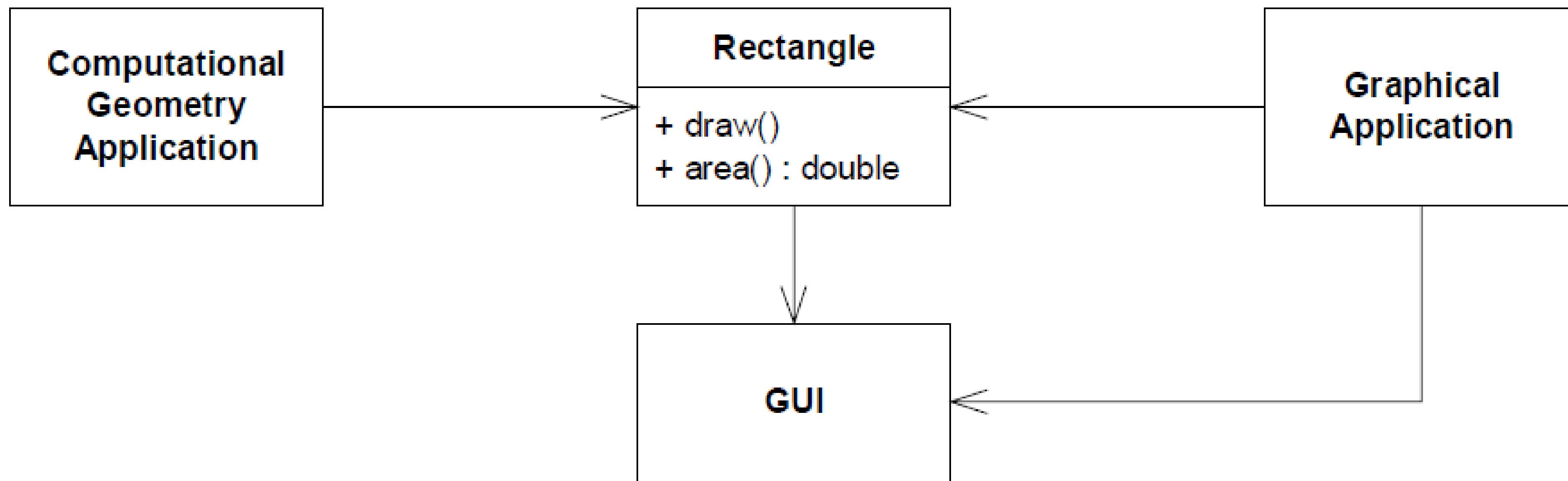


# 面向对象的模块化

刘钦



---

Figure 9-1

# 课前练习

# Outline

- 面向对象中的模块与耦合
- 访问耦合
- 继承耦合
- 内聚
- 耦合和内聚的度量

# Module

- A piece of code
  - Methods
  - Class
  - Module(package)
- Coupling:
  - among pieces
- Cohesion:
  - internal a piece

# Structural methods vs OO methods in Coupling

- Coupling
  - Coupling is the measure of the strength of association established by a connection from one module to another
- Structural methods
  - A connection is a reference to some label or address defined elsewhere
- OO methods
  - Component coupling (访问耦合)
  - Inheritance coupling (继承耦合)

# 降低耦合的设计原则

- 1: 《Global Variables Consider Harmful》
- 2: 《To be Explicit》
- 3: 《Do not Repeat》
- 4: Programming to Interface

# Outline

- 面向对象中的模块与耦合
- 访问耦合
- 继承耦合
- 内聚
- 耦合和内聚的度量

表 14-1 访问耦合

类 型	耦 合 性	解 释	例 子
隐式访问	最高 ↑ ↓ 最低	B 既没在 A 的规格中出现，也没在实现中出现	Cascading Message
实现中访问		B 的引用是 A 方法中的局部变量	1) 通过引入局部变量，避免 Cascading Message 2) 方法中创建一个对象，将其引用赋予方法的局部变量，并使用
成员变量访问		B 的引用是 A 的成员变量	类的规格中包含所有需接口和供接口（需要特殊语言机制）
参数变量访问		B 的引用是 A 的方法的参数变量	类的规格中包含所有需接口和供接口（需要特殊语言机制）
无访问		理论最优，无关联耦合，维护时不需要对方任何信息	完全独立

注：源自 [ Eder1992 ]。

# 访问耦合

**Example:** Consider the class `EMPLOYEE` as defined above with the additional instance variable `involvedInProject`, which references the project for which an employee is currently working, and the additional method `numberColleagues`, which returns the number of colleagues in the current project. The implementation of `numberColleagues` may be given as follows:

```
int numberColleagues () {  
    return (involvedInProject->getProjectMembers->count - 1)  
}
```

# Cascading Message问题

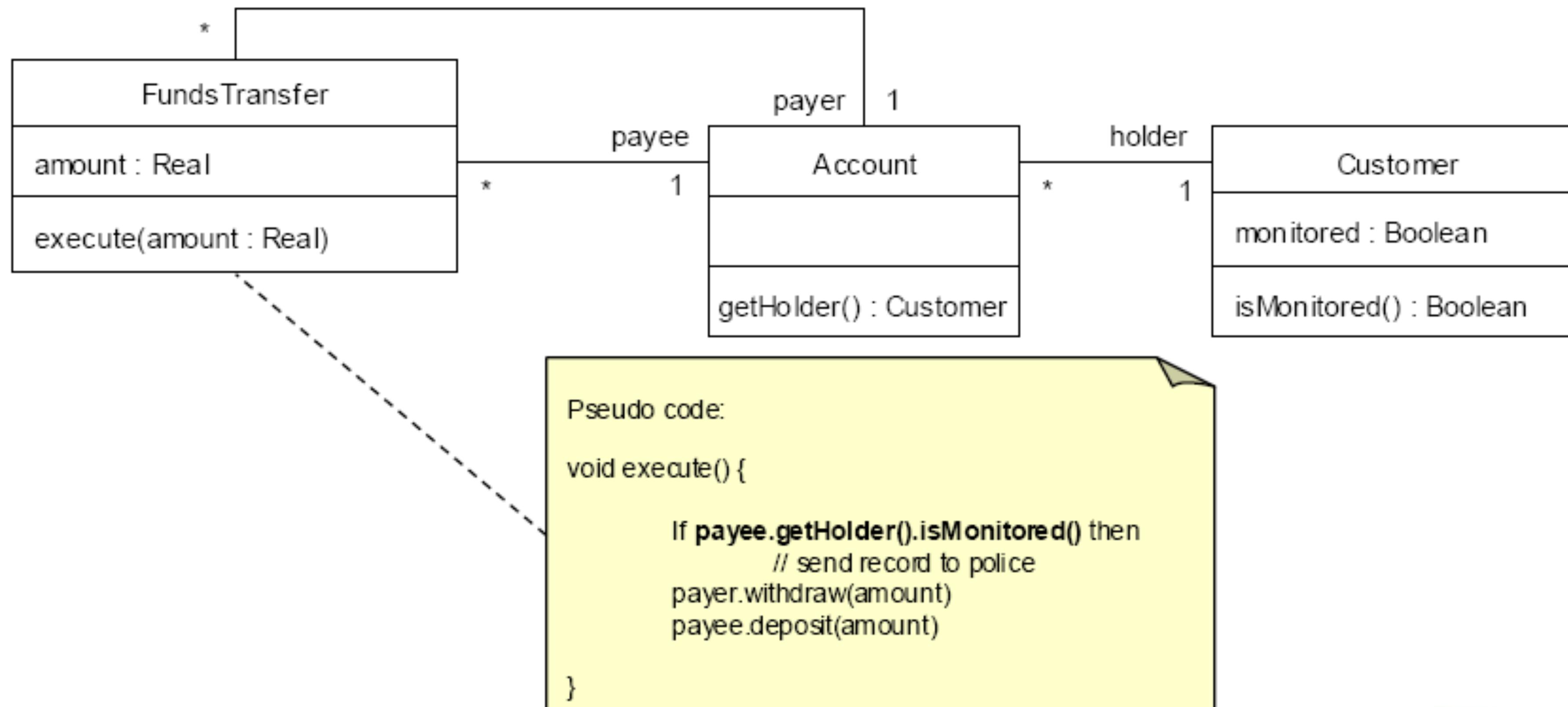
**Example:** Consider the previous example where the classes `EMPLOYEE` and `SET(EMPLOYEE*)` are hidden coupled due to the implementation of the method `numberColleagues`. The implementation may be improved by disallowing cascading messages as follows:

```
int numberColleagues () {
    SET<EMPLOYEE*> * projectMembers;
    projectMembers = involvedInProject->getProjectMembers;
    return (projectMembers->count - 1)
}
```

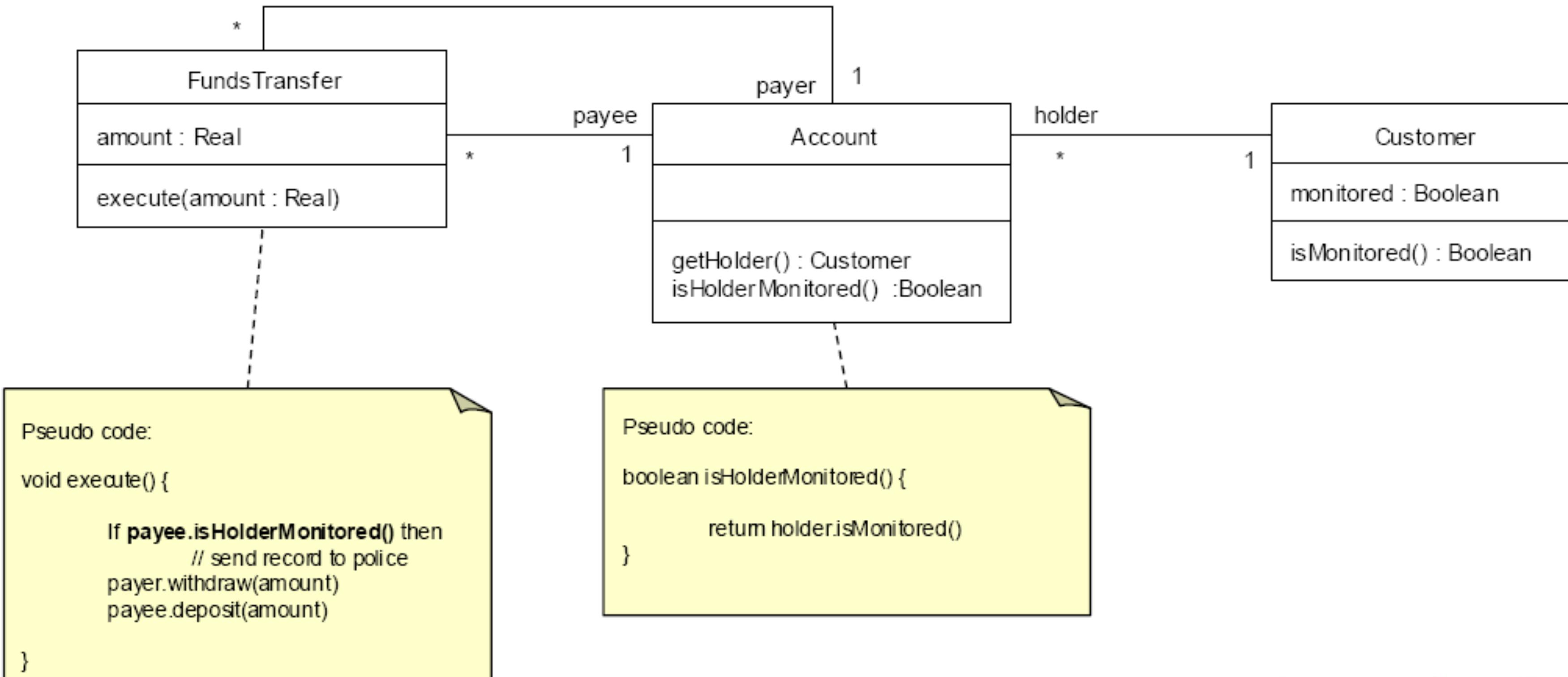
By introducing local variables and disallowing cascading messages the coupling between the classes `EMPLOYEE` and `SET(EMPLOYEE*)` can be improved from hidden to scattered. □

