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 it's function, in it's 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(笨拙的)and cumbersome(笨重的) 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

- Rigidity (僵化) – make it hard to change
- Fragility (脆弱) – make it easy to break
- Immobility (固化) – make it hard to reuse
- Viscosity (黏滞)– make it hard to do the right thing
- Needless Complexity (非必要复杂性) – over design
- Needless Repetition (非必要重复) – error prone
- Not 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

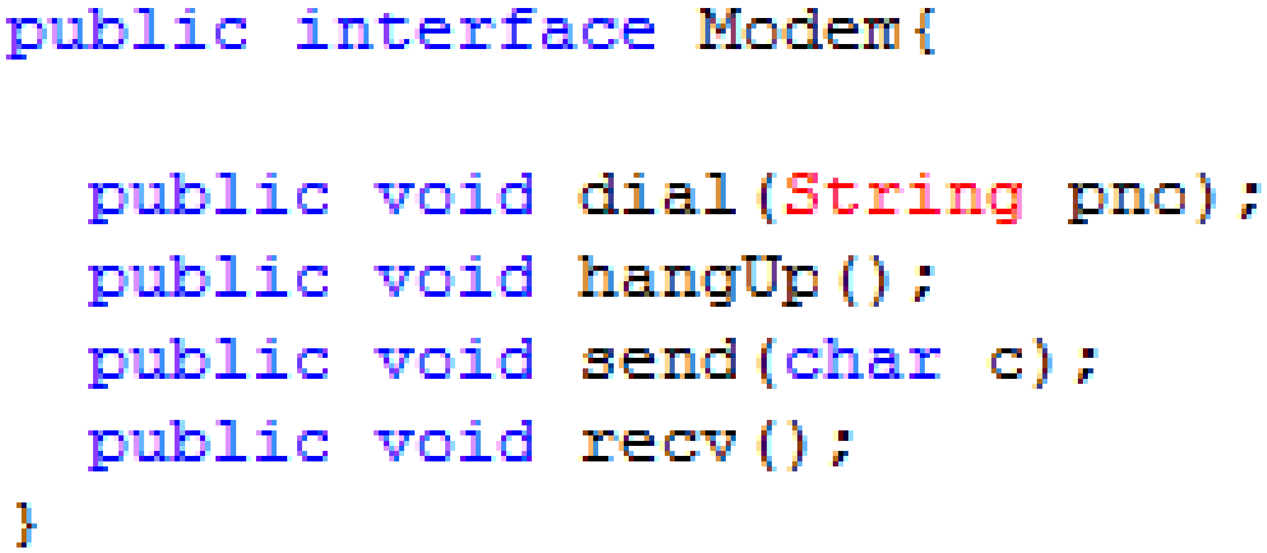
- SRP: Single Responsibility Principle
- A class should have one reason to change
  - A responsibility is a reason to change

- 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。
- 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。

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

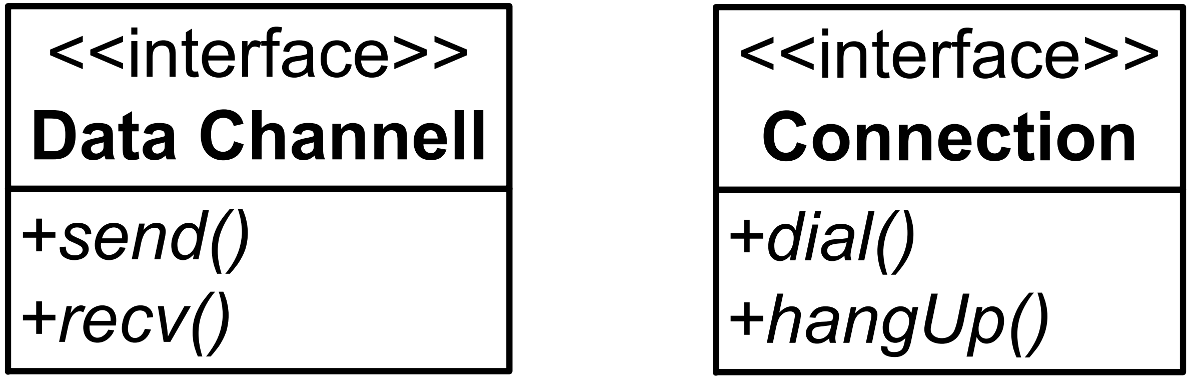


```
public interface Modem{

   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



<<interface>>
Data Channell

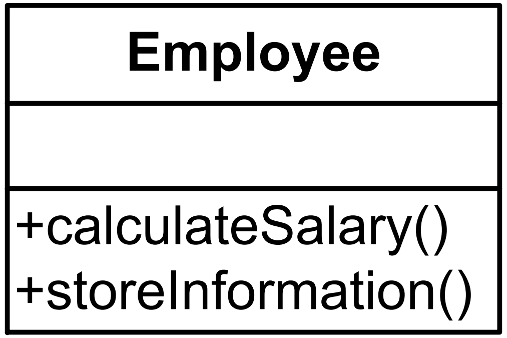
+send()
+recv()

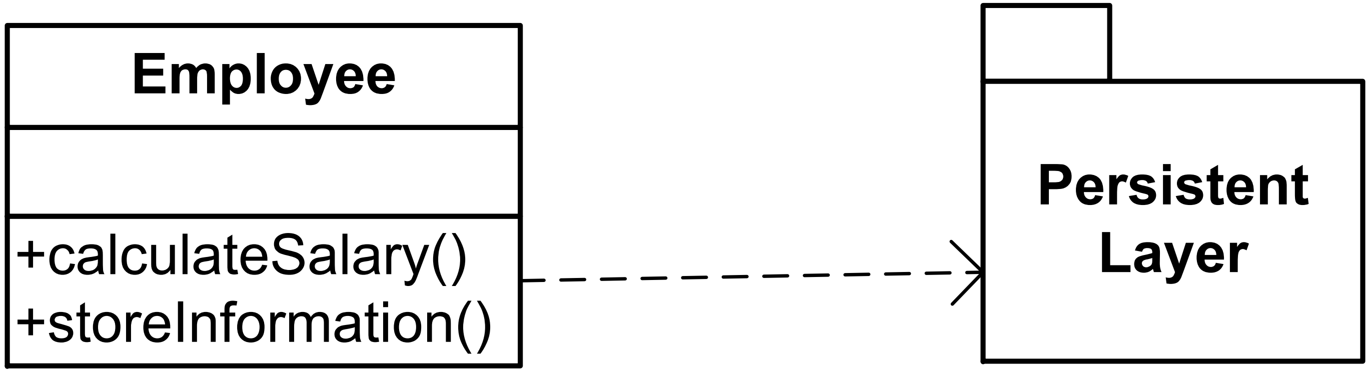
<<interface>>
Connection

+dial()
+hangUp()

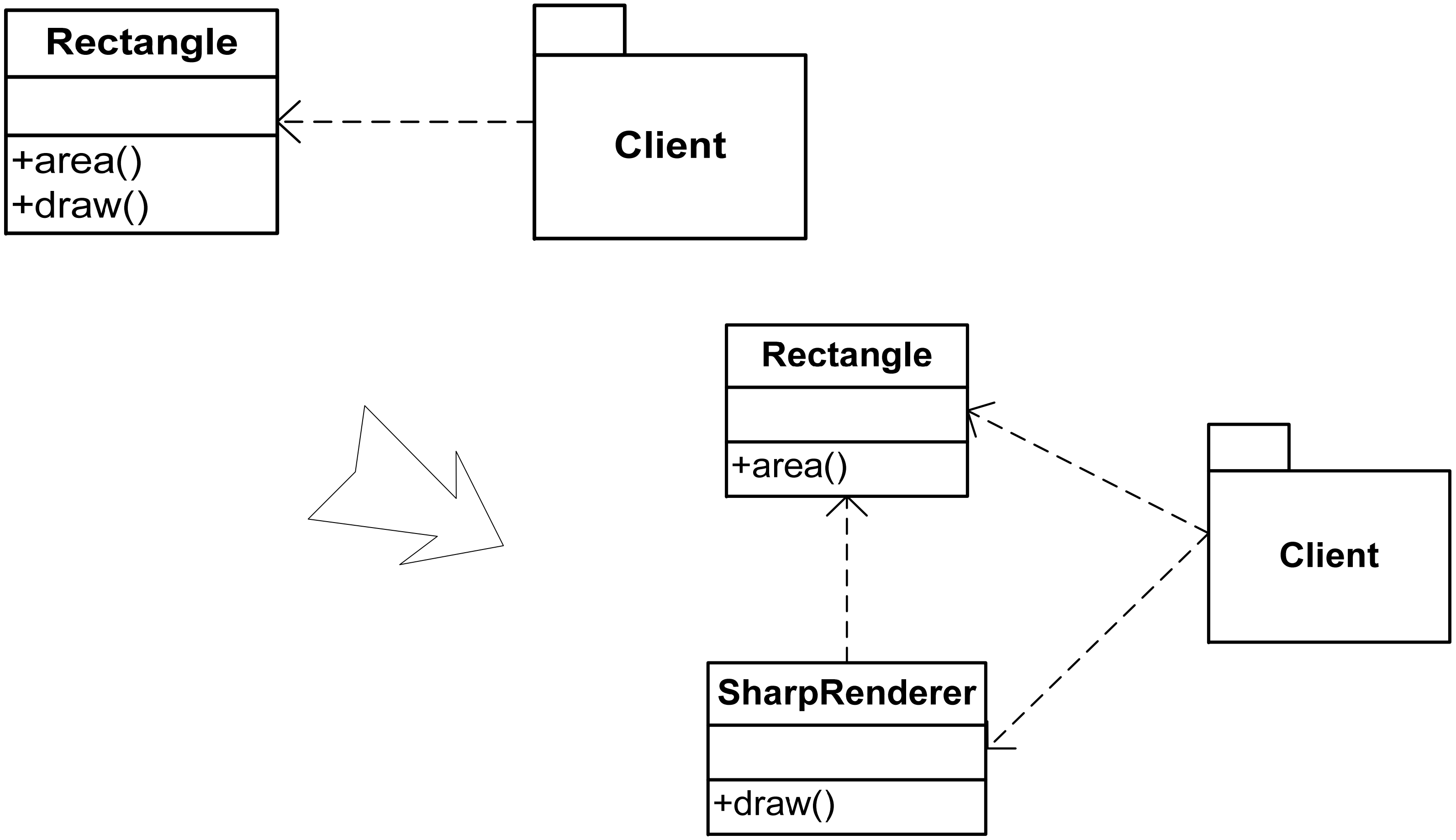
Thinking: Where is the Modem?
How to implement the Modem?

SRP Example (implemented aspect): business and persistent methods





SRP Example (both two aspects): Rectangle



Rectangle

+area()
+draw()

Client

Rectangle

+area()

SharpRenderer

+draw()

Client

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

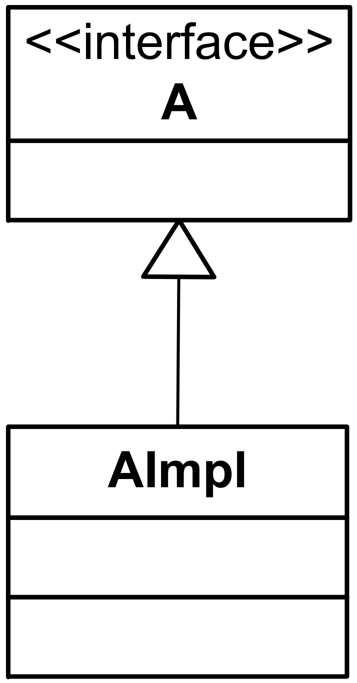
- OCP: Open-Closed Principle
- Software entities (Classes, Modules, Methods, etc.) should be open for extension, but closed for modification.
  - Open For Extension: Satisfying the new requirements by adding new modules.
