南京大学
NANJING UNIVERSITY

《软件工程与计算II》

Ch14　面向对象的模块化

Module

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

What's the difference between Structural methods and OO methods

- 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
  - Interaction coupling
  - Component coupling
  - Inheritance coupling

Main Contents

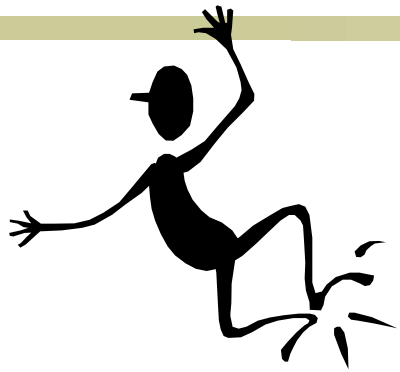
- Coupling of OO
  - Interaction Coupling
  - Component Coupling
  - Inherit Coupling
  - Coupling Metric of OO
- Cohesion of OO

Single interaction coupling

- Method Invoke or Attribute Access between two class
- Most similar to the classical definition of coupling between modules

- Not including inheritance
- Parameter and Attribute not including Class Type

Strength of Single interaction coupling



	Interface Complexity	Type of Connection	Type of Communication
Low	Simple, Obvious	To module, by name	Data
COUPLING			Control
High	Complicated, Obscure	To Internal Elements	Hybrid

Principles of interaction coupling

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

Main Contents

- Coupling of OO
  - Interaction Coupling
  - Component Coupling
  - Inherit Coupling
  - Coupling Metric of OO
- Cohesion of OO

Component coupling

- Abstraction
  - Define one place, using many place
  - Class-Level , Object-Level
- Component coupling
  - The component relationship between classes is defined by the use of a class as domain of some instance variable of another class
  - Not including Inheritance
  - Usually Class-Level needed
    - Attribute Type: how to explicit

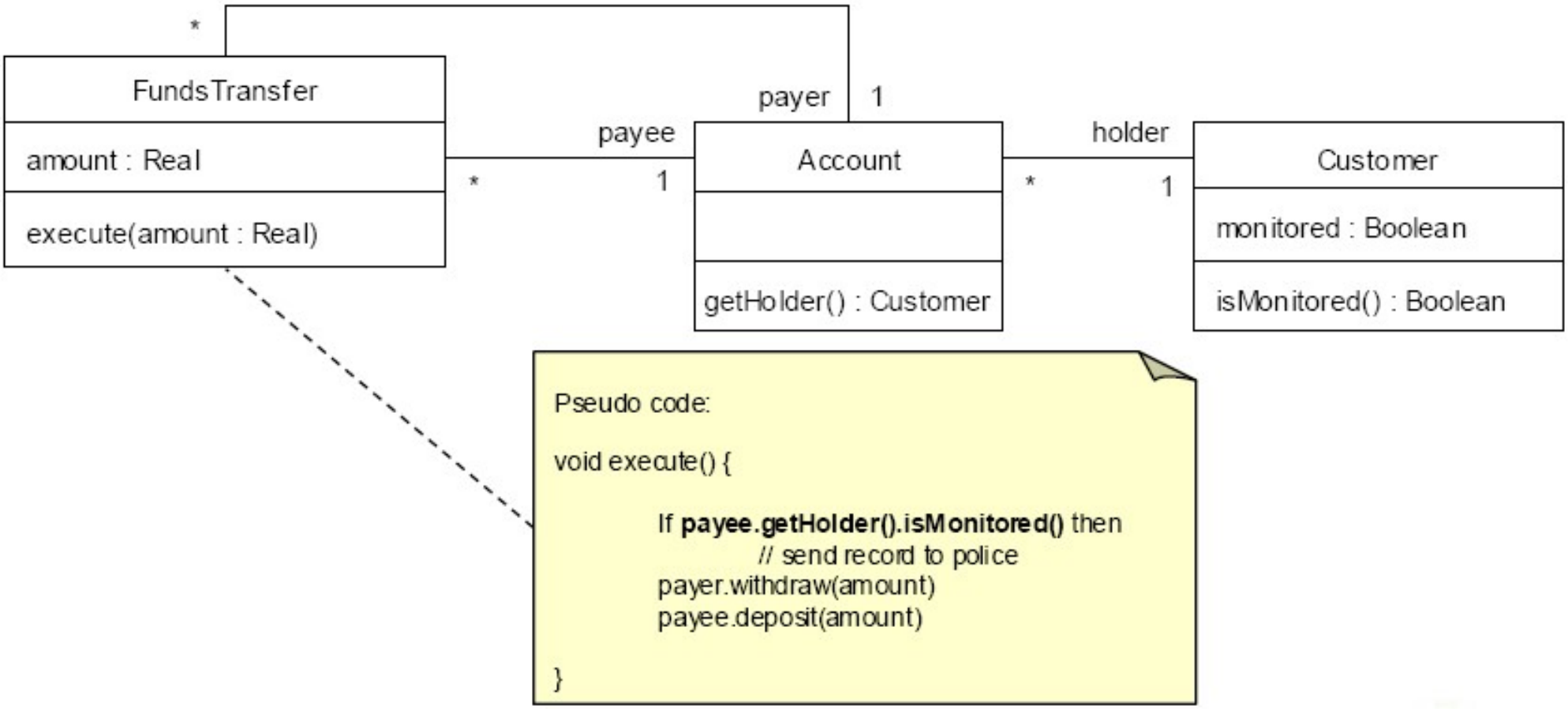
Four Kinds of Component coupling

- Whole variable: Aggregation
- Parameter: Method Parameter
- Creator: Creator in some method’s local
- Hidden: Given by another object

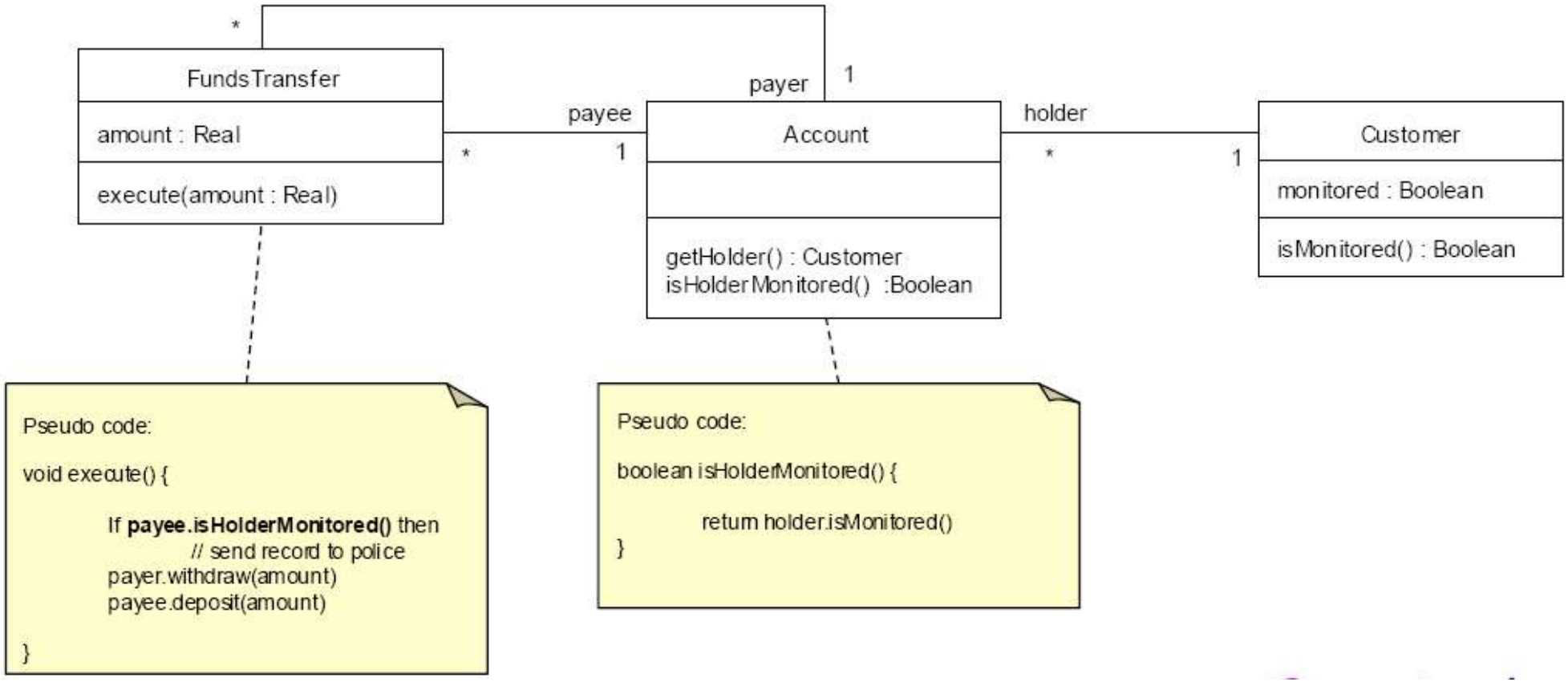
Hidden Component coupling

- The worst coupling: Implicit
- The coupling between two classes C and C' is rated hidden if C' shows up neither in the specification nor in the implementation of C although an object of C' is used in the implementation of a method of C
- A similar problem is encountered if the return value of a method invocation is immediately used as input parameter in another method invocation
- Disallow the use of cascading messages
  - To be explicit!

