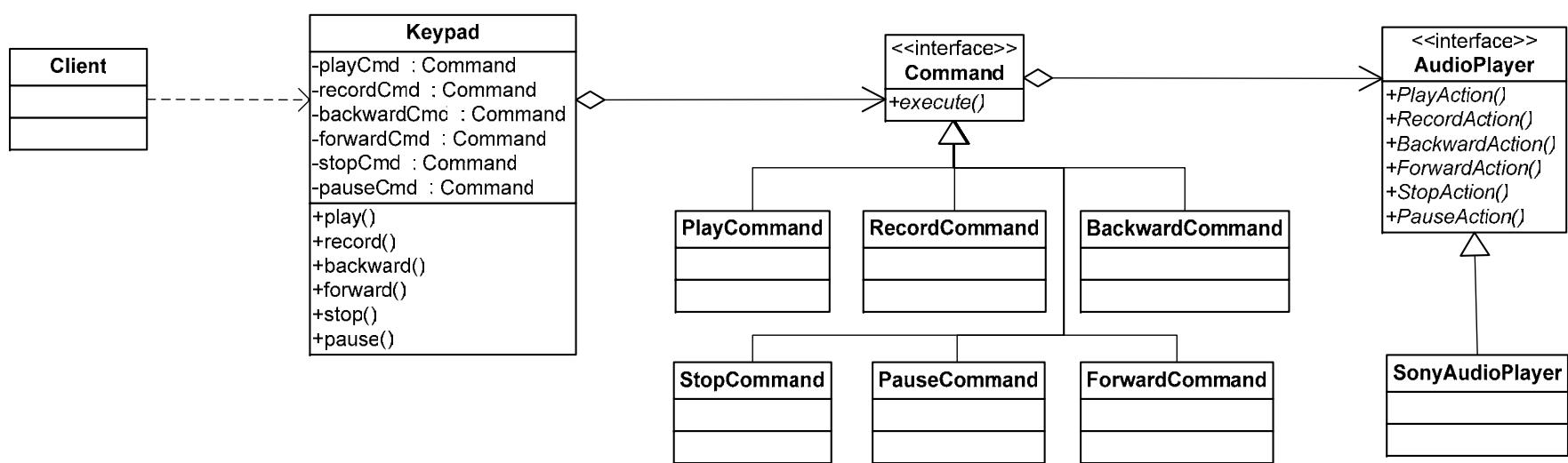
- Errors can accumulate as commands are executed, unexecuted, and re-executed repeatedly, so that an application's state eventually diverges from original values.
 - It may be necessary to store more information in the command to ensure that objects are restored to their original state.
 - **Memento pattern**
-

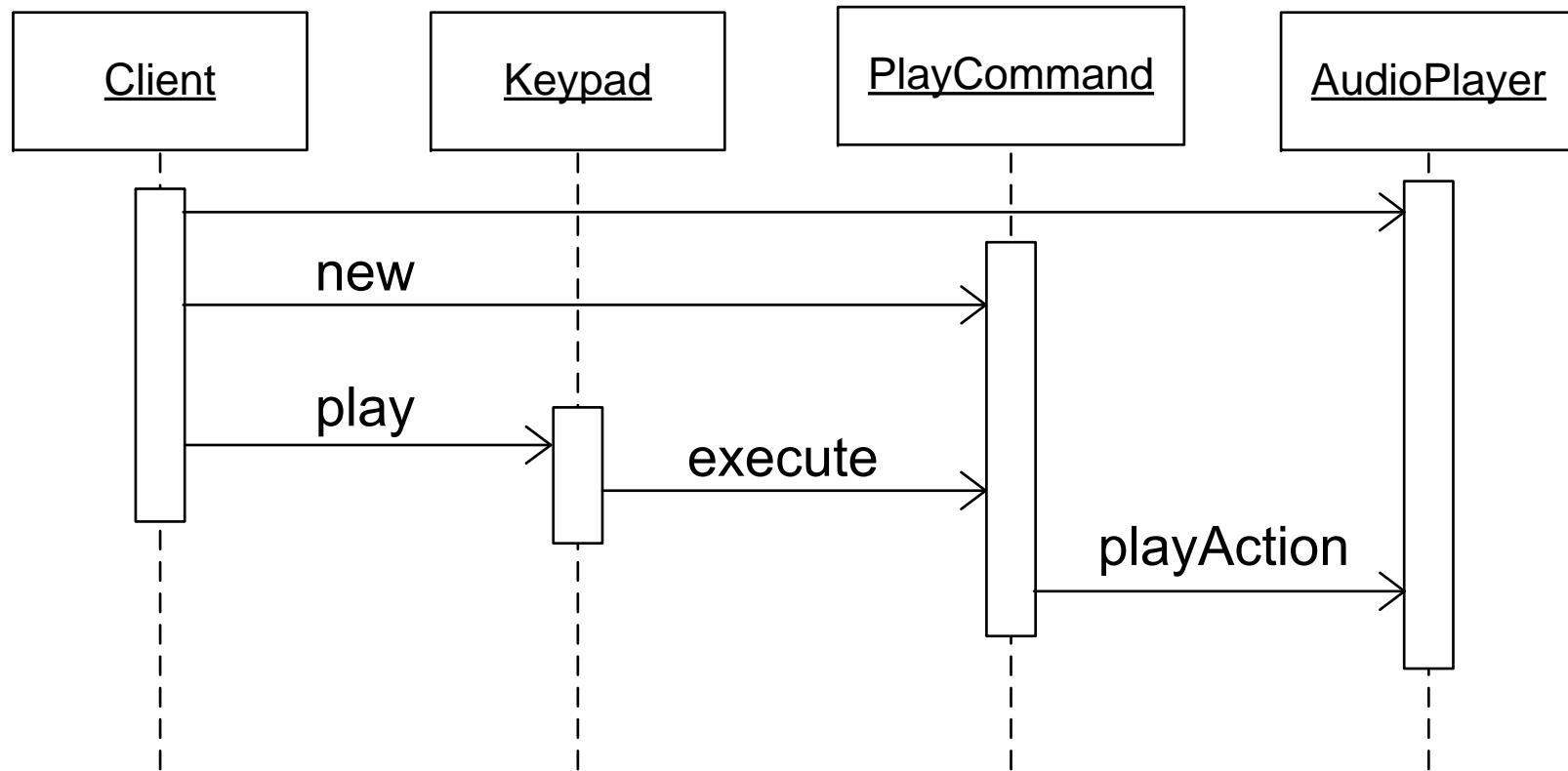
Example: AudioPlayer system

- Play
- Record
- Backward
- Forward
- Stop
- Pause

- Client: Person
- Invoker: Keypad
- Command: Functionalities
- Receiver: AudioPlayer







```
interface AudioPlayer{
    public void playAction();
    public void recordAction();
    public void backwardAction();
    public void forwardAction();
    public void stopAction();
    public void pauseAction();
}
class SonyAudioPlayer implements AudioPlayer{
    public void backwardAction() {
        // TODO backward
    }
    public void forwardAction() {
        // TODO forward
    }
    public void pauseAction() {
        // TODO pause
    }
    public void playAction() {
        // TODO play
    }
    public void recordAction() {
        // TODO record
    }
    public void stopAction() {
        // TODO stop
    }
}
```

```
abstract class PlayerCommand{
    protected AudioPlayer player;
    public PlayerCommand(AudioPlayer player) {
        this.player = player;
    }
    public abstract void execute();
}

class PlayCommand extends PlayerCommand{
    public PlayCommand(AudioPlayer player) {
        super(player);
    }
    public void execute() {
        player.playAction();
    }
}
class RecordCommand extends PlayerCommand{
    public RecordCommand(AudioPlayer player) {
        super(player);
    }
    public void execute() {
        player.recordAction();
    }
}
```

```
class Keypad{
    private PlayerCommand playCmd;
    private PlayerCommand recordCmd;
    private PlayerCommand forwardCmd;
    private PlayerCommand backwardCmd;
    private PlayerCommand stopCmd;
    private PlayerCommand pauseCmd;

    private AudioPlayer player;

    public Keypad(AudioPlayer player){
        this.player = player;
        playCmd = new PlayCommand(player);
        recordCmd = new RecordCommand(player);
        forwardCmd = new ForwardCommand(player);
        backwardCmd = new BackwardCommand(player);
        stopCmd = new StopCommand(player);
        pauseCmd = new PauseCommand(player);
    }
}
```

```
    public void play(){
        playCmd.execute();
    }

    public void record(){
        recordCmd.execute();
    }

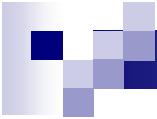
    public void backward(){
        backwardCmd.execute();
    }

    public void forward(){
        forwardCmd.execute();
    }

    public void stop(){
        stopCmd.execute();
    }

    public void pause(){
        pauseCmd.execute();
    }
}
```

```
class Client{
    public void testCommand() {
        Keypad keypad = new Keypad(new SonyAudioPlayer());
        keypad.play();
        keypad.stop();
    }
}
```



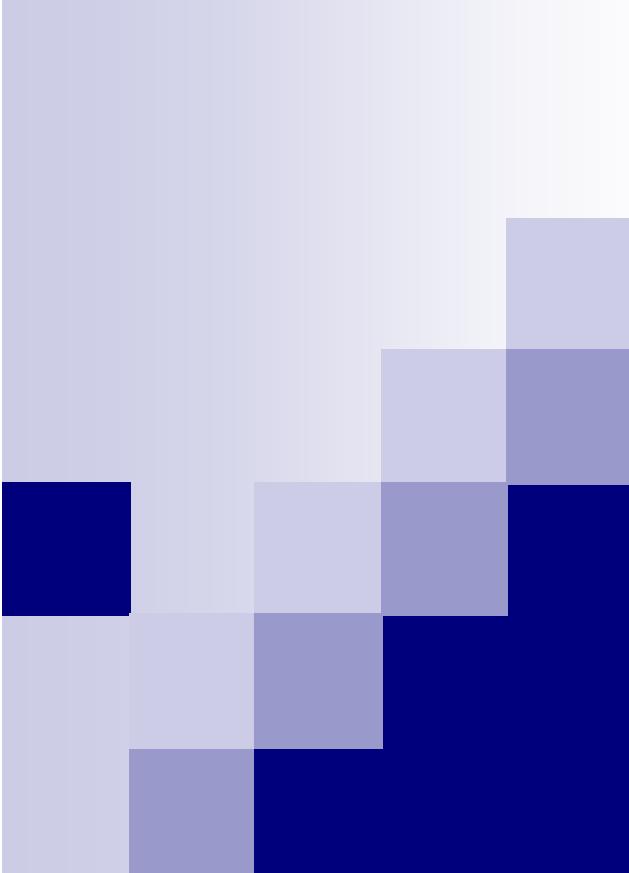
Extension: Macro command set

- A macro is a rule or pattern that specifies how a certain input sequence should be mapped to an output sequence.
 - A macro command set is pre-defined sequence which contains certain commands in specified order.
 - Command pattern is easy to implemented macro command set.
 - Macro command can be implemented by **Aggregate** and **Iterator** pattern.
-



Let's go to next...





Design Patterns

宋 杰

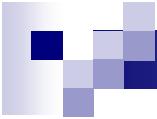
Song Jie

东北大学 软件学院

Software College, Northeastern
University

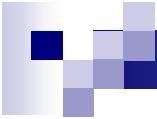


19. Observer Pattern



Intent

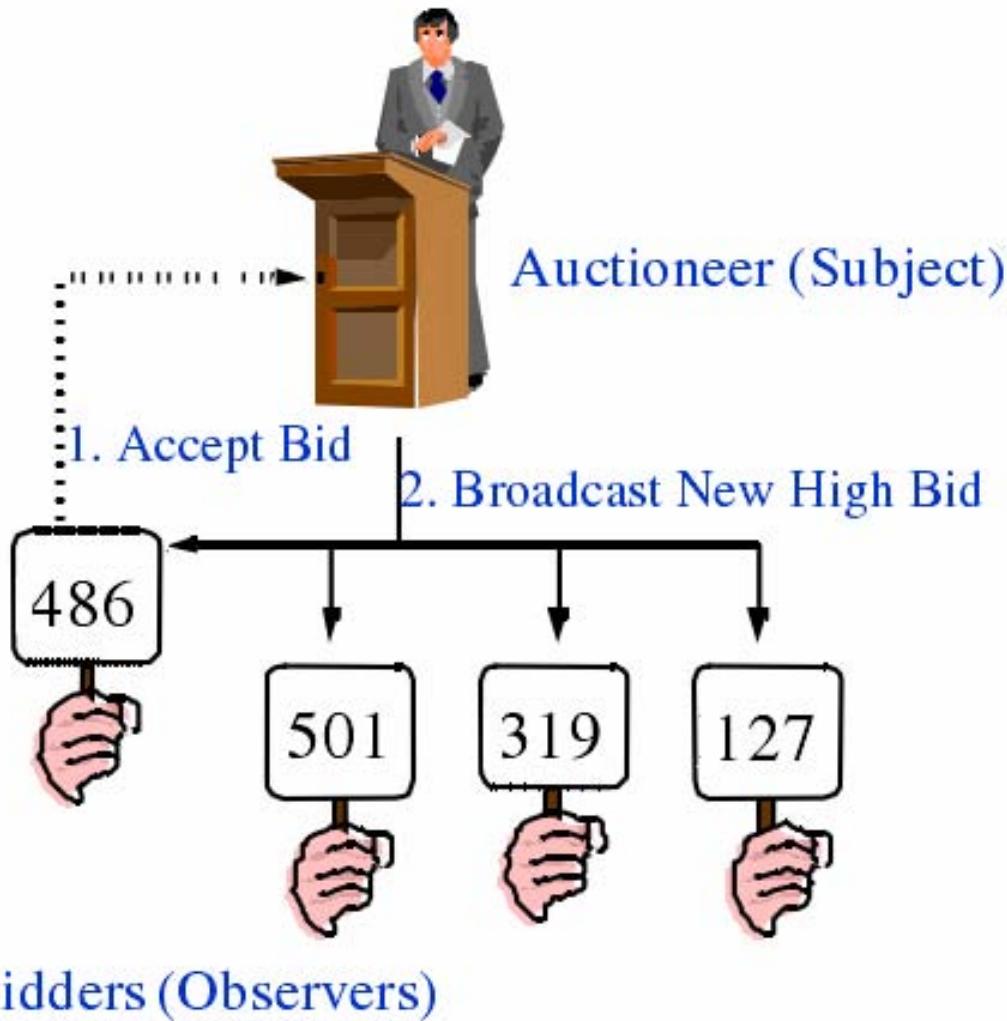
- Define a **one-to-many dependency** between objects so that when one object changes state, all its dependents are notified and updated automatically.
 - 观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。
-