# 解决方案—引入局部变量

# Cascading Message|问题案例

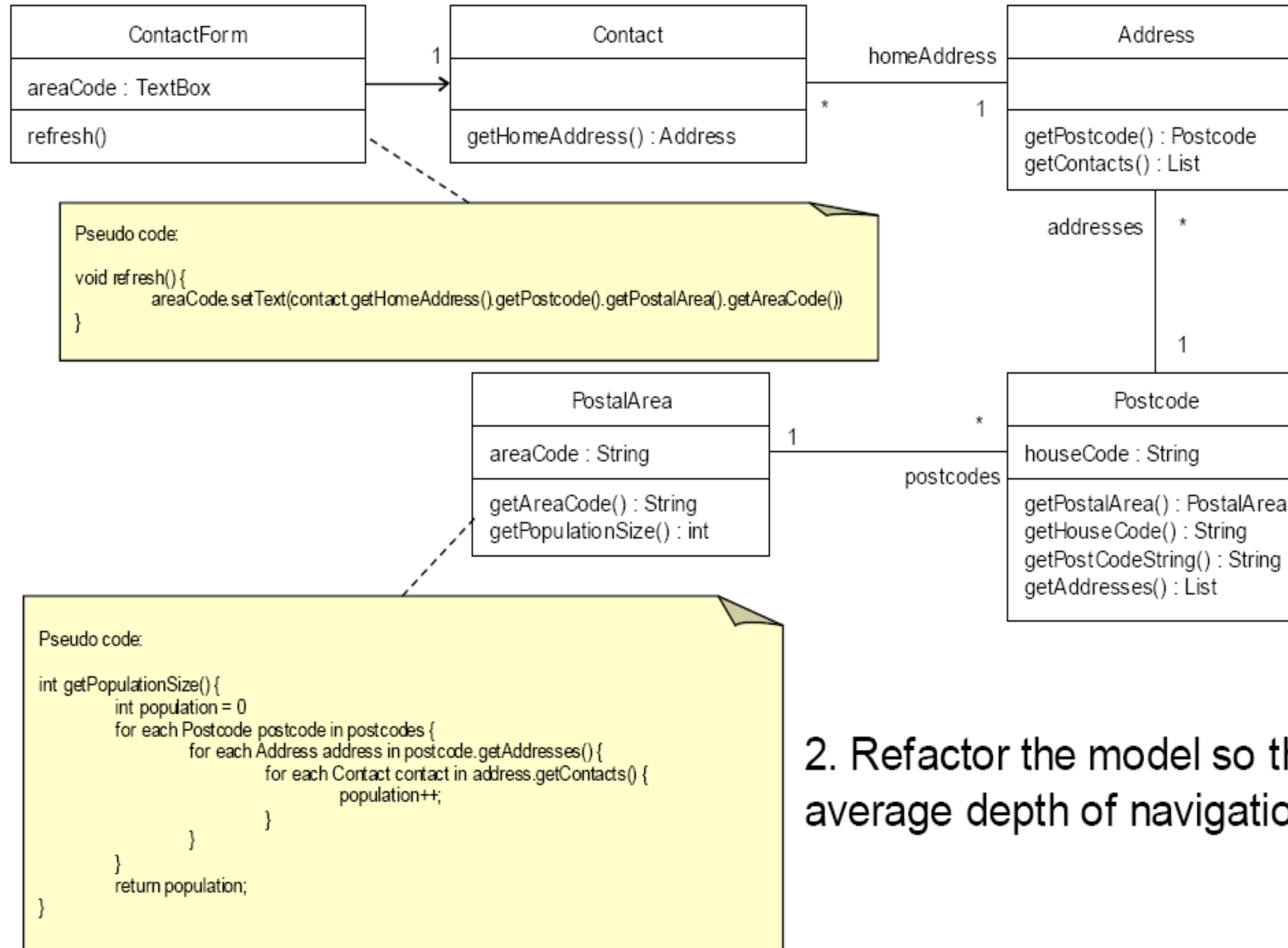


# 解决方案 — 委托



# Principles of Component Coupling

- Principle 5: The Law of Demeter
  - 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.
  - You can play with toys you've made yourself.



2. Refactor the model so that the average depth of navigation is 1

# 问题案例

# Principles of Component Coupling

- Principle 4: Programming to Interface

面向接口编程

- Programming to Required Interface, not only Suffered Interface
- Design by Contract
- Contract of Module/ Class
  - Required methods / Provided methods
  - Contract of Methods
    - PreCondition, PostCondition, Invariant

**Example:** In the previous example the classes EMPLOYEE and SET(EMPLOYEE\*) are scattered coupled. We may improve their coupling property to specified coupling by changing the specification of EMPLOYEE as follows:

```
class EMPLOYEE {  
    suffered interface:          /* corresponds to public in C++ */  
        int computeSalary ();  
        int numberColleagues ();  
        ...  
    required interface:          /* not available in C++ */  
        SET<EMPLOYEE*>* class PROJECT::getProjectMembers ();  
        int class SET<EMPLOYEE*>::count ();  
        ...  
};
```

# 案例

# Principles of Component Coupling

**Clients should not be forced to depend  
upon interfaces that they do not use.**

**R. Martin, 1996**

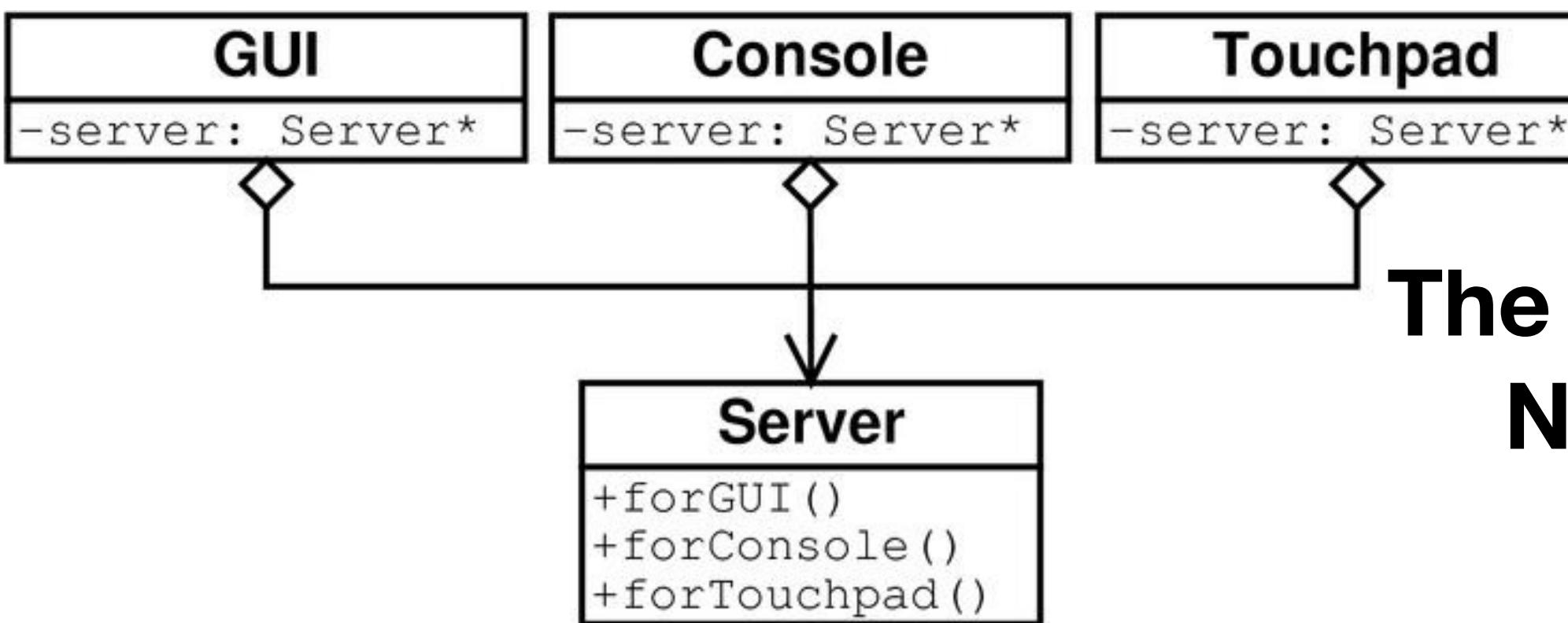
接口分离原则

- Principles 6: Interface Segregation Principle(ISP)
  - Programming to Simpler Interface
  - Many client-specific interfaces are better than one general purpose interface