Hidden Component coupling

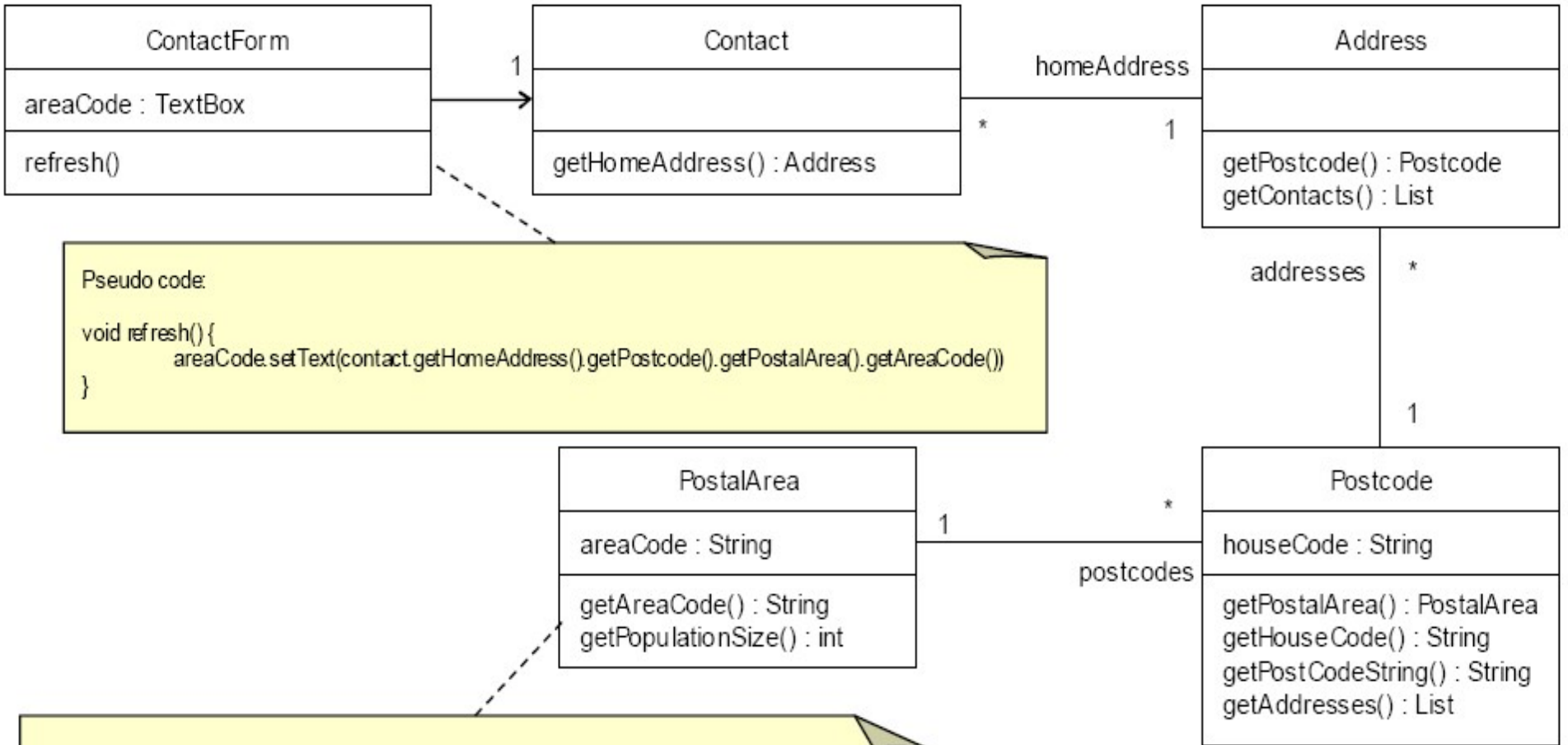


The Law of Demeter Example



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

Scattered Component Coupling

- We rate two classes C and C' as scattered coupled if C' is used as domain in the definition of some local variable or instance variable in the implementation of C yet C' is not included in the specification of C
- Common and acceptable
- Aggregation: Global
- Local instance: Local
- Which is better?
  - Many Connections VS Stronger Single Connection

Specified Component Coupling

- We rate two classes C and C’ as specified coupled if C’ is included in the specification of C whenever it is a component of C
- Get Specified, Design by Contract——Suffered Interface VS Required Interface
  - Suffered Interface: Specified
  - Required Interface: Used
- Scattered Component Coupling can be improved to Specified component coupling with comments

Principles of Component Coupling

- Principle 4: Programming to Interface
  - Programming to Required Interface, not only Suffered Interface
  - Design by Contract
    - Contract of Module/ Class
      - Required mothods / Provided methods
    - Contract of Methods
      - PreCondition, PostCondition, Invariant

Principles of Component Coupling

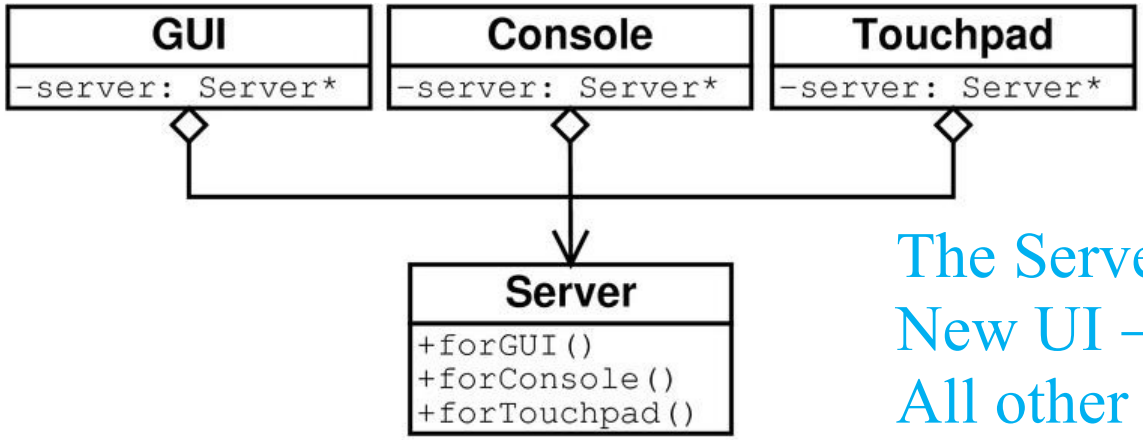
> *Clients should not be forced to depend upon interfaces that they do not use.*
>
> **R. Martin**, 1996