  - Closed For Modification: No need and can not modify current modules for new 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

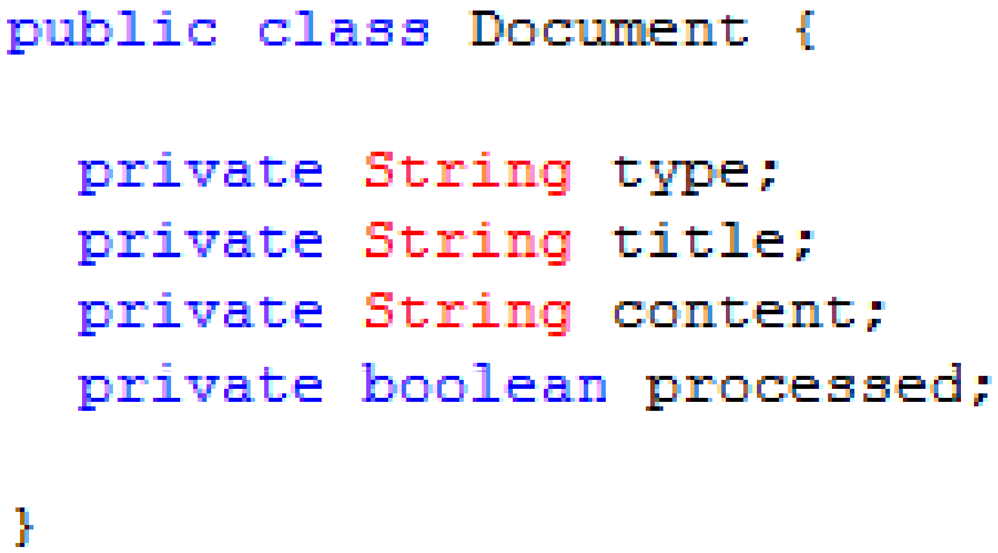


OCP: Implementation

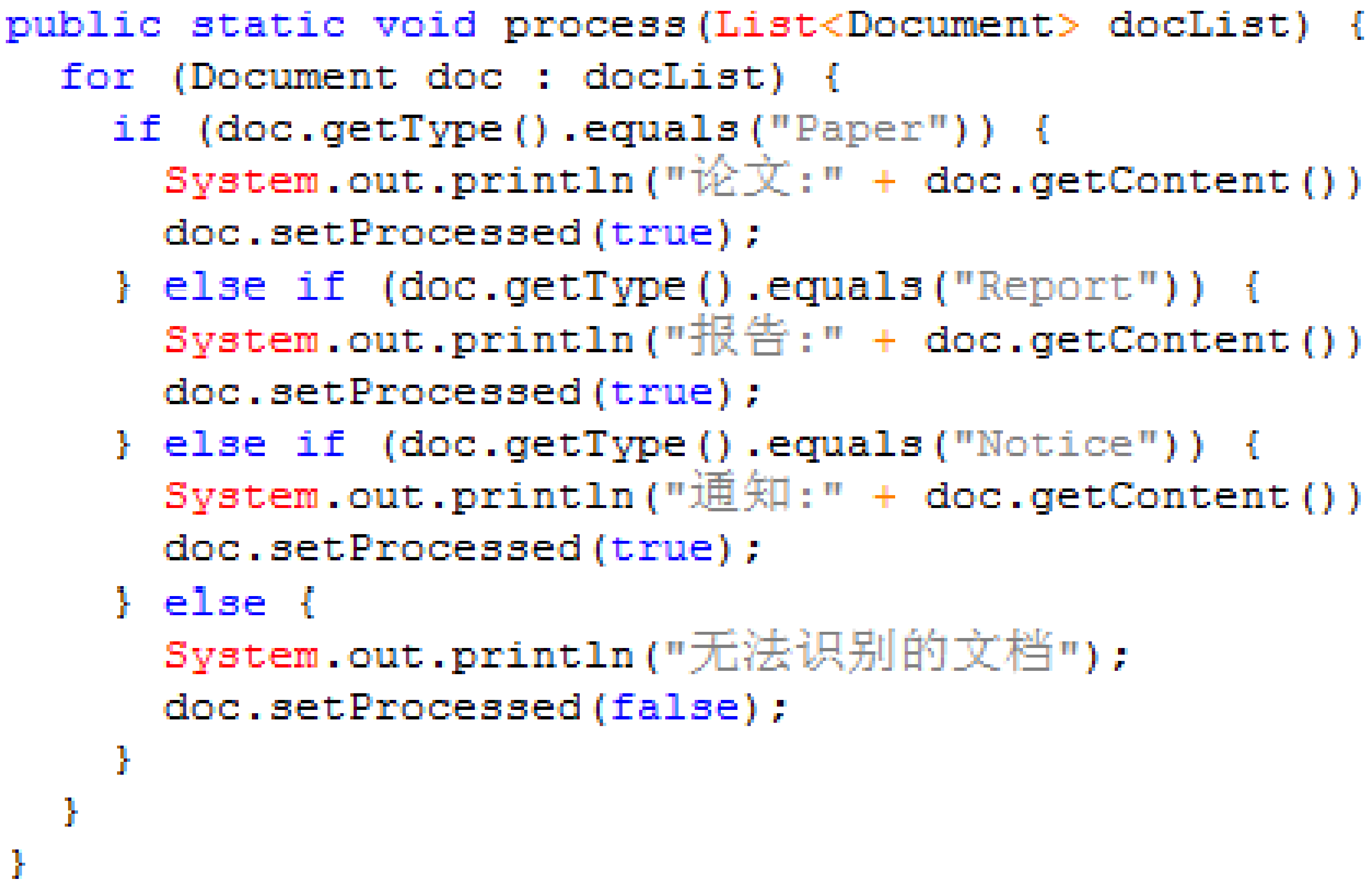
- Interface and abstract class are the abstraction which are fixed but have many possible behaviors
  - These behaviors are presented as the implemented class or inherited class.
- A Interface is open for extension because it have flexible number of implementations;
- A Interface is closed for modification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

OCP Example: Document Processor

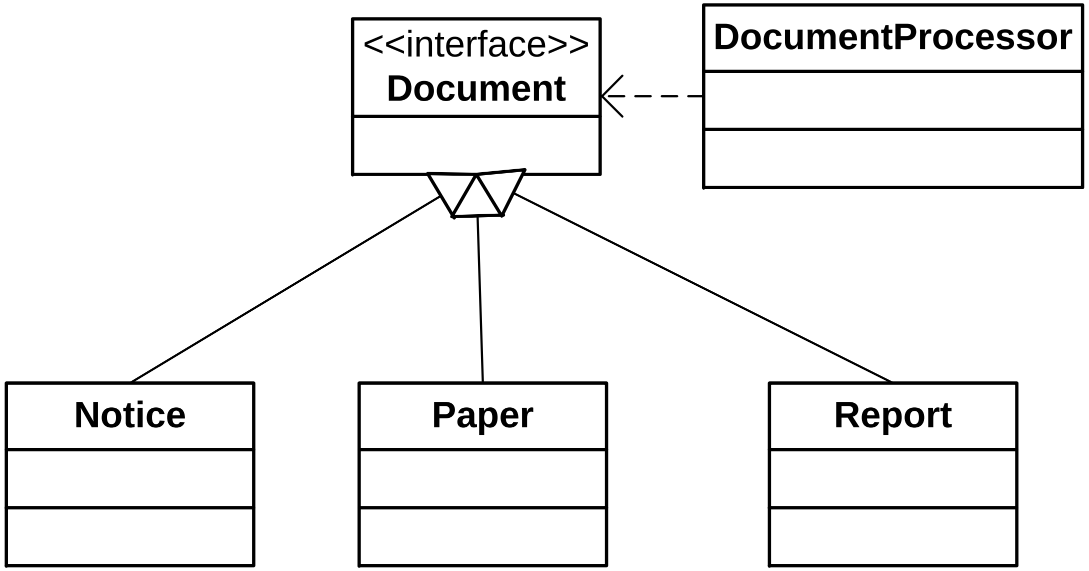
- A system which can process the documents one by one;
- There is three kinds of Documents, including Paper, Report and Notice.

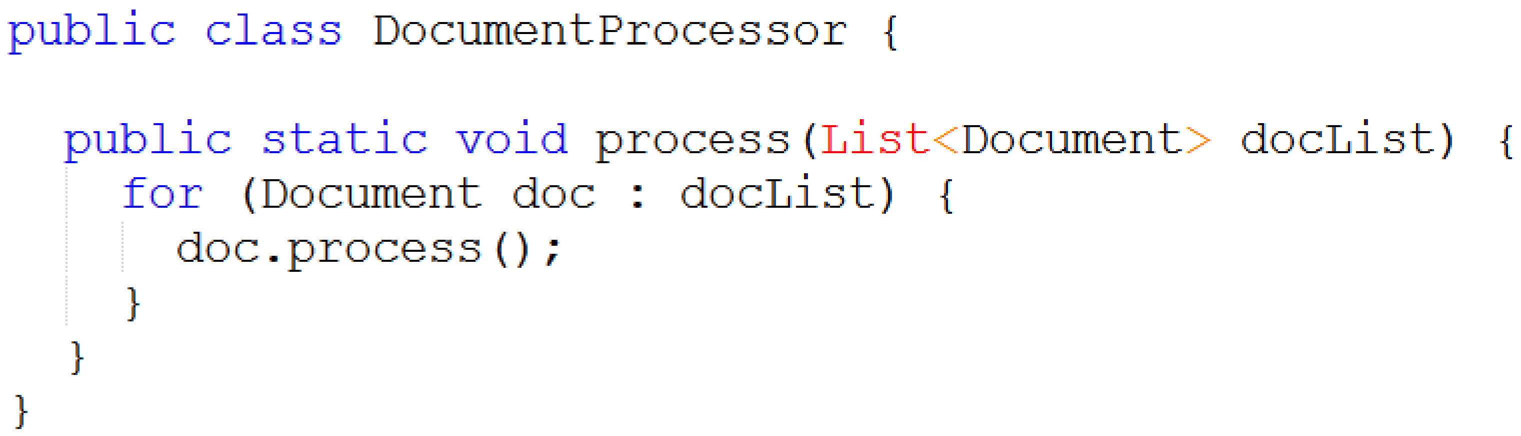


```java
public class Document {

  private String type;
  private String title;
  private String content;
  private boolean processed;

}
```



```java
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);
    }
  }
}
```





```
public class DocumentProcessor {

  public static void process(List<Document> docList) {
    for (Document doc : docList) {
      doc.process();
    }
  }
}
```

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

- OCP is the kernel of OOD (Object Oriented Design), abstraction is the kernel of OCP;
- OCP means the better reusability and maintainability.
- 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”；
- 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;

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

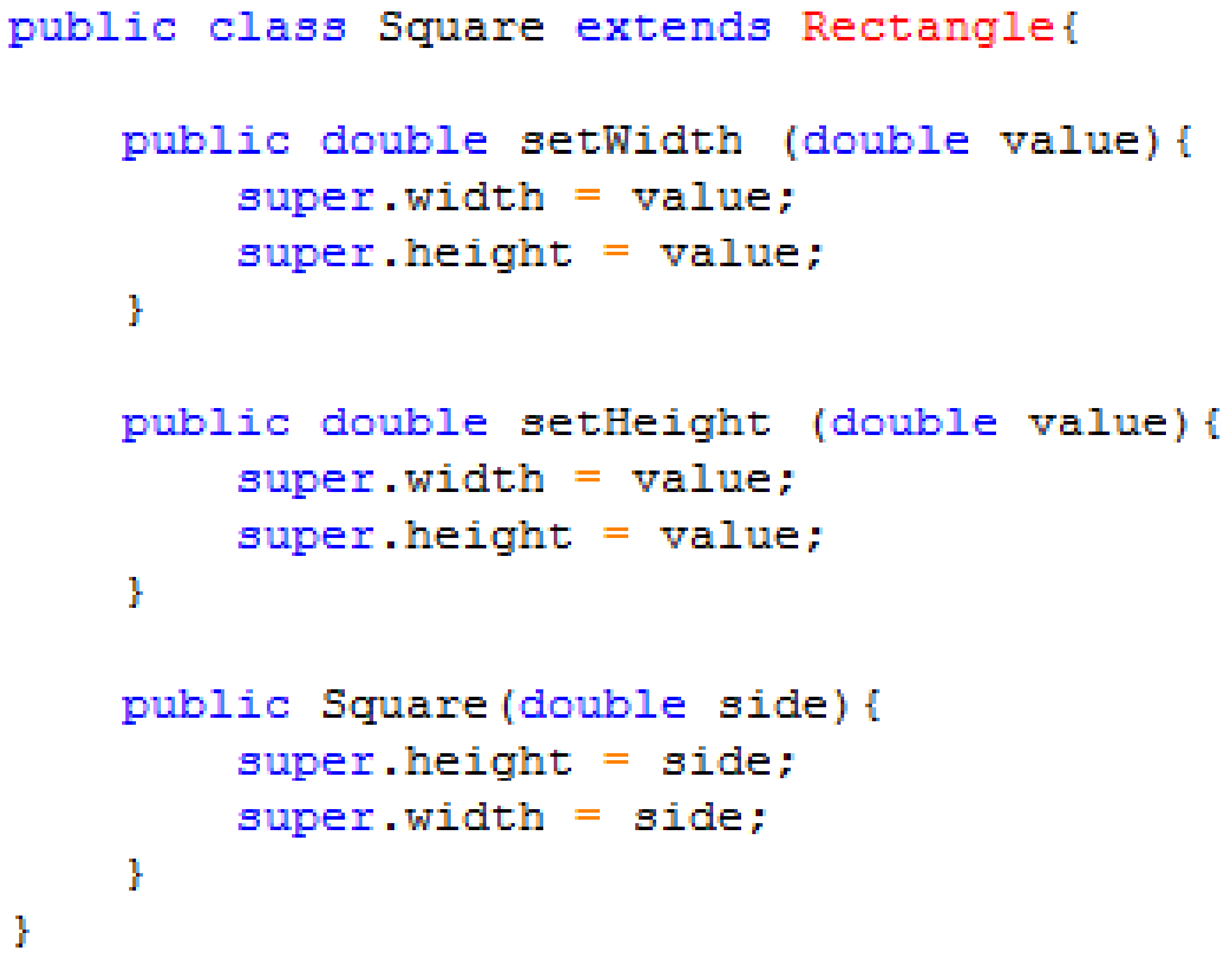
- Any subclass should always be usable instead of its parent class.