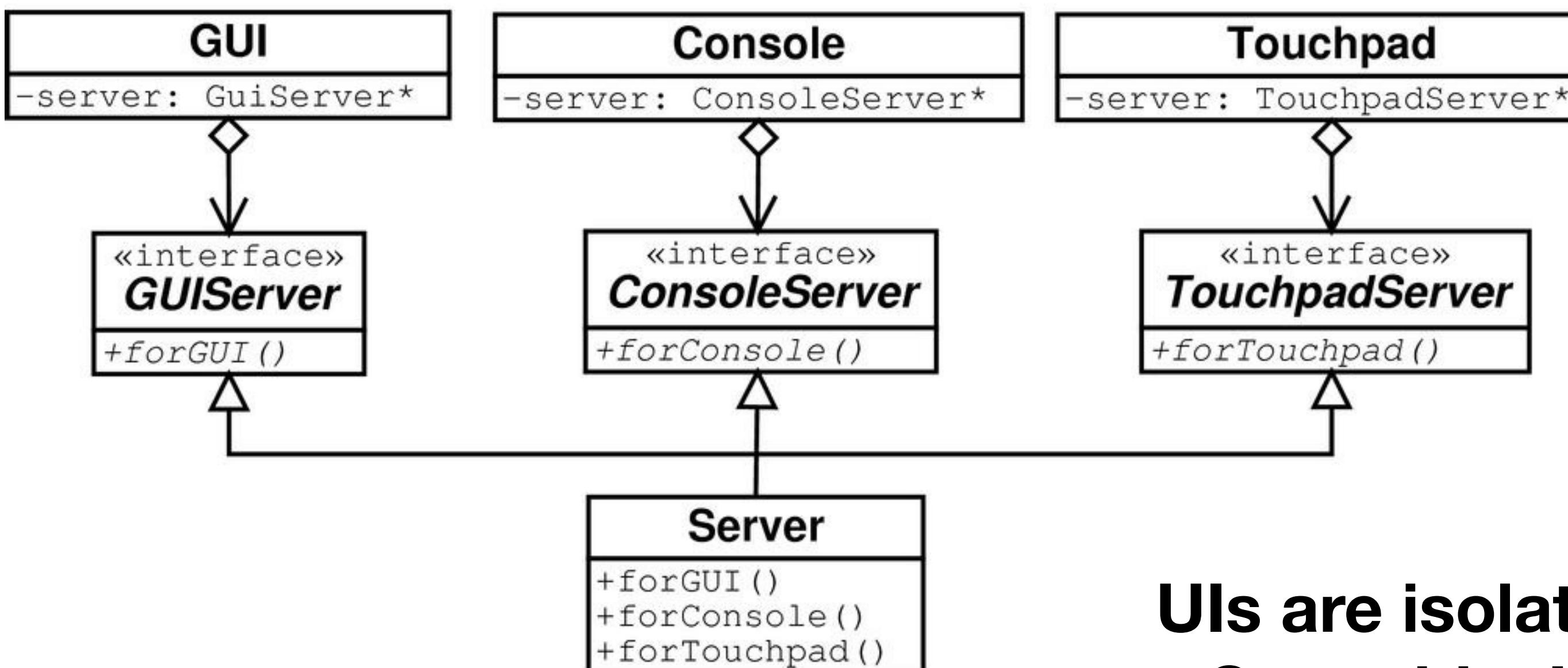
# Principles of Component Coupling

## — — ISP Explained

- Multipurpose classes
  - Methods fall in different groups
  - Not all users use all methods
- Can lead to unwanted dependencies
  - Clients using one aspect of a class also depend indirectly on the dependencies of the other aspects
- ISP helps to solve the problem
  - Use several client-specific interfaces

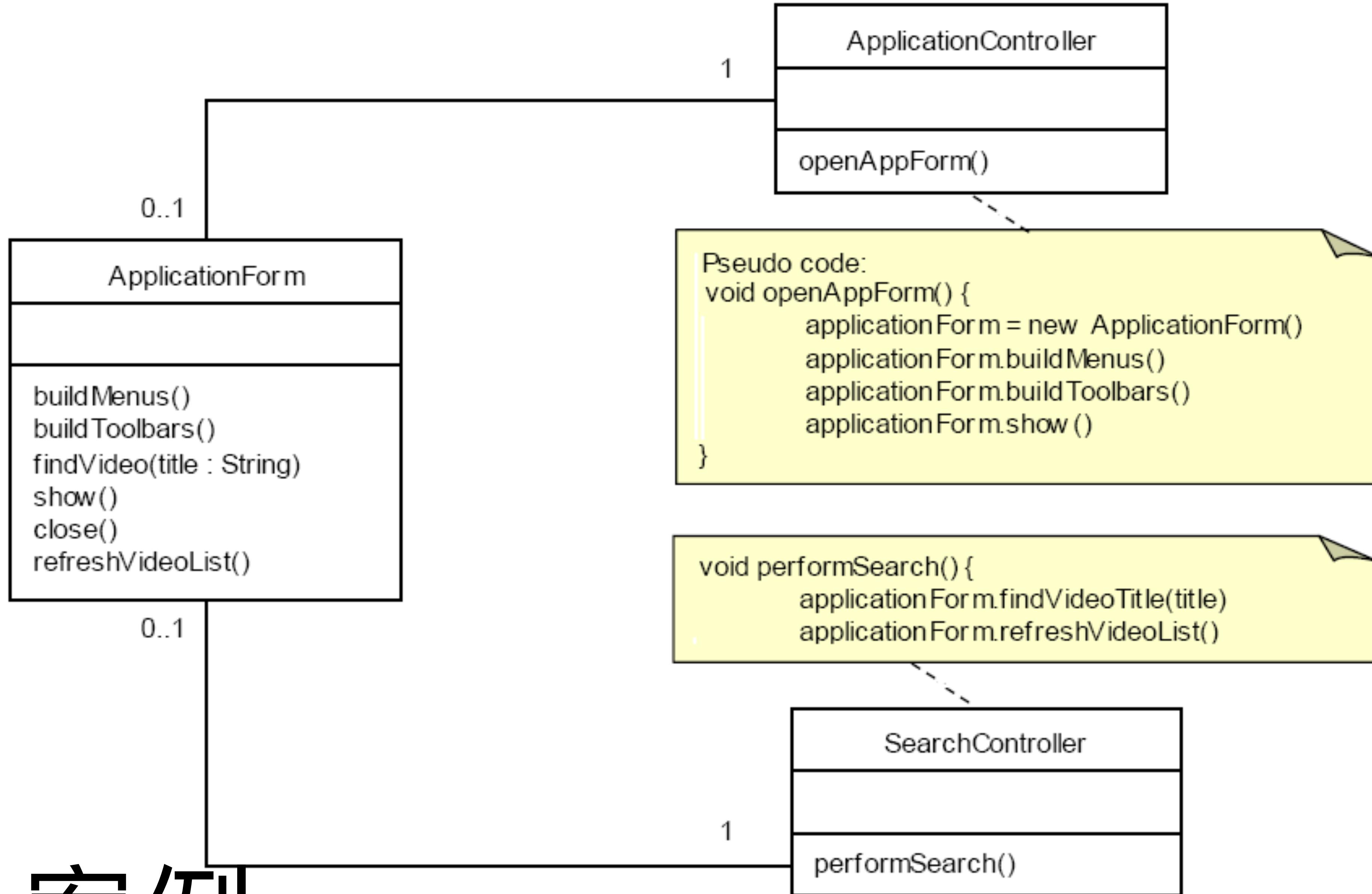


**The Server "collects" interfaces**  
**New UI → Server interface**  
**changes**  
**All other UIs recompile**



**UIs are isolated from each other**  
**Can add a UI with changes in**  
**Server → other UIs not affected**

案例



案例

# Outline

- 面向对象中的模块与耦合
- 访问耦合
- 继承耦合
- 内聚
- 耦合和内聚的度量

表 14-2 继承耦合

类 型		耦 合 性	解 释
修改 (modification)	规格	最高	子类任意修改从父类继承回来的方法的接口
	实现		子类任意修改从父类继承回来的方法的实现
精化 (refinement)	规格	最低	子类只根据已经定义好的规则 (语义) 来修改父类的方法,且至少有一个方法的接口被改动
	实现		子类只根据已经定义好的规则 (语义) 来修改父类的方法,但只改动了方法的实现
扩展 (extension)			子类只是增加新的方法和成员变量, 不对从父类继承回来的任何成员进行更改
无 (nil)			两个类之间没有继承关系

# 继承耦合

# Modification Inheritance Coupling

- Modifying without any rules and restricts
- Worst Inheritance Coupling
- If a client using a parent ref, the parent and child method are all needed
  - Implicit
  - There are two connections, more complex
- Harm to polymorphism

**Example:** Consider class **STACK** inheriting from class **ARRAY**. Since **ARRAY** is only used to implement **STACK**'s internal data structure, and since the methods of **ARRAY** are semantically not meaningful when used with a stack (e.g., the method `putAt` of **ARRAY** does not exist for a stack) the methods of **ARRAY** are only inherited for private use but are deleted from the suffered, i.e., public interface of **STACK**. Thus **STACK** and **ARRAY** are signature modification coupled. To improve their coupling the definition of **STACK** should include an instance variable `a` with domain **ARRAY** instead of inheriting from **ARRAY**. □

# 问题案例

# Refinement Inheritance Coupling

- defining new information
- the inherited information is only changed due to predefined rules
- If a client using a parent ref, the whole parent and refinement of child are needed
  - 1+connections
- Necessary!

```

class PERSON {
    [0..120] age;           /* for simplicity we assume */
    ...
public;                  /* the existence of an enumeration type [0..120] */
                           /* and [15..65] */
    [0..120] getAge ();
    void setAge ([0..120] a);
    ...
}

class EMPLOYEE : public PERSON {
    [15..65] age;
    ...
public;
    [15..65] getAge ();
    void setAge ([15..65] a);
    ...
}

```

Since employees may only be active from 15 to 65 (at least in Austria) the subclass **EMPLOYEE** of class **PERSON** refines the signatures of the inherited access operations of **age** according to the covariant style. Thus, **EMPLOYEE** and **PERSON** are signature refinement coupled based on the covariant style.