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

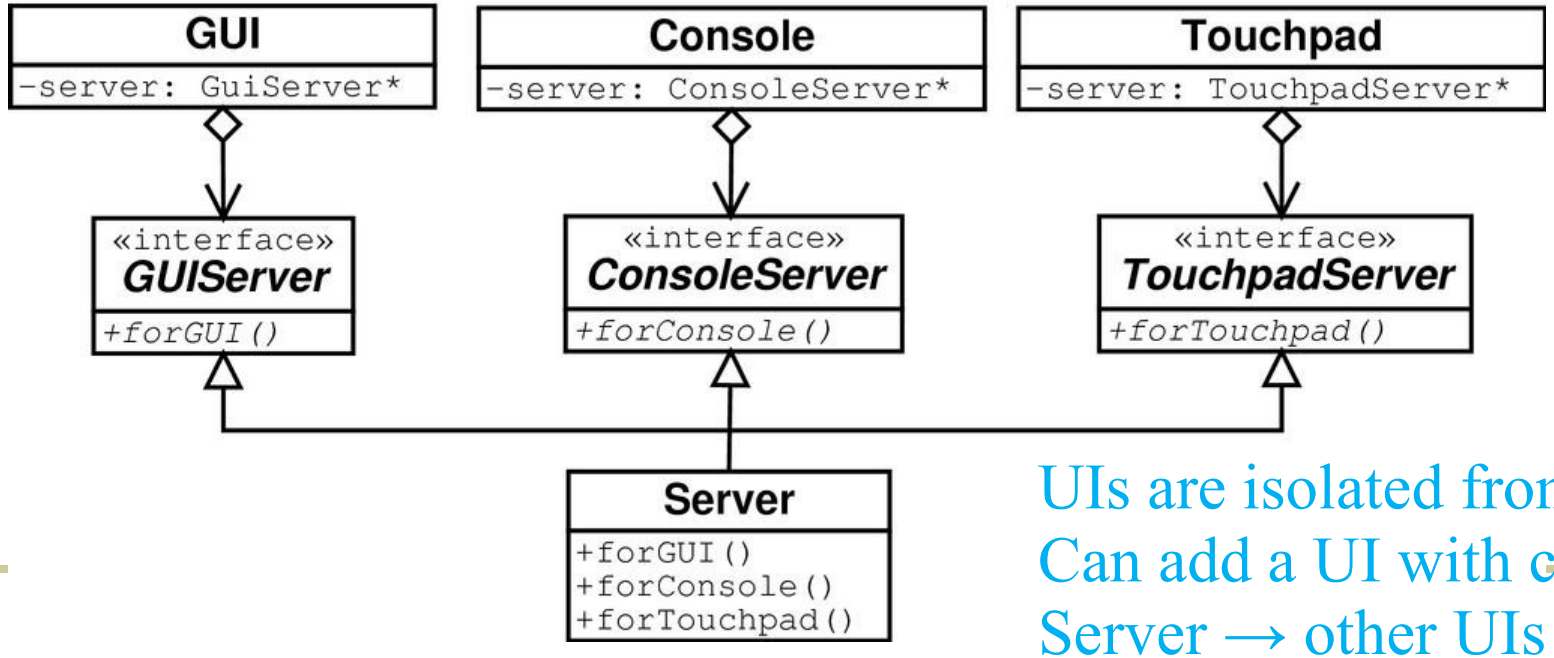
Principles of Component Coupling
—— ISP Explained

- 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

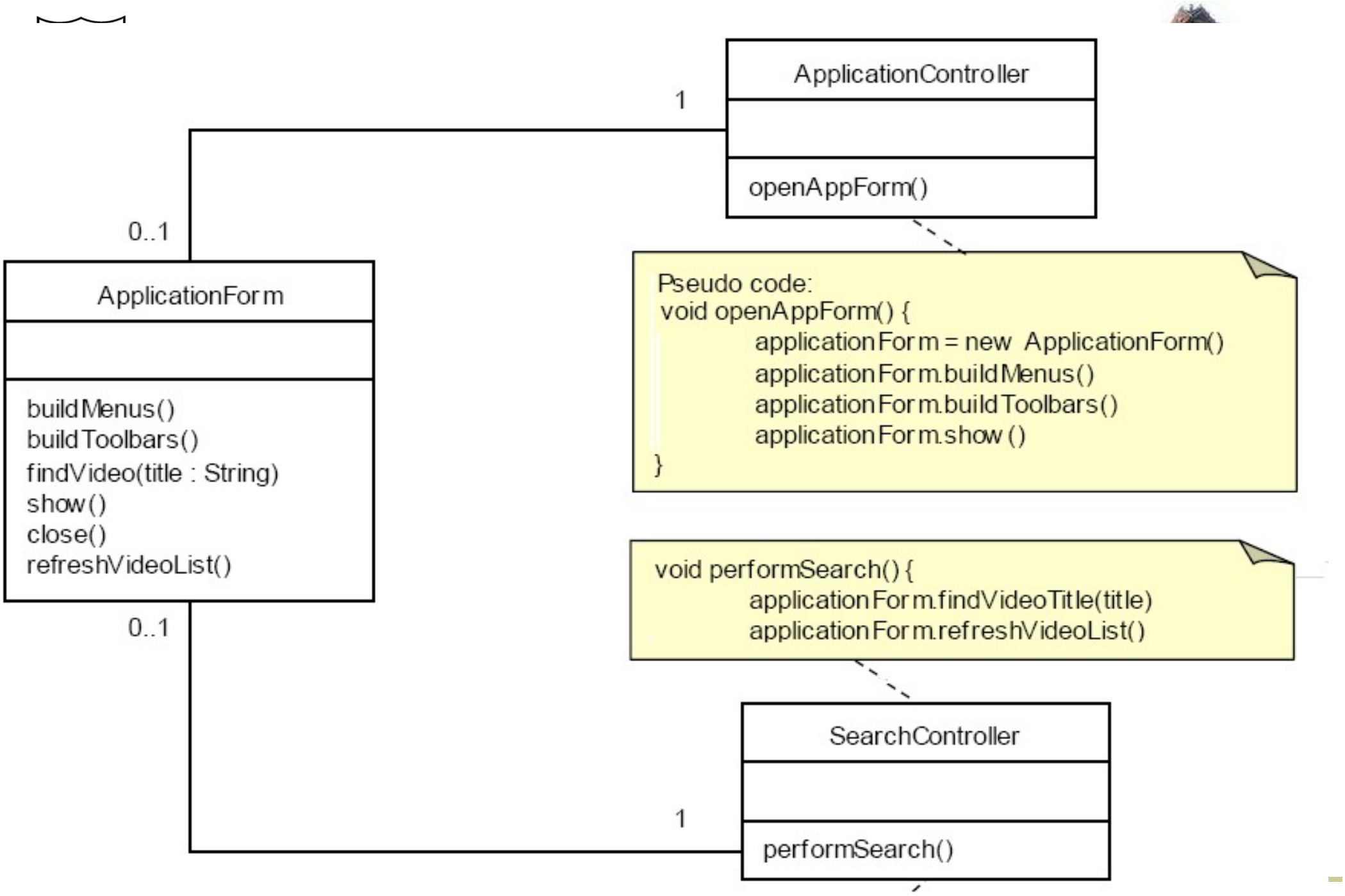
Principles of Component Coupling
—— ISP Example: UIs



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


ApplicationController
openAppForm()
1
0..1
ApplicationForm
buildMenus()
buildToolbars()
findVideo(title : String)
show()
close()
refreshVideoList()
Pseudo code:
void openAppForm() {
applicationForm = new ApplicationForm()
applicationForm.buildMenus()
applicationForm.buildToolbars()
applicationForm.show()
}
void performSearch() {
applicationForm.findVideoTitle(title)
applicationForm.refreshVideoList()
0..1
1
SearchController
performSearch()

Main Contents

- Coupling of OO
  - Interaction Coupling
  - Component Coupling
  - Inherit Coupling
  - Coupling Metric of OO
- Cohesion of OO

Inheritance Coupling

- Parent information is specified for children
- With a parent reference, how many information a client needed when interacting?
  - Modification
  - Refinement
  - Extension

