# Tecniche di Progettazione: Design Patterns

## Design principles, part 2

# Design principles part 1

- Basic (architectural) design principles
  - Encapsulation
  - Accessors & Mutators (aka *getters and setters)*
  - Cohesion
  - Uncoupling
- SOLID
  - Single Responsibility Principle (1 class 1 reason to change).
  - Open Closed Principle (Extending $\neq\Longrightarrow$ modifying.)
  - Liskov Substitution Principle
  - Interface Segregation Principle (Make fine grained interfaces).
  - Dependency Inversion Principle (Program to the interface).

# Design principles part 1 (cont'd)

- GRASP
  - General Responsibility Assignment Software Patterns
  - First four:
    - Creator
    - Information Expert
    - High Cohesion
    - Low Coupling

# The nine GRASP Patterns

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

# Controller: problem

- **Who should be responsible for handling an input system event?**
- What first object beyond the UI layer receives and coordinates a system operation?
  - An input system event (system operation) is an event generated by an external actor.
  - Examples
    - when a cashier using a POS terminal presses the "End Sale" button to indicate "the sale has ended".
    - a writer using a word processor presses the "spell check" button, he is generating a system event indicating "perform a spell check."
- A Controller object is a non-user interface object responsible for receiving or handling a system event.

# The controller object: two alternate solutions

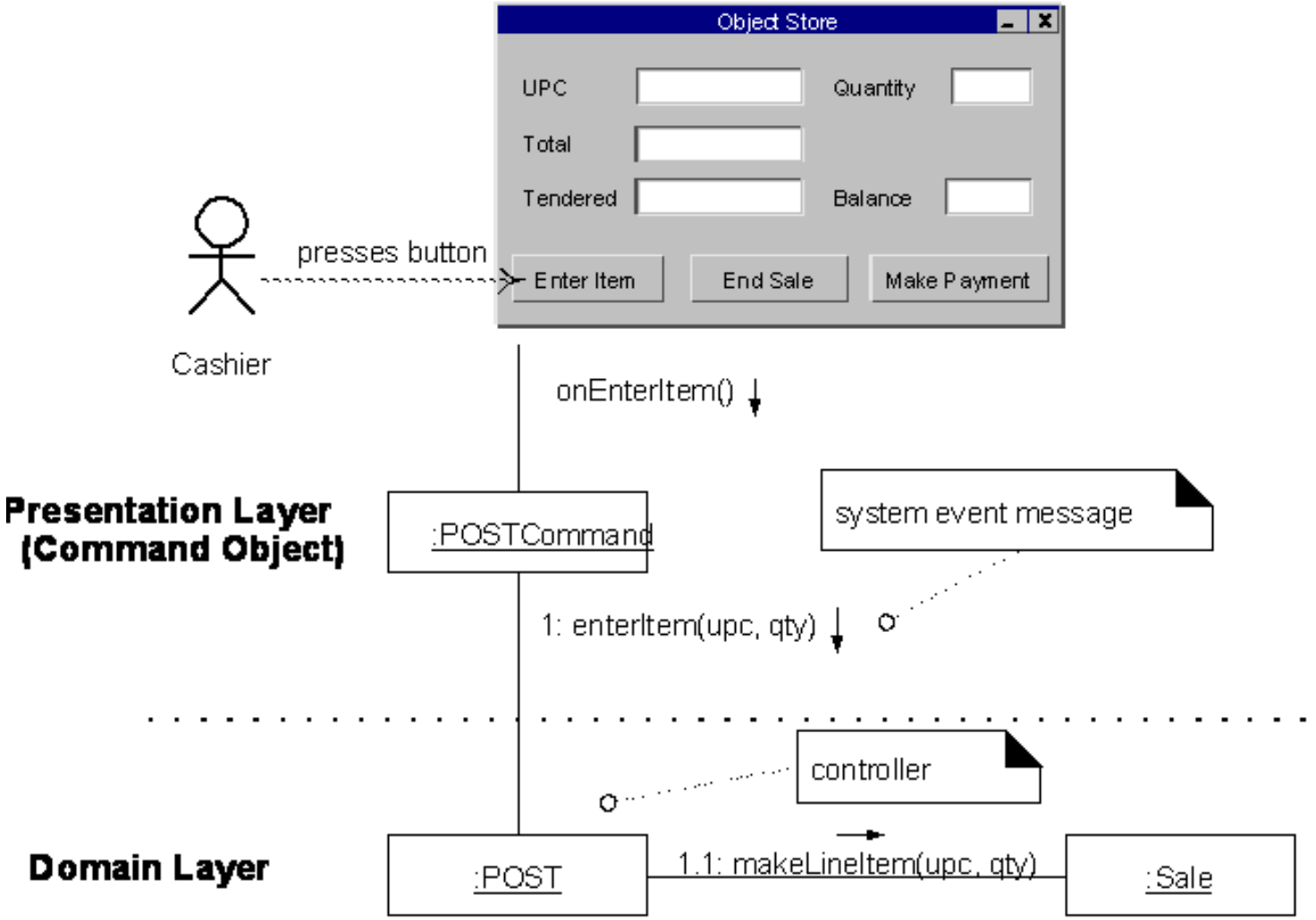
- Assign the responsibility for receiving or handling a system event message to a controller class that:
  - Represents the **overall system**, device, or subsystem
    - This class is called façade controller.
  - Represents a **use case scenario** within which the s. e. occurs
    - Often this class is named <UseCaseName>Handler, <UseCaseName>Coordinator, or <Use-CaseName>Session
    - Use the same class for all system events originating in the same use case. (A session is an instance of a conversation with an actor. )
- Note that "window," "applet," "widget," "view," and "document" classes typically receive these events and delegate them to a controller.

# Controller : Example

- System events in Buy Items use case
  - enterItem()
  - endSale()
  - makePayment()