- 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

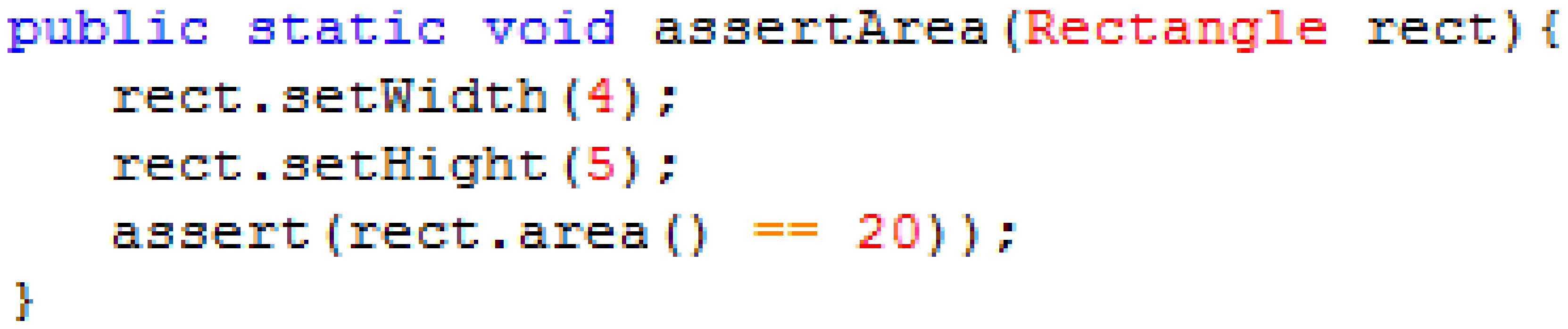
- IS-A relationship is based on the Concepts, not Behavior. LSP is based on Behavior.
- The behavior is relay on the context and applied situation, Some concepts are obviously satisfy IS-A relationship but not inherited relationship because theirs behaviors are inconsistent in some situation.
- When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

LSP Example: Square is a Rectangle

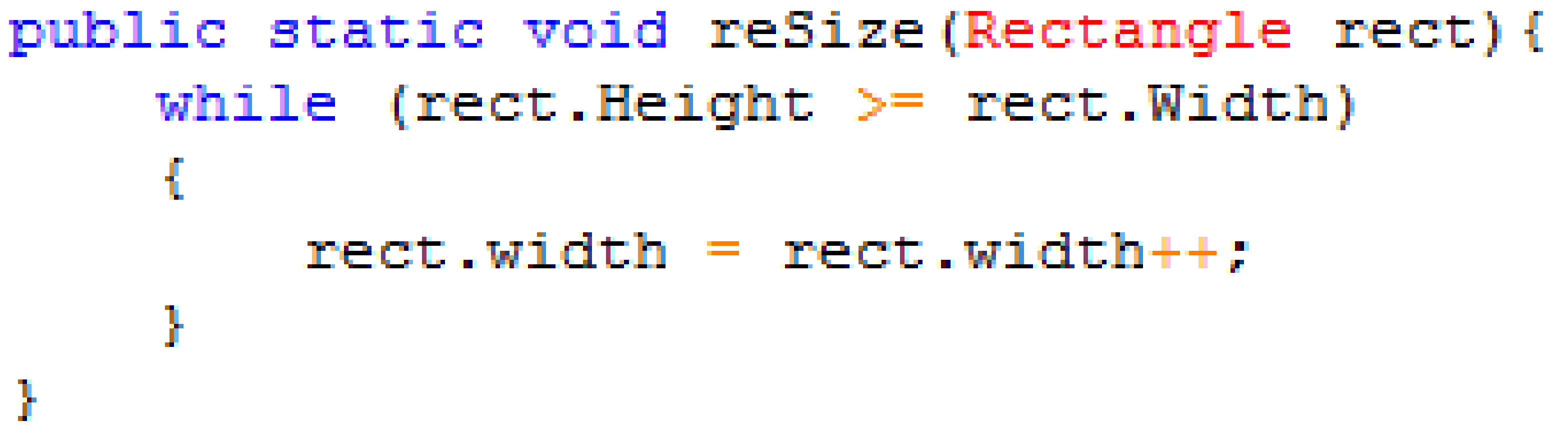


```
public class Square extends Rectangle{

    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;
    }
}
```

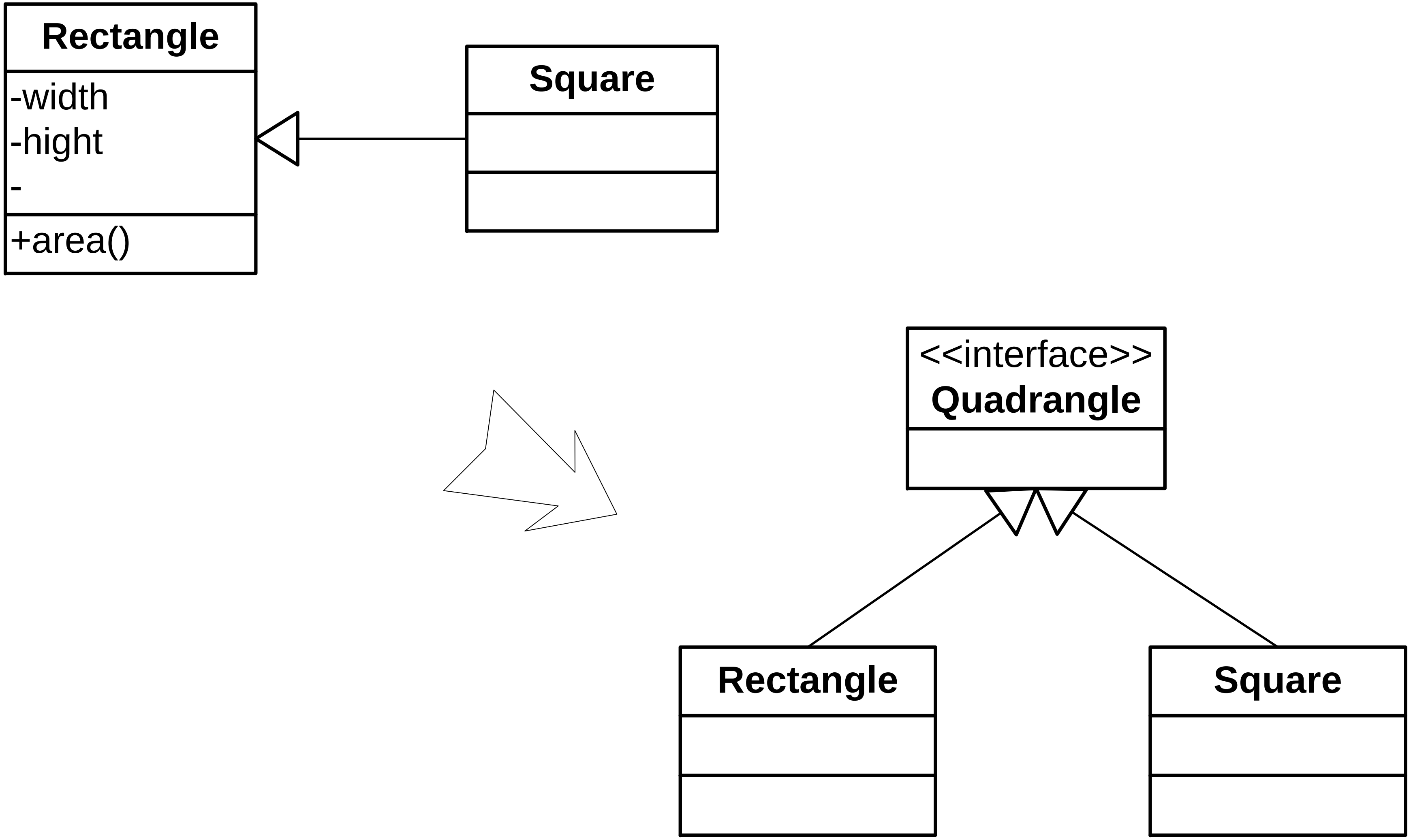
LSP Example: Square is a Rectangle