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

Refinement Inheritance Coupling

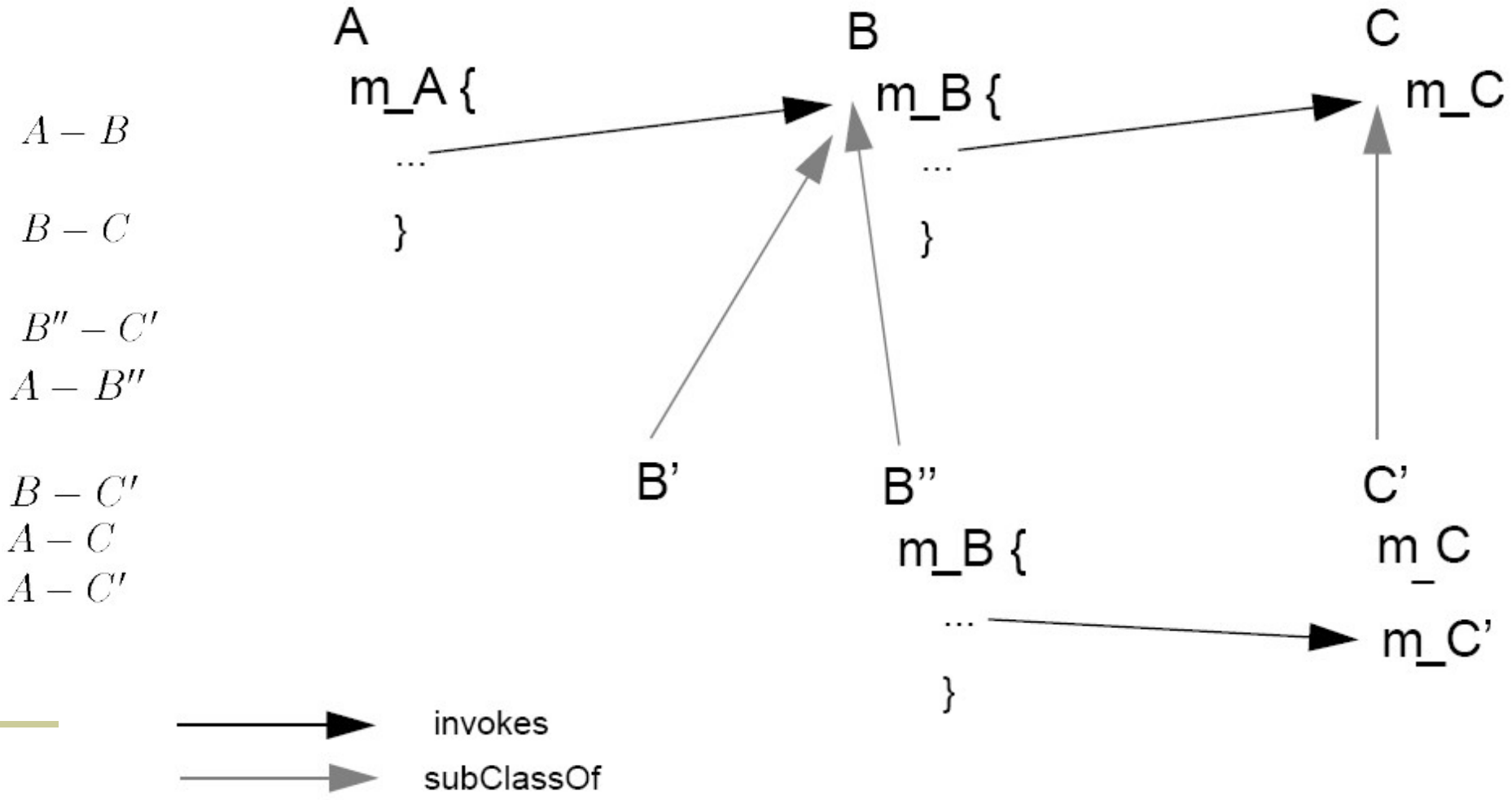
- defining new information the inherited information is only changed due to predefined rules
- If a client using a parent ref, the whole parent and refinement of child are needed
  - 1+connections
- Necessary!

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

How Inheritance reduce coupling ?

Remember: in Refinement and Extension inheritance coupling, the interaction coupling between super-class and subclass is ignored



Principles of Inherit Coupling

Principle 7: Liskov Substitution Principle (LSP)

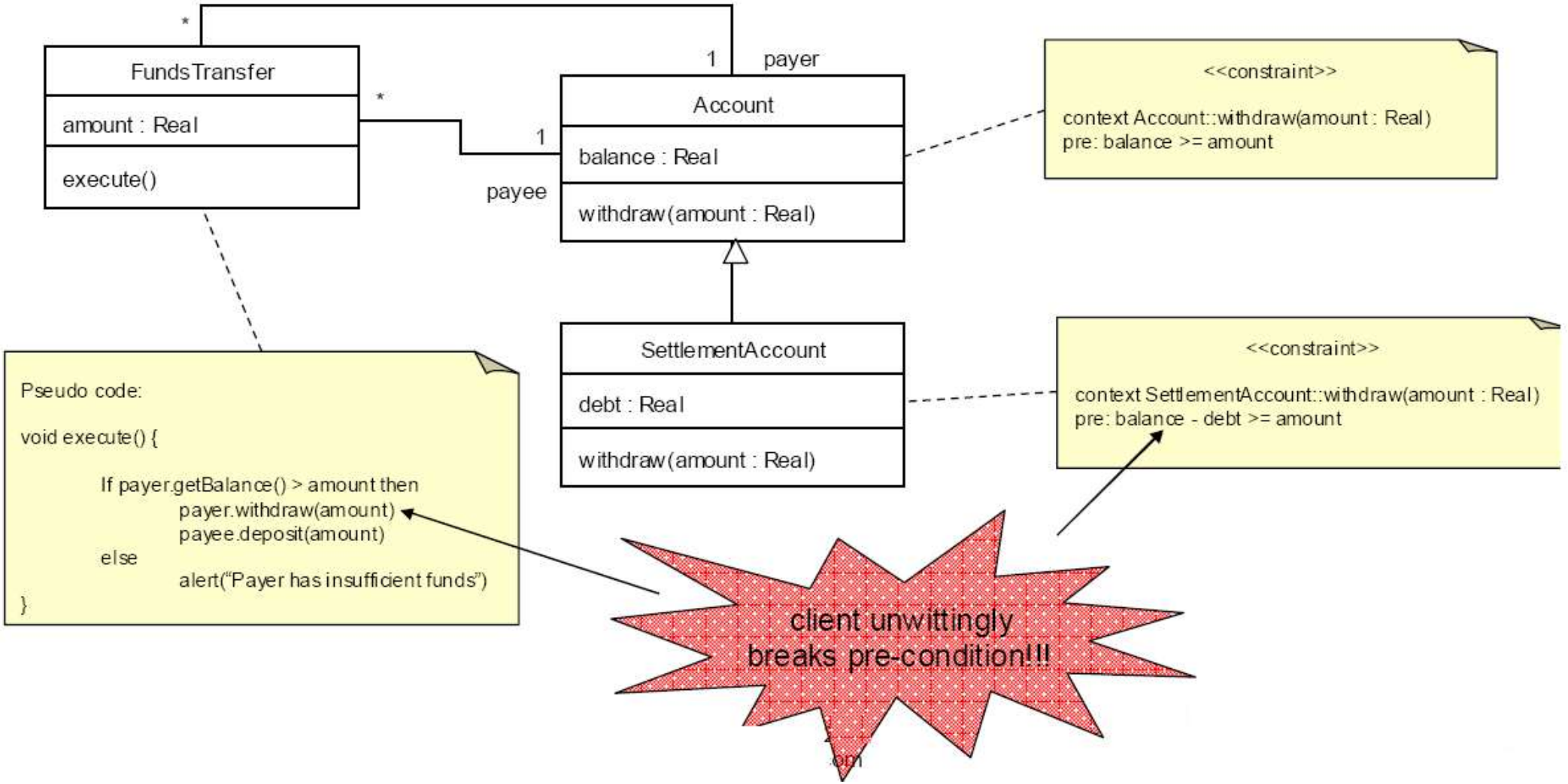
All derived classes must be substitutable for their base class