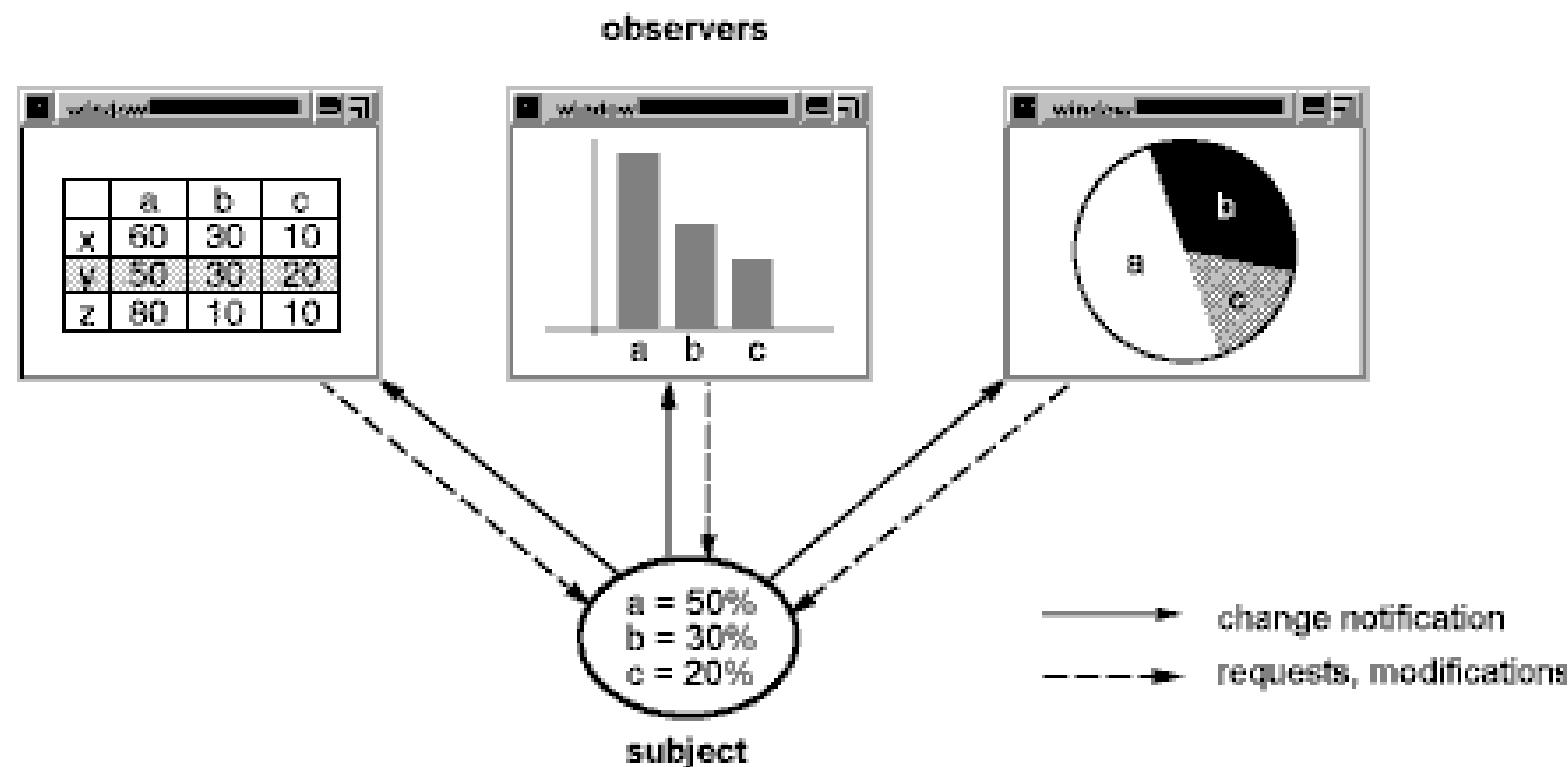
Intent

- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
 - You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.
-

Example



Example



```
interface Auctioneer {  
    public void attach(Bidder bidder);  
    public void detach(Bidder bidder);  
    public void clear();  
    public void notifying();  
    public void asking();  
    public boolean accept();  
    public Bidder currentBidder();  
}  
  
abstract class AbstractAuctioneer implements Auctioneer {  
    protected List<Bidder> bidders;  
    public AbstractAuctioneer() {  
        bidders = new ArrayList<Bidder>();  
    }  
    public void attach(Bidder bidder) {  
        if (!bidders.contains(bidder)) {  
            bidders.add(bidder);  
        }  
    }  
    public void detach(Bidder bidder) {  
        bidders.remove(bidder);  
    }  
    public void clear() {  
        bidders.clear();  
    }  
}
```

AuctioneerImpl Part 1

```
class AuctioneerImpl extends AbstractAuctioneer implements Auctioneer {  
    private Bidder currentBidder;  
    private int notifiedCount = 0;  
    private int maxNotifiedCount = 3;  
    public AuctioneerImpl(int initPrice, int maxNotifiedCount) {  
        currentBidder = new BidderImpl("Init", 0, 0);  
        currentBidder.updatePrice(initPrice);  
        this.maxNotifiedCount = maxNotifiedCount;  
    }  
    public Bidder currentBidder() {  
        return currentBidder;  
    }  
}
```

AuctioneerImpl Part 2

```
public void asking() {
    boolean bidderChanged = false;
    for (Iterator<Bidder> it = bidders.iterator(); it.hasNext();) {
        Bidder bidder = it.next();
        boolean state = bidder.bidding();
        if (!state) {
            it.remove();
            System.out.println(bidder + " quit!");
        } else if (currentBidder.getPrice() < bidder.getPrice()) {
            currentBidder = bidder;
            bidderChanged = true;
        }
    }
    if (!bidderChanged) {
        notifiedCount++;
        System.out.println("Notified" + notifiedCount);
    }
}
```

AuctioneerImpl Part 3

```
public void notifying() {
    for (Bidder bidder : bidders) {
        bidder.updatePrice(currentBidder.getPrice());
    }
}
public boolean accept() {
    if (notifiedCount >= maxNotifiedCount) {
        this.clear();
        System.out.println("Accept:" + currentBidder);
        return true;
    }
    return false;
}
```

```
interface Bidder {  
    public void setAuctioneer(Auctioneer auctioneer);  
    public String getName();  
    public int getPrice();  
    public void updatePrice(int price);  
    public boolean bidding();  
    public void plan();  
}
```

```
class BidderImpl implements Bidder {

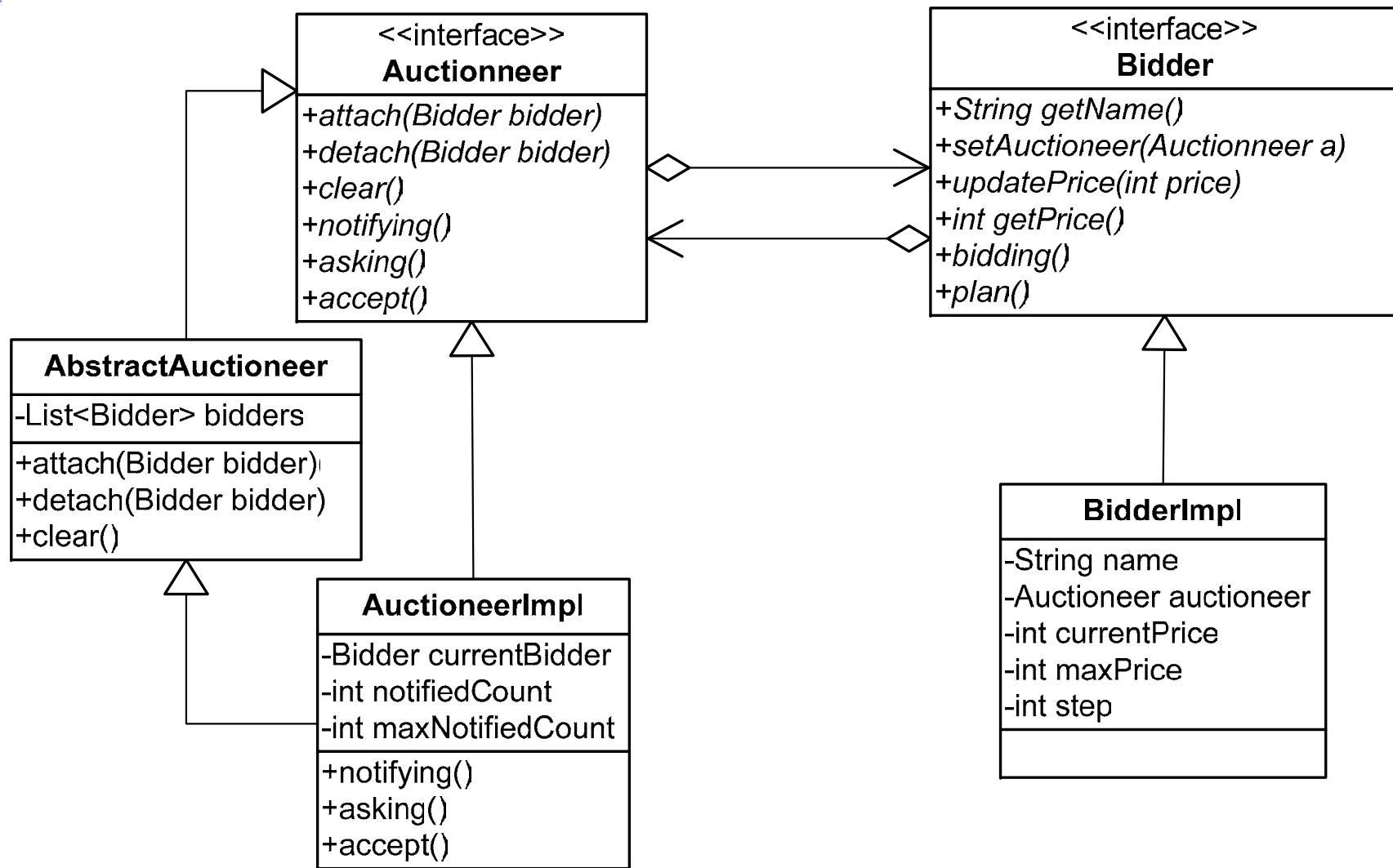
    private String name;
    private Auctioneer auctioneer;
    private int currentPrice;
    private int maxPrice;
    private int step;

    public BidderImpl(String name, int maxPrice, int step) {
        this.name = name;
        this.maxPrice = maxPrice;
        this.step = step;
    }
    public void setAuctioneer(Auctioneer auctioneer) {
        this.auctioneer = auctioneer;
        auctioneer.attach(this);
    }
    public String getName() {
        return this.name;
    }
    public int getPrice() {
        return currentPrice;
    }
}
```

```
public boolean bidding() {
    if (auctioneer == null) {
        return false;
    }
    if (auctioneer.currentBidder() == this) {
        return true;
    }
    if (currentPrice > maxPrice) {
        return false;
    }
    int price = currentPrice + step;
    currentPrice = price < maxPrice ? price : maxPrice;
    System.out.println(this);
    return true;
}
public void updatePrice(int price) {
    this.currentPrice = price;
}
// Use Strategy pattern or Template Method pattern here
public void plan() {
    // Defining the bidding strategy dynamically
    // currentPrice, maxPrice, step
}
public String toString() {
    return this.name + ": " + this.currentPrice;
}
}
```

```
public void testBidding() {  
    Auctioneer auctioneer = new AuctioneerImpl(50, 3);  
    Bidder tom = new BidderImpl("Tom", 100, 5);  
    Bidder jack = new BidderImpl("Jack", 120, 10);  
    Bidder marry = new BidderImpl("Marry", 150, 20);  
    Bidder aike = new BidderImpl("Aike", 200, 20);  
    tom.setAuctioneer(auctioneer);  
    jack.setAuctioneer(auctioneer);  
    marry.setAuctioneer(auctioneer);  
    aike.setAuctioneer(auctioneer);  
    while (!auctioneer.accept()) {  
        auctioneer.notifying();  
        auctioneer.asking();  
    }  
}
```

- Tom: 55
- Jack: 60
- Marry: 70
- Aike: 70
- Tom: 75
- Jack: 80
- Marry: 90
- Aike: 90
- Tom: 95
- Jack: 100
- Marry: 110
- Aike: 110
- Tom: 110 quit!
- Jack: 120
- Marry: 130
- Aike: 130
- Jack: 130 quit!
- Aike: 150
- Marry: 150
- Notified1
- Marry: 150
- Notified2
- Marry: 150
- Notified3
- Accept:Aike:
150



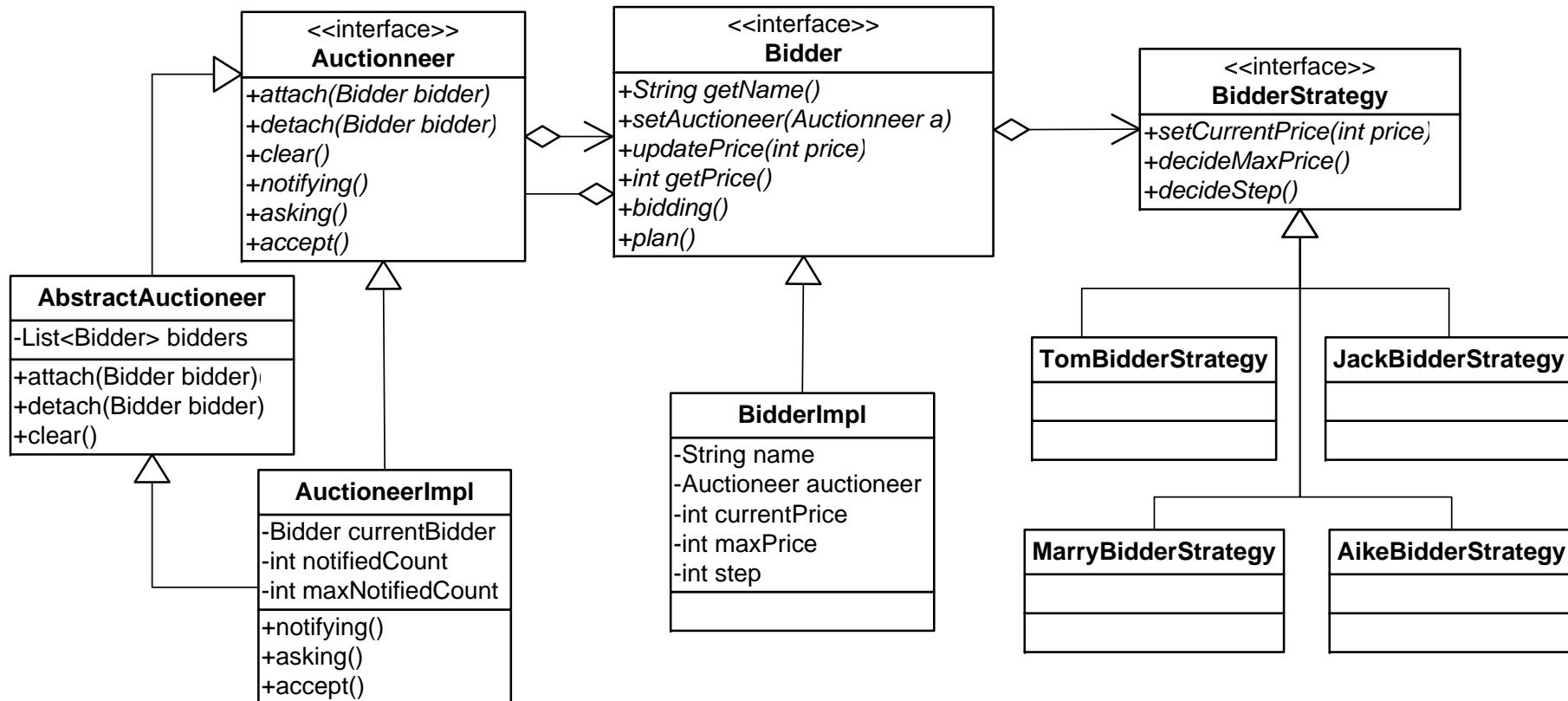
```
interface BidderStrategy {  
    public void setCurrentPrice(int price);  
    public int decideMaxPrice();  
    public int decideStep();  
}  
class TomBidderStrategy implements BidderStrategy {  
    private int currentPrice;  
    public int decideMaxPrice() {  
        return currentPrice *2;  
    }  
    public int decideStep() {  
        return currentPrice / 10;  
    }  
    public void setCurrentPrice(int price) {  
        this.currentPrice = price;  
    }  
}
```

```
class StrategyBidder implements Bidder {  
    private String name;  
    private Auctioneer auctioneer;  
    private BidderStrategy strategy;  
    private int currentPrice;  
    private int maxPrice;  
    private int step;  
    public StrategyBidder(String name, BidderStrategy strategy) {  
        this.name = name;  
        this.strategy = strategy;  
    }  
    public void setAuctioneer(Auctioneer auctioneer) {  
        this.auctioneer = auctioneer;  
        auctioneer.attach(this);  
    }  
    public String getName() {  
        return this.name;  
    }  
    public int getPrice() {  
        return currentPrice;  
    }
```