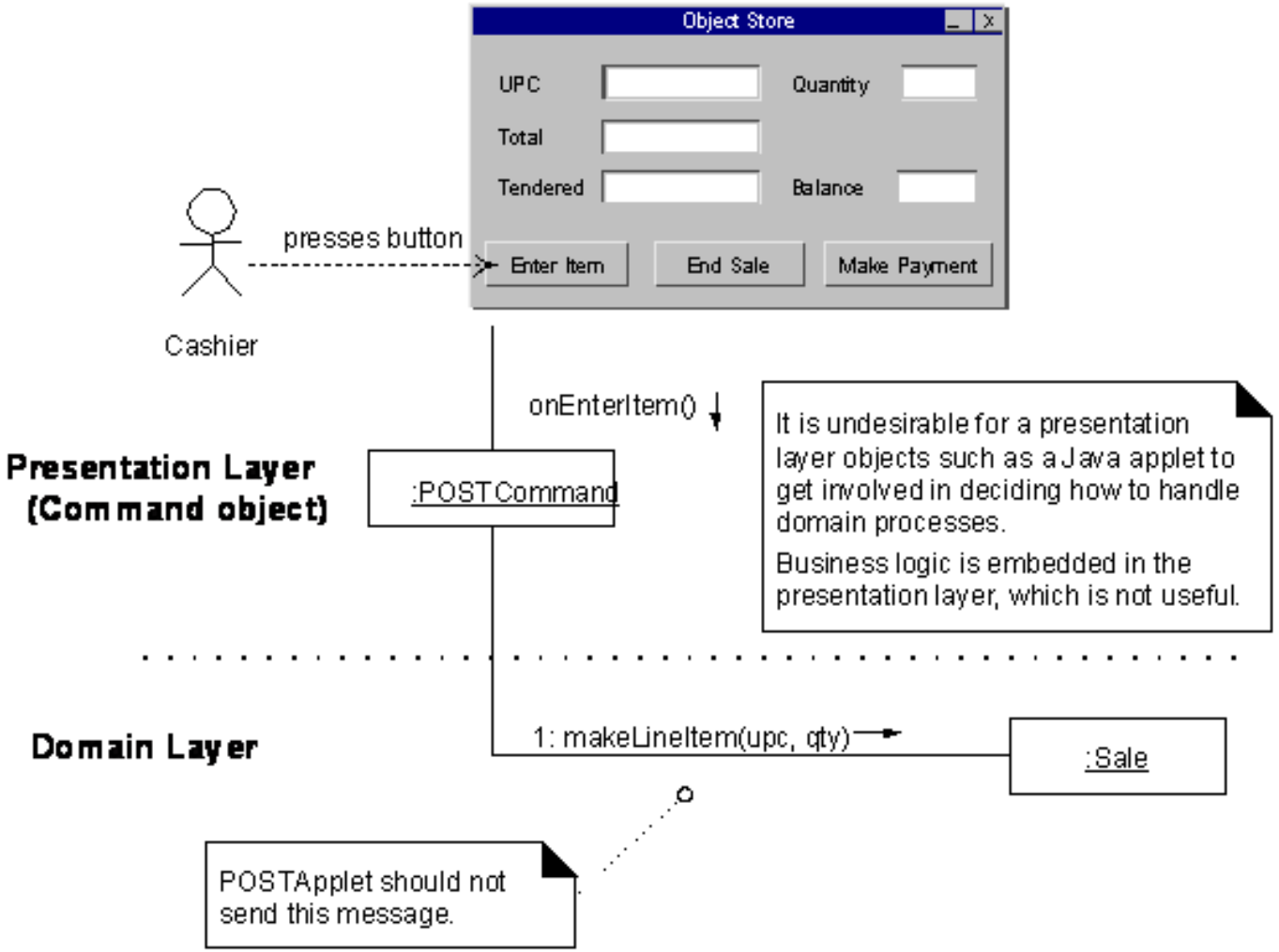
# Good design

- presentation layer decoupled from problem domain

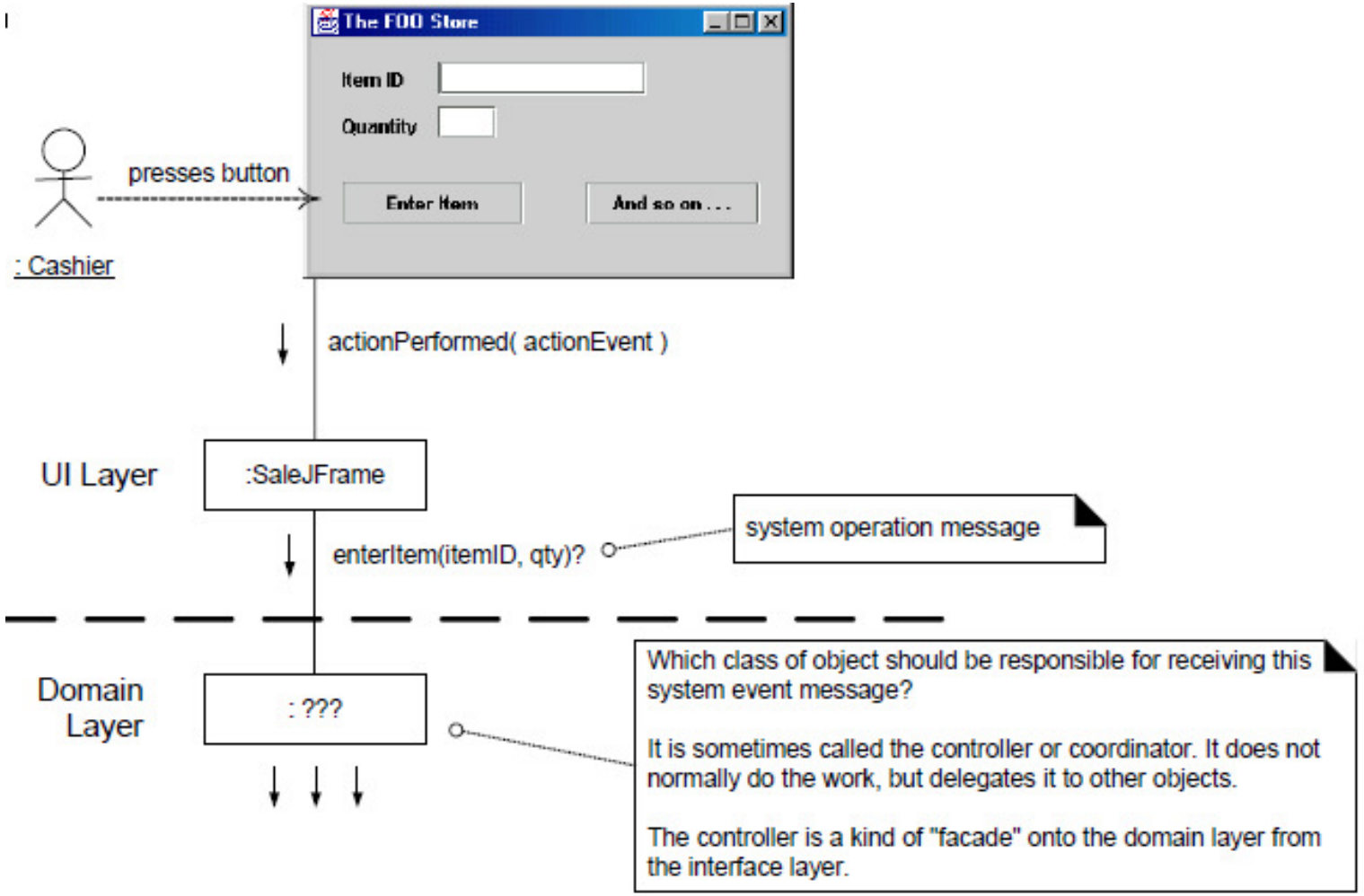


# Bad design

## – presentation layer coupled to problem domain

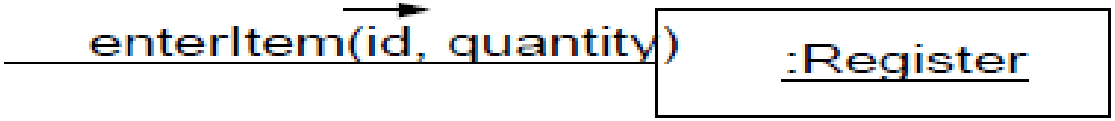


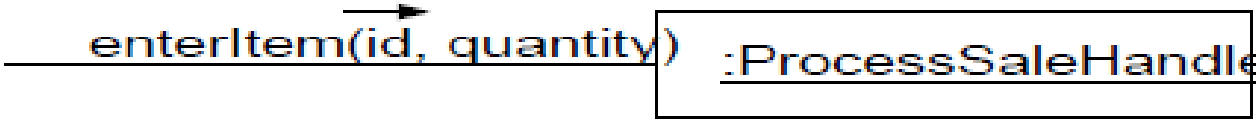
# But then: What object should be the controller for enterItem?



# Controller object: 2 choices

- By the controller pattern, there are choices
  - A controller class to represent the whole system, some root object … Register for example.
  - A controller to handle all system events of a use case, ProcessSaleHandler for example
- Which choice is more appropriate depend on many other factors. **The value of the pattern is to make you consider the alternatives.**



# Discussion

- A controller delegates to other objects the work that needs to be done. It coordinates or controls the activity. It should not do much work itself.
- Increased potential for reuse.
  - Using a controller object keeps external event sources and internal event handlers independent of each other's type and behaviour.
  - It ensures that application logic is not handled in the interface layer
- Reason about the states of the use case.
  - Ensures that the system operations occur in legal sequence, and permits to reason about the current state of activity and operations within the use case.
  - For example, it may be necessary to guarantee that the makePayment operation does not occur until the endSale operation has occurred.

# Discussion (cont'd)

- The first category of controller is a façade controller representing the overall system.
  - Façade controllers are suitable where there are not too many system events or it is not possible for the GUI to redirect system event messages to distinguished controllers
  - The controller objects can become highly coupled and uncohesive with more responsiblities
- The second category of controller is a use-case controller; in this case there is a different controller for each use case.
  - It is desirable to use the same controller class for all the system events of one use case.

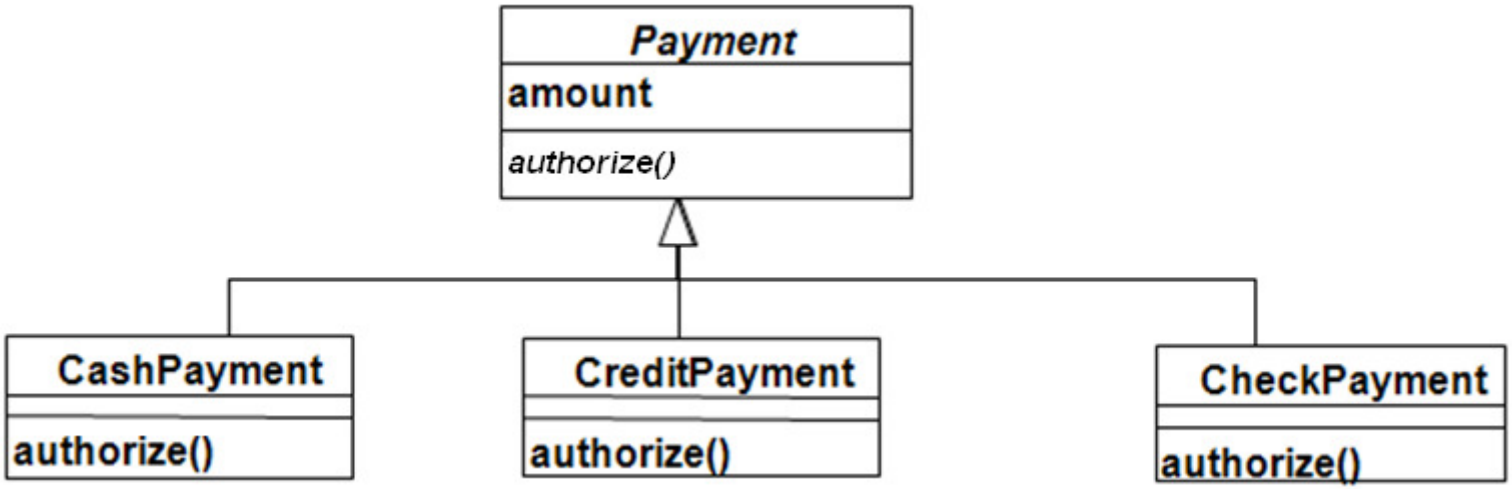
# The nine GRASP Patterns

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

# Def of polymorphism

- is one of the fundamental features of the OO paradigm
  - an abstract operation may be implemented in different ways in different classes
  - applies when several classes, each implementing the operation, either have a common superclass in which the operation exists, or else implement an interface that contains the operation
- gets power from dynamic binding