Barbara Liskov, 1988

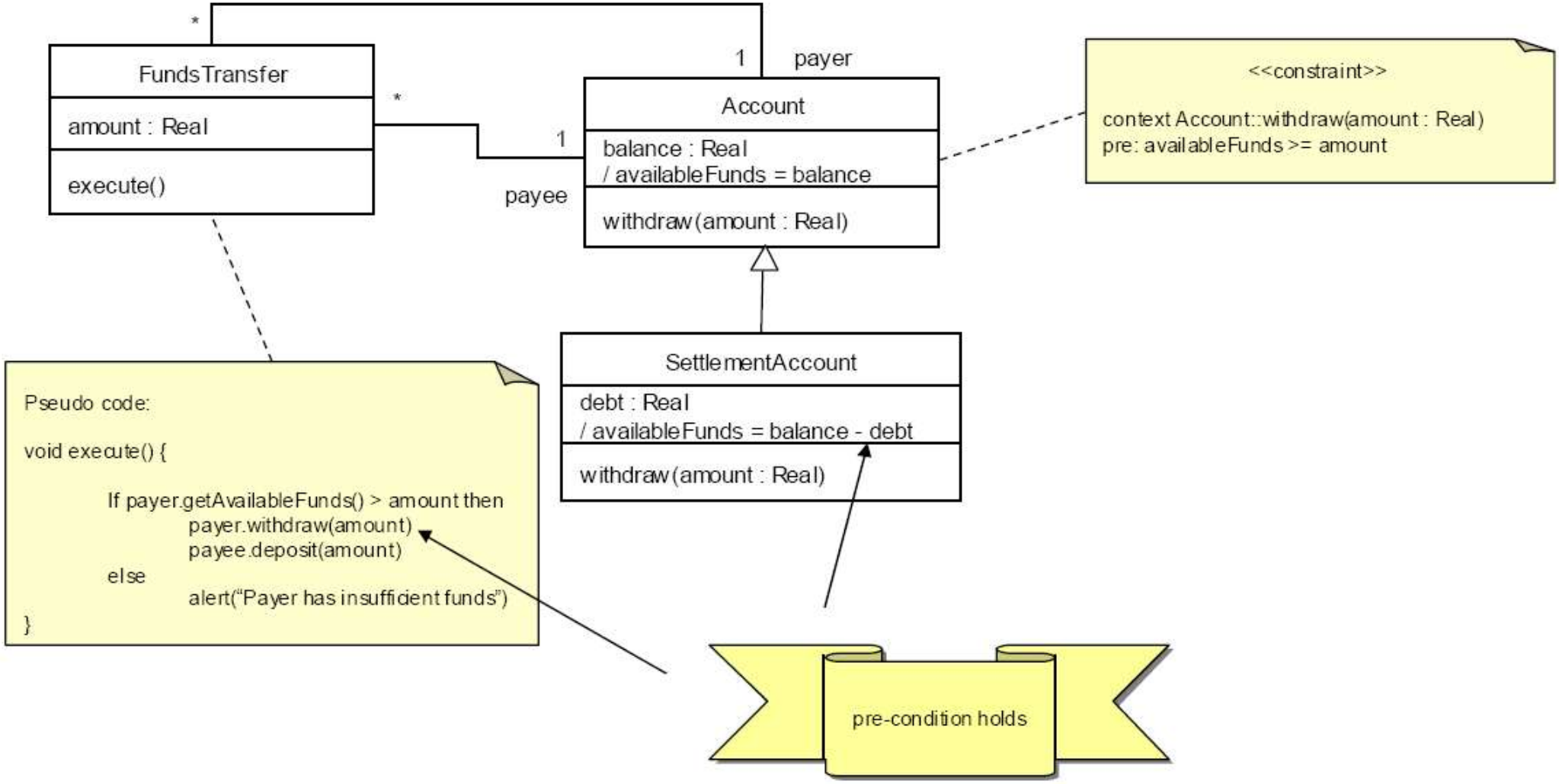
Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

R. Martin, 1996

LSP: example


*
1
payer
FundsTransfer
amount : Real
execute()
*
1
payee
Account
balance : Real
withdraw(amount : Real)
<<constraint>>
context Account::withdraw(amount : Real)
pre: balance >= amount
SettlementAccount
debt : Real
withdraw(amount : Real)
<<constraint>>
context SettlementAccount::withdraw(amount : Real)
pre: balance - debt >= amount
Pseudo code:
void execute() {
If payer.getBalance() > amount then
payer.withdraw(amount)
payee.deposit(amount)
else
alert("Payer has insufficient funds")
}
client unwittingly
breaks pre-condition!!!

LSP: example


*
1
payer
FundsTransfer
amount : Real
execute()
*
1
payee
Account
balance : Real
/ availableFunds = balance
withdraw(amount : Real)
<<constraint>>
context Account::withdraw(amount : Real)
pre: availableFunds >= amount
SettlementAccount
debt : Real
/ availableFunds = balance - debt
withdraw(amount : Real)
Pseudo code:
void execute() {
If payer.getAvailableFunds() > amount then
payer.withdraw(amount)
payee.deposit(amount)
else
alert("Payer has insufficient funds")
}
pre-condition holds

Inheritance *Appears* Simple
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?
```

Inheritance *Appears* Simple

```
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

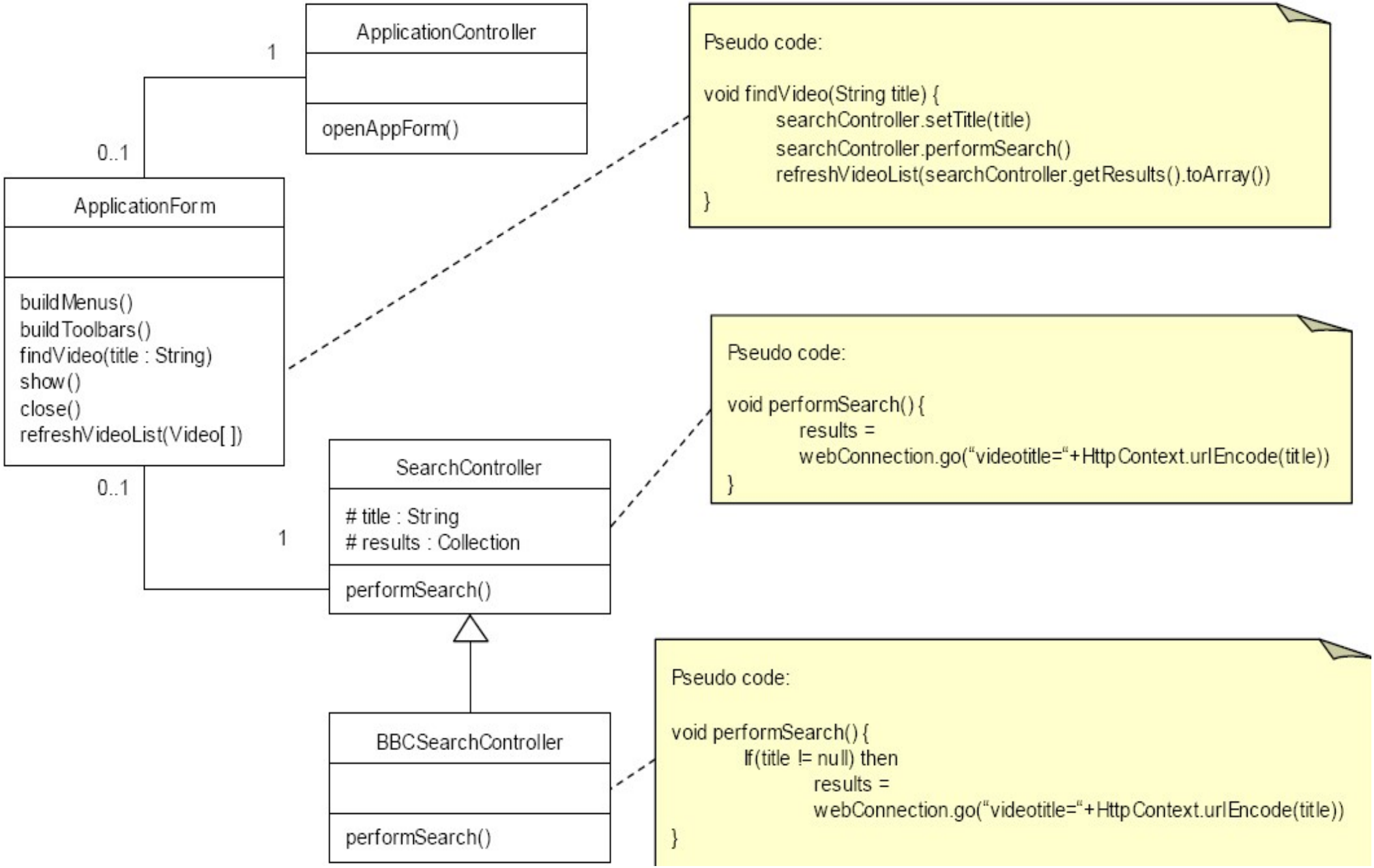
- LSP is about Semantics and Replacement
  - Understand before you design
    - The meaning and purpose of every method and class must be clearly documented
    - Lack of user understanding will induce de facto violations of LSP
  - Replaceability is crucial
    - Whenever any class is referenced by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