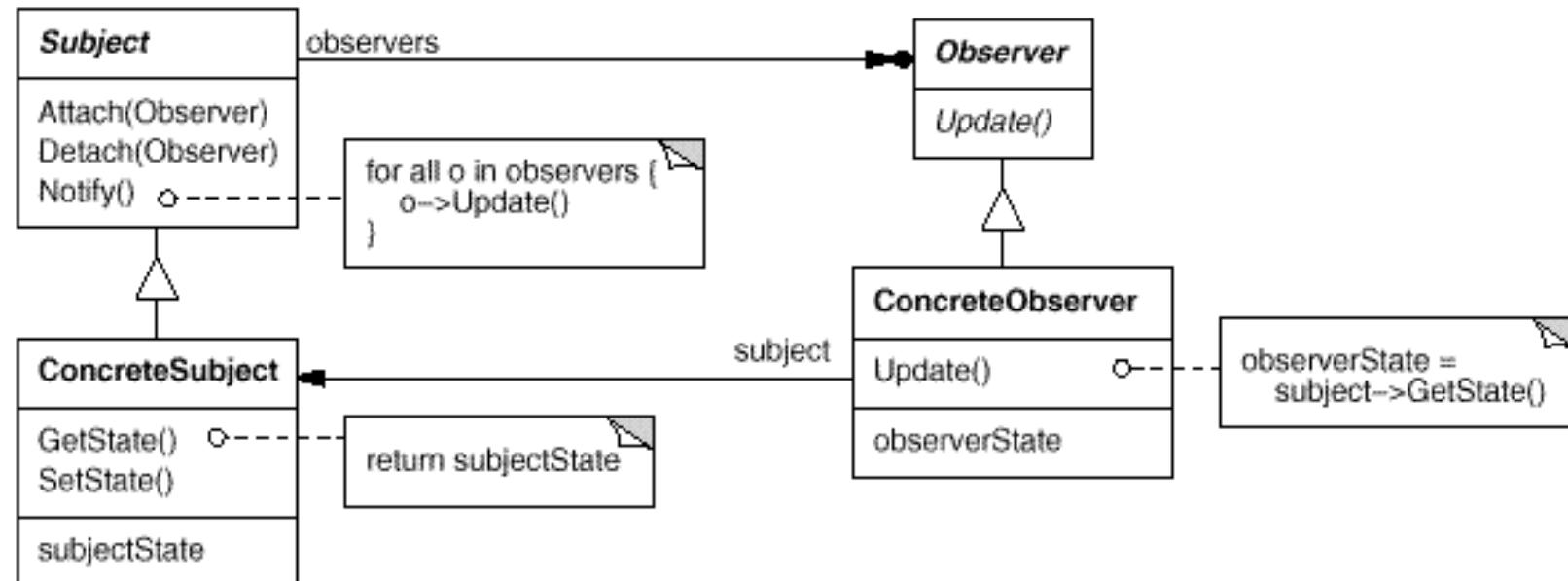
```
public boolean bidding() {
    if (auctioneer == null) {
        return false;
    }
    if (auctioneer.currentBidder() == this) {
        return true;
    }
    if (currentPrice > maxPrice) {
        return false;
    }
    int price = currentPrice + step;
    currentPrice = price < maxPrice ? price : maxPrice;
    System.out.println(this);
    return true;
}

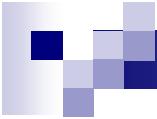
public void updatePrice(int price) {
    this.currentPrice = price;
    plan();
}
public void plan() {
    strategy.setCurrentPrice(currentPrice);
    maxPrice = strategy.decideMaxPrice();
    step = strategy.decideStep();
}
public String toString() {
    return this.name + ": " + this.currentPrice;
}
```

```
public void testStrategyBidding() {  
    Auctioneer auctioneer = new AuctioneerImpl(50, 3);  
    Bidder tom = new StrategyBidder("Tom", new TomBidderStrategy());  
    // Bidder jack = new StrategyBidder("Jack",new JackBidderStrategy());  
    // Bidder marry = new StrategyBidder("Marry",new MarryBidderStrategy());  
    // Bidder aike = new StrategyBidder("Aike",new AikeBidderStrategy());  
    tom.setAuctioneer(auctioneer);  
    // jack.setAuctioneer(auctioneer);  
    // marry.setAuctioneer(auctioneer);  
    // aike.setAuctioneer(auctioneer);  
    while (!auctioneer.accept()) {  
        auctioneer.notifying();  
        auctioneer.asking();  
    }  
}
```



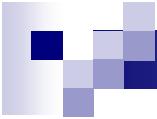
Structure





Participants

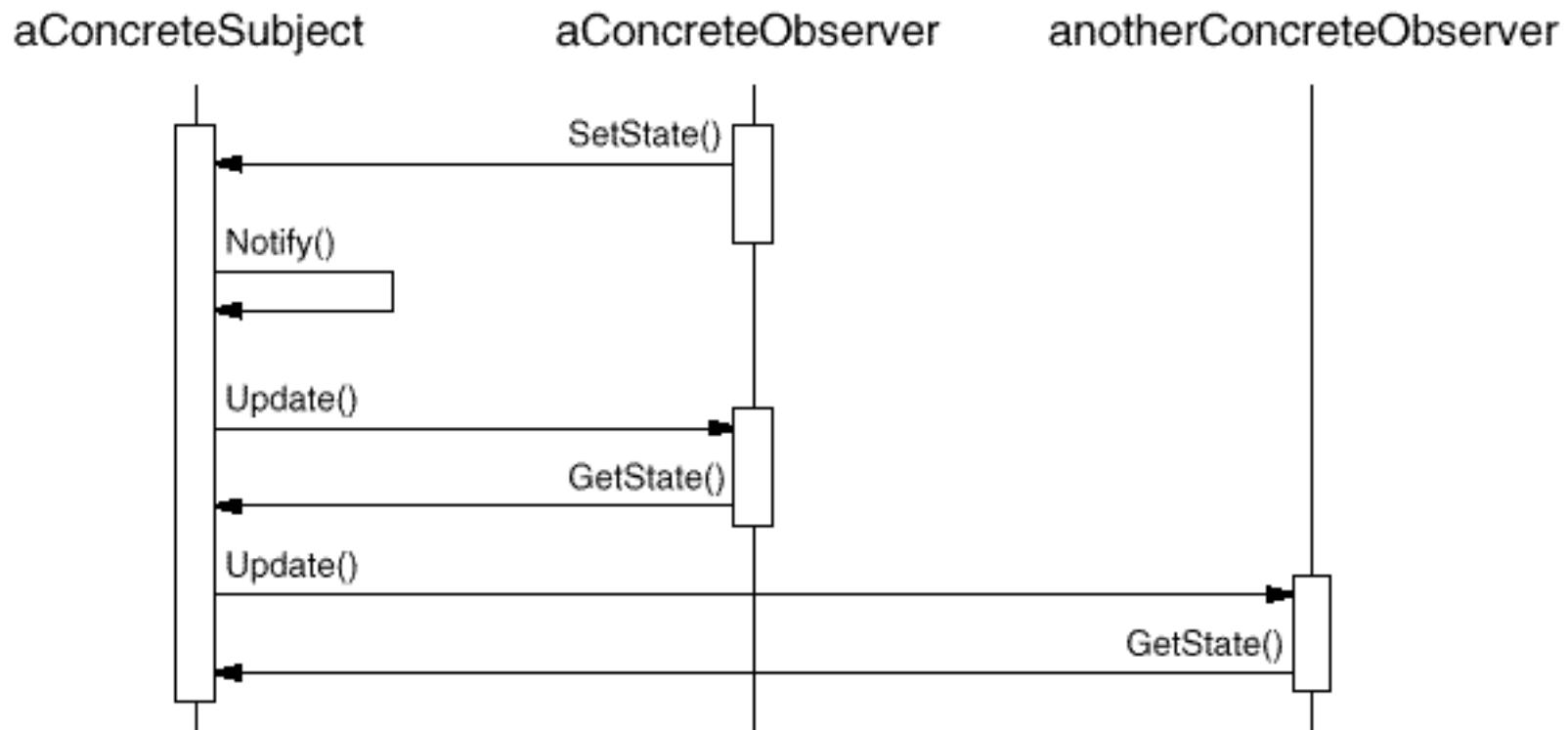
- **Subject**
 - Knows its **observers**. Any number of **Observer** objects may observe a **subject**.
 - Provides an interface for attaching and detaching **Observer** objects.
 - **Observer**
 - Defines an updating interface for objects that should be notified of changes in a **subject**.
 - **ConcreteSubject**
 - Stores state of interest to **ConcreteObserver** objects.
 - Sends a notification to its **observers** when its state changes.
 - **ConcreteObserver**
 - Maintains a reference to a **ConcreteSubject** object.
 - Stores state that should stay consistent with the **subject**'s.
 - Implements the **Observer** updating interface to keep its state consistent with the **subject**'s.
-



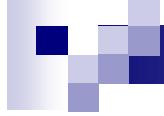
Collaborations

- **ConcreteSubject** notifies its **observers** whenever a change occurs that could make its **observers'** state inconsistent with its own.
 - After being informed of a change in the **ConcreteSubject**, a **ConcreteObserver** object may query the **subject** for information. **ConcreteObserver** uses this information to reconcile its state with that of the **subject**.
-

Collaborations

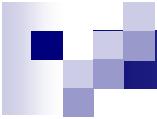


- Note how the **Observer** object that initiates the change request postpones its update until it gets a notification from the **subject**.



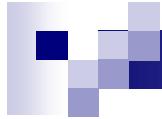
Consequences - advantages

- Abstract coupling between **Subject** and **Observer**.
 - All a **subject** knows is that it has a list of **observers**, each conforming to the simple interface of the abstract **Observer** class.
 - The **subject** doesn't know the concrete class of any **observer**.
 - The coupling between **subjects** and **observers** is abstract and minimal.
 - **Subject** and **Observer** belong to different layers of abstraction in a system. (DIP)
 - If **Subject** and **Observer** are lumped together, then the resulting object must either span two layers (and violate the layering), or it must be forced to live in one layer or the other (which might compromise the layering abstraction).
-



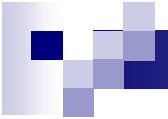
Consequences - advantages

- Support for broadcast communication.
 - The notification that a **subject** sends needn't specify its receiver.
 - The notification is broadcast automatically to all interested objects that subscribed to it.
 - The **subject** doesn't care how many interested objects exist; its only responsibility is to notify its **observers**.
 - This gives you the freedom to add and remove **observers** at any time.
-



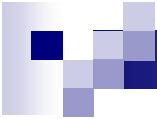
Consequences – drawbacks

- Unexpected updates
 - Observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject.
 - A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects.
 - Moreover, dependency criteria that aren't well-defined or maintained usually lead to spurious (伪造的) updates, which can be hard to track down.
 - This problem is aggravated by the fact that the simple update protocol provides no details on what changed in the subject.
-



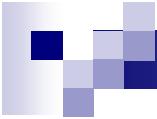
Consequences – drawbacks

- If a **subject** have many **observers**, it is time-costly to notify them all, especially when some method should be synchronized;
 - If the **observers** depends each other circularly, the method will be invoked circularly (dead lock);
 - Concurrence of the **observers** accessing the **subject** should be well considered ;
 - The **observers** is only to know the **subject** is changed, but hard to know how a **subject** is modified.
-



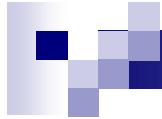
Applicability

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
 - When a change to one object requires changing others, and you don't know how many objects need to be changed.
 - When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.
-



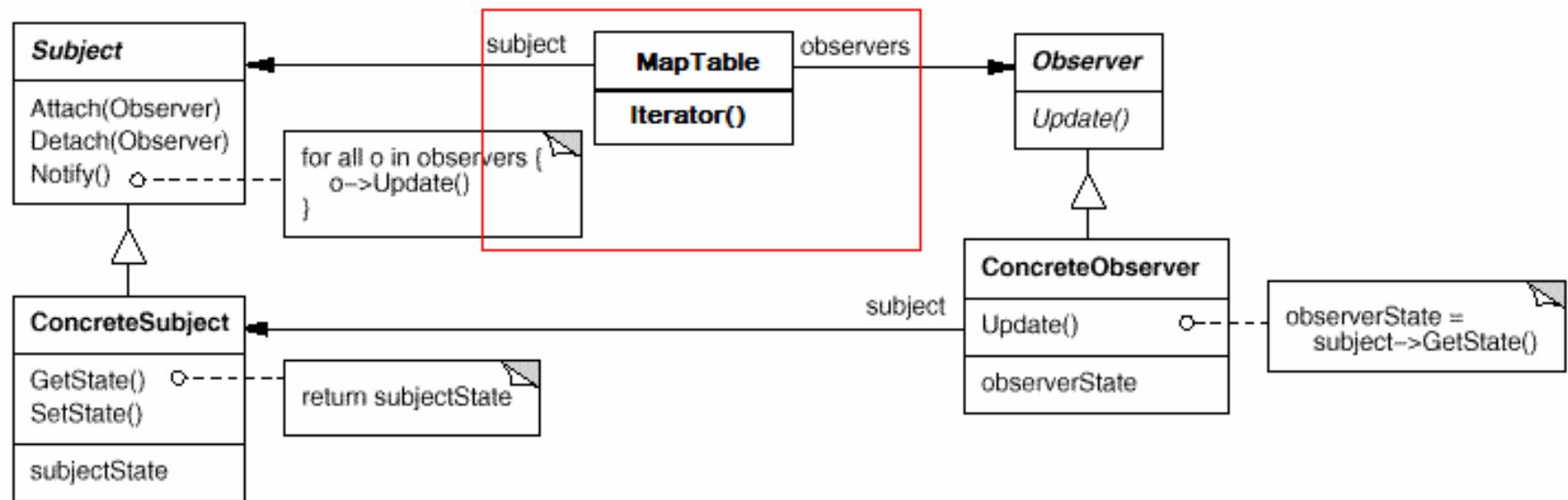
Implementation 1: Observing more than one **subject**

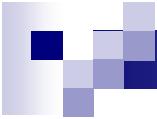
- It might make sense in some situations for an **observer** to depend on more than one **subject**.
 - For example, a spreadsheet may depend on more than one data source.
 - It's necessary to extend the Update interface in such cases to let the **observer** know which **subject** is sending the notification.
-



Implementation 2: Mapping **subjects** to their **observers**.

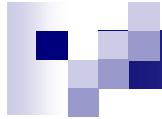
- Store references to **observers** explicitly in the **subject**.
 - Such storage may be too expensive when there are many **subjects** and few **observers**.
 - Using an associative look-up (hash table) to maintain the **subject-to-observer** mapping.
(trade space for time)
 - This approach increases the cost of accessing the **observers**.
-





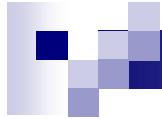
Implementation 3: Who triggers the update?

- Setting operations on **Subject** call notify after they change the **subject's** state automatically.
- Make clients (or one of **observers**) responsible for calling **Notify** at the right time.
 - Avoiding needless intermediate updates.
 - Clients have an added responsibility to trigger the update, clients might forget to call **Notify**.



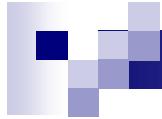
Implementation 4: Deleting a **subject** or a **observer**

- Deleting a **subject** should not produce dangling references in its **observers**. vice versa.
 - One way to avoid dangling references is to make the **subject** notify its **observers** as it is deleted so that they can reset their reference to it.
-



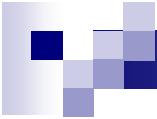
Implementation 5: Making sure **Subject** state is self-consistent while notification

- It's important to make sure **Subject** state is self-consistent while calling *Notify*, because **observers** query the **subject** for its current state in the course of updating their own state.
 - Change state first, then notify it later;
 - Every changes should be notified;



Implementation 6: Avoiding observer-specific update protocols

- Different **observers** may interest different *Update*, the amount of information may vary widely.
 - Update without information: the update only be treat as an notification without state (data).
 - Push model (Extreme condition): the **subject** sends **observers** detailed information about the change, whether they want it or not.
 - Pull model (Extreme condition): the **subject** sends nothing but the most minimal notification, and **observers** ask for details explicitly thereafter.
-



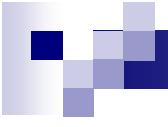
Push vs Pull

■ Pull model

- Emphasizes the **subject's** ignorance of its **observers**
- May be inefficient, because **Observer** classes must ascertain what changed without help from the **Subject**.

■ Push model

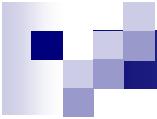
- Assumes **subjects** know something about their **observers'** needs
 - Make **observers** less reusable. because **Subject** classes make assumptions about **Observer** classes that might not always be true.
-



Implementation 7: Specifying modifications of interest explicitly

- Improve update efficiency by extending the **subject**'s registration interface to allow registering **observers** only for specific events of interest.