# Polymorphism : Example




- Who should be responsible for authorising different kinds of payments? Payments may be in
  - cash (authorising involves determining if it is counterfeit)
  - credit (authorising involves communication with bank)
  - check (authorising involves driver license record)

# Polymorphism

- Problem:
  - How to handle alternatives based on type? How to create pluggable software components?
    - Alternatives based on type – avoiding if-then-else conditional logic that makes extension difficult
    - Pluggable components – how can you replace one component with another without affecting the client code?
- Solution:
  - When alternate behaviours are selected based on the type of an object, use polymorphic method call to select the behaviour, rather than using if statement to test the type.

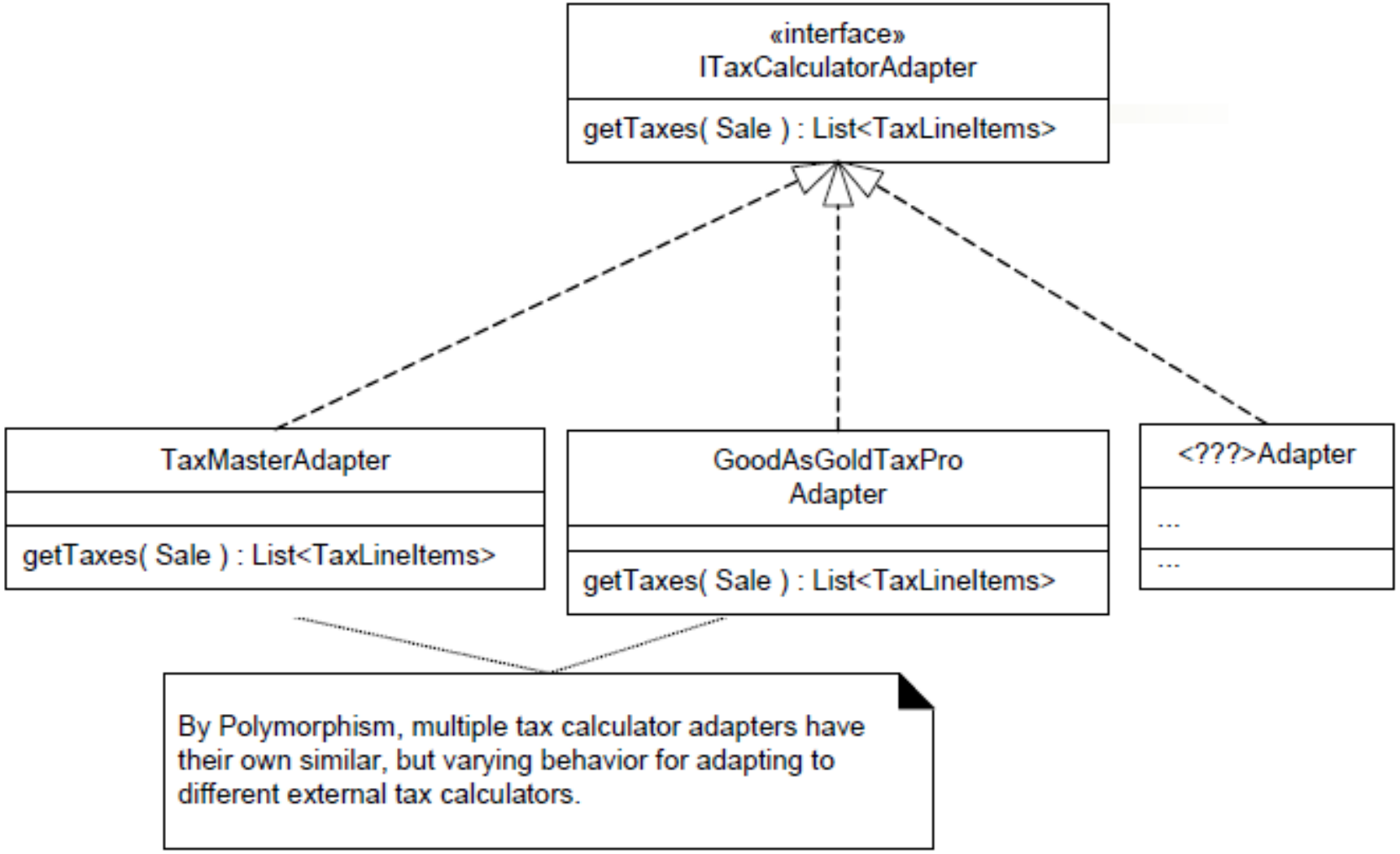
# Broader use of polymorphism

- In the GRASP context polymorphism has also a broader meaning
  - Give the same name to services in different objects when the services are similar or related

# Broader use of polymorphism: Ex.

- There are multiple external third-party tax calculators that must be supported – the system needs to be able to integrate with all of these.
  - The calculators have different interfaces but similar, though varying behavior.
  - What object should be responsible for handling this variation?
- Since the behavior of calculator adaptation varies by the type of calculator, by polymorphism the responsibility of this adaptation is assigned to different calculator (adapter) objects themselves.

# Ex


«interface»
ITaxCalculatorAdapter
getTaxes( Sale ) : List<TaxLineItems>
TaxMasterAdapter
getTaxes( Sale ) : List<TaxLineItems>
GoodAsGoldTaxPro
Adapter
getTaxes( Sale ) : List<TaxLineItems>
<???>Adapter
...
...
By Polymorphism, multiple tax calculator adapters have their own similar, but varying behavior for adapting to different external tax calculators.

# Discussion

- Easier and more reliable than using explicit selection logic
- Extensions required for new variations are easy to add
- New implementations can be introduced without affecting clients.

- aka:
  - "Do it myself"
    - Example: payments authorise themselves
  - "Choosing Message"
  - "don't ask 'what kind?'"

# The nine GRASP Patterns

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

# Pure Fabrication

- Problem:
  - Not to violate High Cohesion and Low Coupling
- Solution:
  - Assign a highly cohesive set of responsibilities to an artificial class that does not represent anything in the problem domain, in order to support high cohesion, low coupling, and reuse.

# Pure Fabrication: discussion

- The design of objects can be roughly partitioned to two groups
  - Those chosen by ***representational decomposition***
  - Those chosen by ***behavioral decomposition***
- The latter group doesn't represent anything in the problem domain, they are simply made up for the convenience of the designer, thus the name ***pure fabrication***.
- The classes are designed to group together related behavior
- A pure fabrication object is a kind of functioncentric (or behavioral) object

# Pure Fabrication: Example

- Suppose, in the point of sale example, that support is needed to save Sale instances in a relational database.
- By Expert, there is some justification to assign this responsibility to Sale class. However.
  - The task requires a relatively large number of supporting database-oriented operations and the Sale class becomes incohesive.
  - The sale class has to be coupled to the relational database increasing its coupling.
  - Saving objects in a relational database is a very general task for which many classes need support.
  - Placing these responsibilities in the Sale class suggests there is going to be poor reuse or lots of duplication in other classes that do the same thing.

# Pure Fabrication : Example


- The Sale remains well designed, with high cohesion and low coupling
- The PersistentStorage class is itself relatively cohesive
- The PersistentStorage class is a very generic and reusable object

# Other ex

- A login class:  not defined by the domain but needed
- Factories

# Discussion

- High cohesion is supported because responsibilities are factored into a class that only focuses on a very specific set of related tasks.
- Reuse potential may be increased because of the presence of Pure Fabrication classes.
- Architectural goals like separation of concerns may be supported by a pure fabrication
  - e.g. a *PersistentStorage class with the sole responsibility* of saving objects in some persistent storage keeps its client classes highly cohesive, removes the ugly coupling from client classes to databases, and may itself be highly reusable.