LSP Summary

- Design by Contract
  - Advertised Behavior of an object:
    - advertised **Requirements** (Preconditions)
    - advertised **Promises** (Postconditions)

> *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

Derived class services should **require no more** and **promise no less**


ApplicationController
openAppForm()
1
0..1
ApplicationForm
buildMenus()
buildToolbars()
findVideo(title : String)
show()
close()
refreshVideoList(Video[ ])
0..1
1
SearchController
# title : String
# results : Collection
performSearch()
BBCSearchController
performSearch()
Pseudo code:
void findVideo(String title) {
searchController.setTitle(title)
searchController.performSearch()
refreshVideoList(searchController.getResults().toArray())
}
Pseudo code:
void performSearch() {
results =
webConnection.go("videotitle="+HttpContext.urlEncode(title))
}
Pseudo code:
void performSearch() {
If(title != null) then
results =
webConnection.go("videotitle="+HttpContext.urlEncode(title))
}

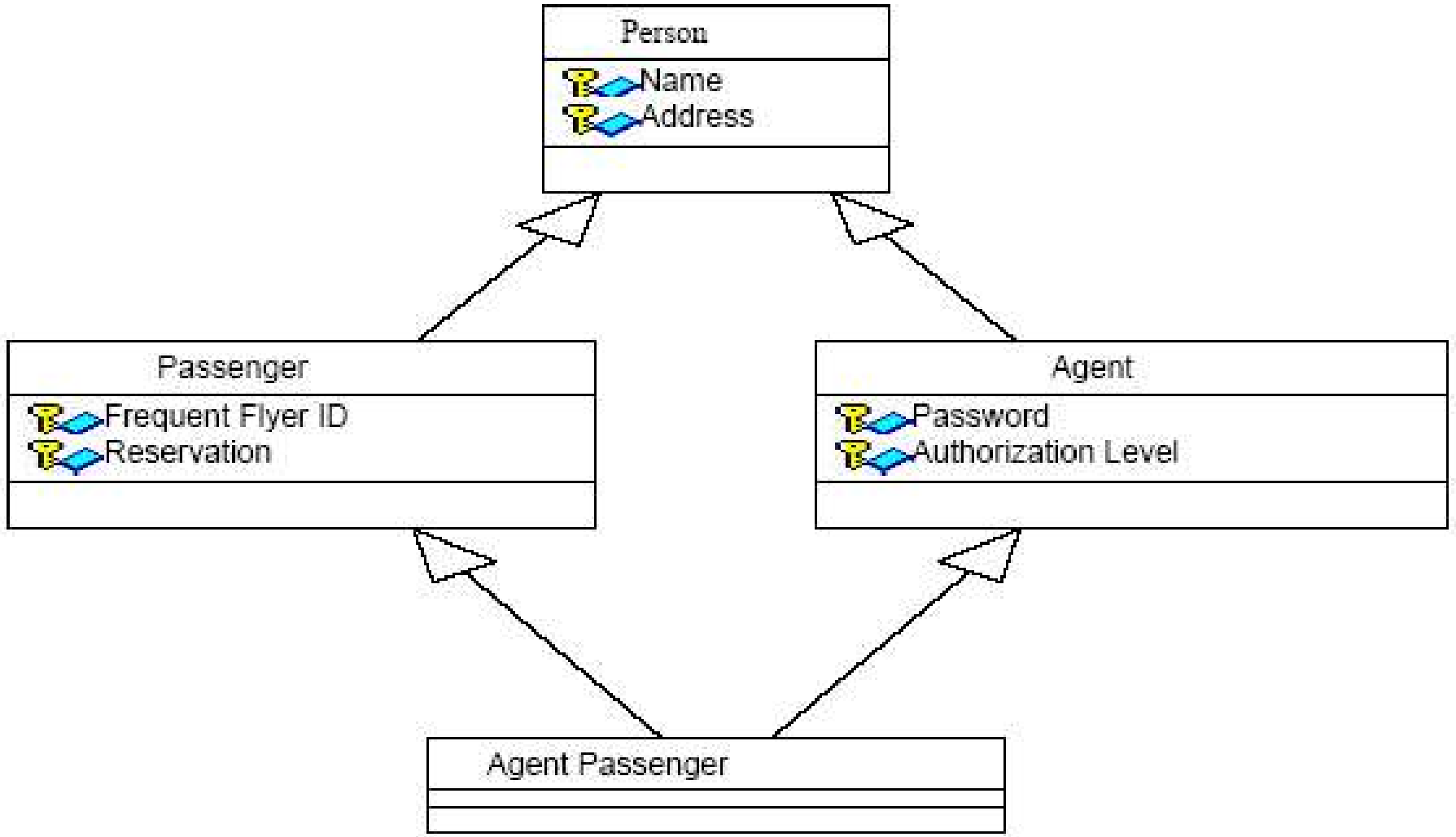
Principle 8 : Favor Composition Over Inheritance

- Use inherit for polymorphism
- Use delegate not inherit to reuse code!

Coad's Rules of Using Inheritance

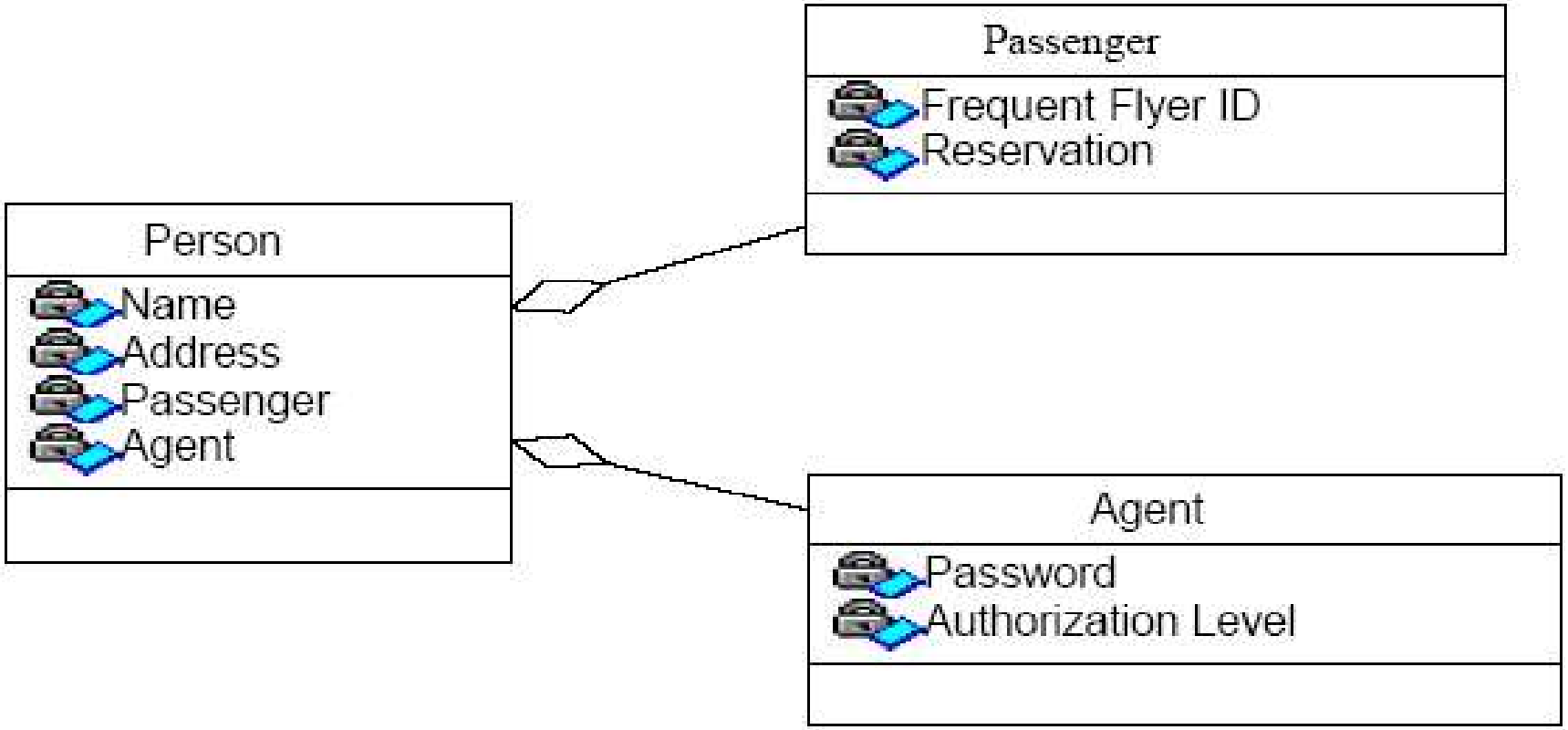
- Use inheritance only when all of the following criteria are satisfied:
  - A subclass expresses "is a special kind of" and not "is a role played by a"
  - An instance of a subclass never needs to become an object of another class
  - A subclass extends, rather than overrides or nullifies, the responsibilities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