# Extension Inheritance Coupling

- the subclass only adds methods and instance variables but neither modifies nor refines any of the inherited ones
- If a client using a parent ref, only the parent is needed
  - 1 connection

# Principles of Inherit Coupling

## Principle 7: Liskov Substitution Principle (LSP)

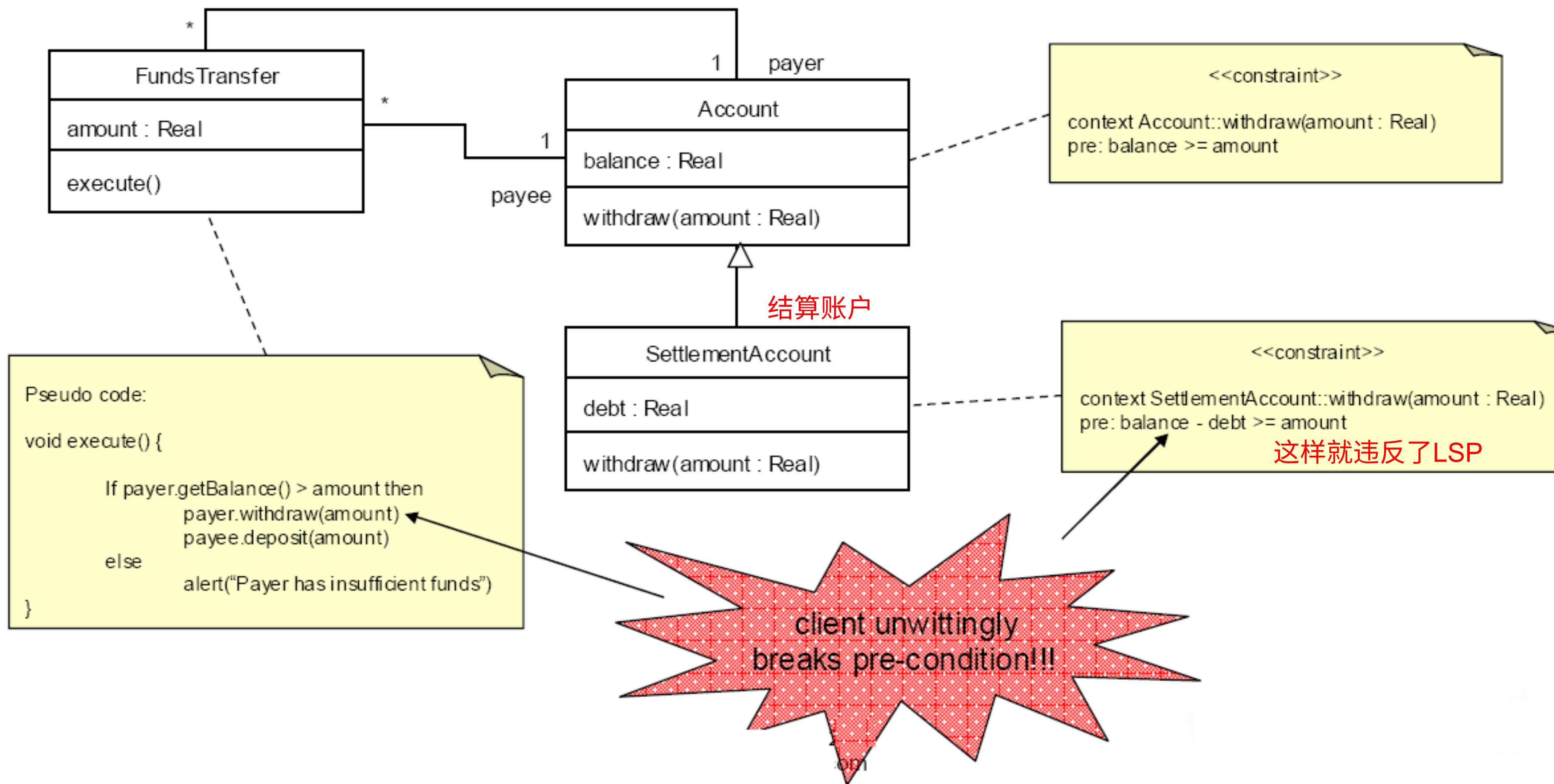
“All derived classes must be substitutable for their base class”

— Barbara Liskov, 1988

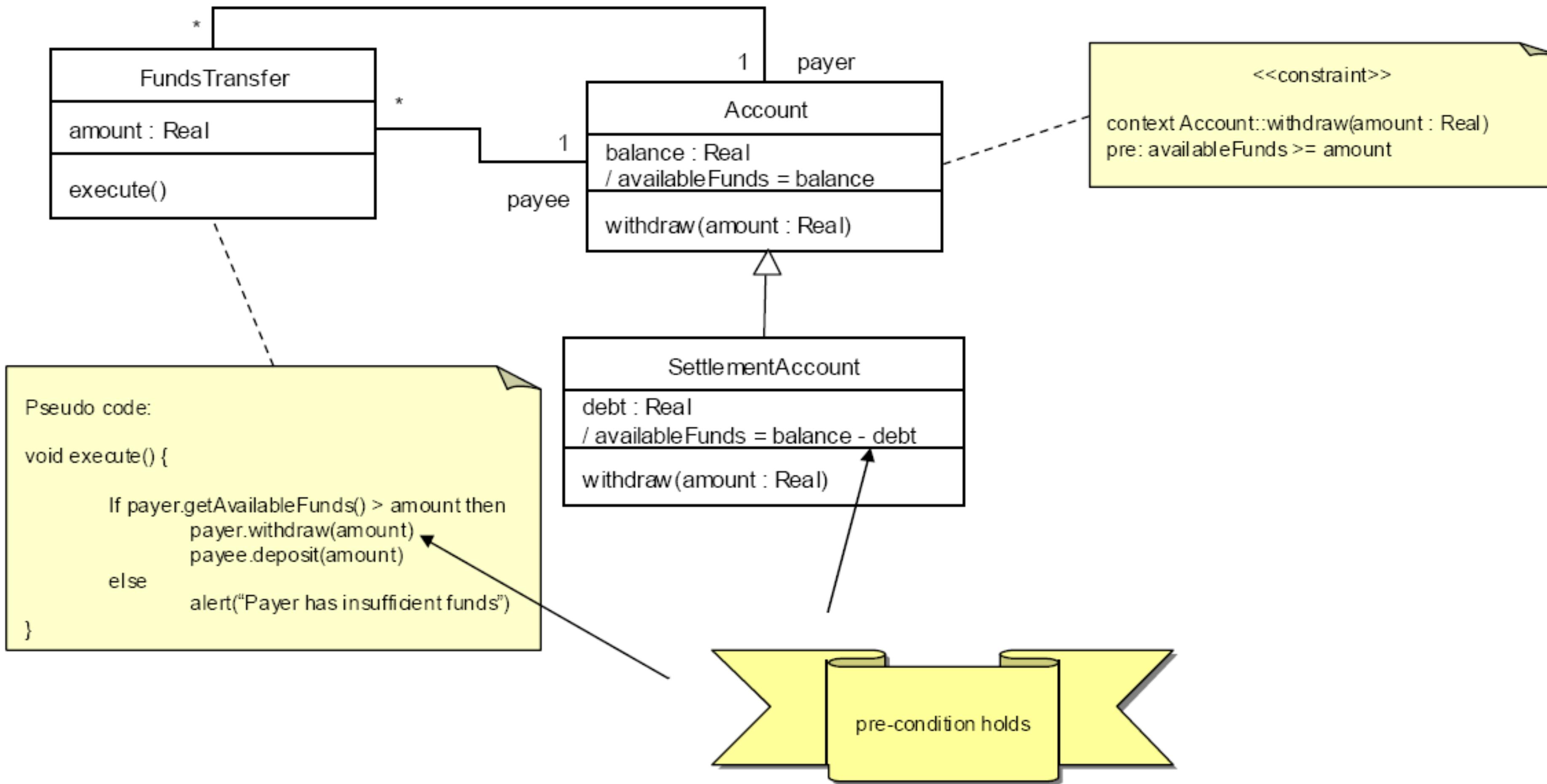
“Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.”

— R. Martin, 1996

# 问题案例



# 解决方案



# 问题案例 Is a Square a Rectangle?

- Rect r = new Rect();
- setWidth = 4;
- setHeight=5;
- assert(20 == getArea());
- class Square extends Rect{
  - // Square invariant, height = width
  - setWidth(x) {setHeight()=x}
  - setHeight(x) {setWidth(x)}
  - } // violate LSP?

# 问题案例 Penguin is a bird?

- class Bird { // has beak, wings,...
- public: virtual void fly(); // Bird can fly
- };
- class Parrot : public Bird { // Parrot is a bird
- public: virtual void mimic(); // Can Repeat words...
- };
- class Penguin : public Bird {
- public: void fly() {
- error ("Penguins don't fly!"); }
- };

# Penguins Fail to Fly!

- void PlayWithBird (Bird abird) {
  - abird.fly(); // OK if Parrot.
  - // if bird happens to be Penguin...OOOPS!!
  - }
- Does not model: “Penguins can’t fly”**  
**It models “Penguins may fly, but if they try it is error”**  
**Run-time error if attempt to fly → not desirable**  
**Think about Substitutability - Fails LSP**