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



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



Rectangle

-width
-hight
-

+area()

Square

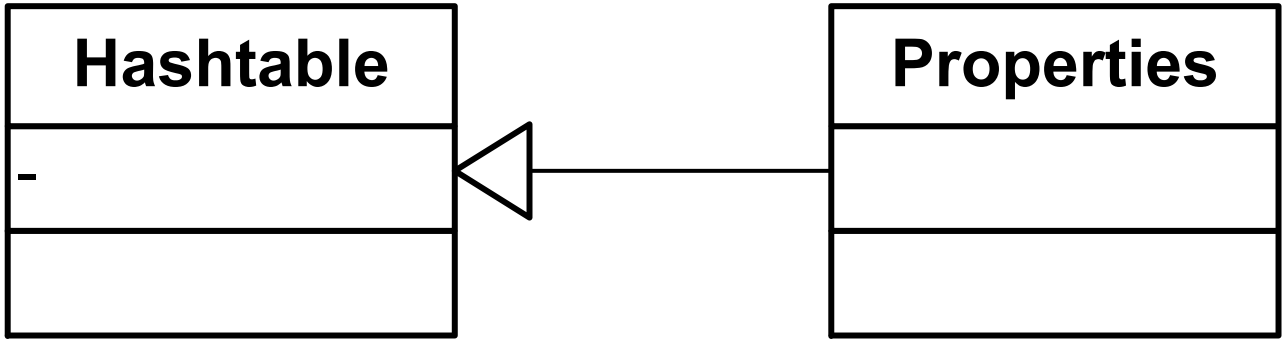
<<interface>>
Quadrangle

Rectangle

Square

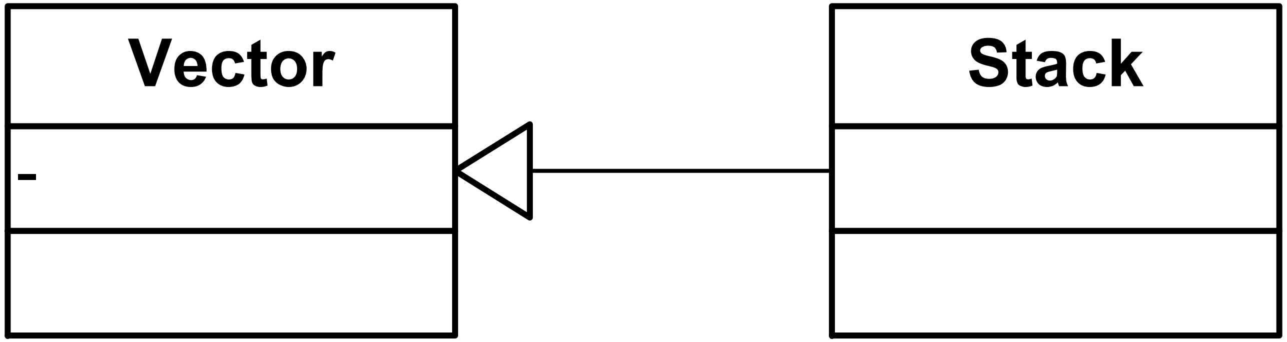
LSP Example :

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



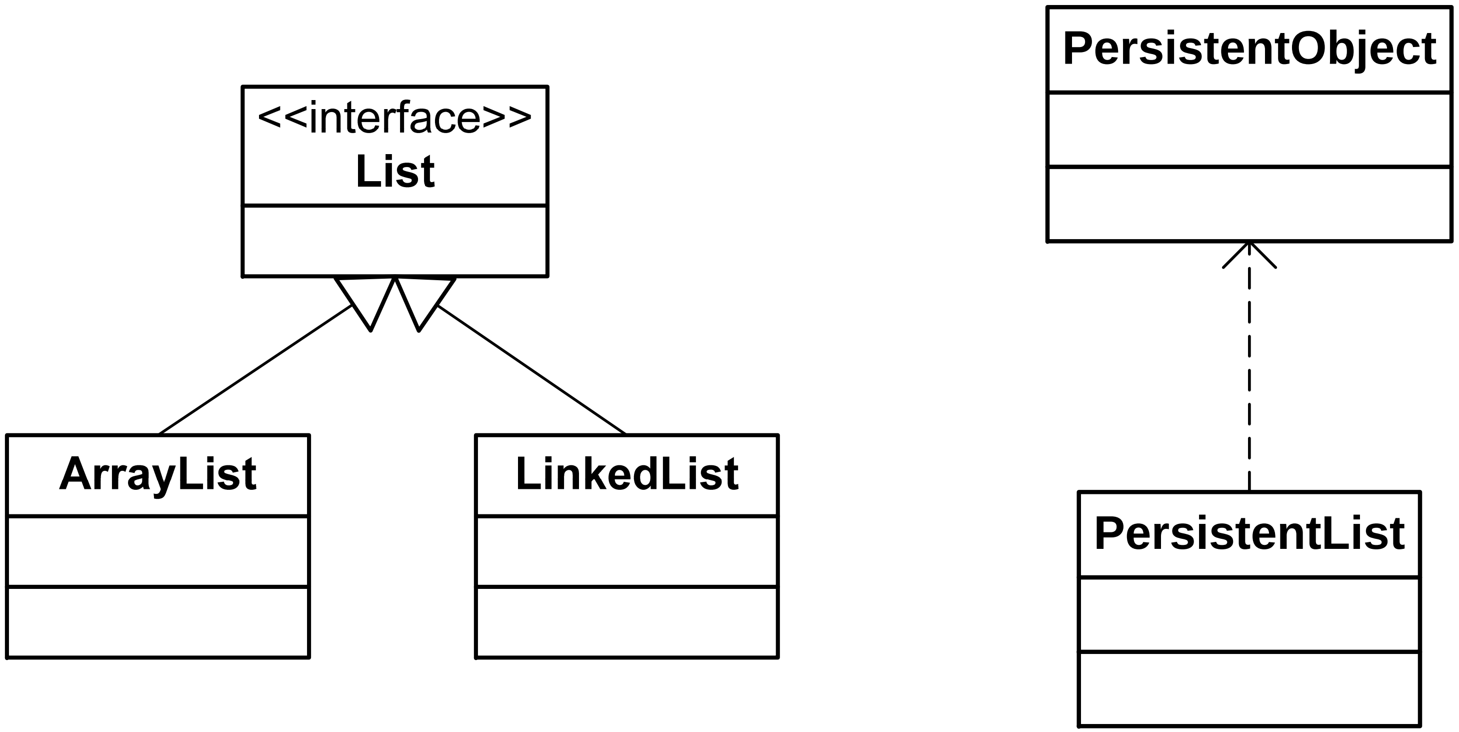
- Hashtable. key:Object,value:Object
- Properties. key:String,value:String

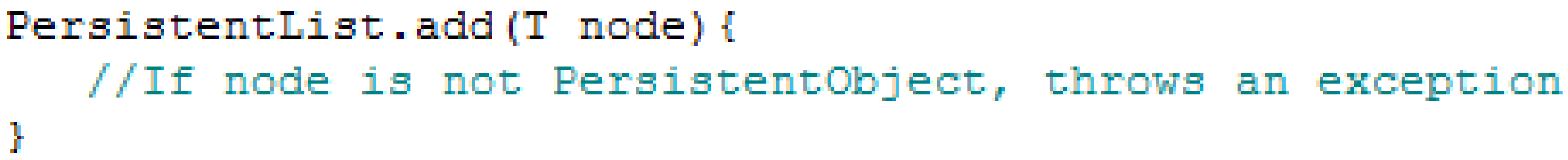
LSP Example :
java.util.Stack is a java.util.Vector



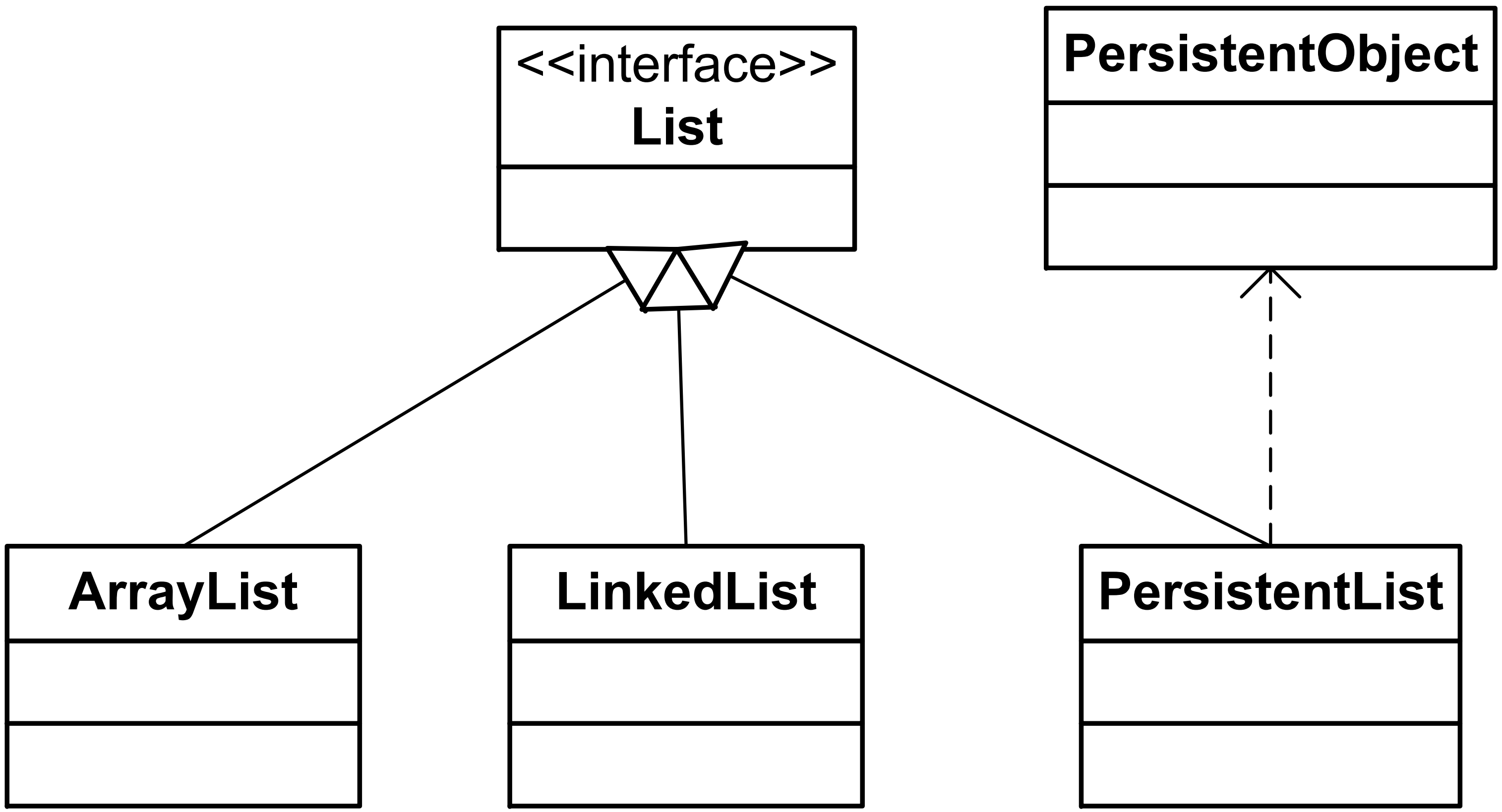
- Vector: FIFO
- Stack: FILO

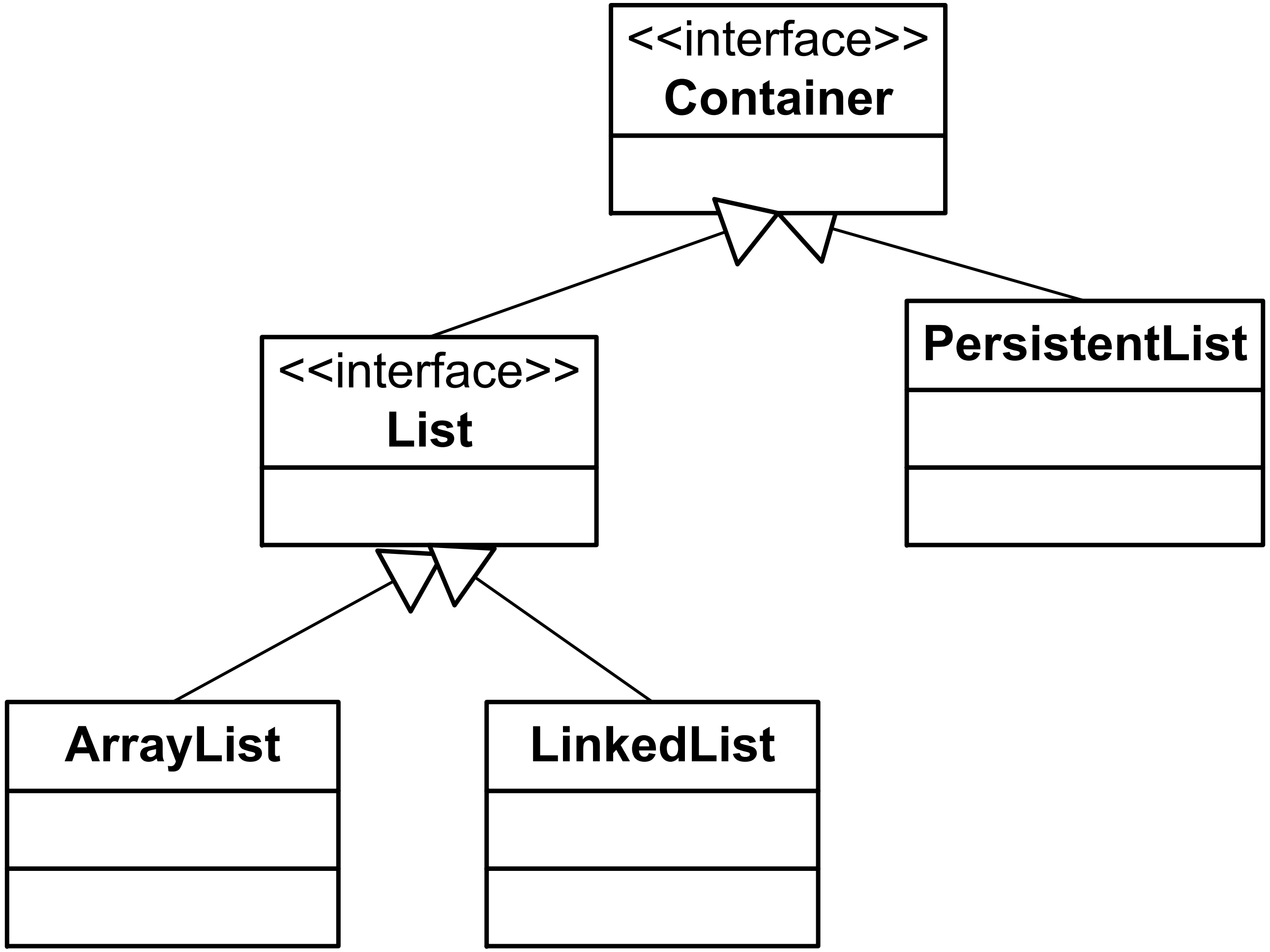
LSP Example: List