# The nine GRASP Patterns

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

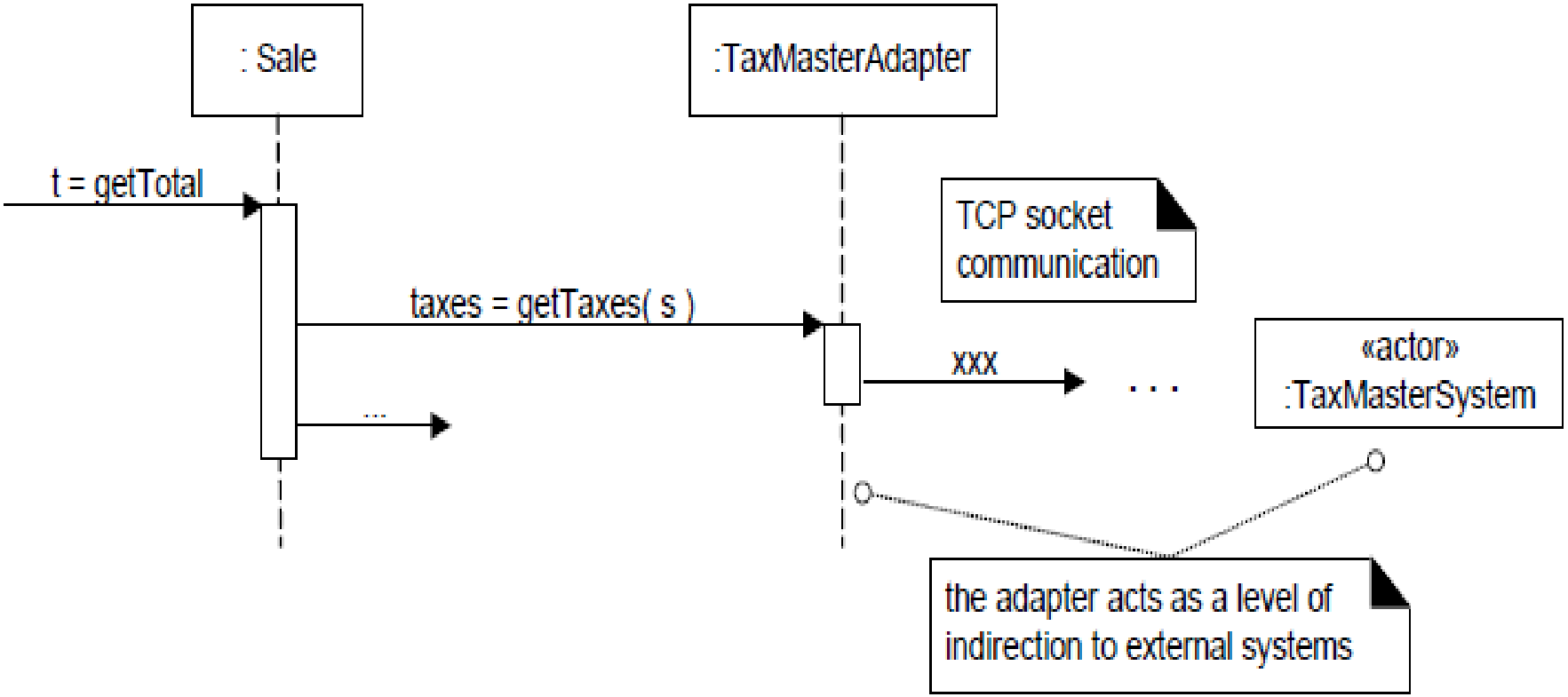
# Indirection

- Problem:
  - How to avoid direct coupling?
  - How to de-couple objects so that Low coupling is supported and reuse potential remains high?
- Solution:
  - Assign the responsibility to an intermediate object to mediate between other components or services, so that they are not directly coupled.
- Many indirection intermediaries are Pure Fabrications.

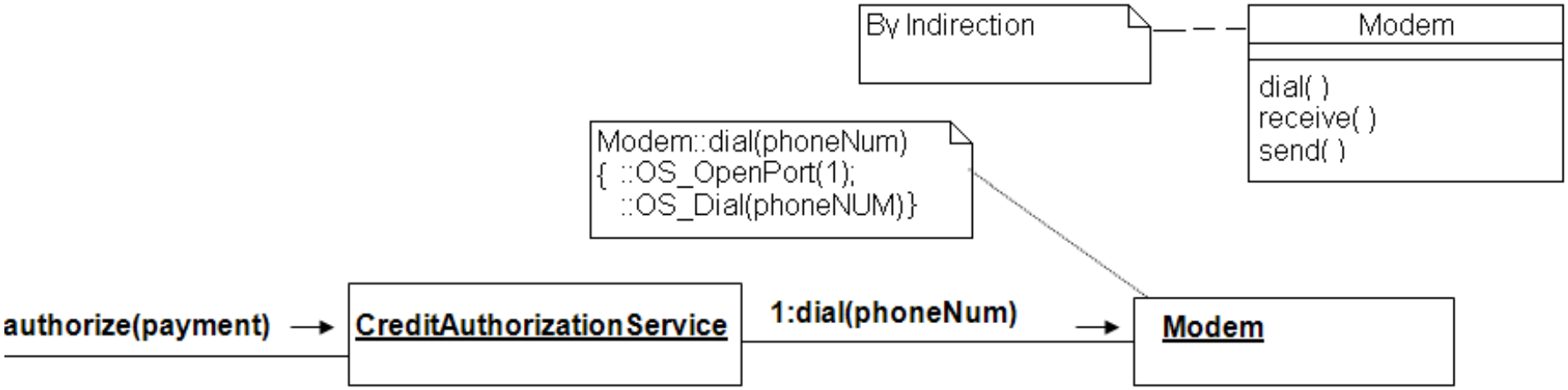
# Example : PersistentStorage

- The Pure fabrication example of de-coupling the Sale from the relational database services through the introduction of a PersistentStorage  is also an example of assigning responsibilities to support Indirection.
- The PersistentStorage acts as a intermediary between the Sale and database

# Tax Ex. adapters are indirection intermediaries



# Indirection : Example




- Assume that :
  - A point-of-sale terminal application needs to manipulate a modem in order to transmit credit payment request
  - The operating system provides a low-level function call API for doing so.
  - A class called CreditAuthorizationService is responsible for talking to the modem
- If CreditAuthorizationService invokes the low –level API function calls directly, it is highly coupled to the API of the particular operating system. If the class needs to be ported to another operating system, then it will require modification.
- Add an intermediate Modem class between the CreditAuthorizationService and the modem API. It is responsible for translating abstract modem requests to the API and creating an Indirection between the CreditAuthorizationService and the modem.

# The nine GRASP Patterns

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

# Protected Variation

- Problem:
  - How to design objects, subsystems and systems so that the variations or instability in these elements does not have an undesirable impact on other elements.
- Solution:
  - **Identify points of predicted variation or instability; assign responsibilities to create a stable interface (or protection mechanism or enveloppe) around them.**
- Data encapsulation, interfaces, polymorphism, indirection and standards are motivated by Protected Variation.

# Protected Variation: Example

- Technology like Service Lookup is an example of Protected Variation because clients are protected from variations in the location of services using the lookup service.
- Benefits:
  - Extensions required for new variations are easy to add.
  - New implementations can be introduced without affecting clients.
  - Coupling is lowered.
  - The impact of cost of changes can be lowered.

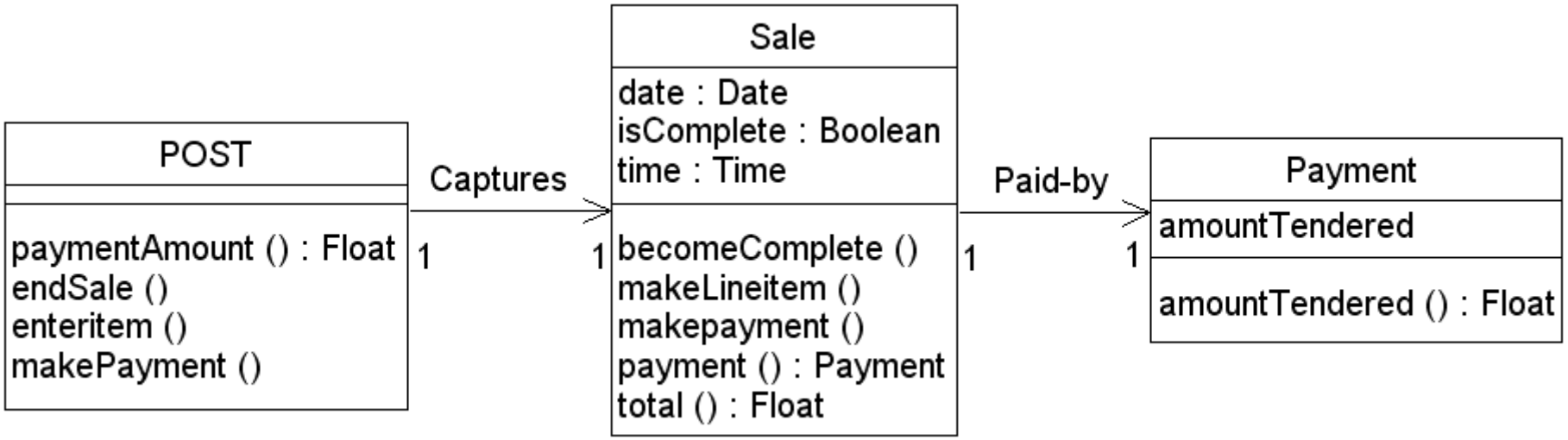
# Protected Variation: Law of Demeter

- Aka Structure-hiding design, aka Don't Talk to Strangers
- Special case of Protected variations.
- **If two classes have no other reason to be directly aware of each other or otherwise coupled, then the two classes should not directly interact.**
- A method should send messages only to:
  - *this*
  - An attribute of *this*
  - An element of a collection which is attribute of *this*
  - A parameter of the method
  - An object created within the method