# LSP Summary

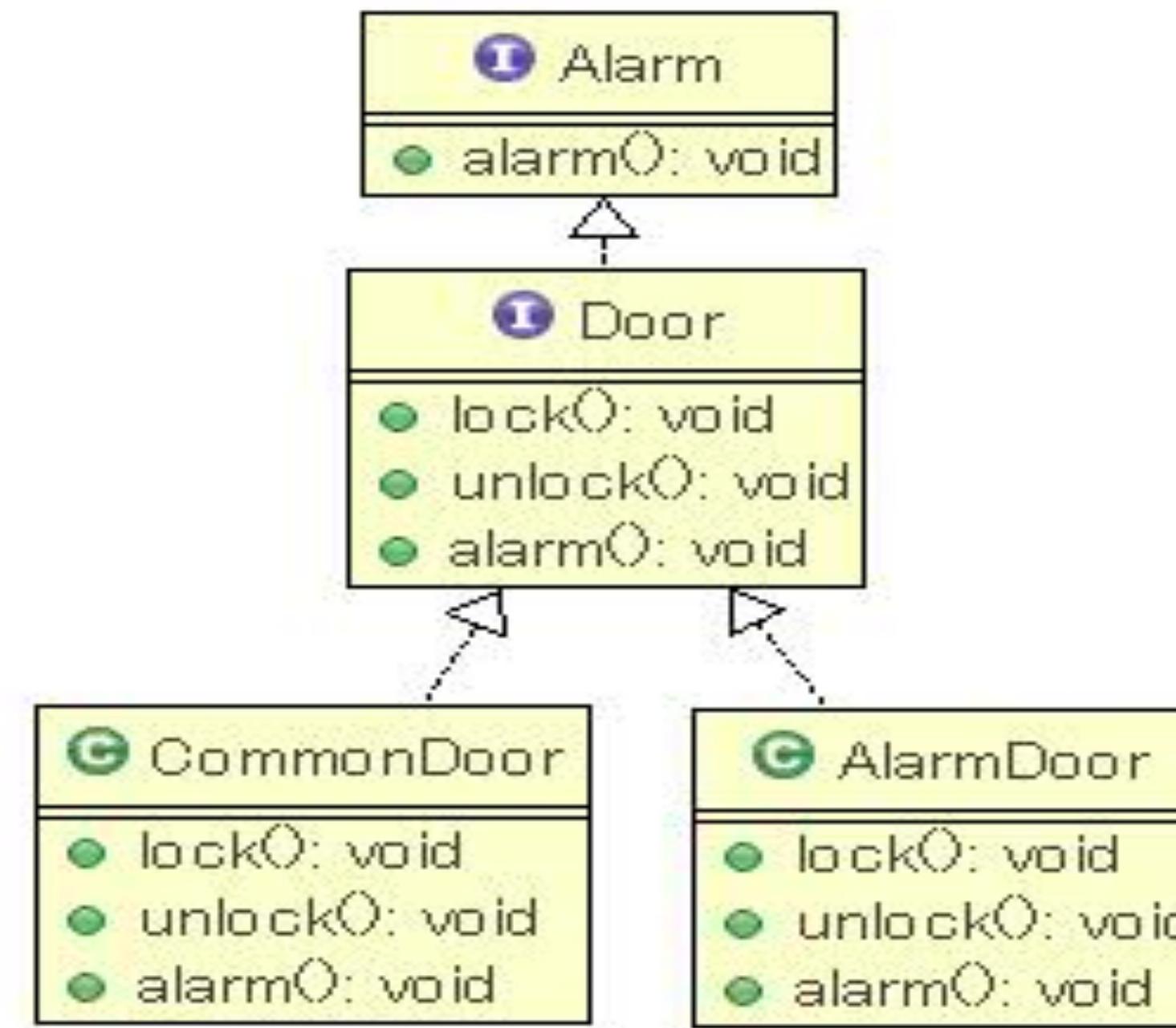
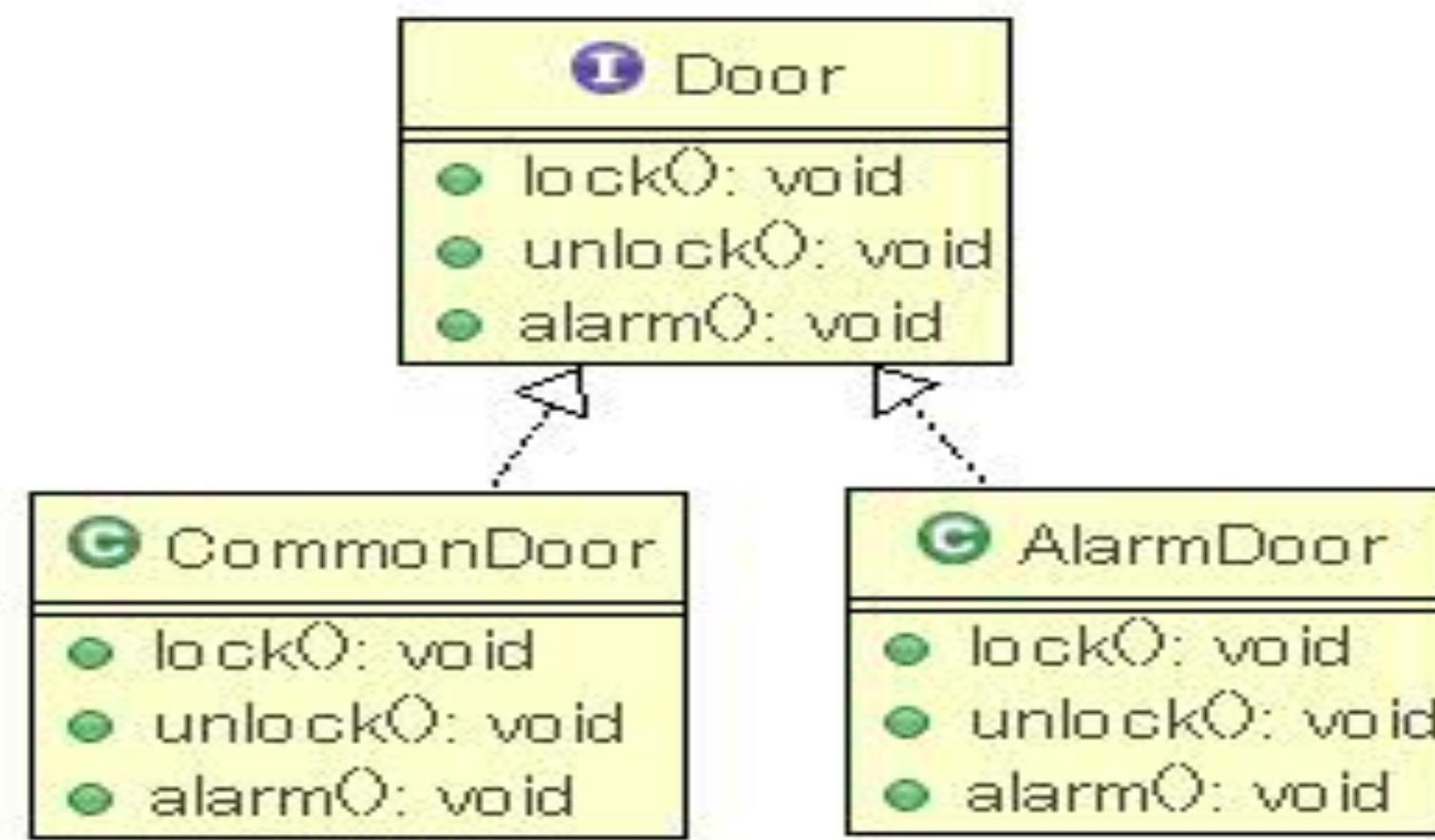
- LSP is about Semantics and Replacement
  - Understand before you design
    - The meaning and purpose of every method and class must be clearly documented
    - Lack of user understanding will induce de facto violations of LSP
  - Replaceability is crucial
    - Whenever any class is referenced by any code in any system,
    - any future or existing subclasses of that class must be 100% replaceable

# LSP Summary

**“When redefining a method in a derivate class, you may only replace its precondition by a weaker one, and its postcondition by a stronger one”**

— B. Meyer, 1988

- Design by Contract
  - Advertised Behavior of an object:
    - advertised Requirements (Preconditions)
    - advertised Promises (Postconditions)
  - Derived class services should require no more and promise no less



# 课堂练习

# Principle 8 : Favor Composition Over Inheritance

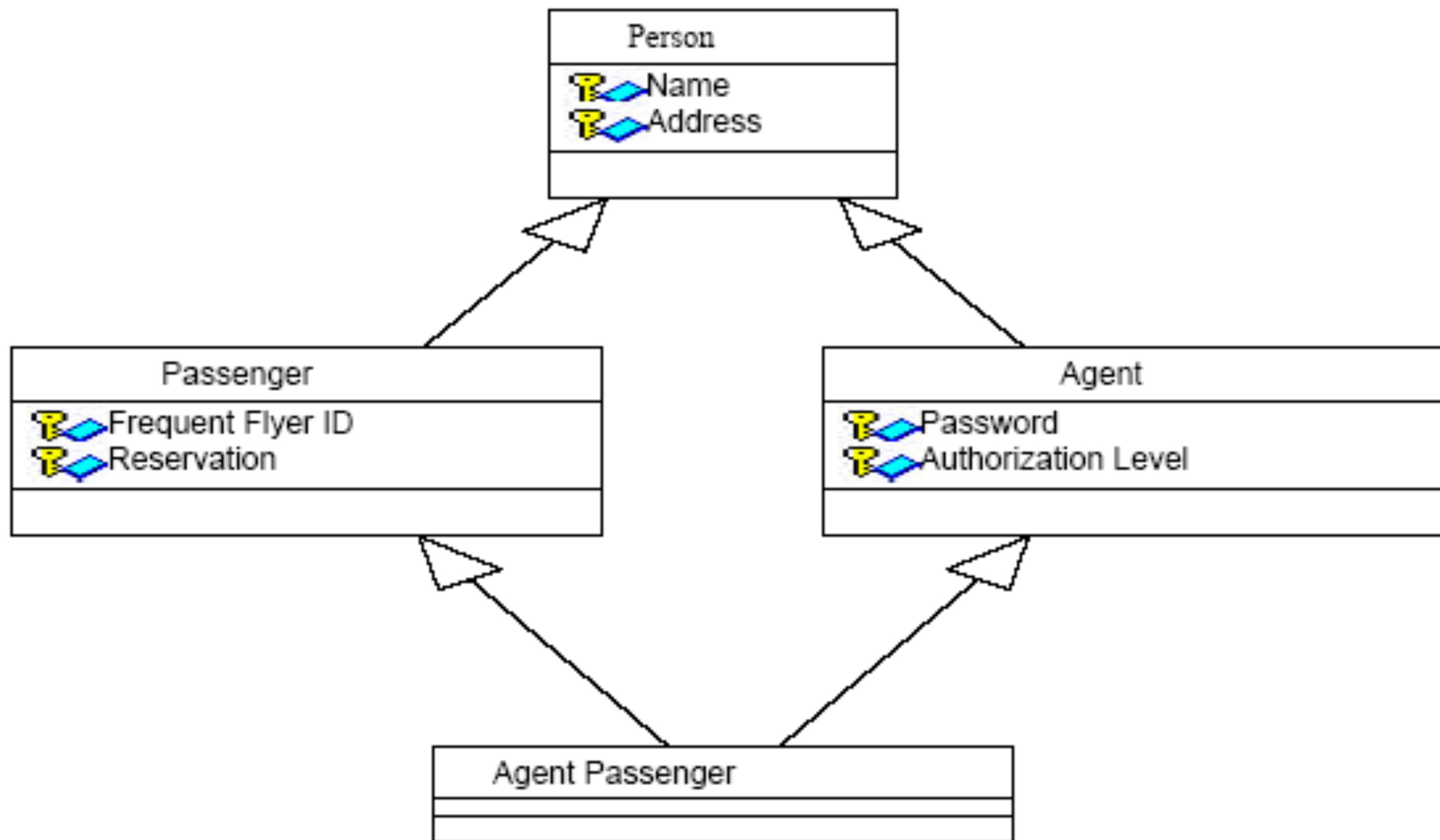
继承是不得已的手段，能不用尽量不用

- Favor Composition Over Inheritance
- Use inherit for polymorphism
- Use delegate not inherit to reuse code!