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

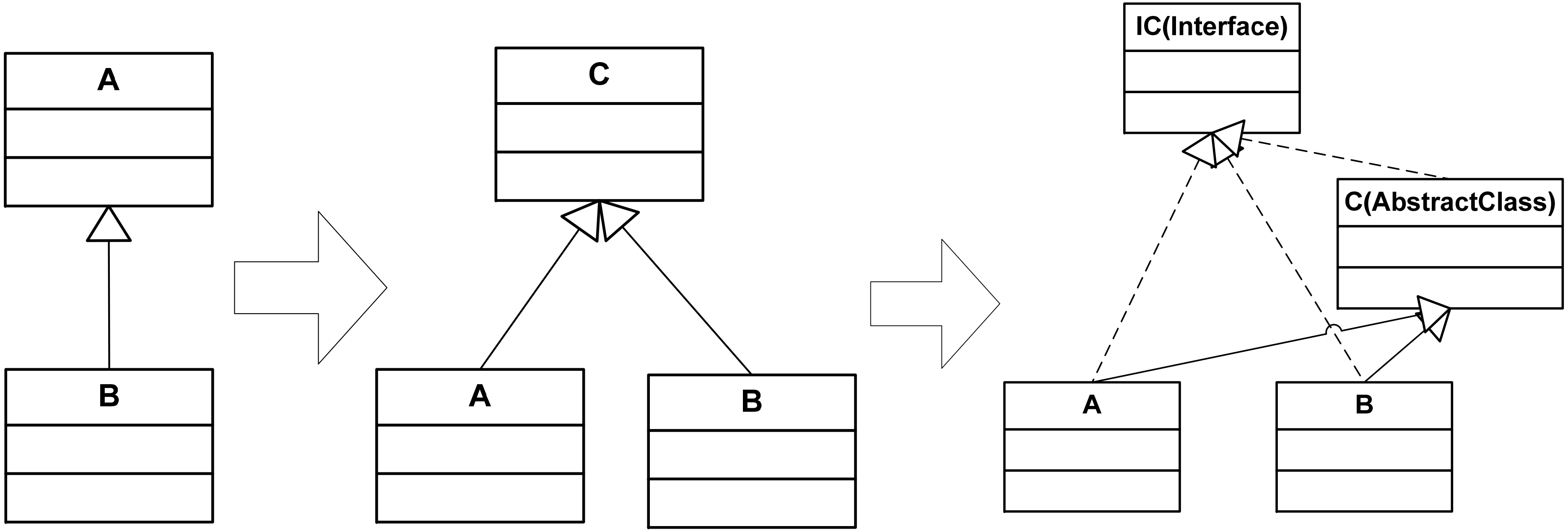

<<interface>>
List
PersistentObject
ArrayList
LinkedList
PersistentList


<<interface>>
Container
<<interface>>
List
PersistentList
ArrayList
LinkedList

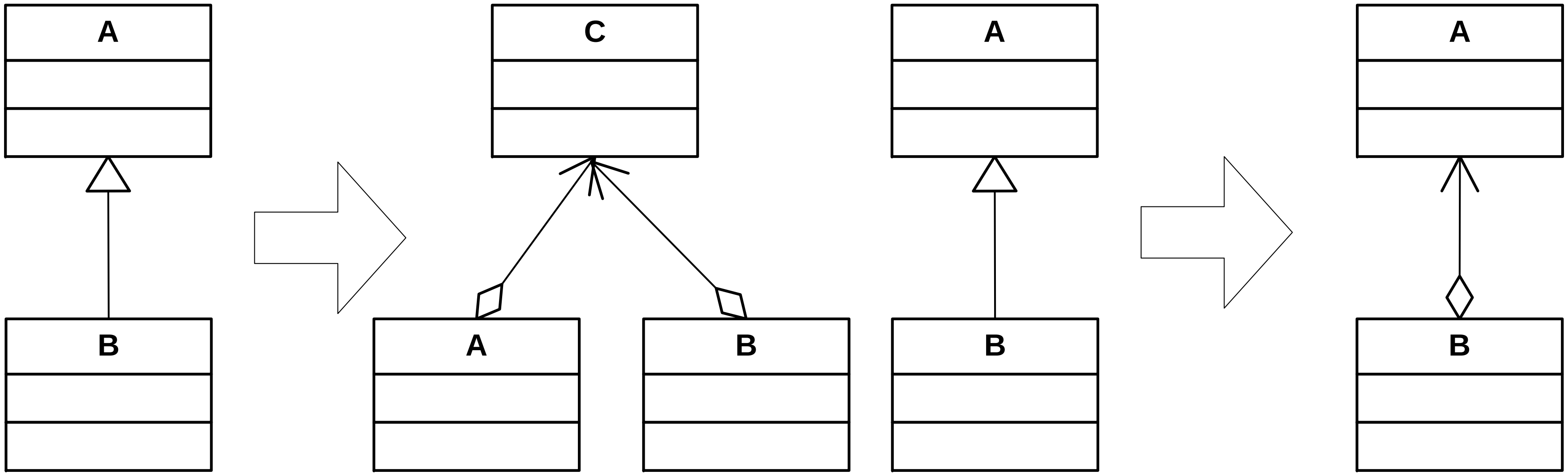
LSP Extension: Refactoring

- 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 concept, not behaviors.
  - 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决，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。

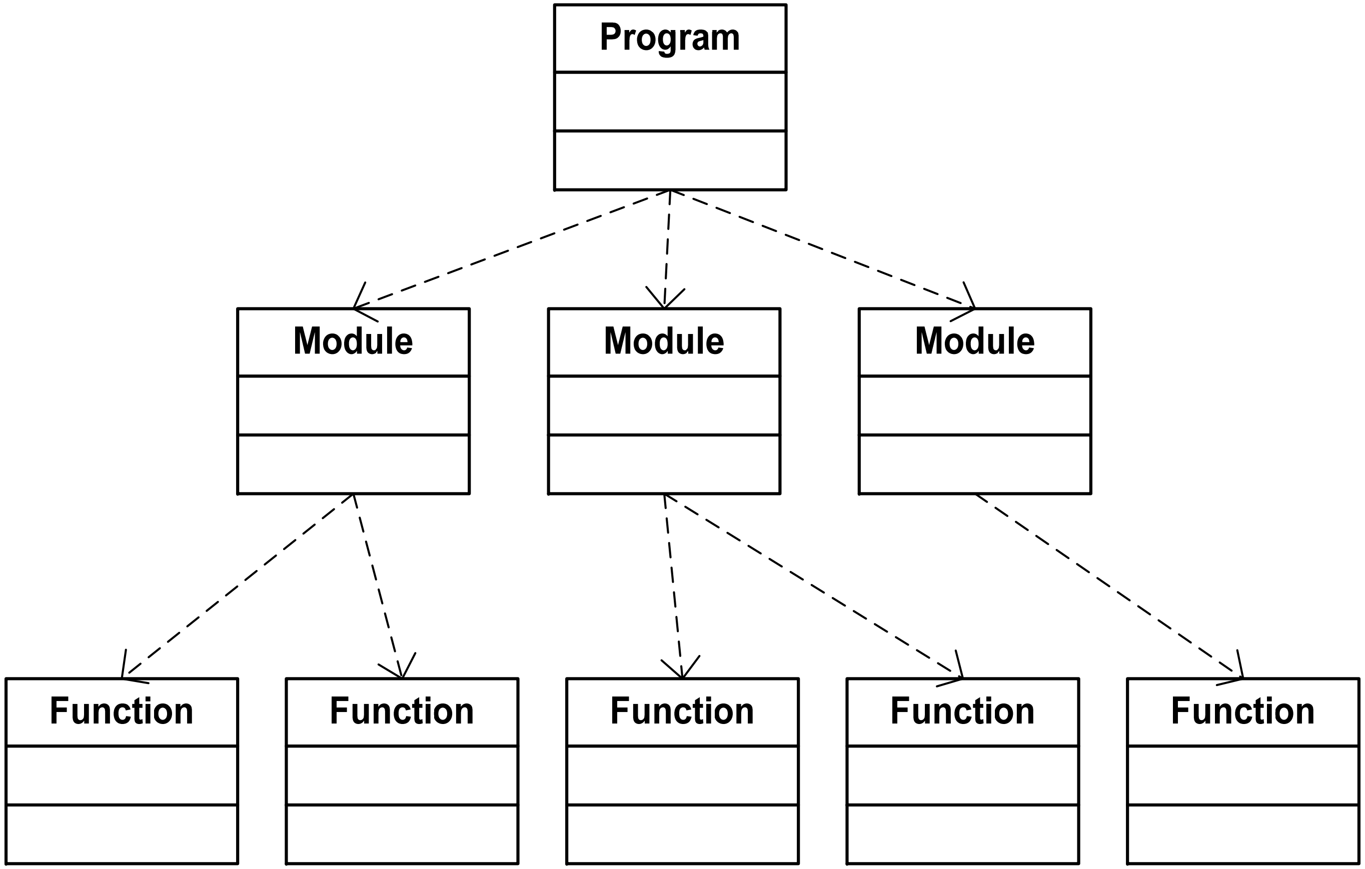
DIP : Description

- 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

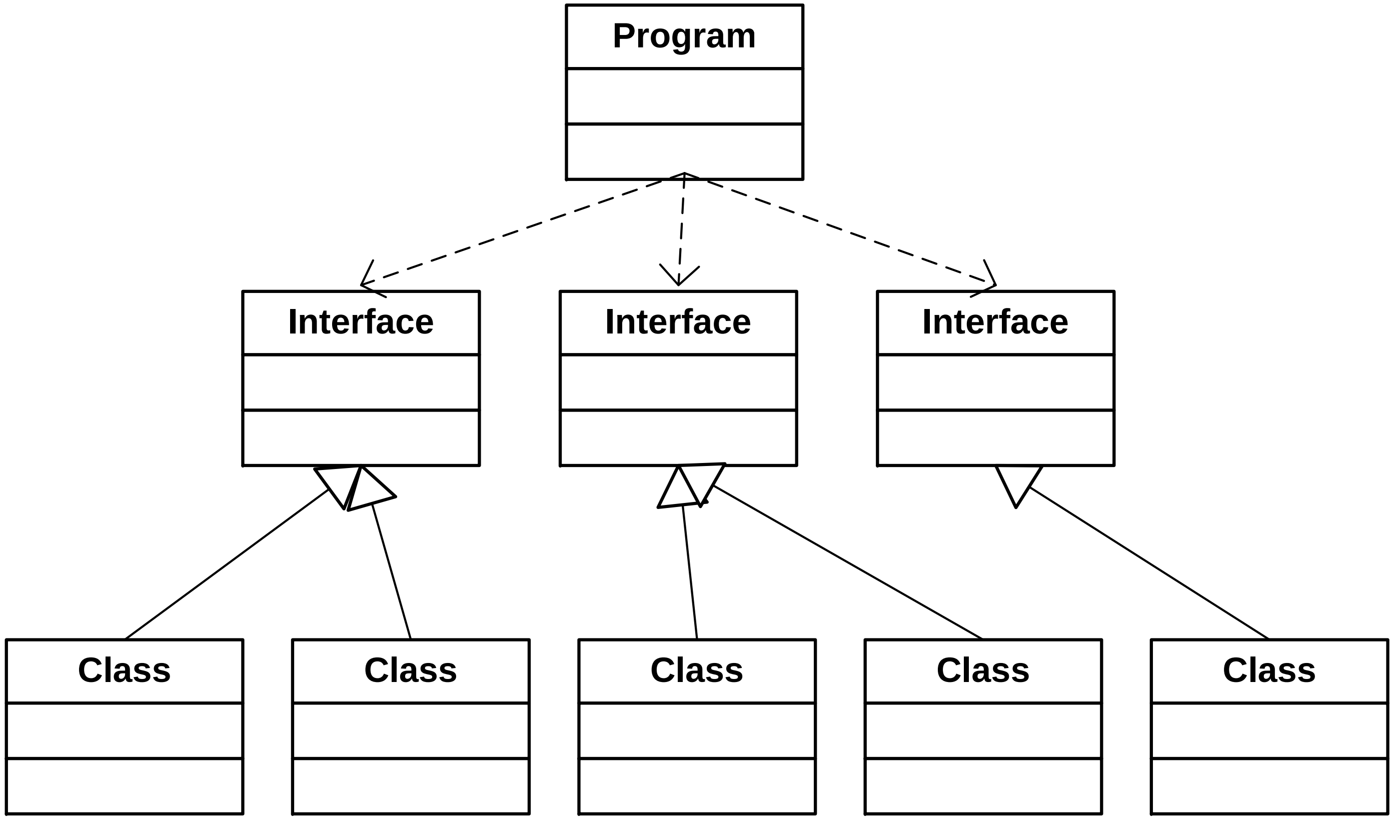
DIP : Implementation

- 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.