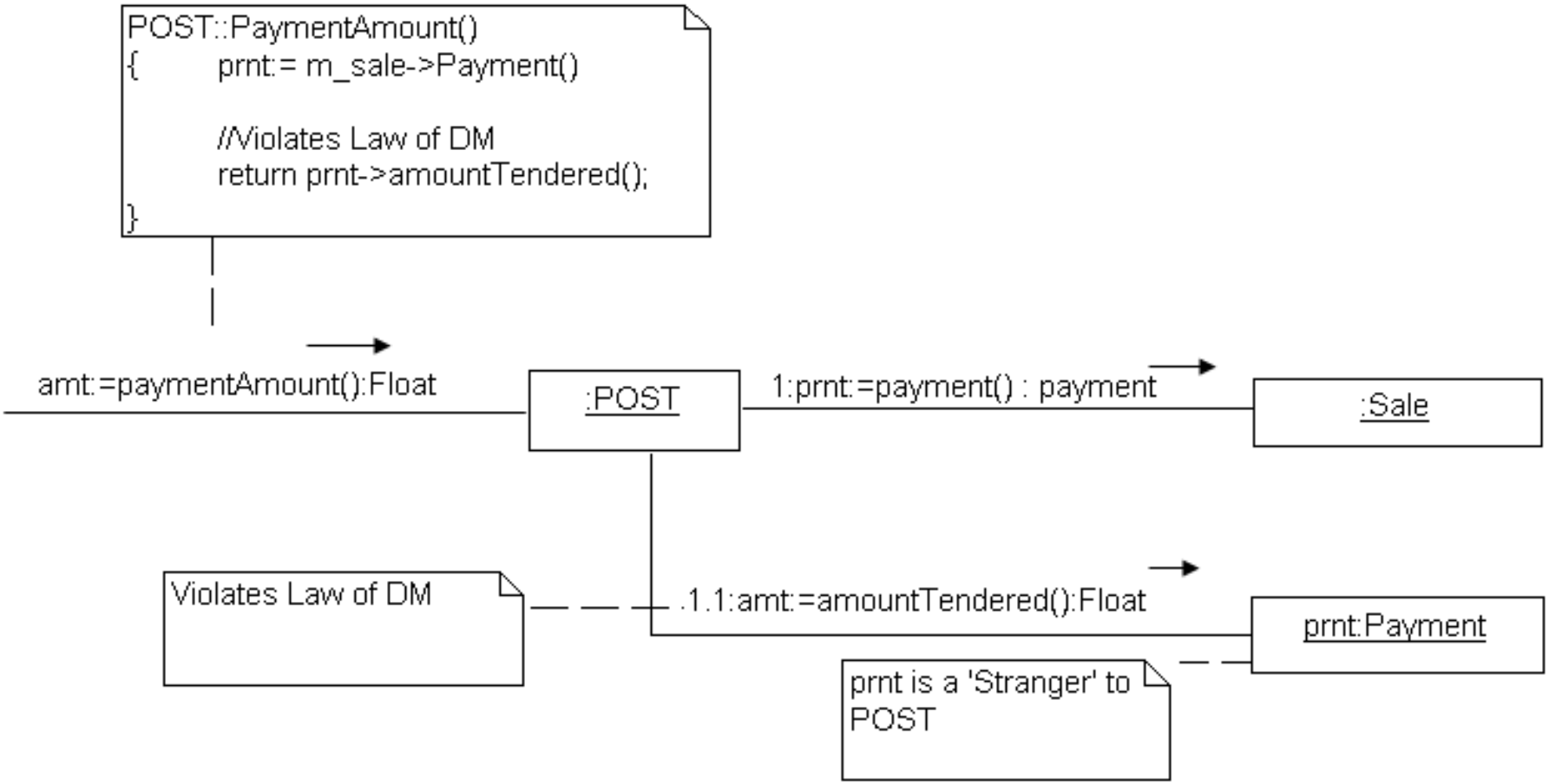
# Protected Variation: Law of Demeter

- Avoid
  - sale.getPayment().getAmount().getCurrency()
- And also the equivalent
  - x=sale.getPayment()
  - y=x. getAmount()
  - z=y. getCurrency()

# Law of Demeter : Example

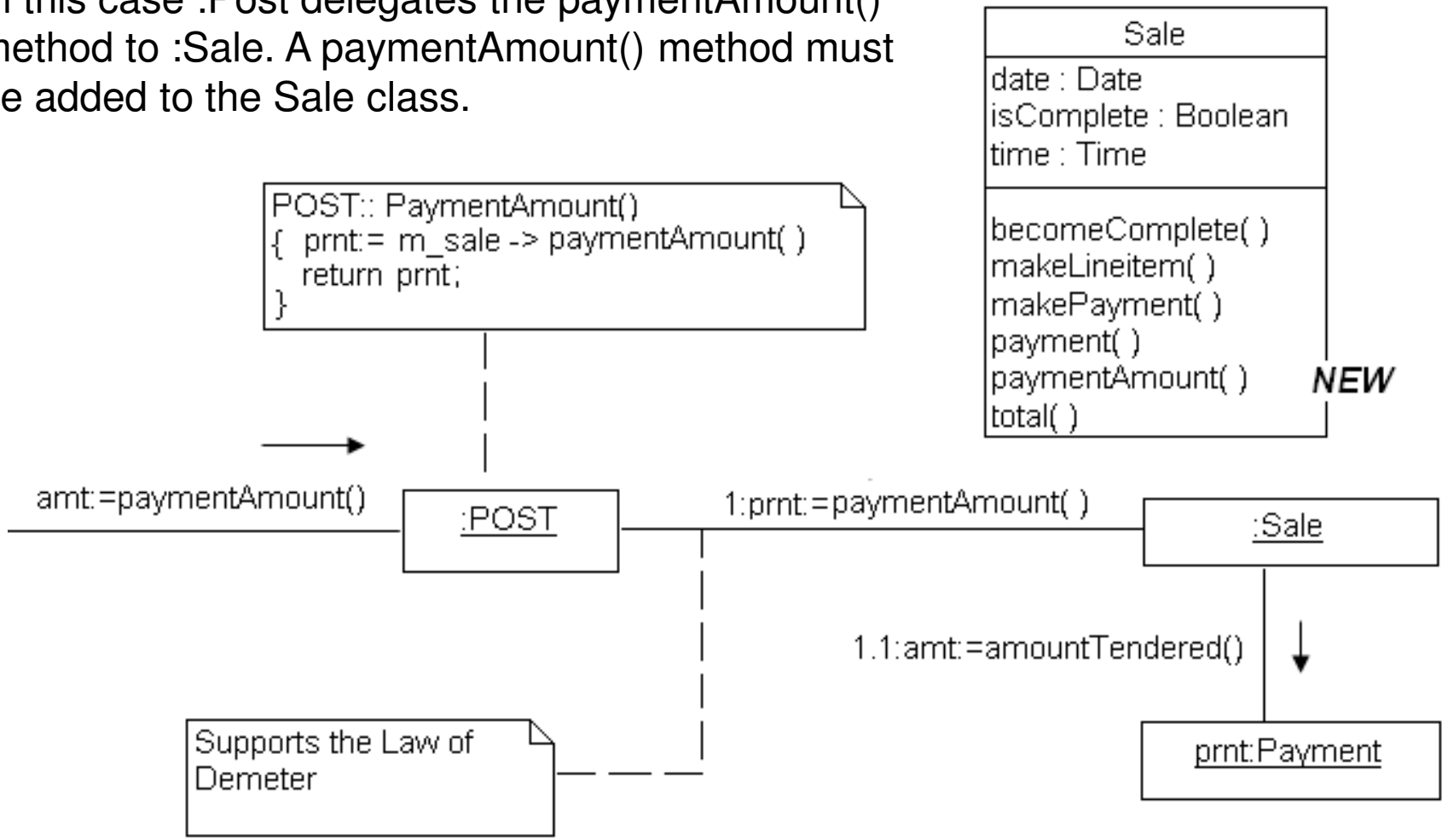


# Violating Law of Demeter: Example



# Supporting Law of Demeter

In this case :Post delegates the paymentAmount() method to :Sale. A paymentAmount() method must be added to the Sale class.



# Law of Demeter: discussion

- Keeps coupling between classes low and makes a design more robust
- Adds a small amount of overhead in the form of indirect method calls

# Delegation vs inheritance

# Delegation vs inheritance [Mark Grand98]

- Inheritance
  - defines a new class, which use the interface of a parent class while adding extra, more problem-specific methods.
- Delegation
  - is a way of reusing and extending the behavior of a class by writing a new class that incorporates the functionality of the original class by using an instance of the original class and calling its methods.
- No.1 issue in OO is if a class A should inherit from B or A should use B.