# Coad's Rules of Using Inheritance

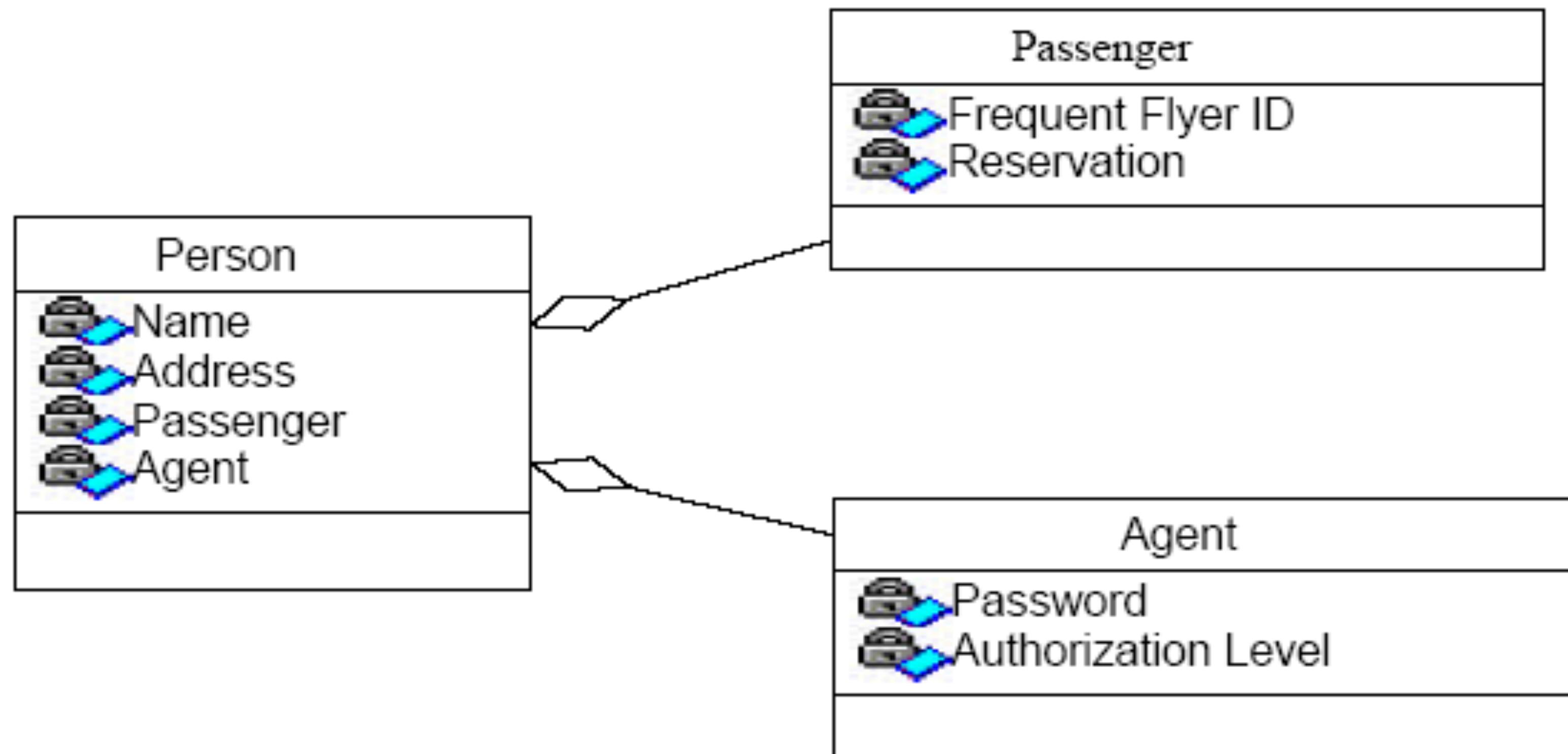
- Use inheritance only when all of the following criteria are satisfied:
  - A subclass expresses "is a special kind of" and not "is a role played by a"
  - An instance of a subclass never needs to become an object of another class
  - A subclass extends, rather than overrides or nullifies, the responsibilities of its superclass
  - A subclass does not extend the capabilities of what is merely an utility class

# Inheritance/Composition Example 1

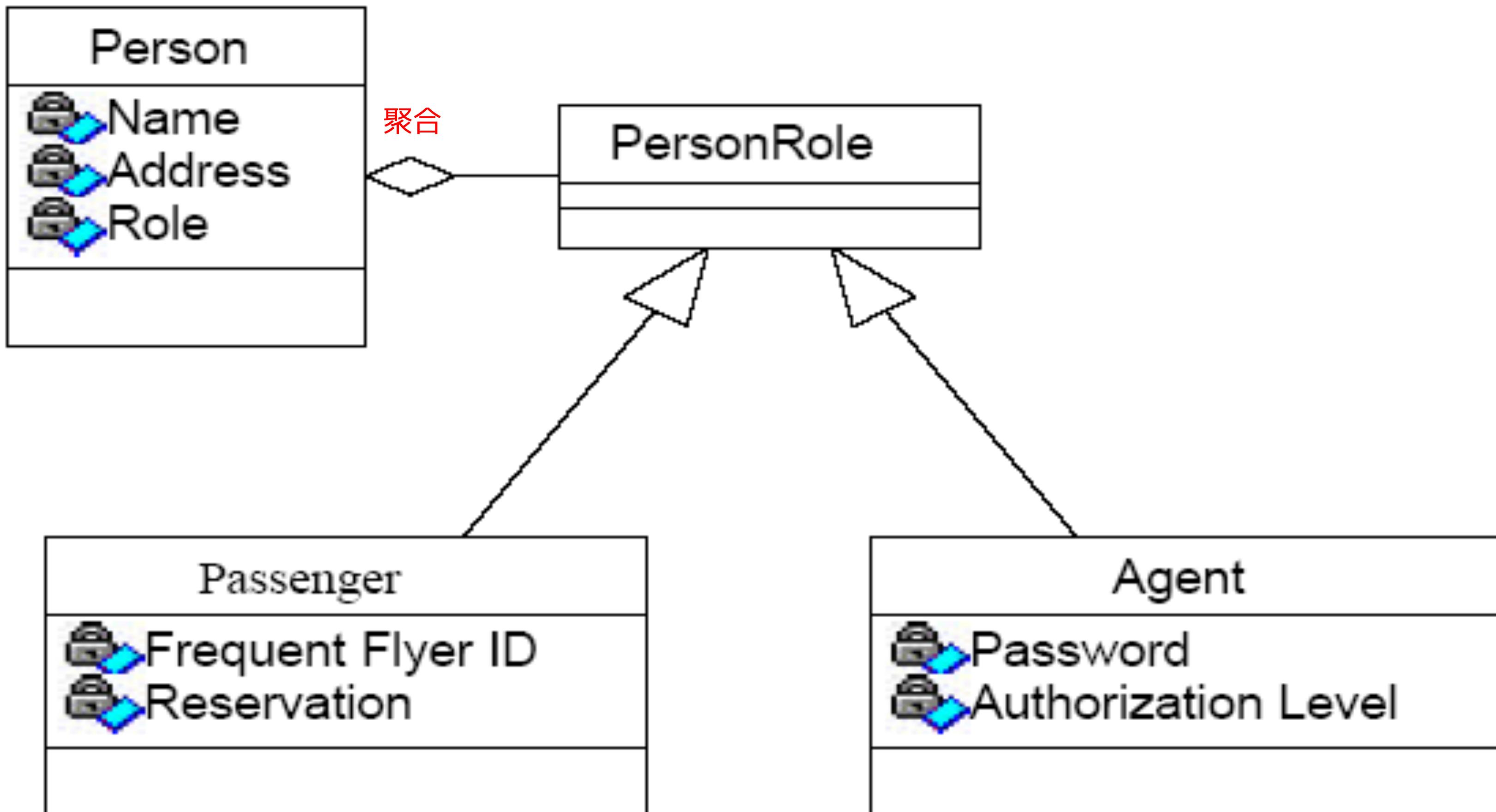


# Inheritance/Composition Example 1 (Continued)

*Composition to the rescue!*



# Inheritance/Composition Example 2



- class Object {
  - public: virtual void update() {};
    - virtual void draw() {};
    - virtual void collide(Object objects[]) {};
  - };
- class Solid : public Object {
  - public: virtual void collide(Object objects[]) {};
    - /\* check and react to collisions with objects \*/;
  - };
- class Movable : public Object {
  - public: virtual void update() {};
    - /\* update position \*/;
  - };
- class Visible : public Object {
  - public: virtual void draw() {};
    - /\* draw model at position of this object \*/;
  - private: Model\* model;
  - };

# Outline

- 面向对象中的模块与耦合
- 访问耦合
- 继承耦合
- 内聚
- 耦合和内聚的度量

衡量标准	内聚低的例子	内聚高的例子
方法和属性是否一致	<p>小计每一购物项金额的方法放在 Sales 类中</p> <pre>class Sales{     HashMap&lt;Integer, SalesLineItem&gt;     map;     getSubtotal(int CommodityID) {         1) 根据 CommodityID 找到         Commodity 的价格         2) 根据 CommodityID 找到         SalesLineItem, 再找到商品购买         的数量         3) 计算小计     } }</pre>	<p>小计每一购物项金额的方法放在 SalesLineItem 中。计算总额的类在 Sales 类中。</p> <pre>class Sales{     HashMap&lt;Integer, SalesLineItem&gt;     map;      getTotal() {         遍历 map 中的 item         total = item.getSubtotal();     } } class SalesLineItem{     Commodity commodity;     Int quantity;     getSubtotal(); }</pre>

衡量标准	内聚低的例子	内聚高的例子
属性之间是否体现一个职责	<p>学号、姓名、成绩、课程编号、课程名在一个类里面</p> <pre data-bbox="1182 656 2104 1270">class SCORE{     int studentID;     String name;     int score;     int courseID;     String courseName; }</pre>	<p>学号、姓名在学生类中；课程编号、课程名在课程类中；学生、课程、成绩在成绩类中</p> <pre data-bbox="2769 656 3691 1808">class Student{     int studentID;     String name; } class Course{     int courseID;     String courseName; } class SCORE{     Student student;     Course course;     int score; }</pre>

属性之间可否抽象

生产年份、生产月份、生产日期、进货年份、进货月份、进货日期在一个类里面

```
class Product{  
    int yearOfProduction;  
    int monthOfProduction;  
    int dayOfProduction;  
    int yearOfImport;  
    int monthOfImport;  
    int dayOfImport;  
}
```