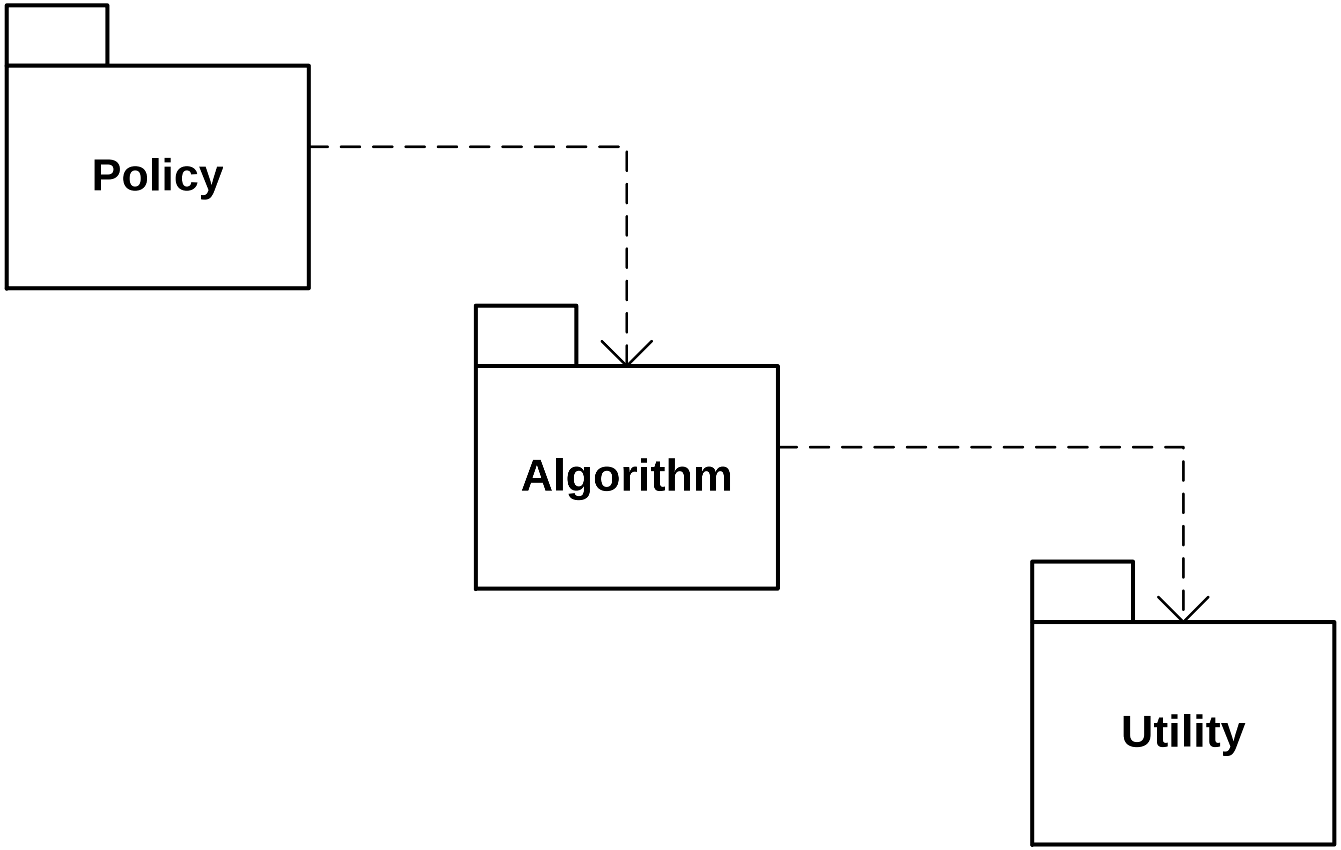
Procedural Design

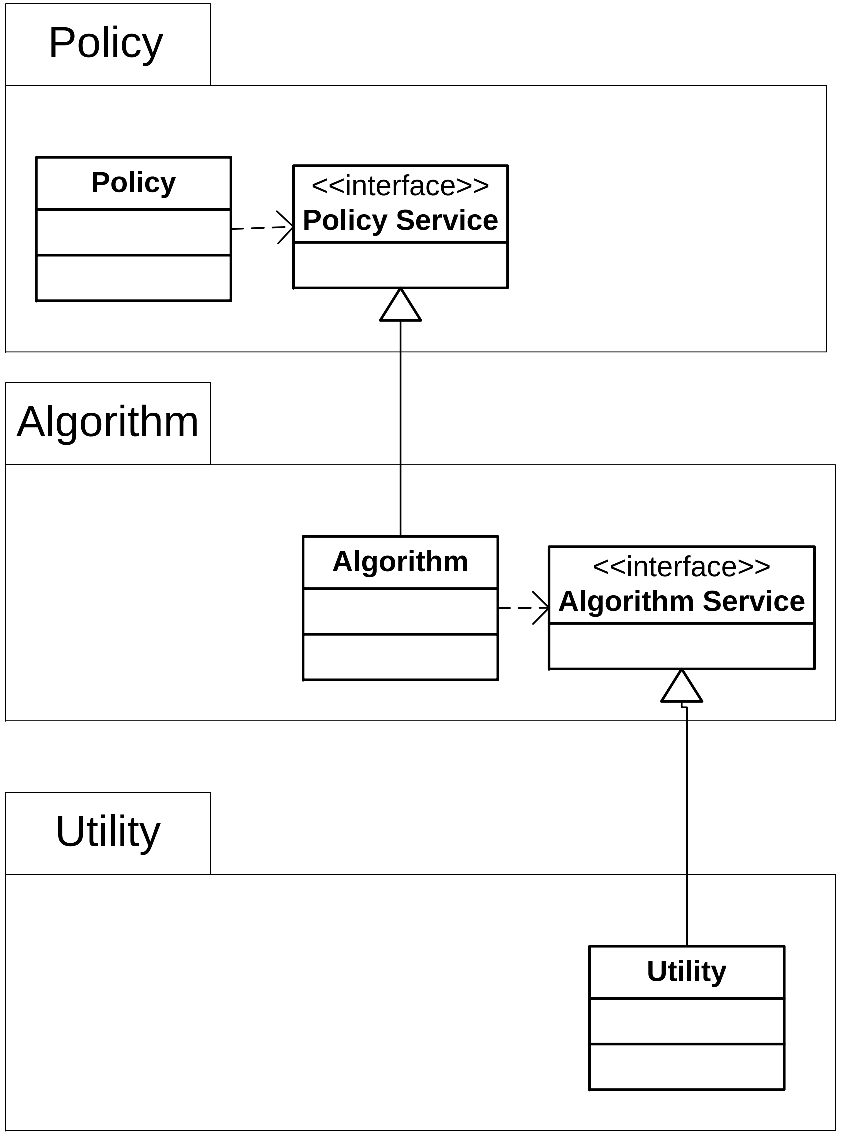


DIP (OOD)

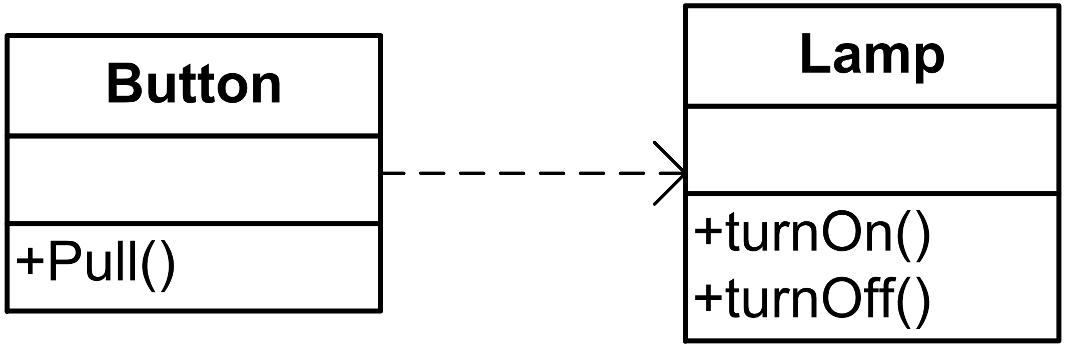

Program
Interface
Interface
Interface
Class
Class
Class
Class
Class

DIP Example:
General layers of an module

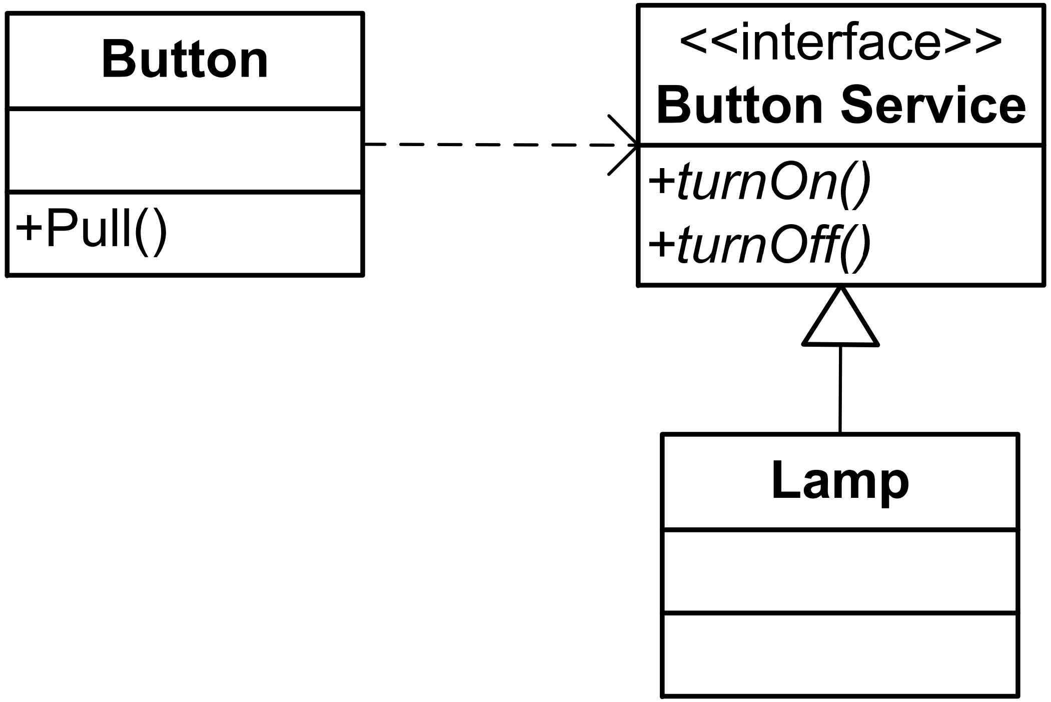



Policy
Policy
<<interface>>
Policy Service
Algorithm
Algorithm
<<interface>>
Algorithm Service
Utility
Utility

DIP Example: Button and Lamp

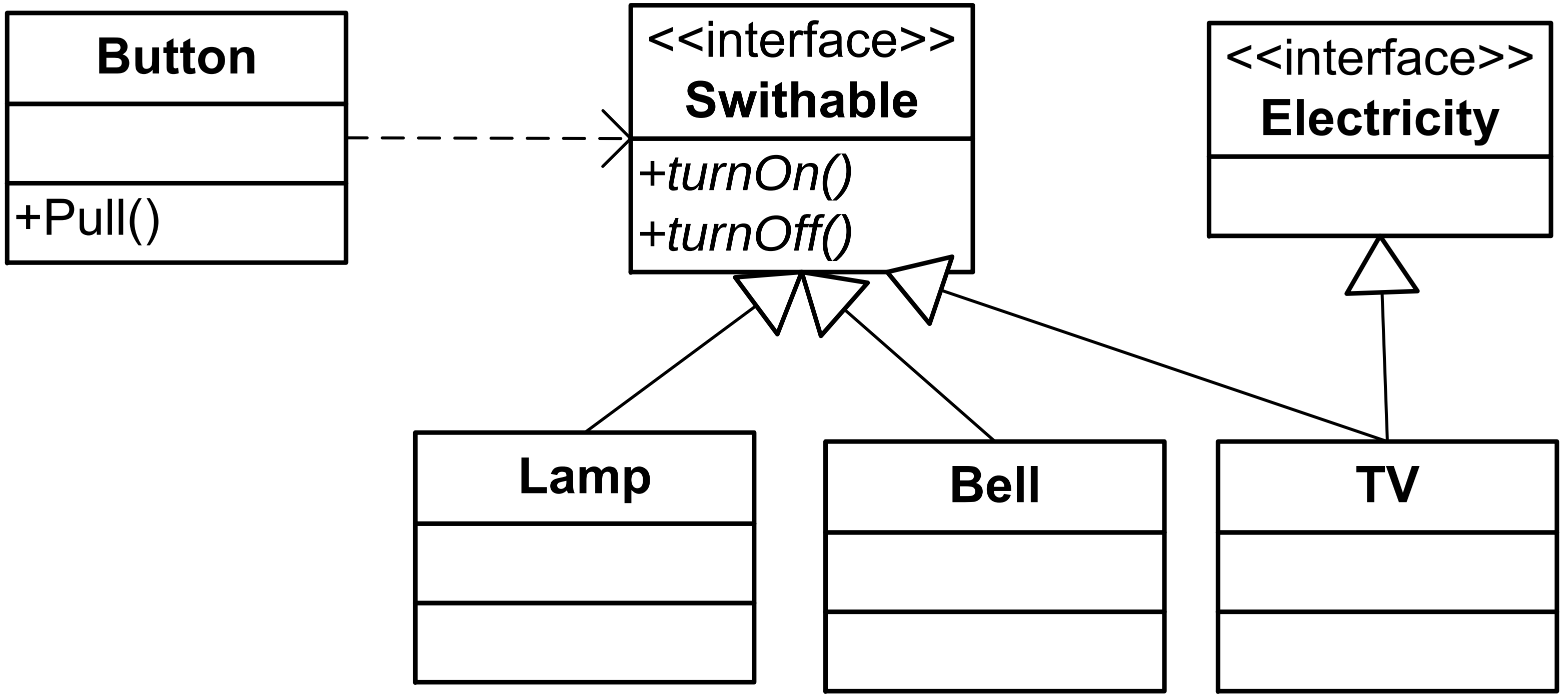


DIP Example: Button and Lamp



隐喻（**Metaphor**）是功能背后的抽象，是那些不随具体细节的改变而改变的真理。它是系统内部的系统。

DIP Example: Button and Lamp



DIP : Rules

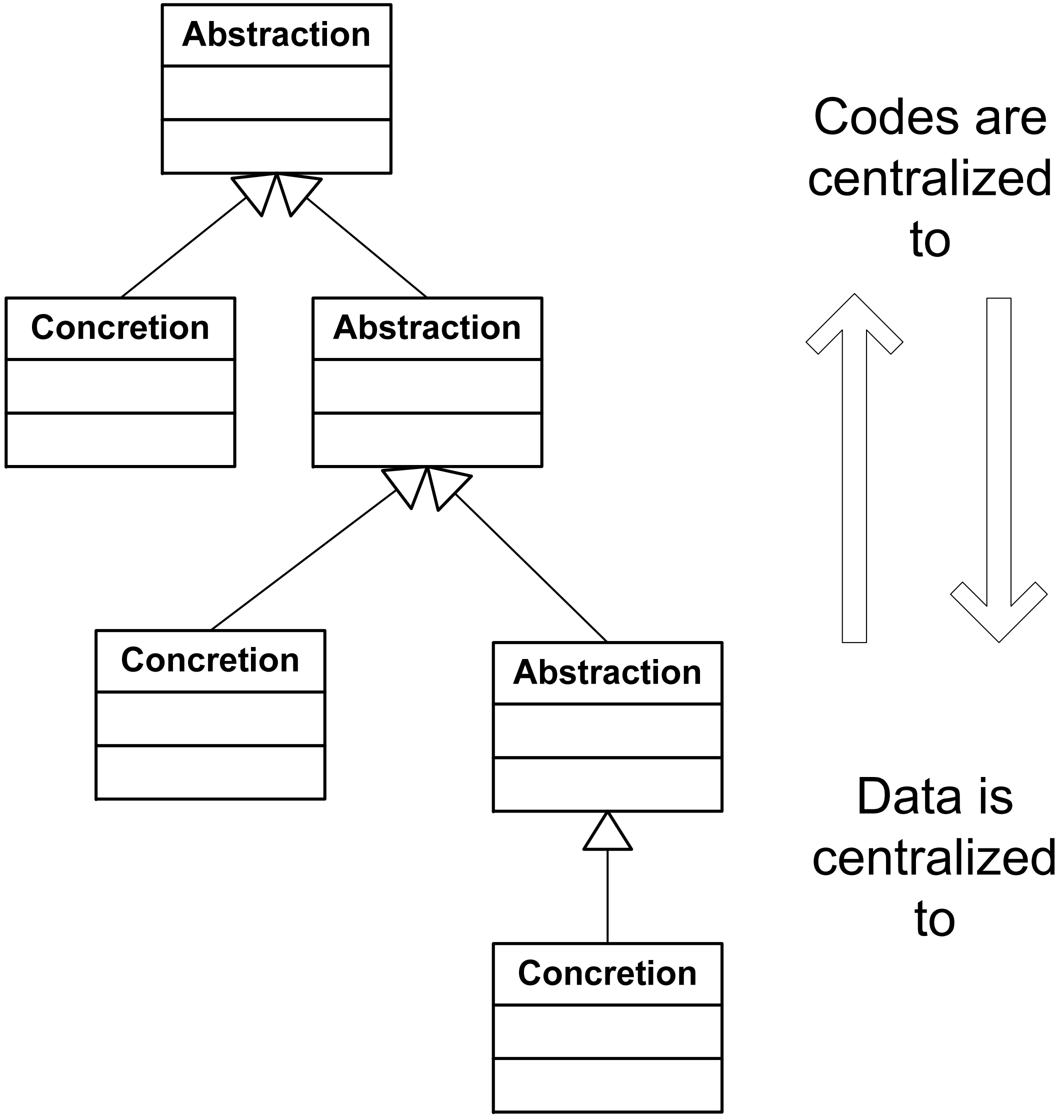
- Any variables should NOT hold a reference which refer to a concrete class, but a interface or abstract class;