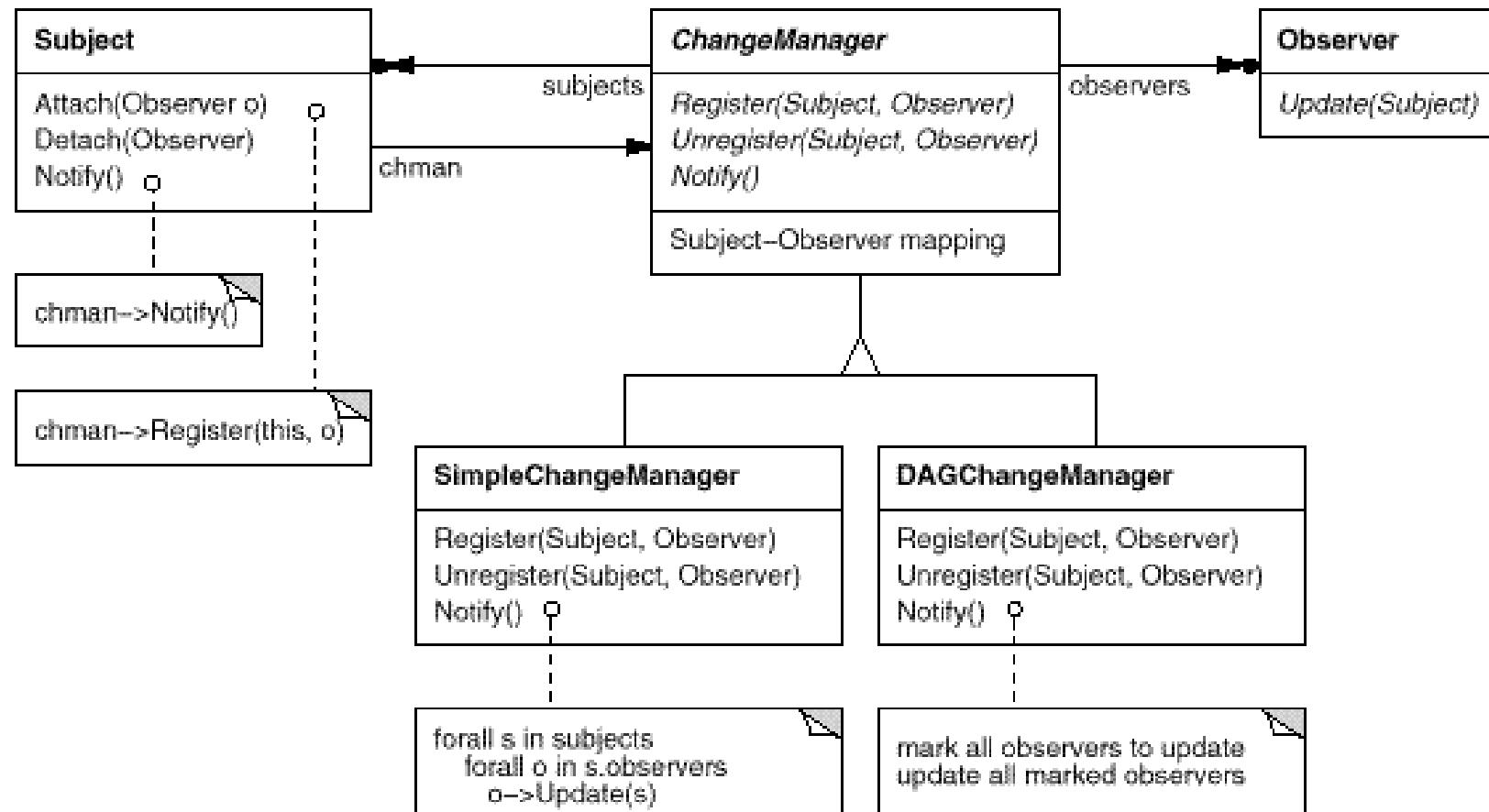
```
public void attach (Observer observer ,Interest interest);
```

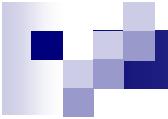


Implementation 8: Encapsulating complex update semantics.

- Dependency relationship between **subjects** and **observers** can be particularly complex.
 - If an operation involves changes to several interdependent **subjects**, you might have to ensure that their **observers** are notified only after *all* the **subjects** have been modified to avoid notifying **observers** more than once.
 - An object that maintains these relationships might be required.
-

ChangeManager





ChangeManager

- It maps a **subject** to its **observers** and provides an interface to maintain this mapping. This eliminates the need for **subjects** to maintain references to their **observers** and vice versa.
 - It defines a particular update strategy.
 - It updates all dependent **observers** at the request of a **subject**.
 - **ChangeManager** is an instance of the Mediator pattern
-

Example: java.util.Observer java.util.Observable

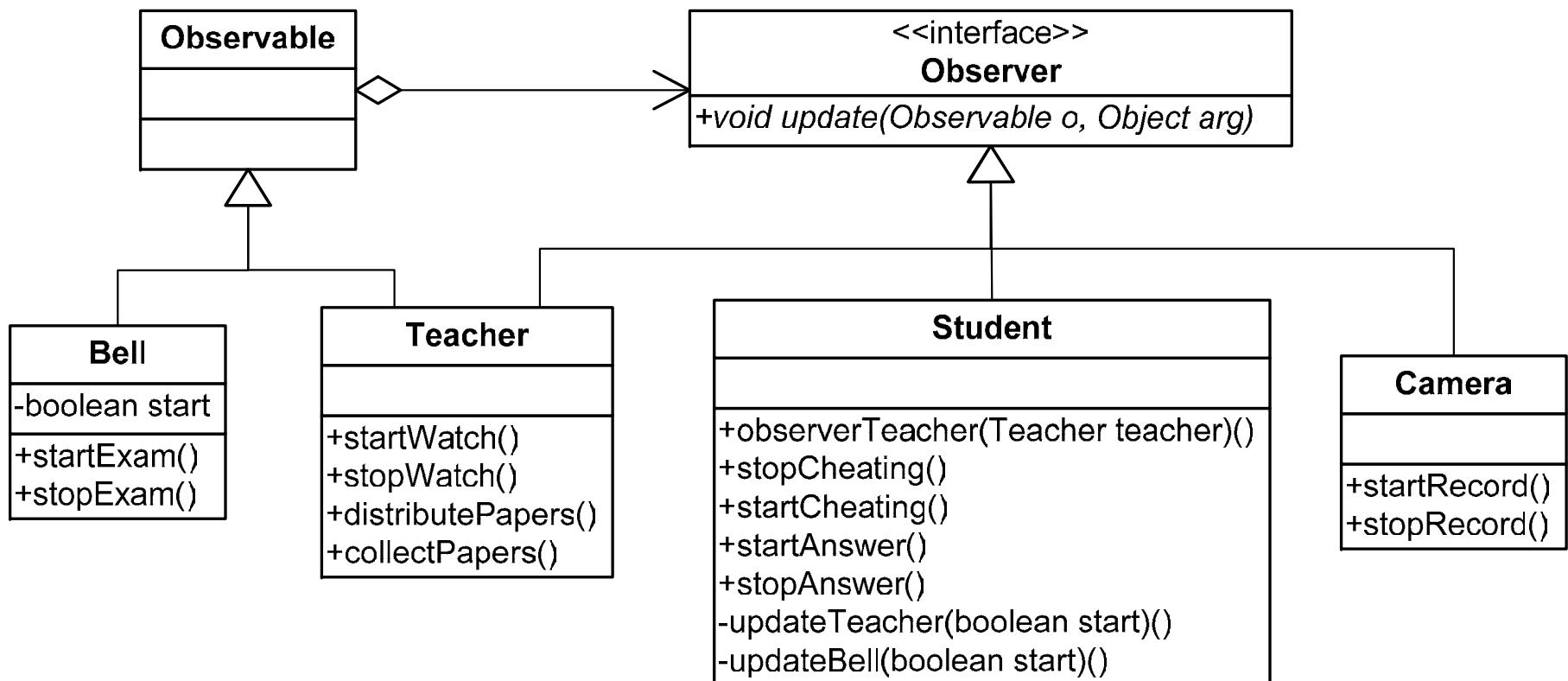
```
<<interface>>
Observer
+void update(Observable o, Object arg)
```

```
Observable
+
+
+	synchronized void addObserver(Observer o)
+	synchronized void deleteObserver(Observer o)
+	void notifyObservers()
+	void notifyObservers(Object arg)
+	synchronized void deleteObservers()
+	synchronized void setChanged()
+	synchronized void clearChanged()
+	synchronized boolean hasChanged()
+	synchronized int countObservers()
```

```
public class Observable {  
    private boolean changed = false;  
    private Vector obs;  
    public Observable() {  
        obs = new Vector();  
    }  
    public synchronized void addObserver(Observer o) {  
        if (o == null)  
            throw new NullPointerException();  
        if (!obs.contains(o)) {  
            obs.addElement(o);  
        }  
    }  
    public synchronized void deleteObserver(Observer o) {  
        obs.removeElement(o);  
    }  
    public synchronized void deleteObservers() {  
        obs.removeAllElements();  
    }  
    protected synchronized void setChanged() {  
        changed = true;  
    }  
}
```

```
protected synchronized void clearChanged() {
    changed = false;
}
public synchronized boolean hasChanged() {
    return changed;
}
public synchronized int countObservers() {
    return obs.size();
}
public void notifyObservers() {
    notifyObservers(null);
}
public void notifyObservers(Object arg) {
    Object[] arrLocal;
    synchronized (this) {
        if (!changed) {
            return;
        }
        arrLocal = obs.toArray();
        clearChanged();
    }
    for (int i = arrLocal.length - 1; i >= 0; i--)
        ((Observer) arrLocal[i]).update(this, arg);
}
}
```

Example using Observer and Observable



```
class Bell extends Observable {  
    public void startExam() {  
        System.out.println(this + ": start exam.");  
        this.setChanged();  
        this.notifyObservers(new Boolean(true));  
    }  
    public void stopExam() {  
        System.out.println(this + ": stop exam.");  
        this.setChanged();  
        this.notifyObservers(new Boolean(false));  
    }  
    public String toString() {  
        return "Bell";  
    }  
}
```

```
class Camera implements Observer {
    private String id;
    private Bell bell;
    public Camera(String id, Bell bell) {
        this.id = id;
        this.bell = bell;
        this.bell.addObserver(this);
        System.out.println(this + ": attach to " + bell + ".");
    }
    public void startRecord() {
        System.out.println(this + ": start record.");
    }
    public void stopRecord() {
        System.out.println(this + ": stop record.");
        this.bell.deleteObserver(this);
        System.out.println(this + ": detach from " + bell + ".");
        this.bell = null;
    }
    public void update(Observable o, Object arg) {
        boolean start = ((Boolean) arg).booleanValue();
        if (o == this.bell) {
            if (start) {
                startRecord();
            } else {
                stopRecord();
            }
        }
    }
    public String toString() {
        return id;
    }
}
```

```
class Teacher extends Observable implements Observer {  
    private String id;  
    private Bell bell;  
  
    public Teacher(String id, Bell bell) {  
        this.id = id;  
        this.bell = bell;  
        this.bell.addObserver(this);  
        System.out.println(this + ": attach to " + bell + ".");  
    }  
    public void startWatch() {  
        System.out.println(this + ": start watch.");  
        this.setChanged();  
        this.notifyObservers(new Boolean(true));  
    }  
    public void stopWatch() {  
        System.out.println(this + ": stop watch.");  
        this.setChanged();  
        this.notifyObservers(new Boolean(false));  
    }  
}
```

```
public void distributePapers() {
    System.out.println(this + ": distribute papers.");
}
public void collectPapers() {
    System.out.println(this + ": collect papers.");
    this.bell.deleteObserver(this);
    System.out.println(this + ": detach from " + bell + ".");
    this.bell = null;
}
public void update(Observable o, Object arg) {
    boolean start = ((Boolean) arg).booleanValue();
    if (o == this.bell) {
        if (start) {
            distributePapers();
        } else {
            collectPapers();
        }
    }
}
public String toString() {
    return id;
}
}
```

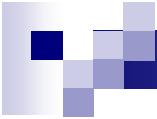
```
class Student implements Observer {
    private List<Teacher> teachers;
    private Bell bell;
    private String id;

    public Student(String id, Bell bell) {
        this.teachers = new ArrayList<Teacher>();
        this.id = id;
        this.bell = bell;
        this.bell.addObserver(this);
        System.out.println(this + ": attach to " + bell + ".");
    }
    public Student observerTeacher(Teacher teacher) {
        teacher.addObserver(this);
        teachers.add(teacher);
        System.out.println(this + ": attach to " + teacher + ".");
        return this;
    }
    public void stopCheating() {
        System.out.println(this + ": stop Cheating.");
    }
    public void startCheating() {
        System.out.println(this + ": start Cheating.");
    }
    public void startAnswer() {
        System.out.println(this + ": start Answer.");
    }
}
```

```
public void stopAnswer() {
    System.out.println(this + ": stop Answer.");
    this.bell.deleteObserver(this);
    System.out.println(this + ": detach from " + bell + ".");
    this.bell = null;
    for (Teacher teacher : teachers) {
        teacher.deleteObserver(this);
        System.out.println(this + ": detach from " + teacher + ".");
    }
    teachers.clear();
}
public void update(Observable o, Object arg) {
    boolean start = ((Boolean) arg).booleanValue();
    if (this.teachers.contains(o)) {
        updateTeacher(start);
    } else if (o == this.bell) {
        updateBell(start);
    }
}
private void updateTeacher(boolean start) {
    if (start) {
        stopCheating();
    } else {
        startCheating();
    }
}
private void updateBell(boolean start) {
    if (start) {
        startAnswer();
    } else {
        stopAnswer();
    }
}
public String toString() {
    return id;
}
}
```

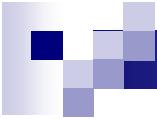
```
public void exam() {  
    Bell bell = new Bell();  
  
    Teacher teacherA = new Teacher("Teacher A", bell);  
    Teacher teacherB = new Teacher("Teacher B", bell);  
  
    new Camera("Camera A", bell);  
    new Camera("Camera B", bell);  
  
    new Student("Student 1", bell).observerTeacher(teacherA);  
    new Student("Student 2", bell).observerTeacher(teacherB);  
    new Student("Student 3", bell).observerTeacher(teacherA)  
        .observerTeacher(teacherB);  
  
    bell.startExam();  
    teacherA.startWatch();  
    teacherB.startWatch();  
    teacherB.stopWatch();  
    teacherA.stopWatch();  
    bell.stopExam();  
}
```

- Teacher A: attach to Bell.
- Teacher B: attach to Bell.
- Camera A: attach to Bell.
- Camera B: attach to Bell.
- Student 1: attach to Bell.
- Student 1: attach to Teacher A.
- Student 2: attach to Bell.
- Student 2: attach to Teacher B.
- Student 3: attach to Bell.
- Student 3: attach to Teacher A.
- Student 3: attach to Teacher B.
- Bell: start exam.
- Student 3: start Answer.
- Student 2: start Answer.
- Student 1: start Answer.
- Camera B: start record.
- Camera A: start record.
- Teacher B: distribute papers.
- Teacher A: distribute papers.
- Teacher A: start watch.
- Student 3: stop Cheating.
- Student 1: stop Cheating.
- Teacher B: start watch.
- Student 3: stop Cheating.
- Student 2: stop Cheating.
- Teacher B: stop watch.
- Student 3: start Cheating.
- Student 2: start Cheating.
- Teacher A: stop watch.
- Student 3: start Cheating.
- Student 1: start Cheating.
- Bell: stop exam.
- Student 3: stop Answer.
- Student 3: detach from Bell.
- Student 3: detach from Teacher A.
- Student 3: detach from Teacher B.
- Student 2: stop Answer.
- Student 2: detach from Bell.
- Student 2: detach from Teacher B.
- Student 1: stop Answer.
- Student 1: detach from Bell.
- Student 1: detach from Teacher A.
- Camera B: stop record.
- Camera B: detach from Bell.
- Camera A: stop record.
- Camera A: detach from Bell.
- Teacher B: collect papers.
- Teacher B: detach from Bell.
- Teacher A: collect papers.
- Teacher A: detach from Bell.