# Motivation

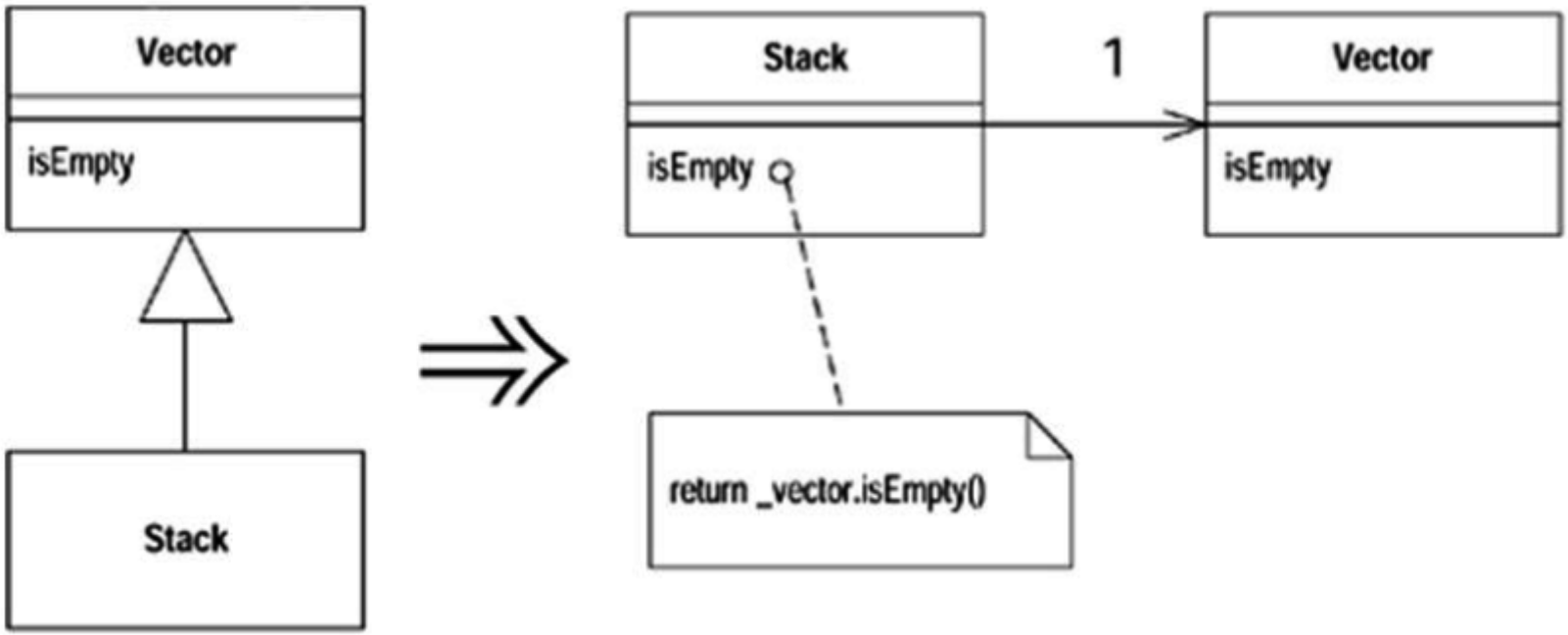
- Inheritance is a wonderful thing, but sometimes it isn't what you want.
  - Often you start inheriting from a class but then find that many of the superclass operations aren't really true of the subclass. In this case you have an interface that's not a true reflection of what the class does.
  - Or you may find that you are inheriting a whole load of data that is not appropriate for the subclass.
  - Or you may find that there are protected superclass methods that don't make much sense with the subclass.

# Motivation (continued)

- You can live with the situation and use convention to say that although it is a subclass, it's using only part of the superclass function.
  - But that results in code that says one thing when your intention is something else—a confusion you should remove.
- **By using delegation** instead, you make it clear that you are making only partial use of the delegated class. You control which aspects of the interface to take and which to ignore.
- **The cost** is extra delegating methods that are boring to write but are too simple to go wrong.

# Replace Inheritance with Delegation

- Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing.



# Example

- One of the classic examples of inappropriate inheritance is making a stack a subclass of vector. Java does this in its utilities (naughty boys!), but in this case I use a simplified form of stack:

```
class MyStack extends Vector {
        public void push(Object element) {insertElementAt(element,0); }
        public Object pop() { Object result = firstElement();
                                        removeElementAt(0); return result; }
    }
```

- Looking at the users of the class, I realize that clients do only four things with stack: push, pop, size, and isEmpty. The latter two are inherited from Vector.

# Example (continued)

- I begin the delegation by creating a field for the delegated vector. I link this field to this so that I can mix delegation and inheritance while I carry out the refactoring:

```
private Vector _vector = this;
```

- Now I start replacing methods to get them to use the delegation. I begin with push:

```
public void push(Object element) {
    _vector.insertElementAt(element,0); }
```

- I can compile and test here, and everything will still work.

# Example (continued)

- Now pop:

```
public Object pop() {
    Object result = _vector.firstElement();
    _vector.removeElementAt(0);
    return result;
    }
```

# Example (continued)

- Once I've completed these subclass methods, I need to break the link to the superclass:

```
class MyStack
private Vector _vector = new Vector();
```

- I then add simple delegating methods for superclass methods used by clients:

```
public int size() { return _vector.size(); }
public boolean isEmpty() { return _vector.isEmpty(); }
```

- Now I can compile and test. If I forgot to add a delegating method, the compilation will tell me.

# Mechanics

- Create a field in the subclass that refers to an instance of the superclass. Initialize it to this.
- Change each method defined in the subclass to use the delegate field. Compile &test after changing each method.
  - *You won't be able to replace any methods that invoke a method on* super *that is defined on the subclass, or they may get into an infinite recurse. These methods can be replaced only after you have broken the inheritance.*
- Remove the subclass declaration and replace the delegate assignment with an assignment to a new object.
- For each superclass method used by a client, add a simple delegating method.
- Compile and test.

# Interface inheritance vs class inheritance

- Class inheritance: implementation reuse
  - dangerous when overiding to nothing
  - See the non flying (rubber) duck in the head first book, p. 4-5.

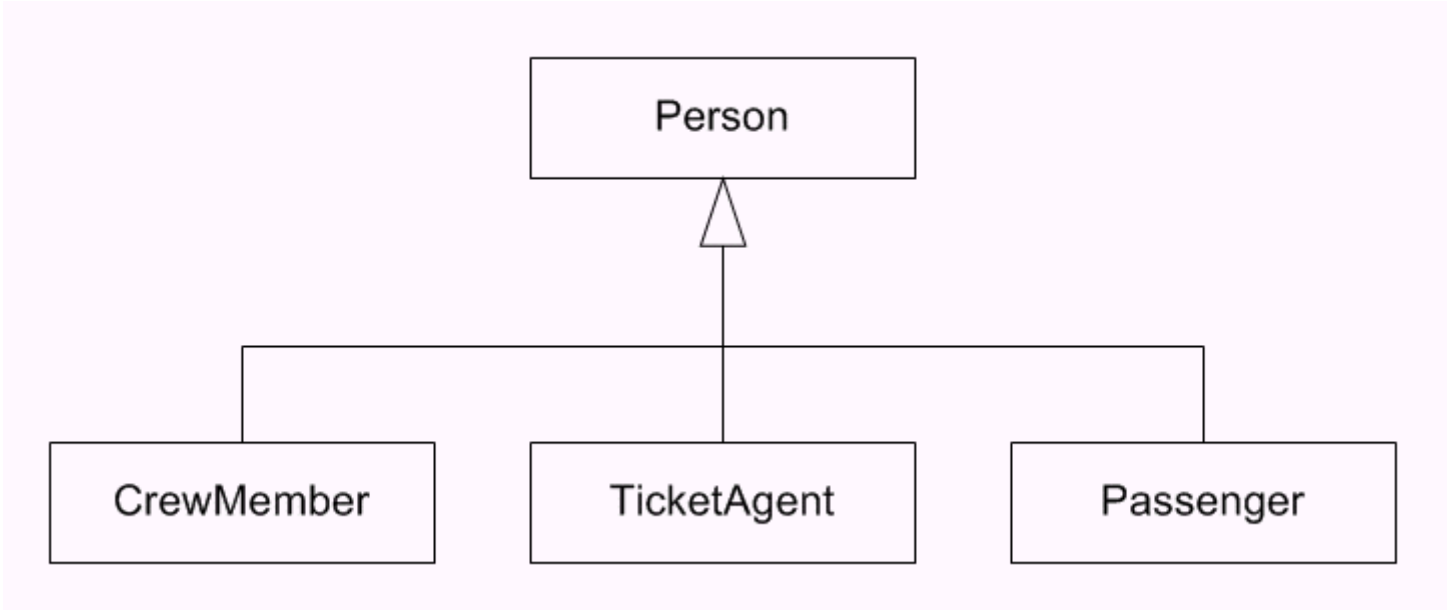
- Interface Inheritance: subtypes
  - Flyable and Quackable duck, subtypes of Duck, with *fly* and *quack* functionalities, resp.
  - But then no code reuse for those funcionalities
  - And… there might be different fly behaviours even among the ducks that do fly.