- Any classes should NOT inherit from a concrete class;
- Any methods should NOT override the base-methods which has been implemented in base class.
- The rules of DIP is over-strict.

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;

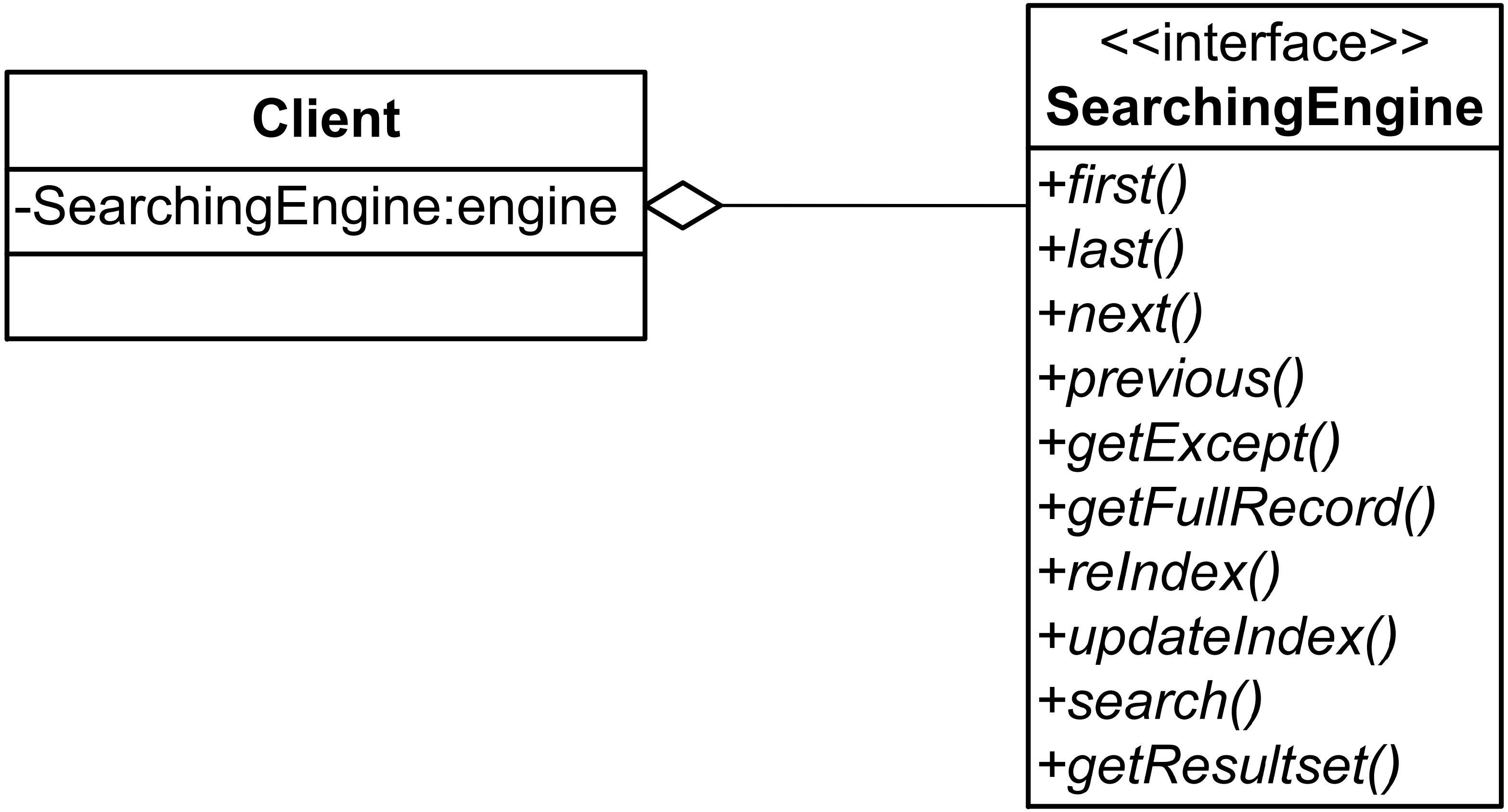
ISP : Description

- 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)。

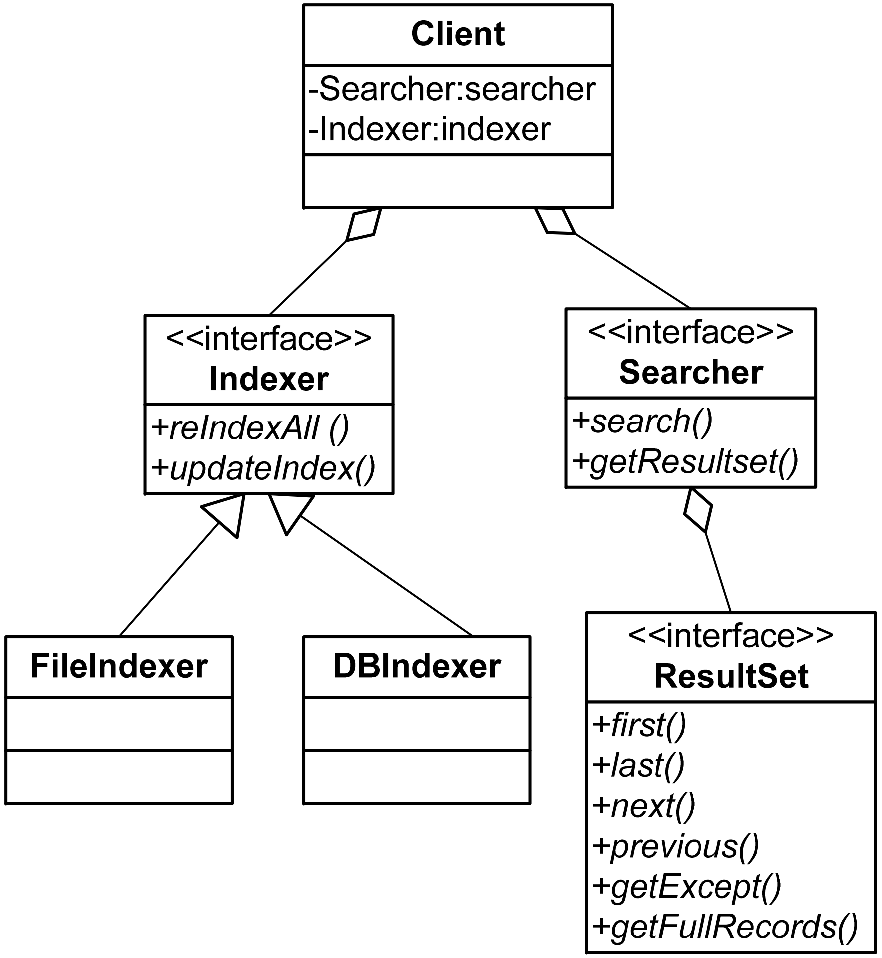
ISP : Interface Pollution

- 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


Client
-SearchingEngine:engine
<<interface>>
SearchingEngine
+first()
+last()
+next()
+previous()
+getExcept()
+getFullRecord()
+reIndex()
+updateIndex()
+search()
+getResultset()

ISP : Example


Client
-Searcher:searcher
-Indexer:indexer
<<interface>>
Indexer
+reIndexAll ()
+updateIndex()