抽象出日期类包含年、月、日三个属性。类里面只有日期类的生产日期和进货日期两个变量

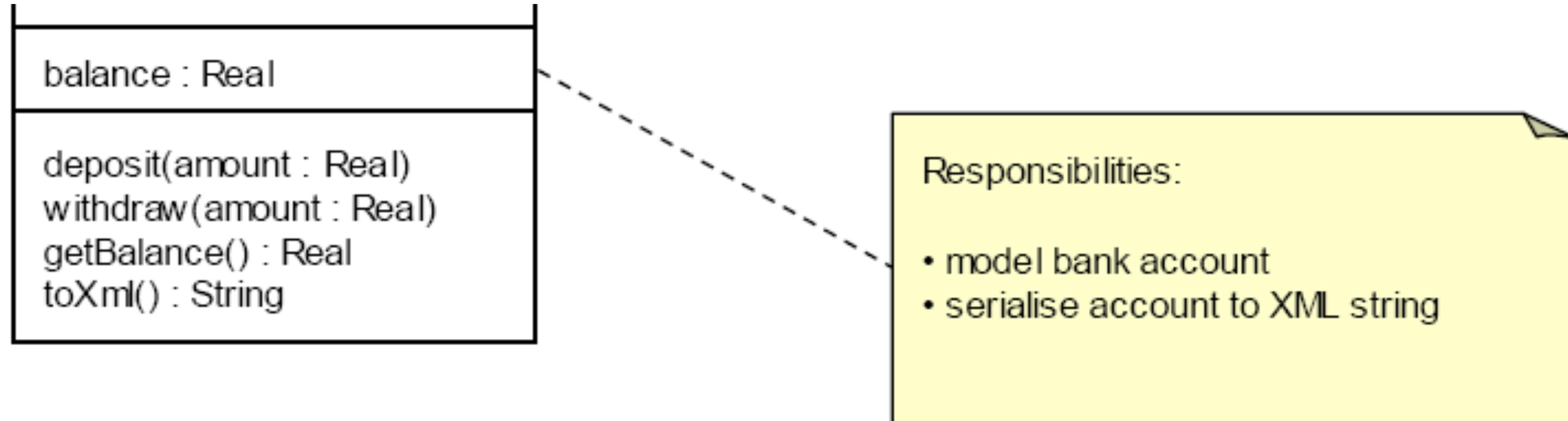
```
class Date{  
    int year;  
    int month;  
    int day;  
}  
class Product{  
    Date productionDate;  
    Date importDate;  
}
```

# Cohesion of methods

- Methods of a Class are Common coupling
- All methods serve One Responsibility
  - Informational Cohesion
  - Relative functions (functional Cohesion)
  - Principle 9: Single Responsibility Principle

# Single Responsibility Principle (SRP)

- “A class should have only one reason to change”
  - – Robert Martin
- Related to and derived from cohesion, i.e. that elements in a module should be closely related in their function
- Responsibility of a class to perform a certain function also a reason for the class to change