# Delegation (When not using inheritance) [Mark Grand98]

- Inheritance is a common way of extending and reusing the functionality of a class. However, inheritance is inappropriate for many situations:
  - Inheritance is useful for capturing is-a-kind-of relationships which are rather static in nature.
  - is-a-role-played-by relationships are awkward to model by inheritance, where delegation could be a better choice. Using instances of a class to play multiple roles.

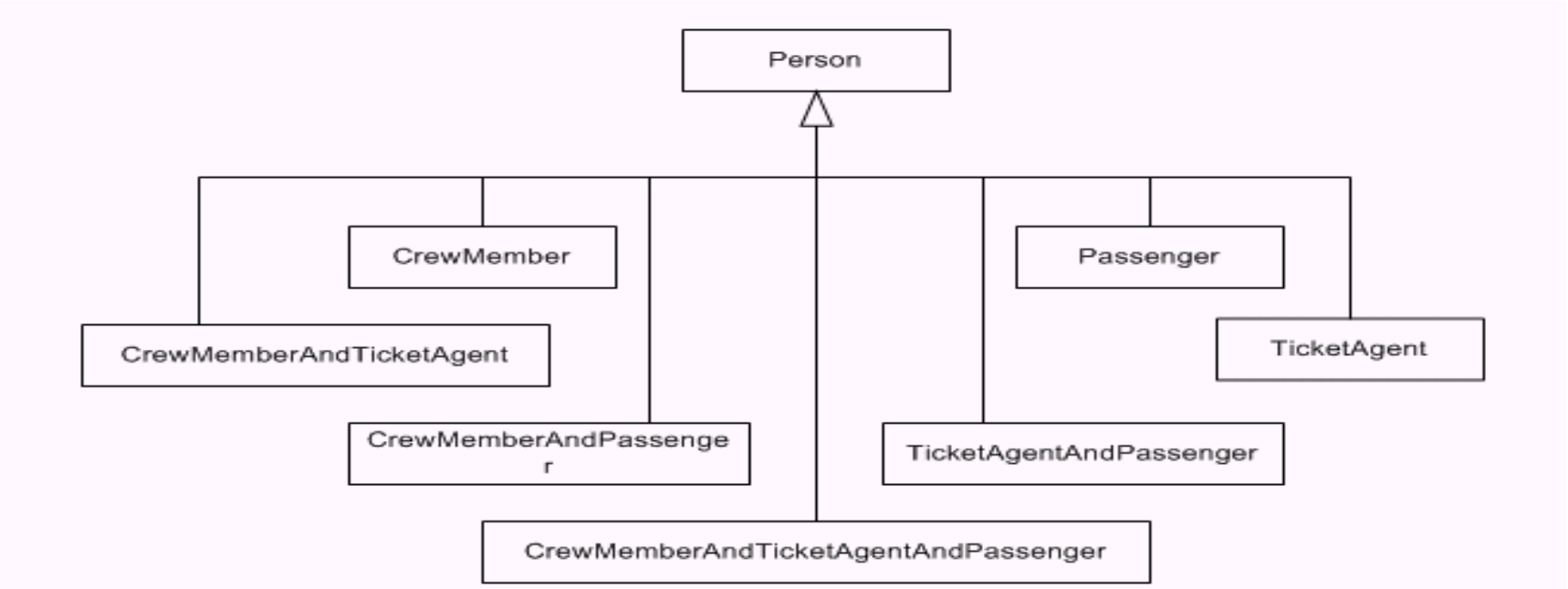
# Inheritance vs delegation: changing roles

- Don't use inheritance where roles interchange.
  - For example, an airline reservation system may include such roles as passenger, ticket selling agent and flight crew.
  - A class called Person may use subclasses corresponding to each of these roles.



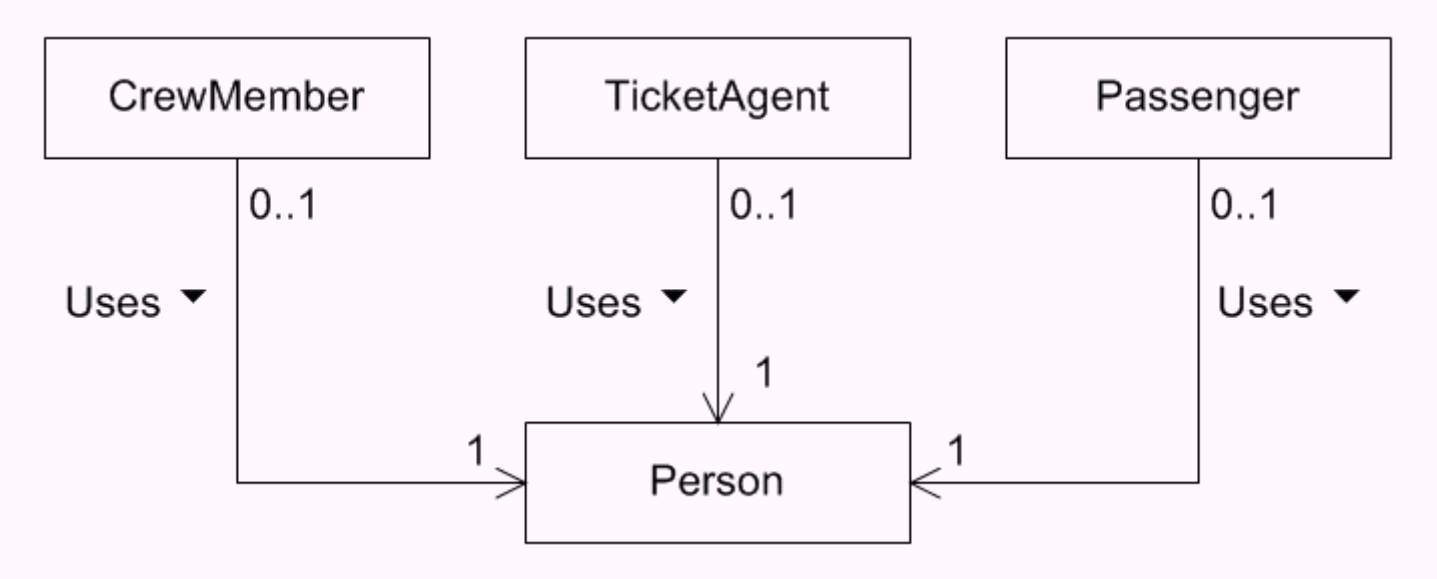
# Example (cont'd)

- The problem is that the same person can fill more than one of these roles.
  - A person who is normally part of a flight crew can also be a passenger…
  - This way, the number of subclasses would increase exponentially.

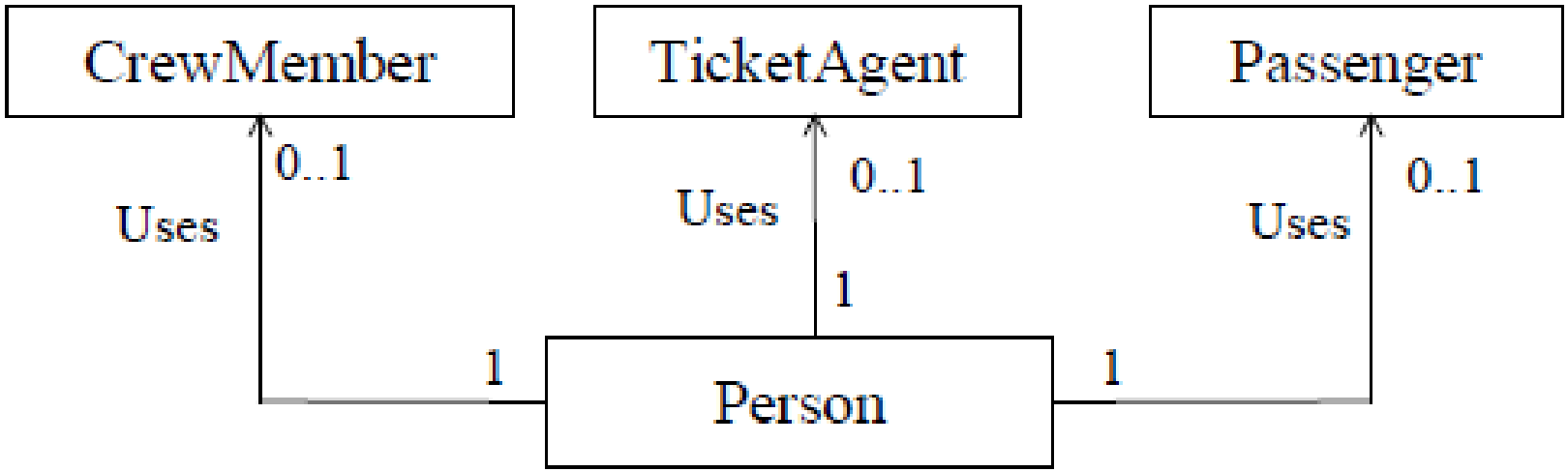


# Example (cont'd)

- If person A, CrewMember, becomes now also a Passenger, a new object Passenger is created, referring A.