Inheritance/Composition Example 1

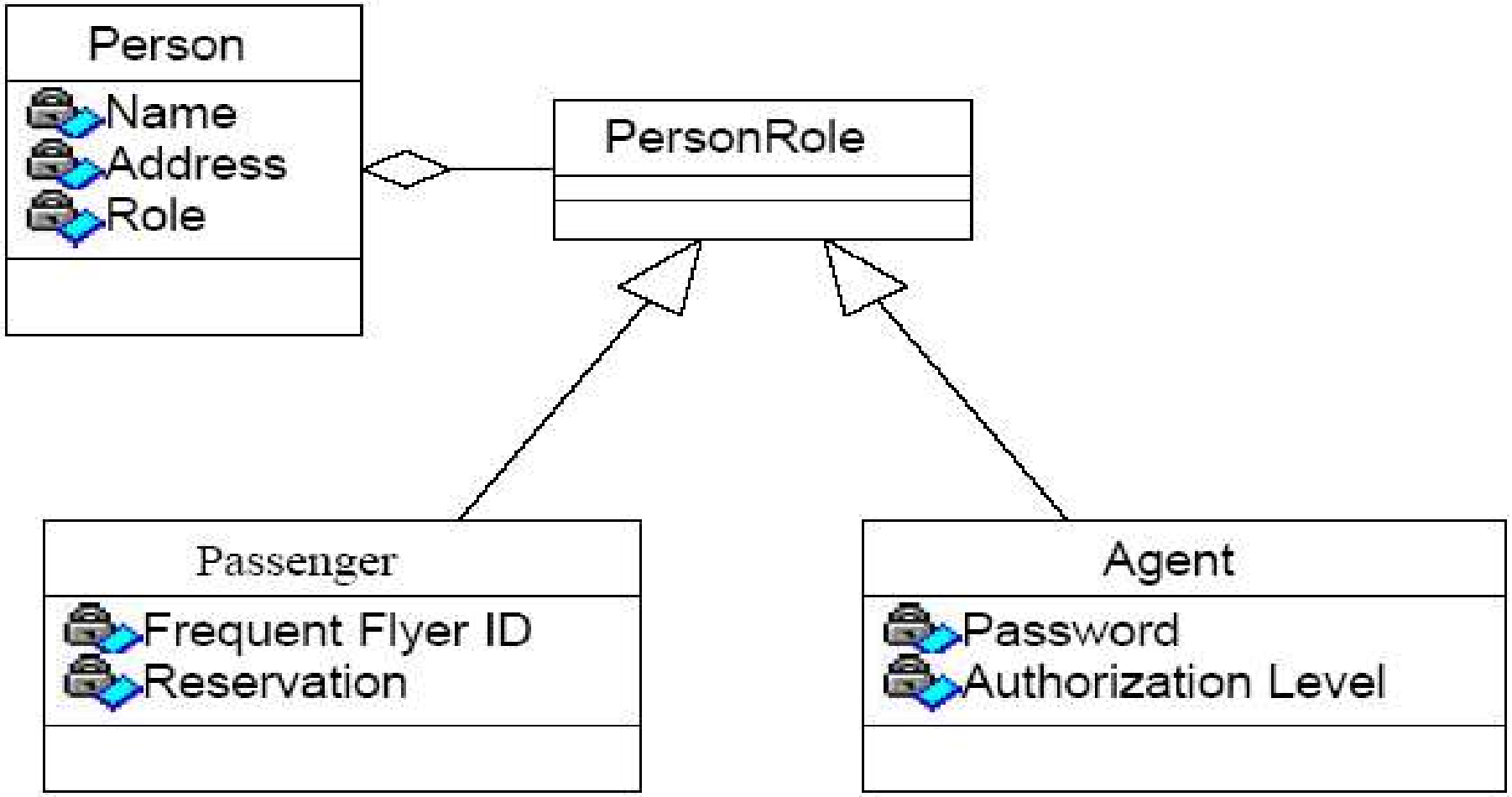


Inheritance/Composition Example 1 (Continued)

Composition to the rescue!



Inheritance/Composition Example 2



```
class Object {
    public: virtual void update() {};
            virtual void draw() {};
            virtual void collide(Object objects[]) {};
        };
class Visible : public Object {
    public: virtual void draw() {
        /* draw model at position of this object */ };
    private: Model* model;
    };
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 */ };
};
```

Main Contents

- Coupling of OO
  - Interaction Coupling
  - Component Coupling
  - Inherit Coupling
  - Coupling Metric of OO
- Cohesion of OO

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 &D (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

Main Contents

- Coupling of OO
  - Interaction Coupling
  - Component Coupling
  - Inherit Coupling
  - Coupling Metric of OO
- Cohesion of OO

Cohesion of Attributes

- Separable
  - represent multiple unrelated data abstractions combined in one object
- multifaceted
  - represent multiple related data abstractions
- Non-delegated
  - Some attribute represent a part of another class: not in third normal form
- Concealed
  - some attribute and referencing methods which may be regarded as a class of its own
- Model: informational strength
  - the class represents a single semantically meaningful conceptual

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)

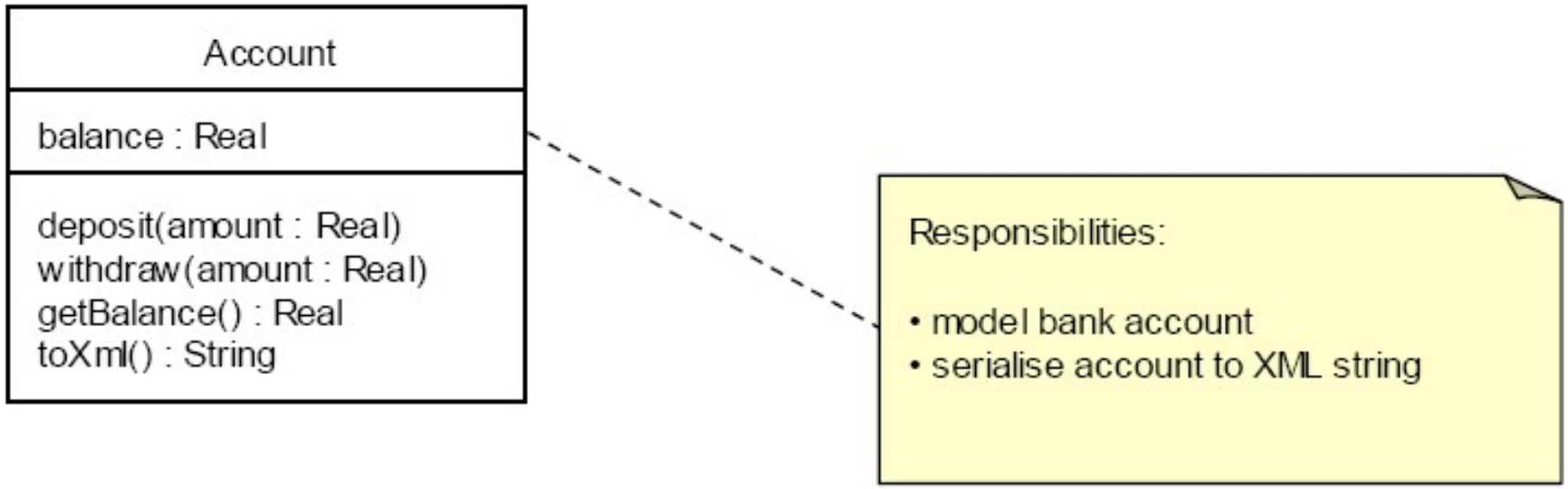
A class should have only one reason to change