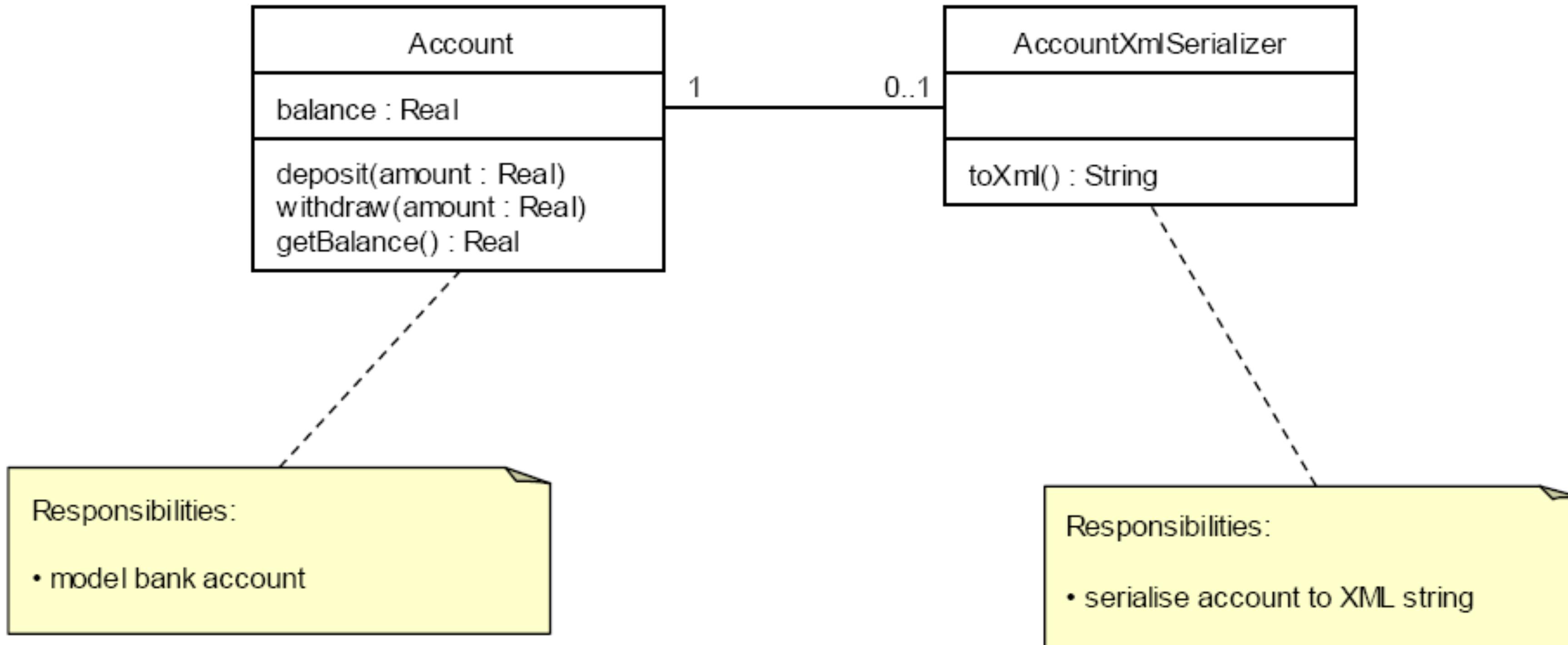


Two reasons why this class might need to change

- changes to domain logic

# SRP Example

## 问题案例

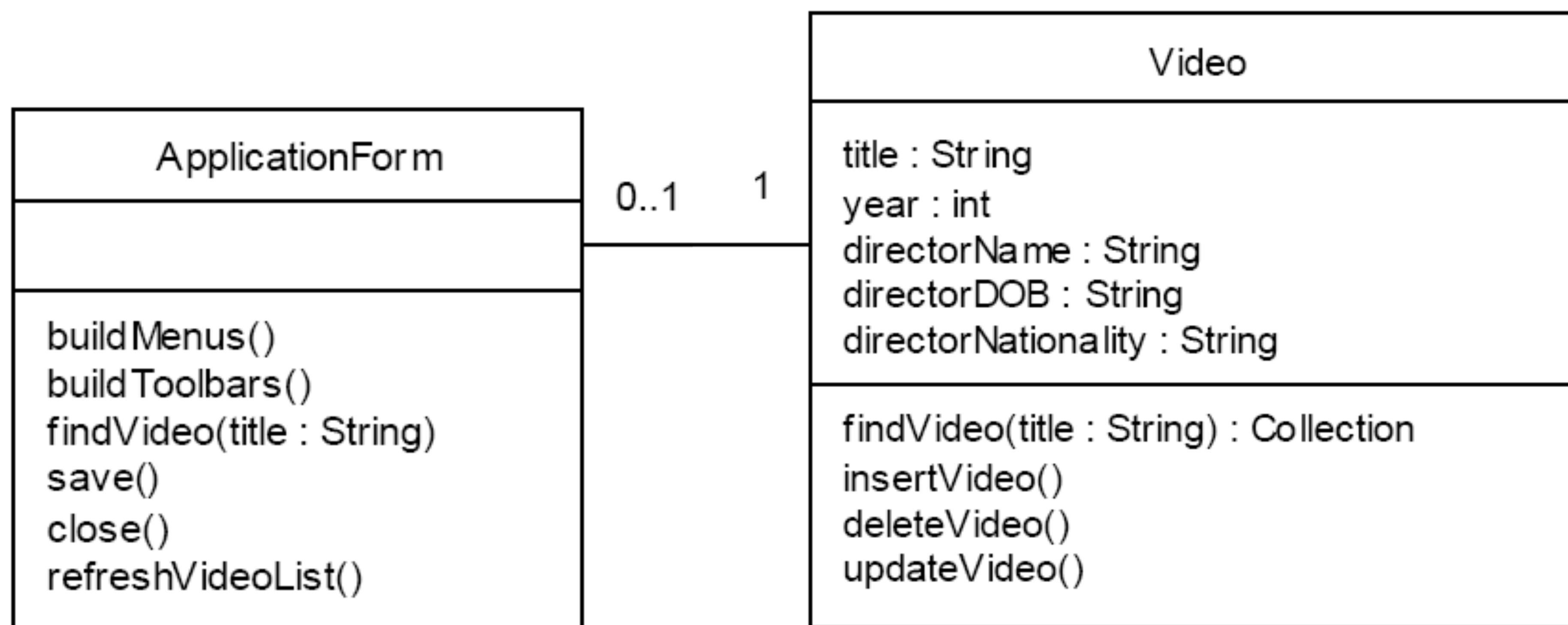


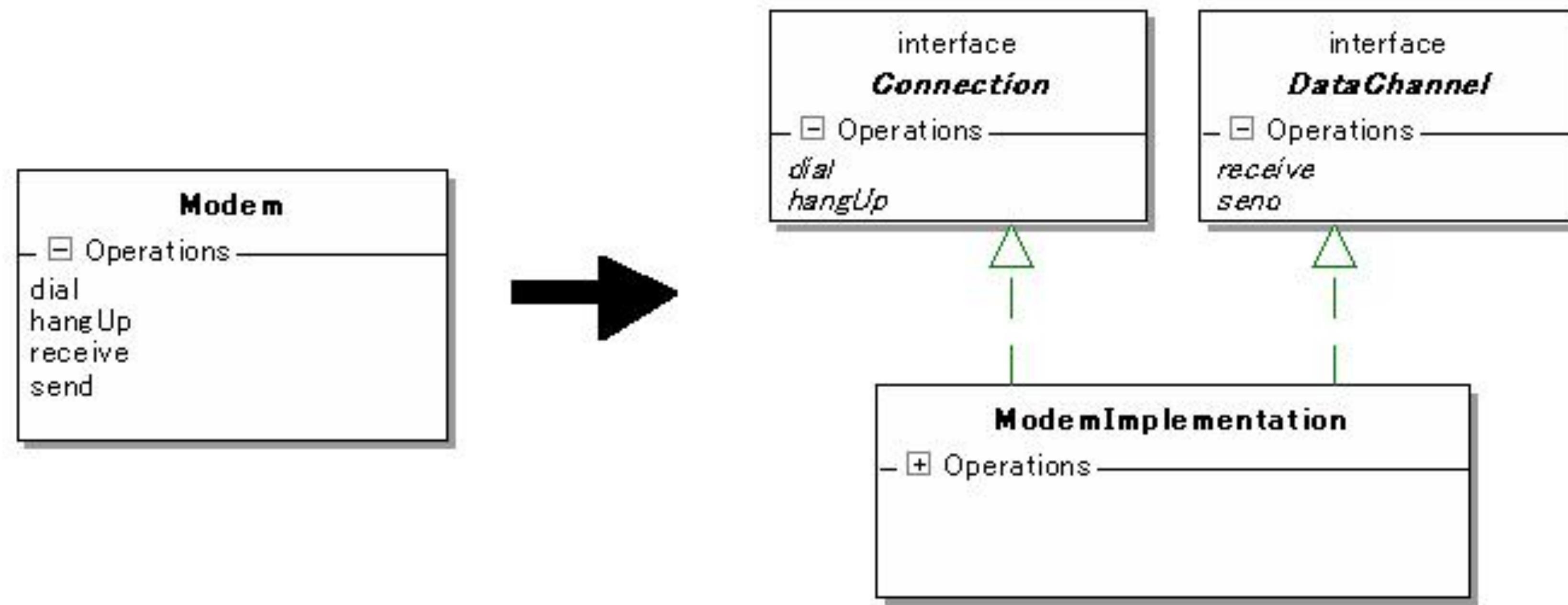
# SRP Example

## 结局方案

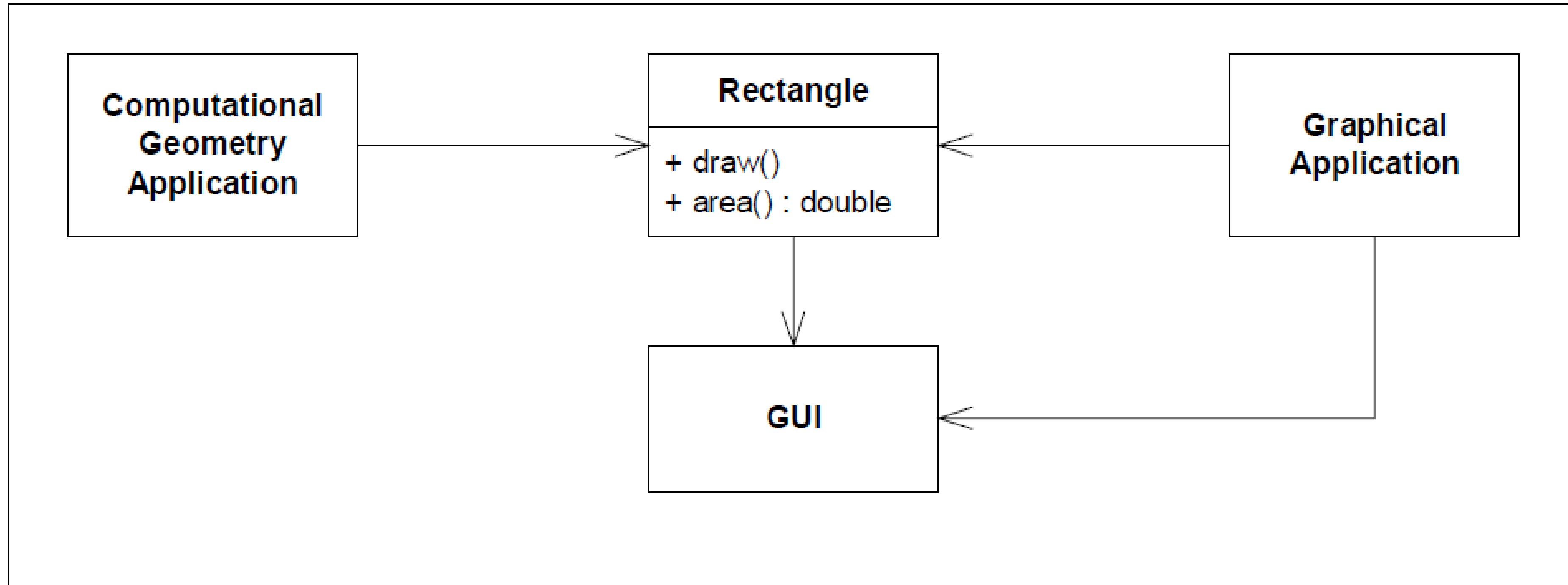
# SRP Summary

- Class should have only one reason to change
  - Cohesion of its functions/responsibilities
  - Several responsibilities
    - mean several reasons for changes → more frequent changes
  - Sounds simple enough
    - Not so easy in real life
    - Tradeoffs with complexity, repetition, opacity





# 课堂练习



**Figure 9-1**  
More than one responsibility

# 课堂练习

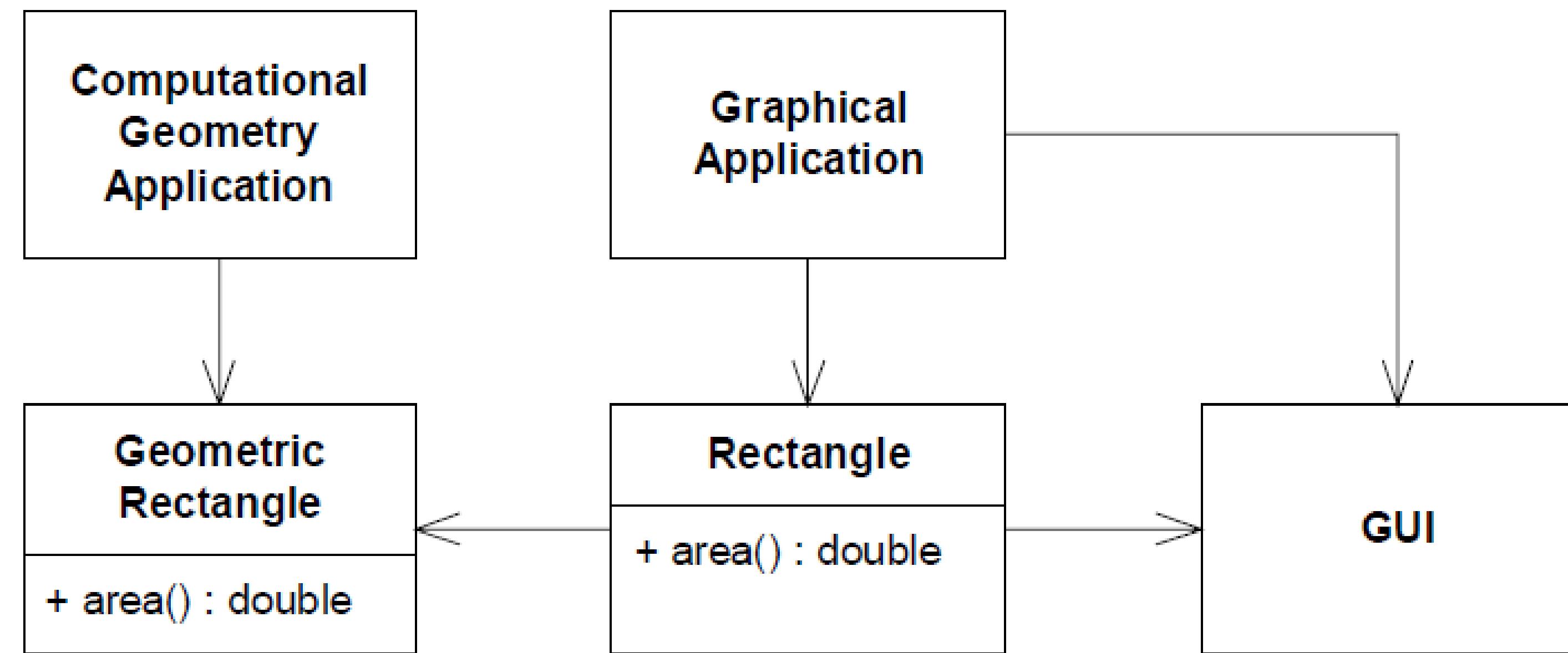


Figure 9-2  
Separated Responsibilities

# 解决方案

# Outline

- 面向对象中的模块与耦合
- 访问耦合
- 继承耦合
- 内聚
- 耦合和内聚的度量

# Coupling Metrics between classes

- Coupling between object classes (CBO)
- A count of the number of other classes:
  - which access a method or variable in this class, or
  - contain a method or variable accessed by this class
  - Not including Inheritance
- Want to keep this low

# Coupling Metrics between classes

- Data abstraction coupling (DAC)
- The number of attribute having an ADT type dependent on the definitions of other classes
- Want to keep this low

# Coupling Metrics between classes

- Ce and Ca (efferent and afferent coupling)
  - Ca:
    - The number of classes outside this category that depend upon classes within this category.
  - Ce:
    - The number of classes inside this category that depend upon classes outside this category
- Want to keep these low

# Coupling Metrics between classes

- Depth of the Inheritance tree (DIT)
  - the maximum length from the node to the root of the tree
  - as DIT grows, it becomes difficult to predict behavior of a class because of the high degree of inheritance
  - Positively, large DIT values imply that many methods may be reused

# Coupling Metrics between classes

- Number of children (NOC)
  - count of the subclasses immediately subordinate to a class
  - as NOC grows, reuse increases
  - as NOC grows, abstraction can become diluted
  - increase in NOC means the amount of testing will increase

# Measure class cohesion

- Lack of cohesion in methods (LCOM)

*“Consider a Class  $C_1$  with  $n$  methods  $M_1, M_2, \dots, M_n$ . Let  $\{I_j\}$  = set of instance variables used by Method  $M_j$ .*

*There are  $n$  such sets  $\{I_1\}, \dots, \{I_n\}$ .*

*Let  $P = \{(I_i, I_j) \mid I_i \cap I_j = \emptyset\}$  and  $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset\}$ .*

*If all  $n$  sets  $\{I_1\}, \dots, \{I_n\}$  are  $\emptyset$  then let  $P = \emptyset$ .*

$$\begin{aligned} LCOM &= |P| - |Q|. \text{ if } |P| > |Q| & P \text{越大, 内聚性越坏。} \\ &= 0 \text{ otherwise.} \end{aligned}$$

P越大, 内聚性越坏。  
Q越大, 内聚性越好。

LCOM越大, 内聚性越差

- Want to keep this low
- Many other versions of LCOM have been defined

# Measure class cohesion

- If  $LCOM \geq 1$ , then the class should be separated

Let  $X$  denote a class,  $I_X$  the set of its instance variables of  $X$ , and  $M_X$  the set of its methods. Consider a simple, undirected graph  $G_X(V, E)$  with

$$V = M_X \text{ and } E = \{ \langle m, n \rangle \in V \times V \mid \exists i \in I_X: (m \text{ accesses } i) \wedge (n \text{ accesses } i) \}.$$

$LCOM(X)$  is then defined as the number of connected components of  $G_X$  ( $1 \leq LCOM(X) \leq |M_X|$ ).

# Summary

- Principles from Modularization
  - 1: 《Global Variables Consider Harmful》
  - 2: 《To be Explicit》
  - 3: 《Do not Repeat》
  - 4: 《Programming to Interface(Design by Contract Design by Contract)》

# Summary

- 5: 《The Law of Demeter》
- 6: 《Interface Segregation Principle(ISP)》
- 7: 《Liskov Substitution Principle (LSP)》
- 8: 《Favor Composition Over Inheritance》
- 9: 《Single Responsibility Principle》