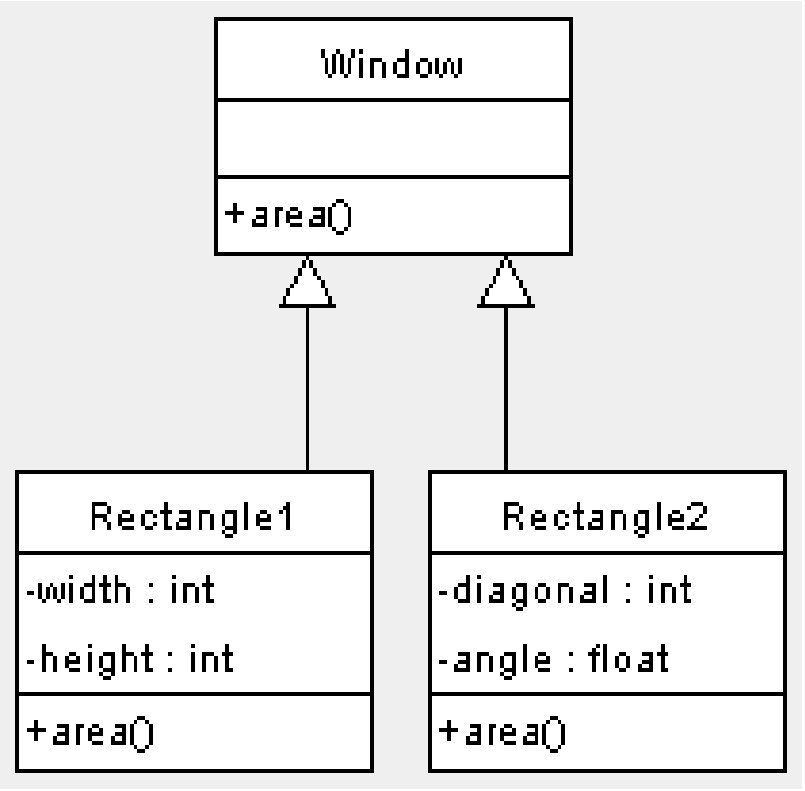
# Example (cont'd)




- But then problems with using the specific methods, which were unforeseen.

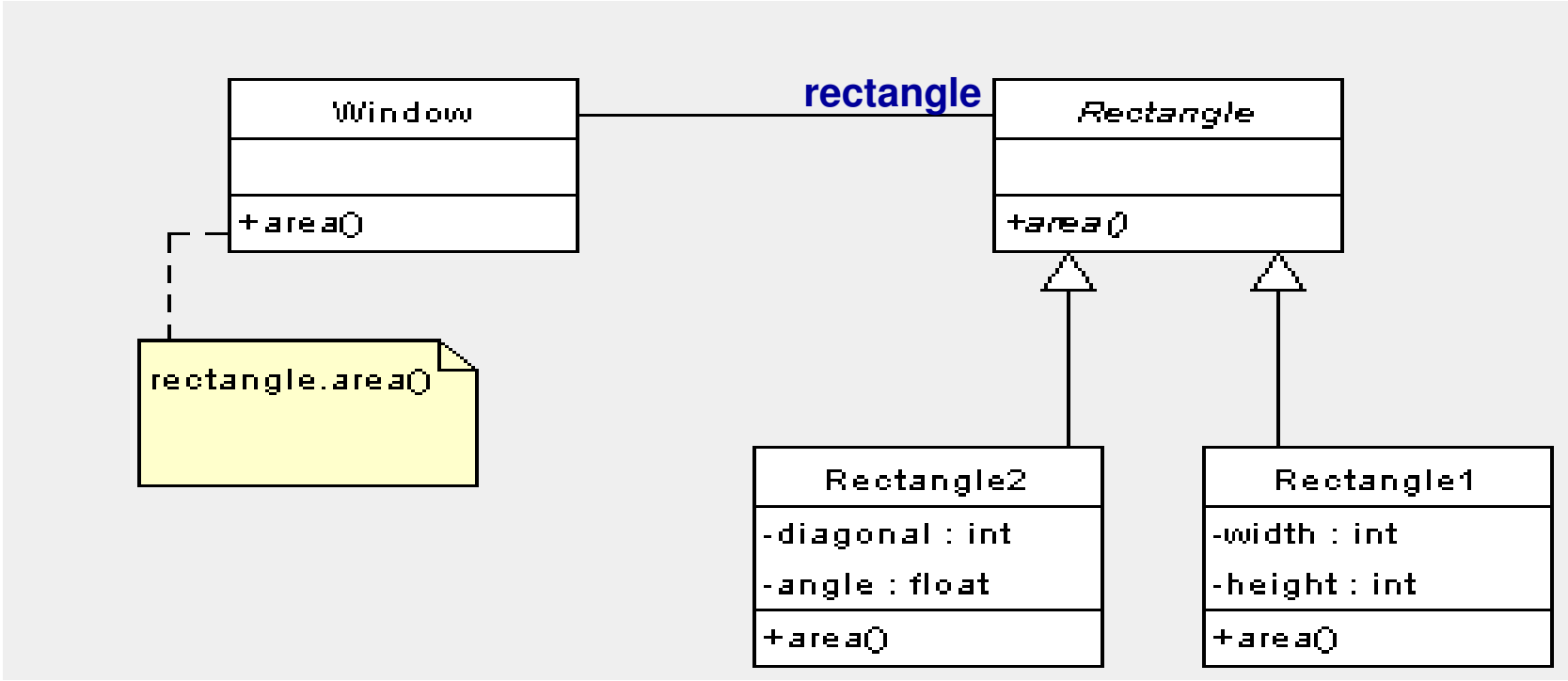
# Inheritance vs delegation: killing ex.

- If you want to let the Window client change the implementation of area, you need define different specializations, and rebuild the whole object to perform the change.




- … and you can't change the implementation inherited from super classes at runtime.

# Inheritance vs delegation: killing ex. (cont'd)




- With delegation, you only need to change the object relative to the delegated operation you want to change

# Inheritance vs delegation:languages

- In Java or C#, an object cannot change its type once it has been instantiated.
- So, if your object need to appear as a different object or behave differently depending on an object state or conditions, then use Composition

- Refer to State and Strategy Design Patterns.

- If the object need to be of the same type, then use Inheritance or implement interfaces

# Inheritance vs delegation: hiding

- Don't use inheritance if you end up in a situation where a class is trying to hide a method or variable inherited from a superclass.
  - If you define a field in a subclass that has the same name as an accessible field in its superclass, the subclass's field *hides* the superclass's version.
    - E.g., if a superclass declares a public field, subclasses will either inherit or hide it. (You can't override a field.)
  - If a subclass hides a field, the superclass's version is still part of the subclass's object data; however methods in the subclass can access the superclass's version only by using the super keyword, as in super.fieldName.

# Inheritance vs delegation: utility classes

- Don't use inheritance of a utility class
  - you're not in control of the parent class and it may change scope later (inheriting java.util.Vector is a very, very bad idea since sun may later declare methods deprecated).
  - It's always easier to replace changing a class you just use – than one you inherit from.
  - Besides, inheritance exposes a subclass to details of its parent's class implementation, that's why it's often said that inheritance breaks encapsulation (in a sense that you really need to focus on interfaces only not implementation, so reusing by sub classing is not always preferred).

# Places where not to use inheritance (but rather delegation) (continued)

- Don't use inheritance from a class, which is written very specifically to a narrow problem - because that will make it more difficult to inherit from another class later.
  - Client classes that use the problem domain class may be written in a way that assumes the problem domain class is a subclass of the utility class.
  - If the implementation of the problem domain changes in a way that results in its having a different superclass, those client classes that rely on its having its original superclass will break.

# Potential Drawbacks of Delegation

- There may be some minor performance penalty for invoking an operation across object boundaries as opposed to using an inherited method.
- Delegation can't be used with partially abstract (uninstantiable) classes
- Delegation does not impose any disciplined structure on the design.

# Homework

- Prepare some few slides on the topic delegation vs inheritance with different examples than the one presented here.
  - Possible starting point
  - http://stackoverflow.com/questions/49002/prefer-composition-over-inheritance

- Send (the pdf) by Tuesday do me (semini@di.unipi.it) with subject
  - DPhomework1

- Send (the pdf) by Tuesday do me (semini@di.unipi.it) with subject
  - DPhomework1