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 is also a reason for the class to change

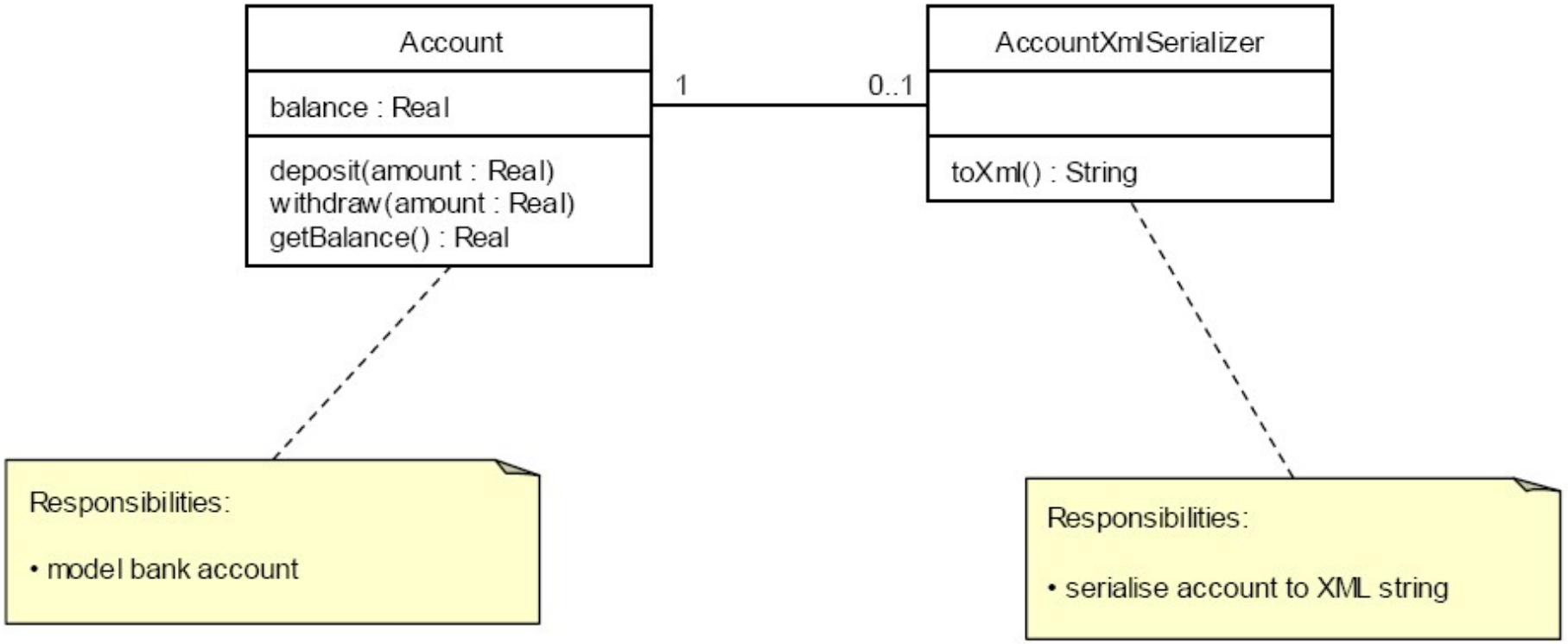
SRP Example



Two reasons why this class might need to change

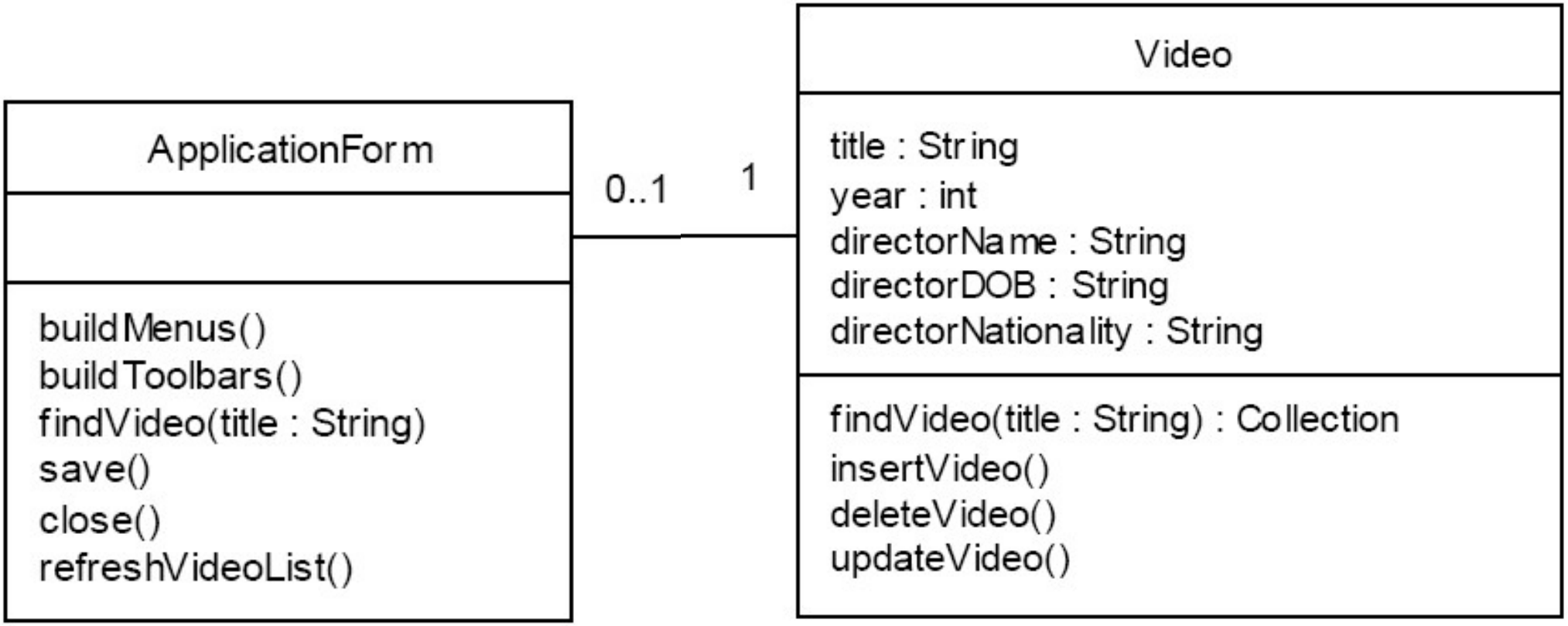
- changes to domain logic
- changes to XML format

SRP Example


Account
balance : Real
deposit(amount : Real)
withdraw(amount : Real)
getBalance() : Real
1
0..1
AccountXmlSerializer
toXml() : String
Responsibilities:
• model bank account
Responsibilities:
• serialise account to XML string

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


ApplicationForm
buildMenus()
buildToolbars()
findVideo(title : String)
save()
close()
refreshVideoList()
0..1
1
Video
title : String
year : int
directorName : String
directorDOB : String
directorNationality : String
findVideo(title : String) : Collection
insertVideo()
deleteVideo()
updateVideo()

Measure class cohesion

- **Lack of cohesion in methods (LCOM)**

  "Consider a Class C_1 with n methods M_1, M_2, ..., M_n. Let
  $\{Ij\}$ = set of instance variables used by Method Mj.
  There are n such sets $\{I_1\}$, ..., $\{I_n\}$.
  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 $\{I_1\}$, ..., $\{I_n\}$ are $\emptyset$ then let $P = \emptyset$.
  $LCOM = |P| - |Q|$. if $|P| > |Q|$
  $= 0$ otherwise."

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

Measure class cohesion

- If LCOM >=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

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

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

Principles of interaction coupling

Principles from Modularization

1: 《Global Variables Consider Harmful》

2: 《To be Explicit》

3: 《Do not Repeat》

4: Programming to Interface

More Principles

4: Programming to Interface (Design by Contract)

5: The Law of Demeter

6: Interface Segregation Principle(ISP)

7: Liskov Substitution Principle (LSP)

8: Favor Composition Over Inheritance

9: Single Responsibility Principle