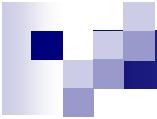
Extension: Delegation Event Model (DEM)

- **Event** : Event are encapsulated in a class hierarchy rooted at `java.util.EventObject`.
 - An event is propagated from a "Source" object to a "Listener" object by invoking a method on the listener and passing in the instance of the event subclass which defines the event type generated.
-



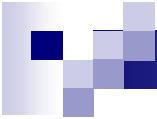
Extension: Delegation Event Model (DEM)

- **Listener** : A Listener is an object that implements a specific **EventListener** interface extended from the generic **java.util.EventListener**.
 - An EventListener interface defines one or more methods which are to be invoked by the event source in response to each specific event type handled by the interface.



Extension: Delegation Event Model (DEM)

- **Source:** An **Event Source** is an object which originates or "fires" events.
 - The source defines the set of events it emits by providing a set of `set<EventType>Listener` (for single-cast) and/or `add<EventType>Listener` (for multi-cast) methods which are used to register specific **listeners** for those events.



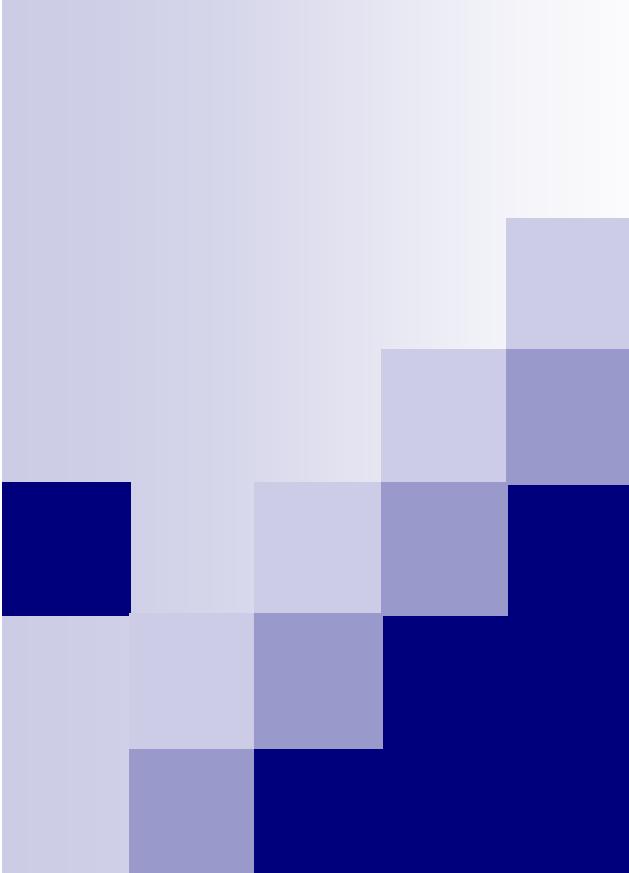
Extension 1: Delegation Event Model (DEM)

- DEM in AWT
 - DEM in Servlet
-



Let's go to next...





Design Patterns

宋 杰

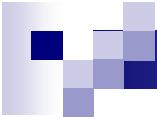
Song Jie

东北大学 软件学院

Software College, Northeastern
University

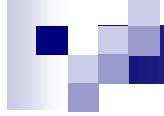


20. Mediator Pattern



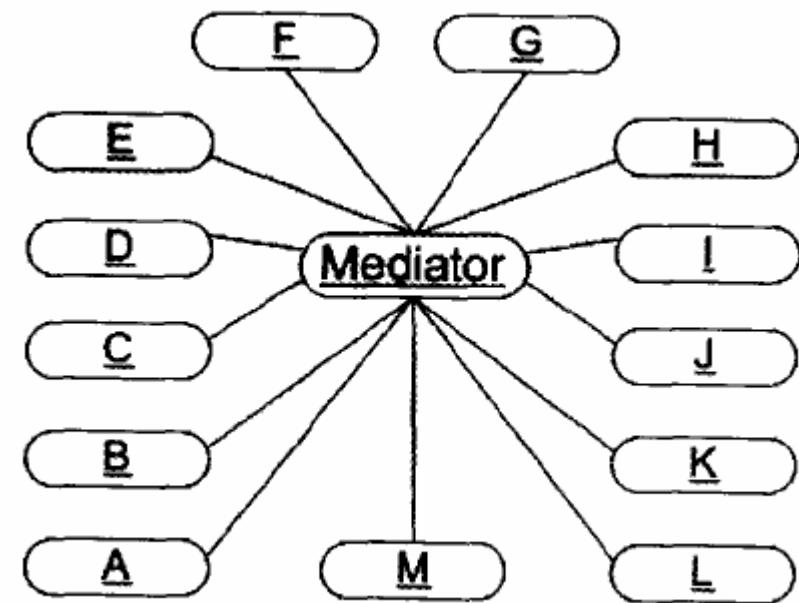
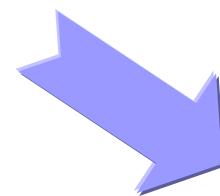
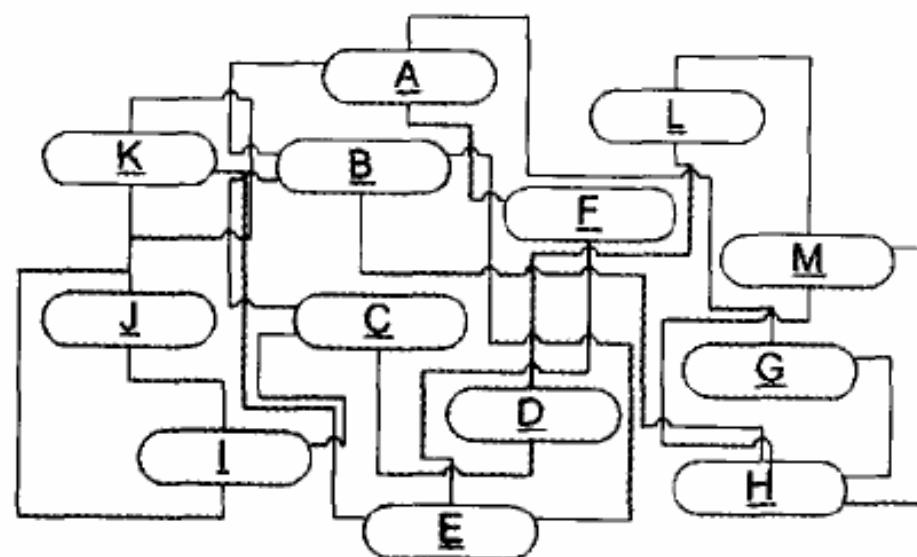
Intent

- Define an object that **encapsulates how a set of objects interact**. Mediator promotes loose coupling by **keeping objects from referring to each other explicitly**, and it lets you vary their interaction independently.
- 调停者模式**包装了一系列对象相互作用的方式**，使得这些对象**不必互相明显引用**。从而使它们可以较松散地耦合。当这些对象中的某些对象之间的相互作用发生改变时，不会立即影响到其他的一些对象之间的相互作用。从而保证这些相互作用可以彼此独立地变化。

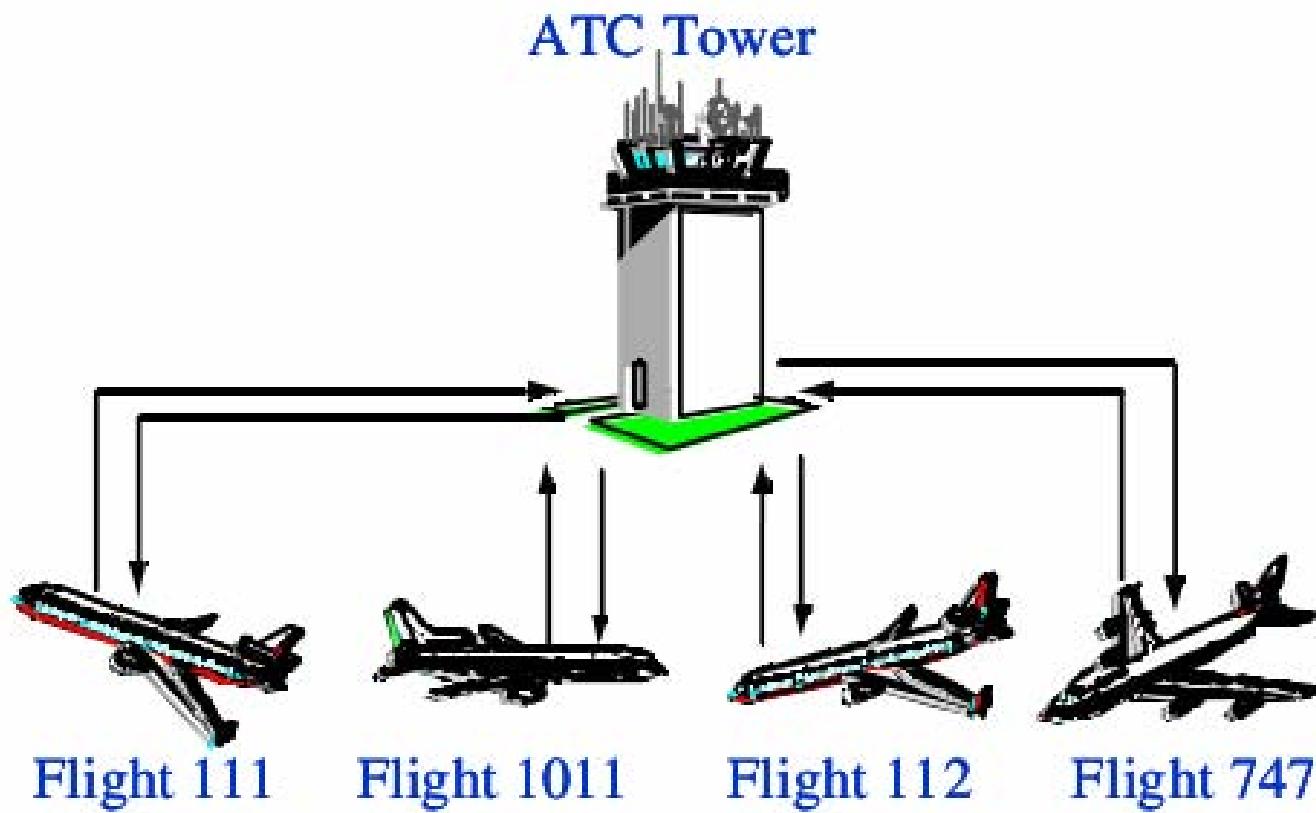


Problems of Object-oriented design

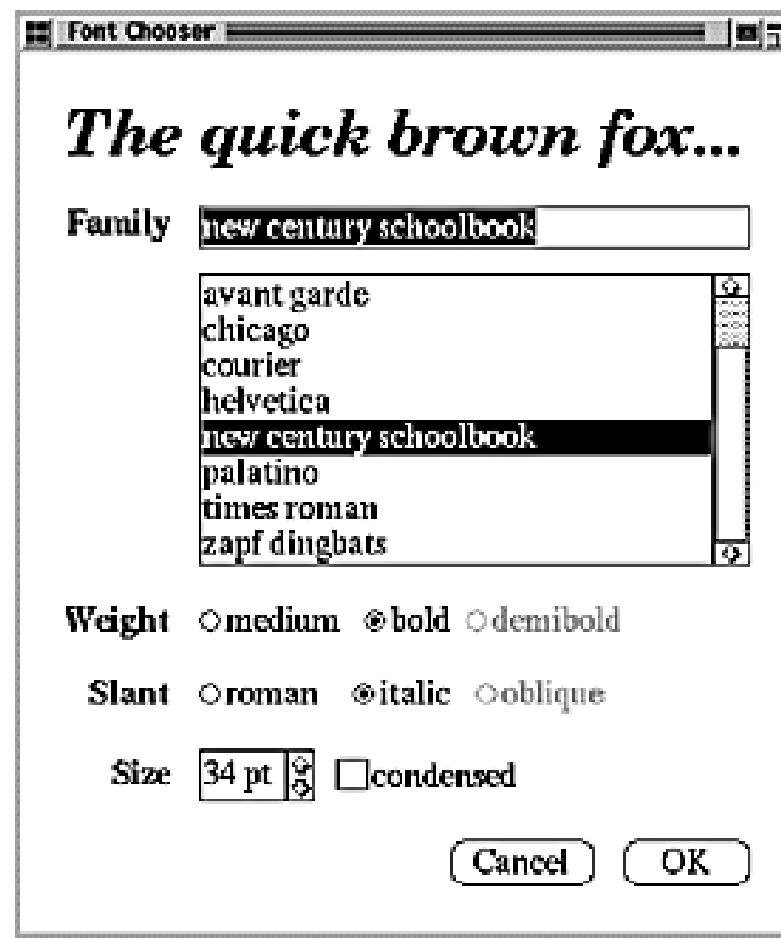
- Object-oriented design encourages the distribution of behavior among objects.
 - Such distribution can result in an object structure with many connections between objects;
 - In the worst case, every object ends up knowing about every other.
 - Though partitioning a system into many objects generally enhances reusability, proliferating interconnections tend to reduce it again.
-



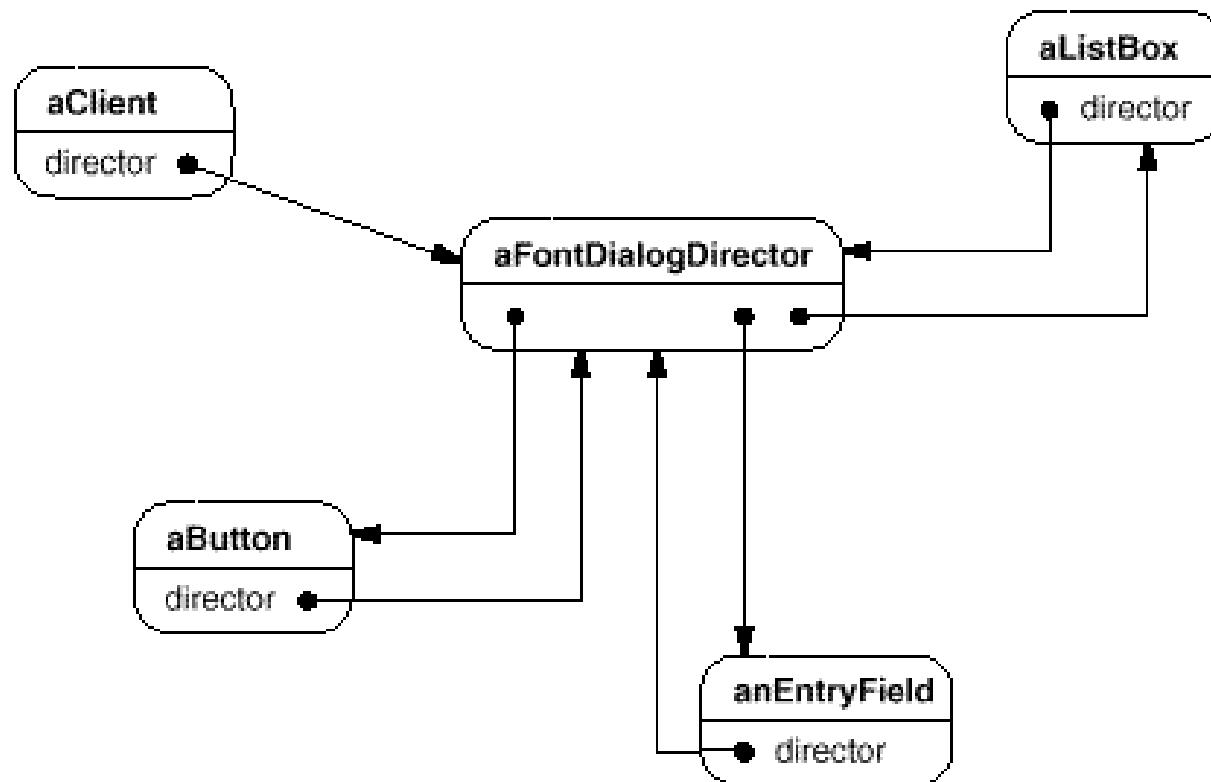
Example:



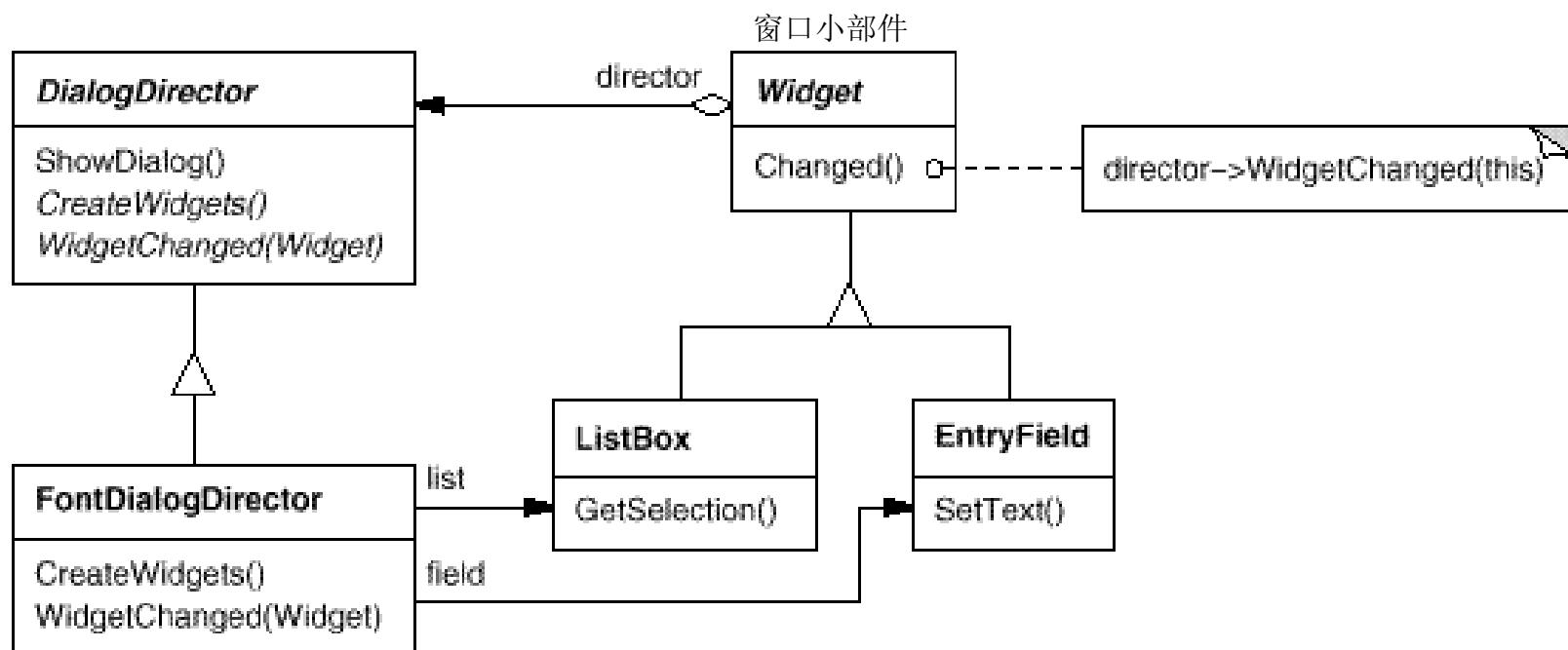
Example: Font box



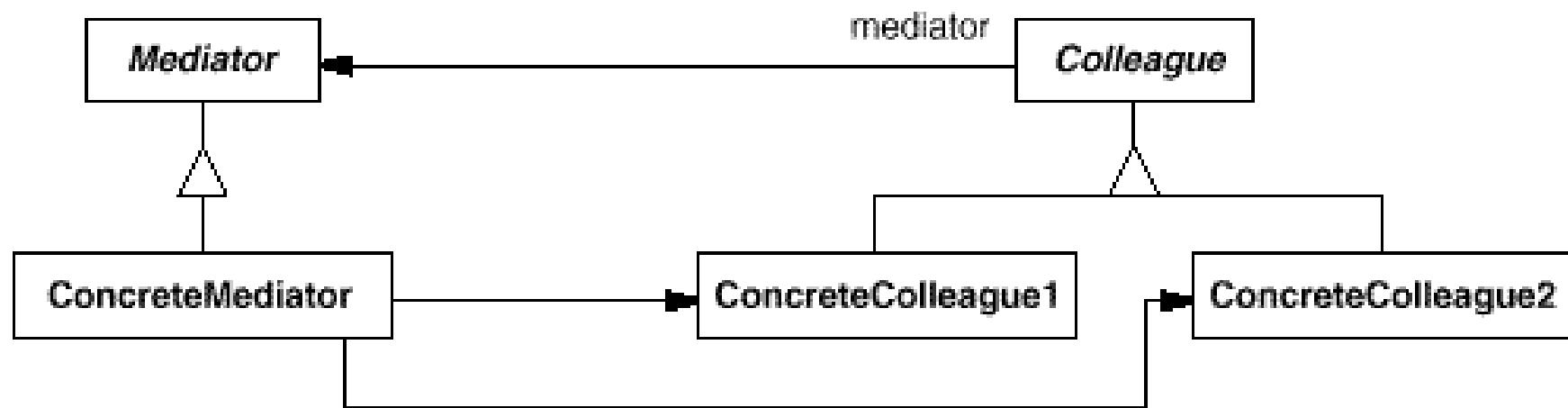
Example: Font box



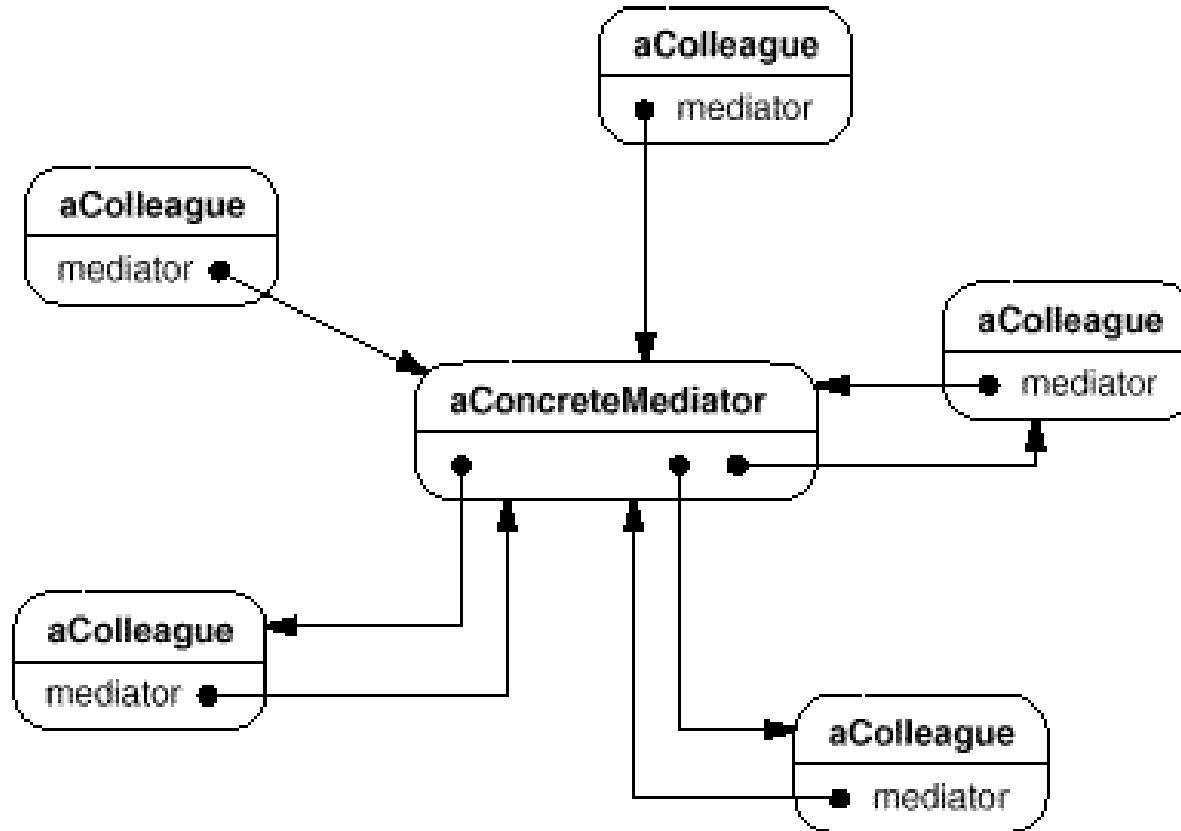
Example: Font box

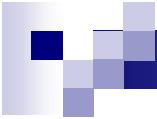


Structure



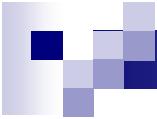
Structure





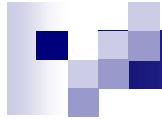
Participants

- **Mediator:** Defines an interface for communicating with **Colleague** objects.
 - **ConcreteMediator:** Implements cooperative behavior by coordinating **Colleague** objects. knows and maintains its **colleagues**.
 - **Colleague classes:**
 - Each **Colleague** class knows its **Mediator** object.
 - Each **Colleague** communicates with its **mediator** whenever it would have otherwise communicated with another **colleague**.
-



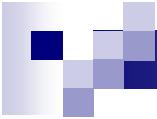
Consequences – advantages

- It decouples colleagues.
 - It simplifies object protocols.
 - It abstracts how objects cooperate.
 - It centralizes control.
 - The Mediator pattern trades complexity of interaction for complexity in the mediator. Because a mediator encapsulates protocols, it can become more complex than any individual colleague. This can make the mediator itself a monolith that's hard to maintain.
-



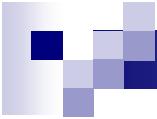
Consequences – drawbacks

- Mediator pattern decrease the complexity between colleagues but increase the complexity of mediator.
 - Sometime, “with a mediator” may worse than “without a mediator”
- Reusing colleagues is possible but reusing the code in Mediator is impractical.
- Mediator providers the extensibility to the colleagues but not itself.
 - Extensibility of Mediator pattern is lean to the colleagues



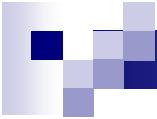
Applicability

- A set of objects communicate in **well-defined but complex ways**. The resulting interdependencies are unstructured and difficult to understand.
 - Reusing an object is difficult because it refers to and communicates with many other objects.
 - A behavior that's distributed between several classes should be customizable without a lot of subclassing.
-



Implementation 1: Omitting the abstract Mediator class.

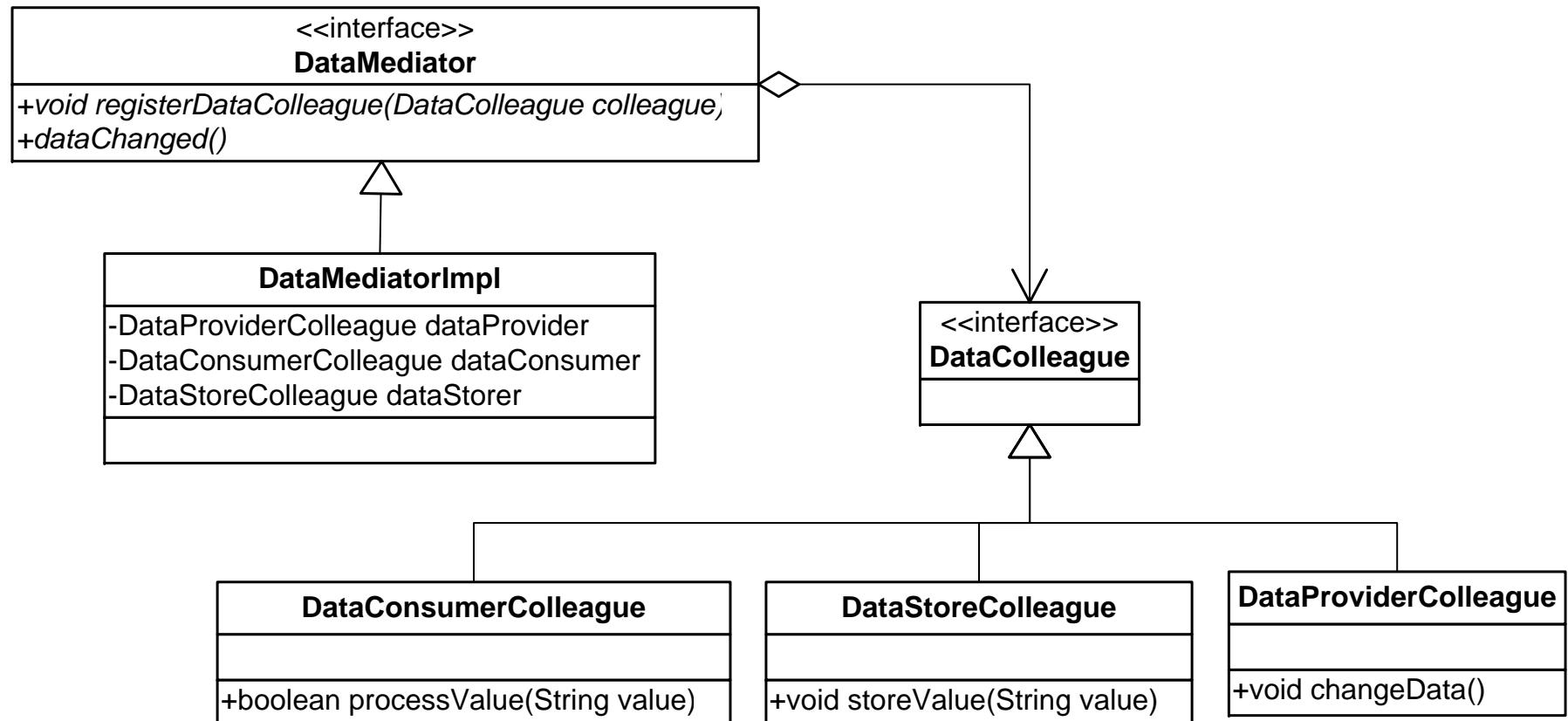
- The abstract coupling that the **Mediator** class provides lets **colleagues** work with different **Mediator** subclasses, and vice versa.
 - There's no need to define an abstract **Mediator** class when **colleagues** work with only one mediator.
-



Implementation 2: Colleague-Mediator communication.

- Colleagues have to communicate with their mediator when an event of interest occurs, using the
 - **Observer pattern:** Colleague classes act as **Subjects**, sending notifications to the mediator whenever they change state. mediator will notify all colleague including the sender.
 - **Notification interface:** Defines a specialized notification interface in **Mediator**, colleagues communicates each other by this interface. a colleague passes itself as an argument, allowing the **mediator** to identify the sender.
-

Example



```
abstract class DataColleague{
    protected DataMediator mediator;
    public DataColleague(DataMediator mediator) {
        this.mediator = mediator;
        mediator.registerDataColleague(this);
    }
}
class DataConsumerColleague extends DataColleague{
    public DataConsumerColleague(DataMediator mediator) {
        super(mediator);
    }
    public boolean processValue(String value) {
        // TODO process the target value
        // if (condition){ return false; }
        return true;
    }
}
class DataStoreColleague extends DataColleague{
    public DataStoreColleague(DataMediator mediator) {
        super(mediator);
    }
    public void storeValue(String value) {
        // TODO store the target value
    }
}
```

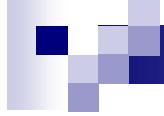
```
class DataProviderColleague extends DataColleague{
    private String target;

    public DataProviderColleague(DataMediator mediator) {
        super(mediator);
    }
    public void changeData() {
        target = "Somthing";
        mediator.dataChanged();
    }
    public String getTarget() {
        return target;
    }
    public void setTarget(String target) {
        this.target = target;
    }
}
interface DataMediator{
    public void registerDataColleague(DataColleague colleague);
    public void dataChanged();
}
```

```
class DataMediatorImpl implements DataMediator{
    private DataProviderColleague dataProvider;
    private DataConsumerColleague dataConsumer;
    private DataStoreColleague dataStorer;

    public void registerDataColleague(DataColleague colleague) {
        Class<?> clazz = colleague.getClass();
        if (clazz.equals(DataProviderColleague.class)) {
            dataProvider = (DataProviderColleague) colleague;
        } else if (clazz.equals(DataConsumerColleague.class)) {
            dataConsumer = (DataConsumerColleague) colleague;
        }else if (clazz.equals(DataStoreColleague.class)) {
            dataStorer = (DataStoreColleague) colleague;
        }else {
            throw new RuntimeException("Unknown DataColleague " + colleague);
        }
    }
    public void dataChanged() {
        String value = dataProvider.getTarget();
        if (dataConsumer != null) {
            if (dataConsumer.processValue(value)) {
                if (dataStorer != null) {
                    dataStorer.storeValue(value);
                }
            }
        }
    }
}
```

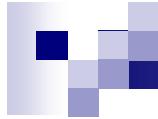
```
public class Client {
    public void test() {
        DataMediator mediator = new DataMediatorImpl();
        DataProviderColleague dataProvider= new DataProviderColleague(mediator);
        //DataConsumerColleague dataConsumer= new DataConsumerColleague(mediator);
        //DataStoreColleague dataStorer= new DataStoreColleague(mediator);
        new DataConsumerColleague(mediator);
        new DataStoreColleague(mediator);
        dataProvider.changeData();
    }
}
```



Extension 1: Law of Demeter (LoD)

A Principle of Object Oriented Design

- Only talk to your immediate friends.
 - One never calls a method on an object you got from another call nor on a global object.
 - You can play with yourself.
 - You can play with your own toys (but you can't take them apart),
 - You can play with toys that were given to you.
 - And you can play with toys you've made yourself.
-



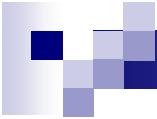
Extension 1: Law of Demeter (LoD)

■ Explanation in plain English:

- Your method can call other methods in its class directly;
- Your method can call methods on its own fields directly (but not on the fields' fields);
- When your method takes parameters, your method can call methods on those parameters directly.
- When your method creates local objects, that method can call methods on the local objects.