<<interface>>
Searcher
+search()
+getResultset()
FileIndexer
DBIndexer
<<interface>>
ResultSet
+first()
+last()
+next()
+previous()
+getExcept()
+getFullRecords()

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

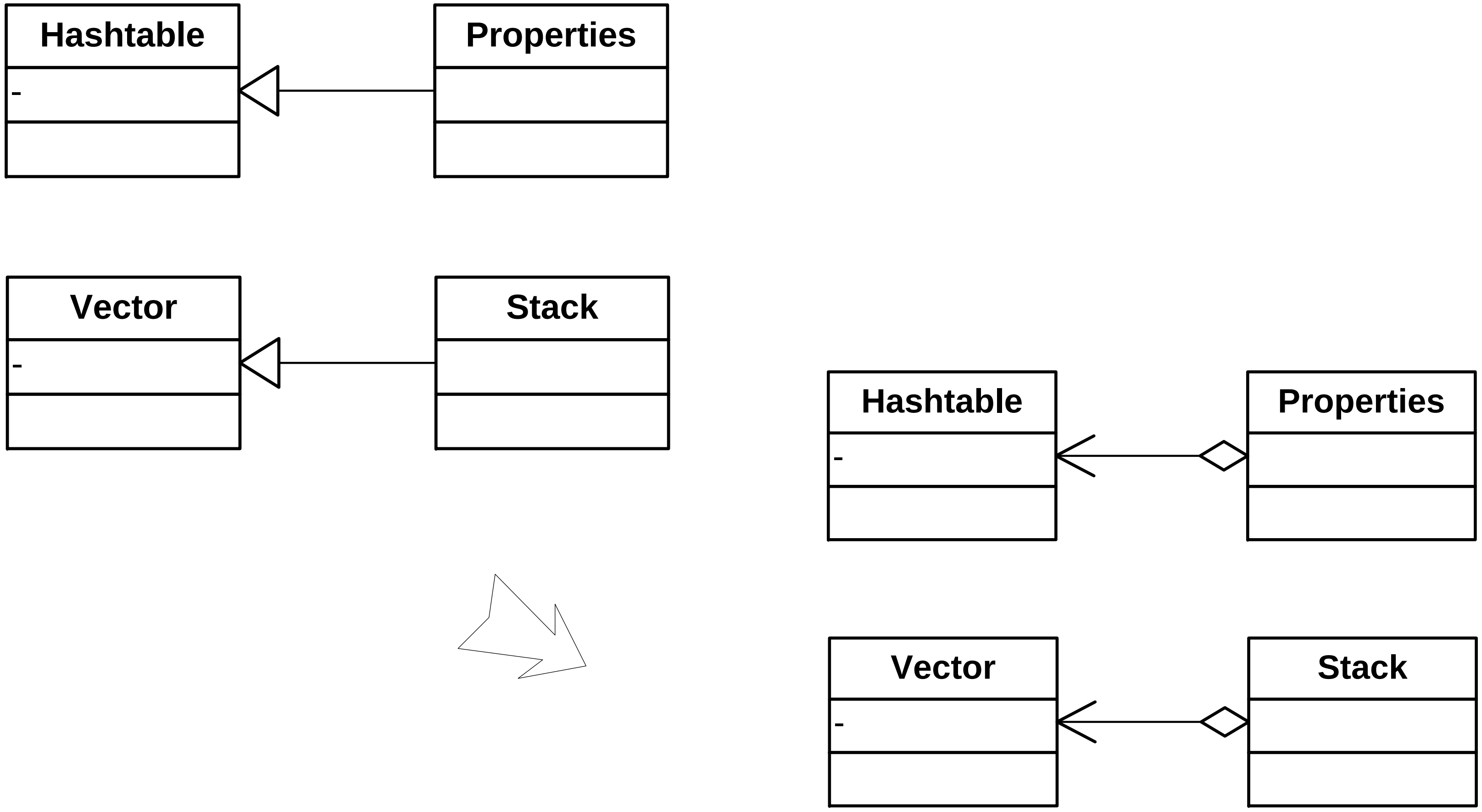
CRP: Definition

- 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。

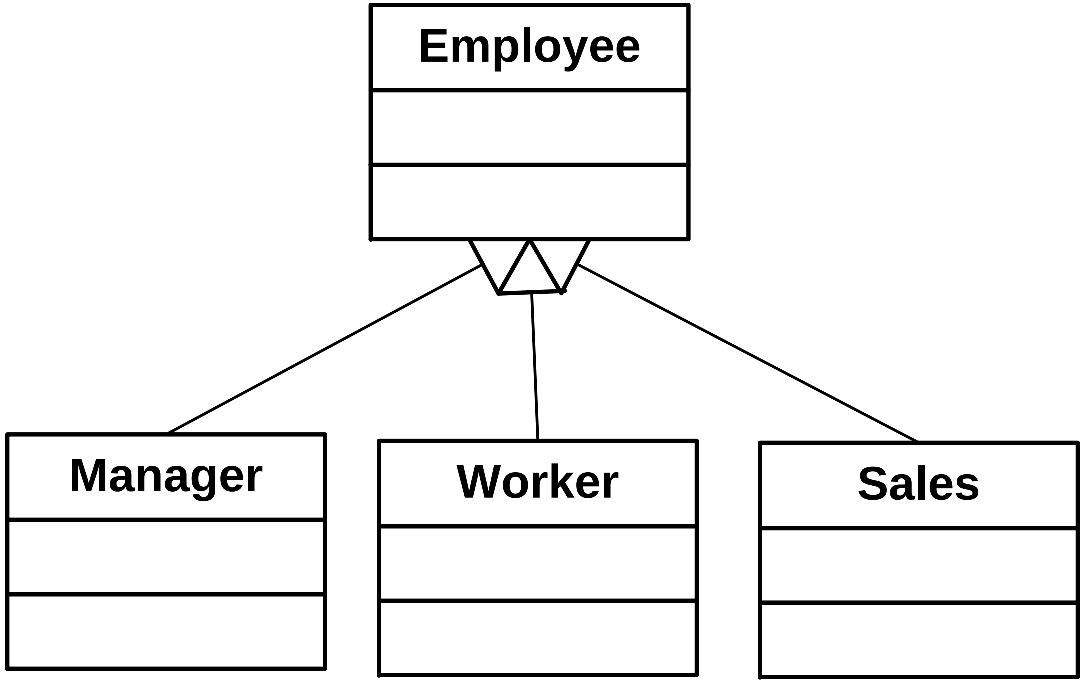
CRP: Description

- 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.

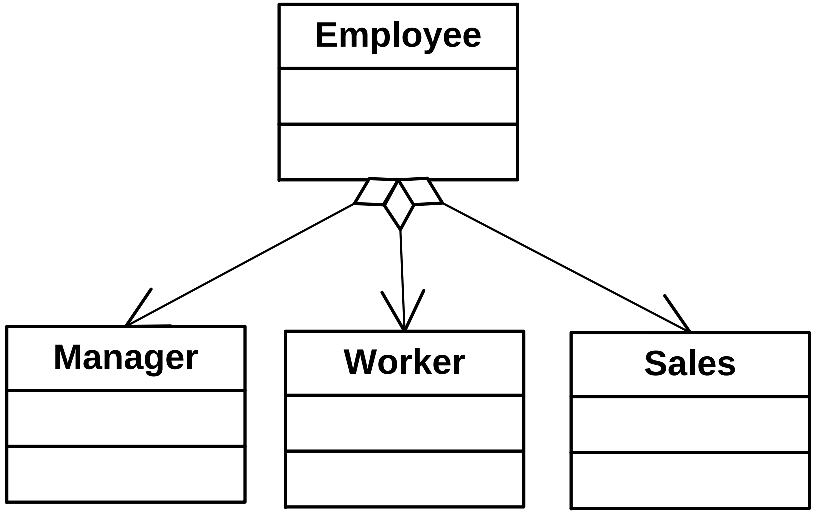
CRP Example (reuse): JDK Container

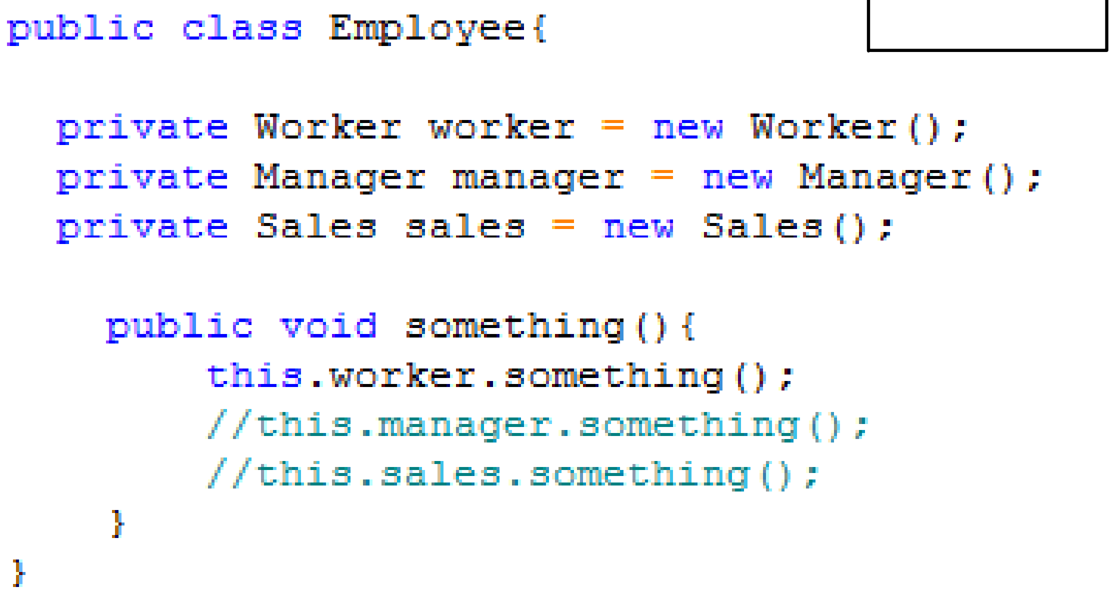


CRP Example (polymorphic) : Employee



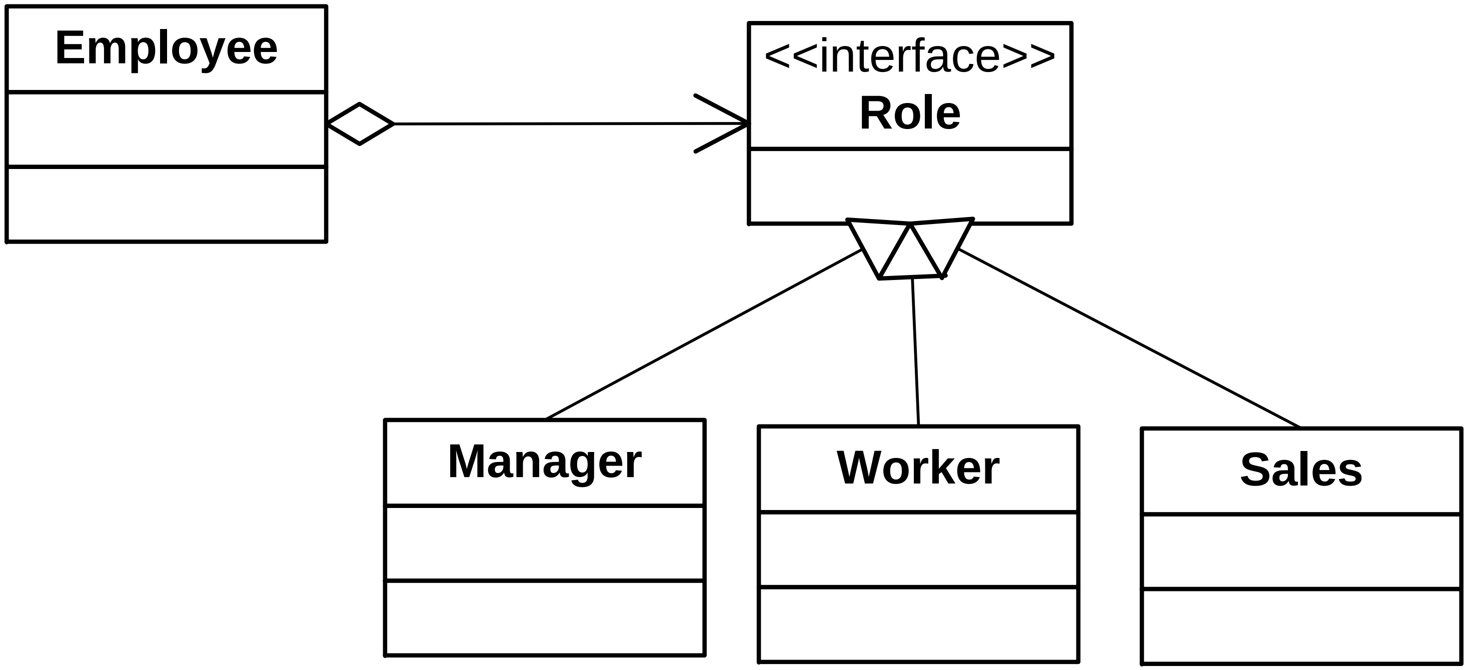
Employee





```
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 it 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…