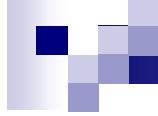
But

- One should not call methods on a global object
 - One should not have a chain of messages
`a.getB().getC().doSomething()` in some class other than a's class.
-



Extension 2: Misusing Mediator

- Mediator pattern is applied to a system for avoiding mess and ugly.
 - Mediator pattern should not be applied to a system which has been mess and ugly.
 - Such system should be re-designed;
 - Responsibilites of classes should be repartitioned;
 - When the system is going to be mess, firstly, try to clarify the functional dependency;
 - **Mediator pattern should be used to avoid the mess system, but not fix it.**
 - Mediator pattern is a pattern but not a sliver bullet.
-



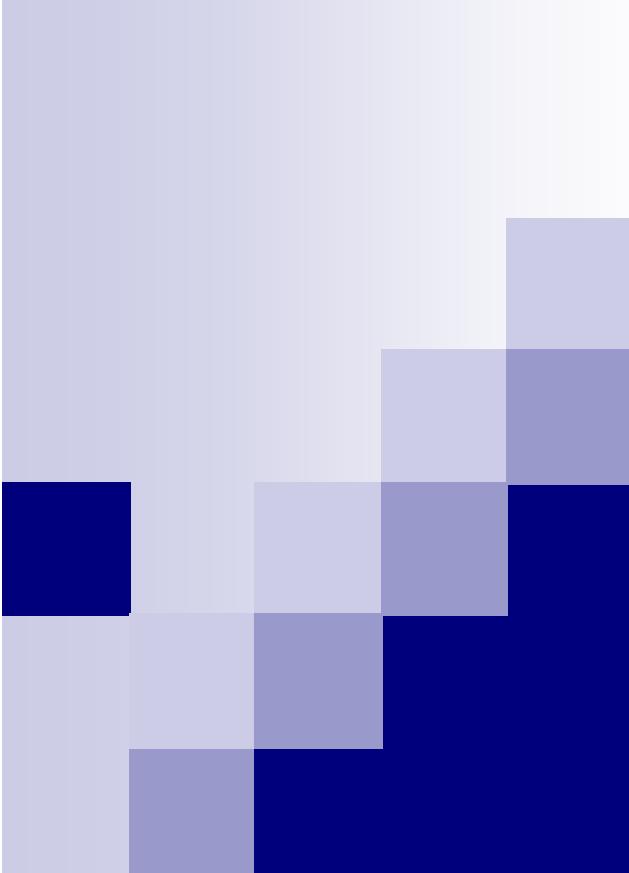
Extension 2: Mediator is not for fixing mess

- 一个初级设计师在对面向对象的技术不熟悉时，会使一个系统在责任的分割上发生混乱。
 - 责任分割的混乱会使得系统中的对象与对象之间产生不适当的复杂关系。
 - 这时候，一个很糟的想法就是继续这个错误，并使用调停者模式“化解”这一团乱麻。实际上，这样一来，责任错误划分的混乱不但不会得到改正，而且还会制造出一个莫名其妙的怪物：一个处于一团乱麻之中的混乱之首。
-



Let's go to next...





Design Patterns

宋 杰

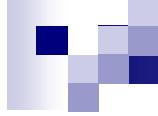
Song Jie

东北大学 软件学院

Software College, Northeastern
University



21. Memento Pattern



Intent

- Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
 - Snapshot, Check Point
 - 在不破坏封装的条件下，将一个对象的状态捕捉住，并外部化，存储起来，从而可以在将来合适的时候把这个对象还原到存储起来的状态。
 - 备忘录对象是一个用来存储另外一个对象内部状态的快照的对象。
-

Example



Step 1: Single Interface

```
class Originator {
    private String state;
    public void setMemento(Memento memento) {
        this.state = memento.getState();
    }
    public Memento createMemento() {
        return new Memento(state);
    }
}

class Memento {
    private String state;
    public Memento(String state) {
        this.state = state;
    }
    public String getState() {
        return state;
    }
}
```

Step 1: Single Interface

```
class Caretaker {  
    private Memento memento;  
    public Memento retrieveMemento() {  
        return memento;  
    }  
    public void saveMemento(Memento memento) {  
        this.memento = memento;  
    }  
}
```

Step 2: Multiple Checkpoints

```
class Originator {
    private String state;
    public void setMemento(Memento memento) {
        this.state = memento.getState();
    }
    public Memento createMemento() {
        return new Memento(new Date().toString(), state);
    }
}

class Memento {
    private String state;
    private String checkpoint;
    public Memento(String checkpoint, string state) {
        this.checkpoint = checkpoint;
        this.state = state;
    }
    public String getCheckpoint() {
        return checkpoint;
    }
    public String getState() {
        return state;
    }
}
```

Step 2: Multiple Checkpoints

```
class Caretaker {
    private Map<String, Memento> mementoPool;
    public Caretaker() {
        mementoPool = new HashMap<String, Memento>();
    }
    public Memento retrieveMemento(String checkpoint) {
        return mementoPool.remove(checkpoint);
    }
    public void saveMemento(Memento memento) {
        mementoPool.put(memento.getCheckpoint(), memento);
    }
    public void clear() {
        mementoPool.clear();
    }
    public Iterator<String> checkpoints() {
        return mementoPool.keySet().iterator();
    }
    public Iterator<Memento> mementos() {
        return mementoPool.values().iterator();
    }
}
```

Step 3: Double interface

```
interface WideMemento {
    public String getCheckpoint();
    public String getState();
}

interface NarrowMemento {
    public String getCheckpoint();
}

class Originator {
    private String state;
    public void setMemento(WideMemento memento) {
        this.state = memento.getState();
    }
    public WideMemento createMemento() {
        return new MementoImpl(new Date().toString(), state);
    }
}
```

Step 3: Double interface

```
class MementoImpl implements WideMemento, NarrowMemento {  
    private String state;  
    private String checkpoint;  
    public MementoImpl(String checkpoint, String state) {  
        this.checkpoint = checkpoint;  
        this.state = state;  
    }  
  
    public String getCheckpoint() {  
        return checkpoint;  
    }  
  
    public String getState() {  
        return state;  
    }  
}
```

Step 3: Double interface

```
class Caretaker {
    private Map<String, NarrowMemento> mementoPool;
    public Caretaker() {
        mementoPool = new HashMap<String, NarrowMemento>();
    }
    public NarrowMemento retrieveMemento(String checkpoint) {
        return mementoPool.remove(checkpoint);
    }
    public void saveMemento(NarrowMemento memento) {
        mementoPool.put(memento.getCheckpoint(), memento);
    }
    public void clear() {
        mementoPool.clear();
    }
    public Iterator<String> checkpoints() {
        return mementoPool.keySet().iterator();
    }
    public Iterator<NarrowMemento> mementos() {
        return mementoPool.values().iterator();
    }
}
```

Step 4: Inner Class

```
interface Memento {
    public String getCheckpoint();
}

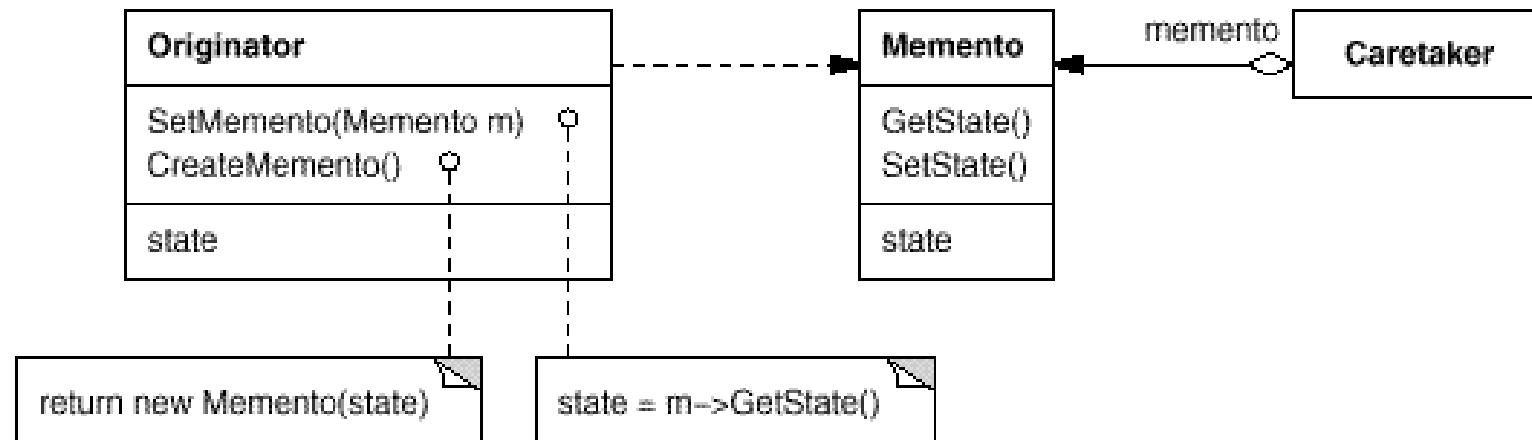
class Originator {
    private String state;
    public void setMemento(Memento memento) {
        this.state = ((InnerMemento) memento).getState();
    }
    public Memento createMemento() {
        return new InnerMemento(new Date().toString(), state);
    }
    class InnerMemento implements Memento {
        private String stateCopy;
        private String checkpoint;

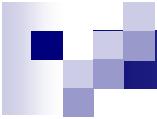
        public InnerMemento(String checkpoint, String state) {
            this.checkpoint = checkpoint;
            this.stateCopy = state;
        }
        public String getCheckpoint() {
            return checkpoint;
        }
        public String getState() {
            return stateCopy;
        }
    }
}
```

Step 4: Inner Class

```
class Caretaker {  
    private Map<String, Memento> mementoPool;  
    public Caretaker() {  
        mementoPool = new HashMap<String, Memento>();  
    }  
    public Memento retrieveMemento(String checkpoint) {  
        return mementoPool.remove(checkpoint);  
    }  
    public void saveMemento(Memento memento) {  
        mementoPool.put(memento.getCheckpoint(), memento);  
    }  
    public void clear() {  
        mementoPool.clear();  
    }  
    public Iterator<String> checkpoints() {  
        return mementoPool.keySet().iterator();  
    }  
    public Iterator<Memento> mementos() {  
        return mementoPool.values().iterator();  
    }  
}
```

Structure





Participants

■ Memento

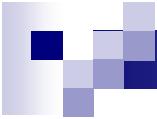
- Stores internal state of the **Originator** object.
- Protects against access by objects other than the **originator**.

■ Originator

- Creates a **memento** containing a snapshot of its current internal state.
- Uses the **memento** to restore its internal state.

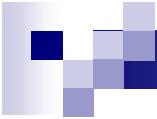
■ Caretaker

- Be responsible for the **memento**'s safekeeping.
 - Never operates on or examines the contents of a **memento**.
-



Two interfaces of Memento

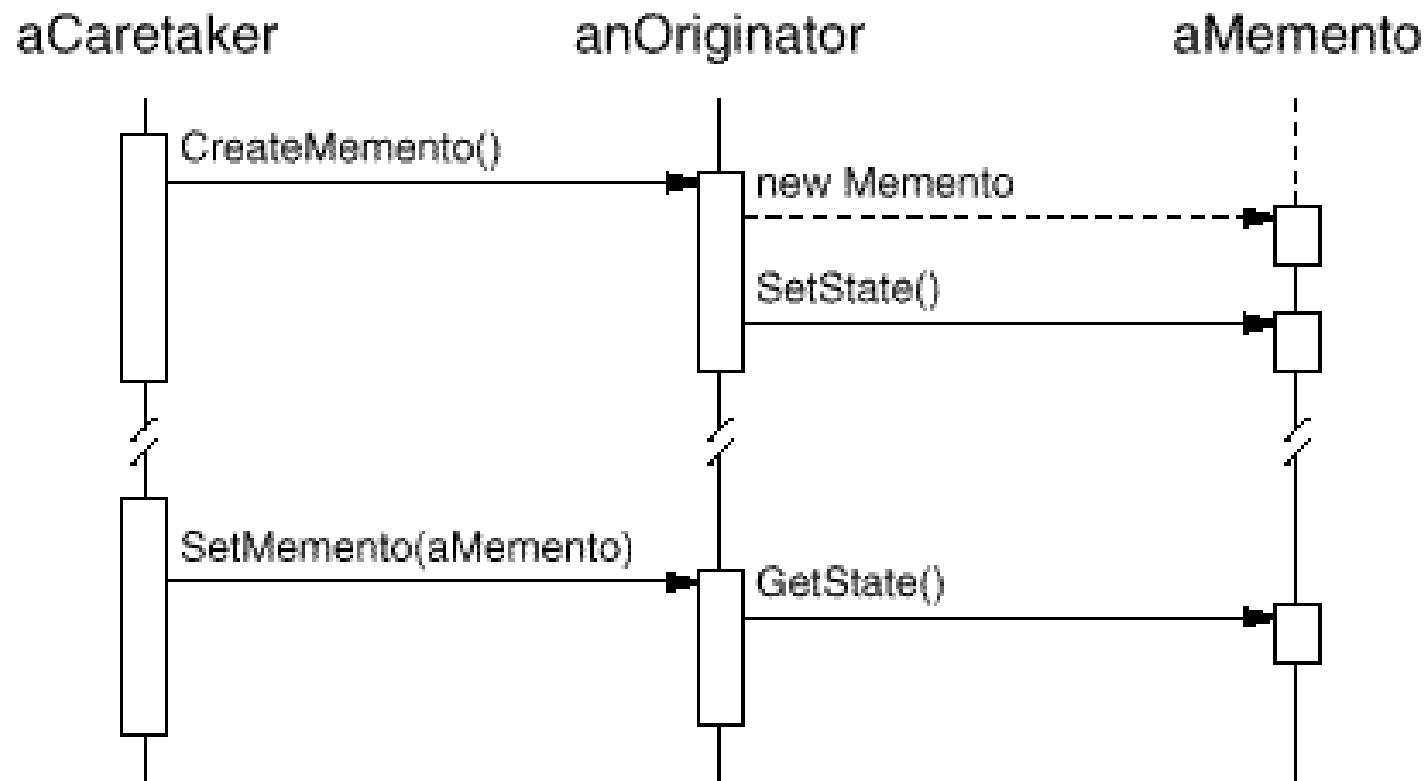
- **Narrow Interface:** **Caretaker** sees a narrow interface to the **Memento**—it can only pass the **memento** to other objects.
 - **Wide Interface:** **Originator**, in contrast, sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state.
 - Ideally, only the originator that produced the memento would be permitted to access the memento's internal state.
-

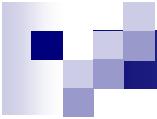


Collaborations

- A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator,
 - Sometimes the caretaker won't pass the memento back to the originator, because the originator might never need to revert to an earlier state.
 - Mementos are passive. Only the originator that created a memento will assign or retrieve its state.
-

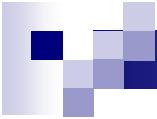
Collaborations





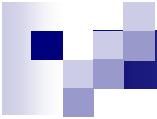
Consequences – advantages

- Preserving encapsulation boundaries.
 - Memento avoids exposing information that only an originator should manage but that must be stored nevertheless outside the originator.
 - It simplifies Originator.
 - Originator need not maintain and management the versions of internal state.
-



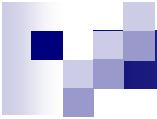
Consequences – drawbacks

- Using mementos might be expensive.
 - Mementos might incur considerable overhead if Originator must copy large amounts of information to store in the memento or if clients create and return mementos to the originator often enough.
 - The caretaker has no idea how much state is in the memento. Hence an otherwise lightweight caretaker might incur large storage costs.
-



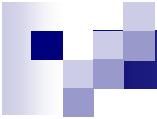
Applicability

- A snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, *and*
 - A direct interface to obtaining the state would expose implementation details and break the object's encapsulation.
-

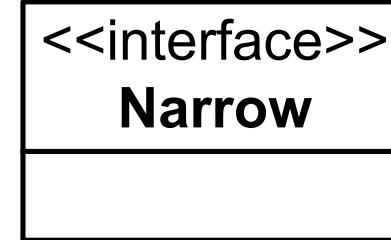
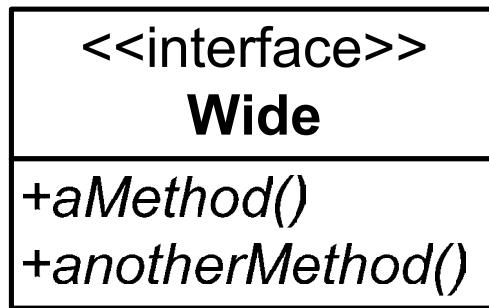


Implementation 1: Storing incremental changes.

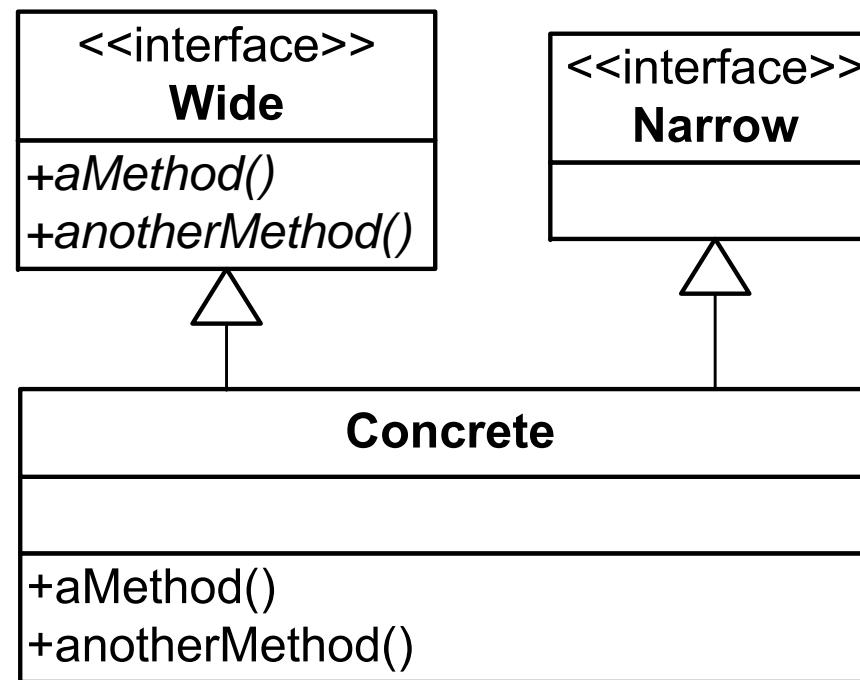
- When mementos get created and passed back to their originator in a predictable sequence, then Memento can save just the incremental change to the originator's internal state.
-



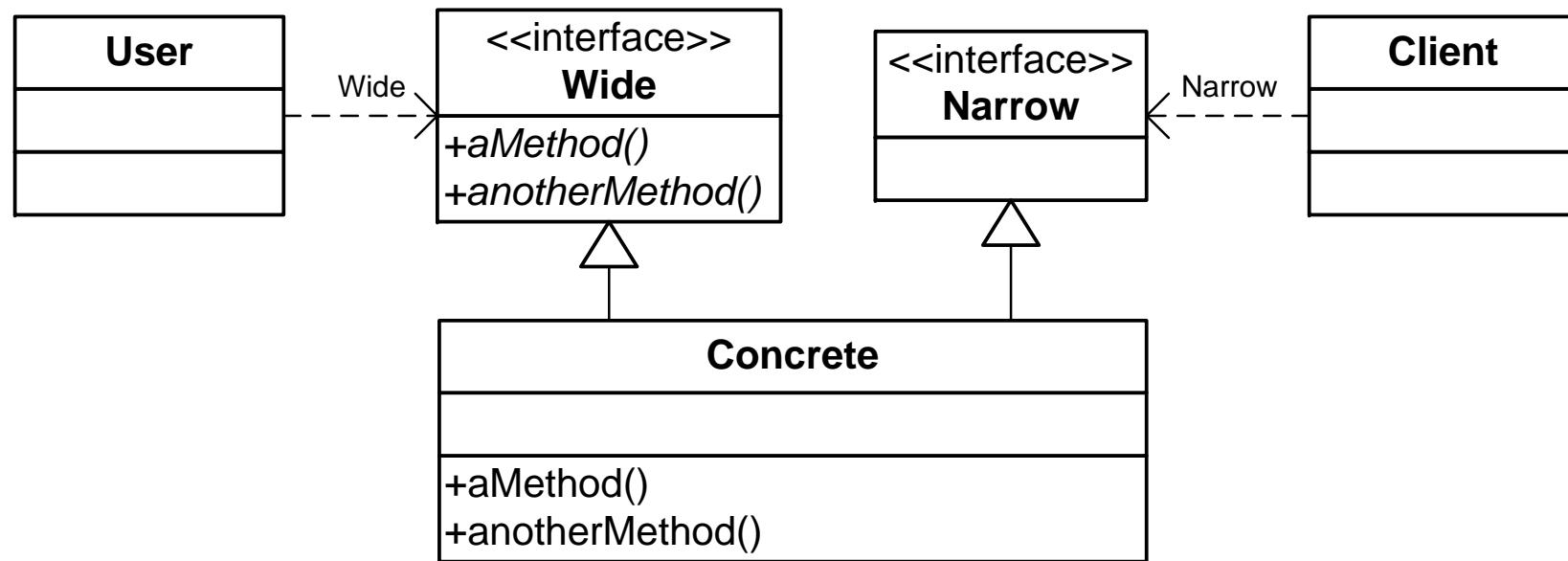
Implementation 2: Wide and narrow interfaces



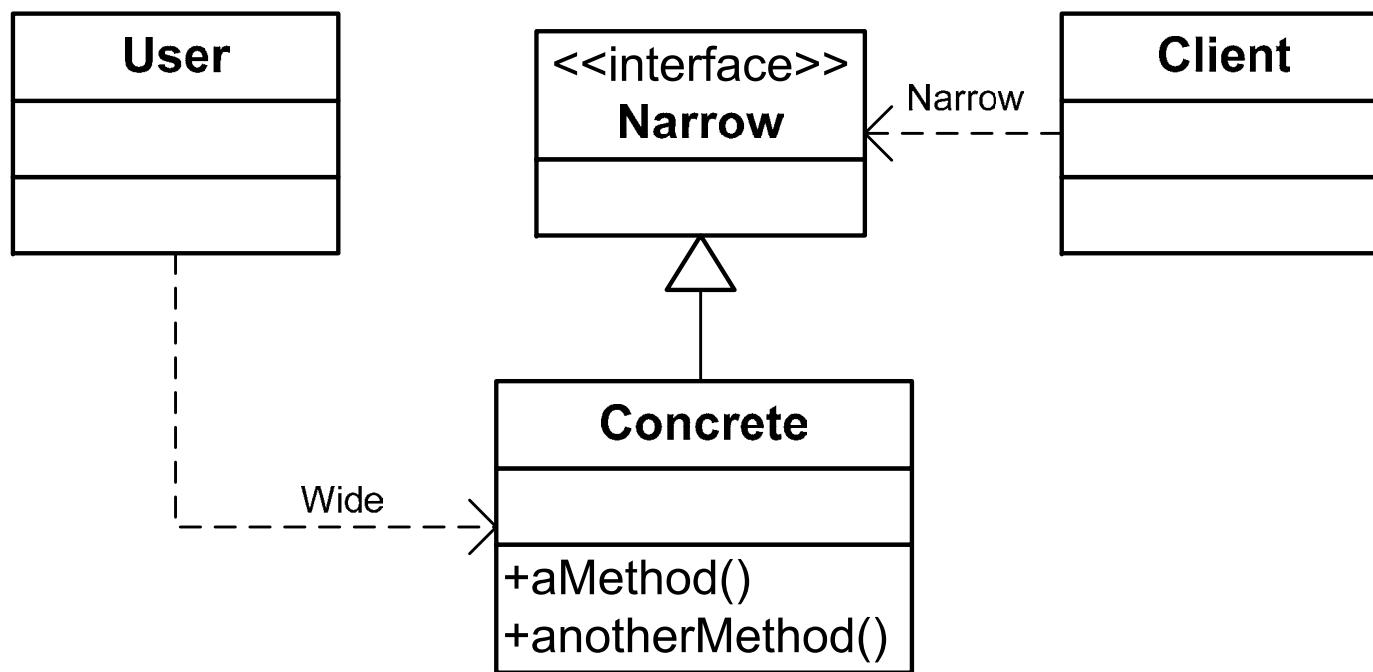
Implementation 2: Wide and narrow interfaces



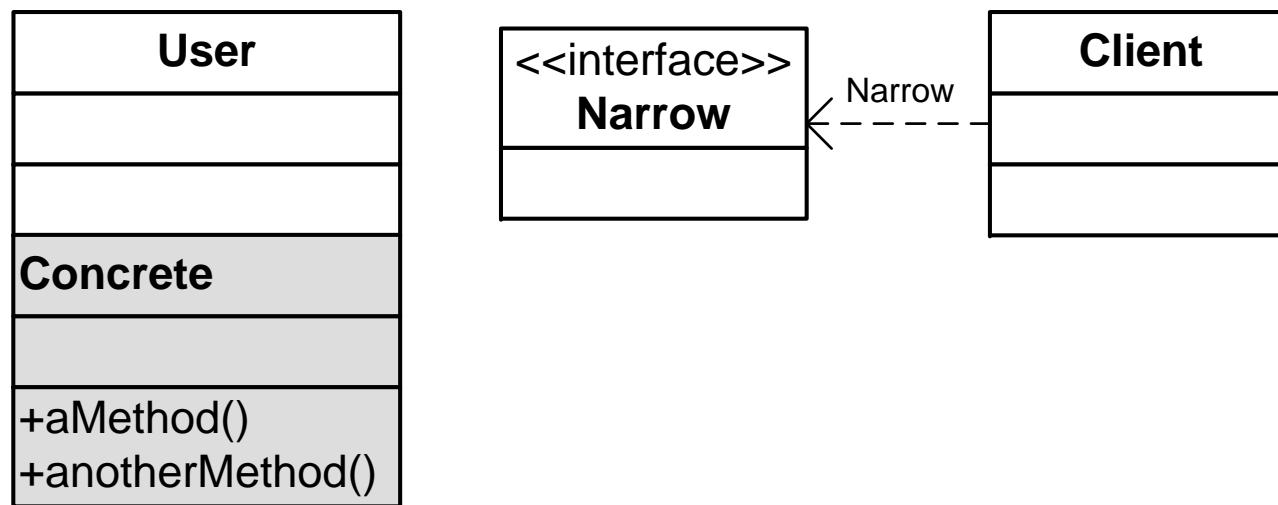
Implementation 2: Wide and narrow interfaces



Implementation 2: Wide and narrow interfaces



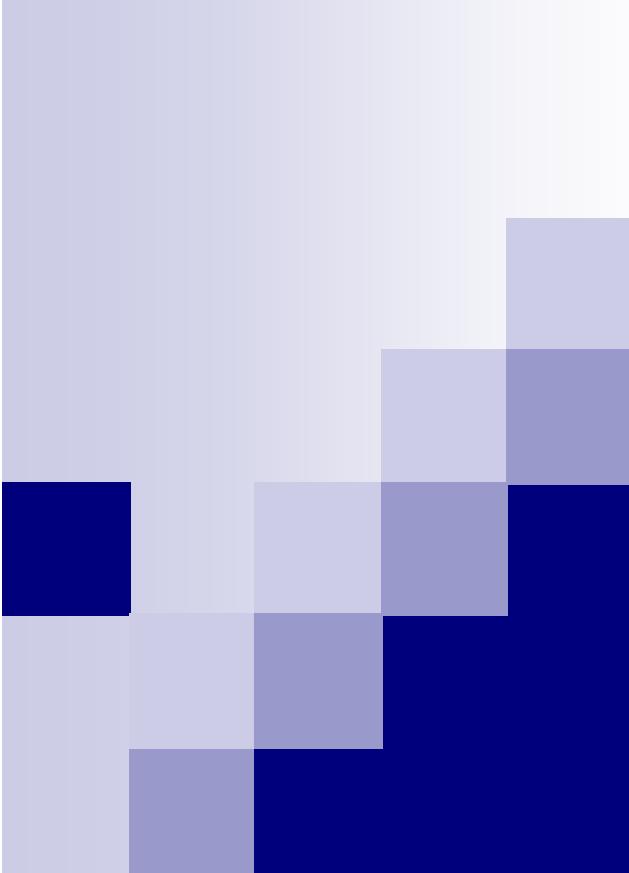
Implementation 2: Wide and narrow interfaces





Let's go to next...





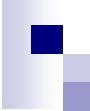
Design Patterns

宋 杰

Song Jie

东北大学 软件学院

Software College, Northeastern
University

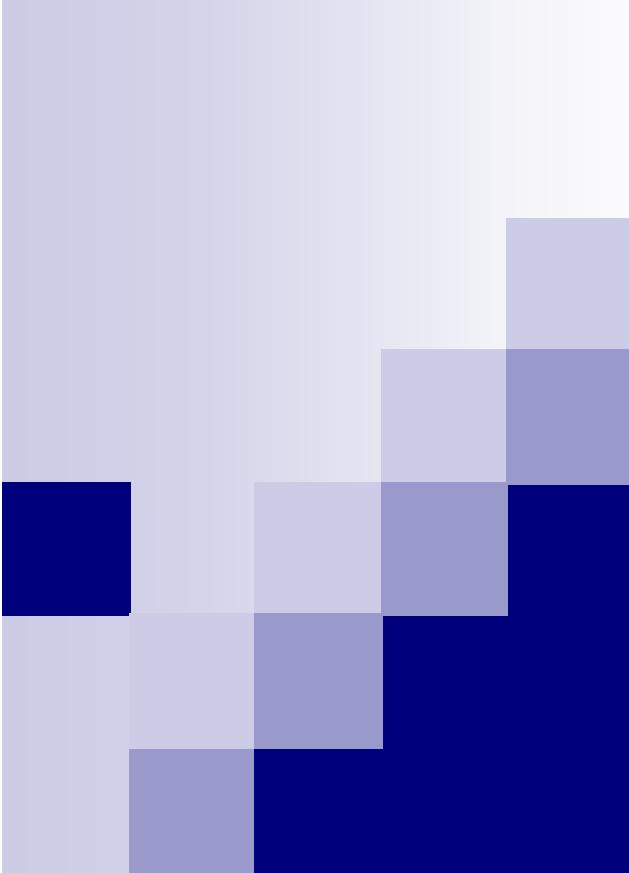


22. Visitor Pattern



To be available soon...





Design Patterns

宋 杰

Song Jie

东北大学 软件学院

Software College, Northeastern
University



23. Interpreter Pattern



To be available soon...

