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 $\not\Longrightarrow$ modification of the class.)
  - 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

- HOMEWORK

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.

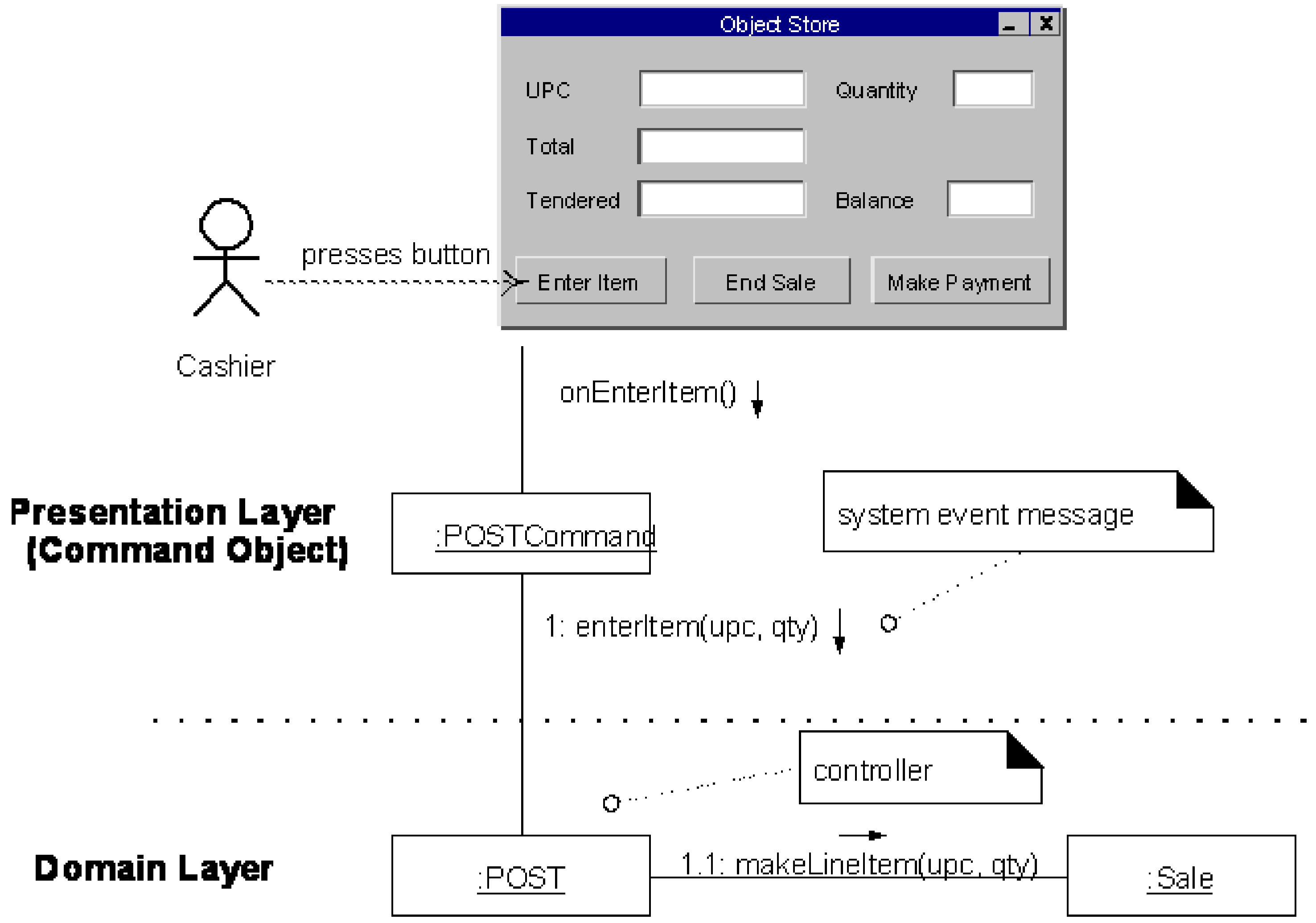
The controller object: two alternate solutions

- 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

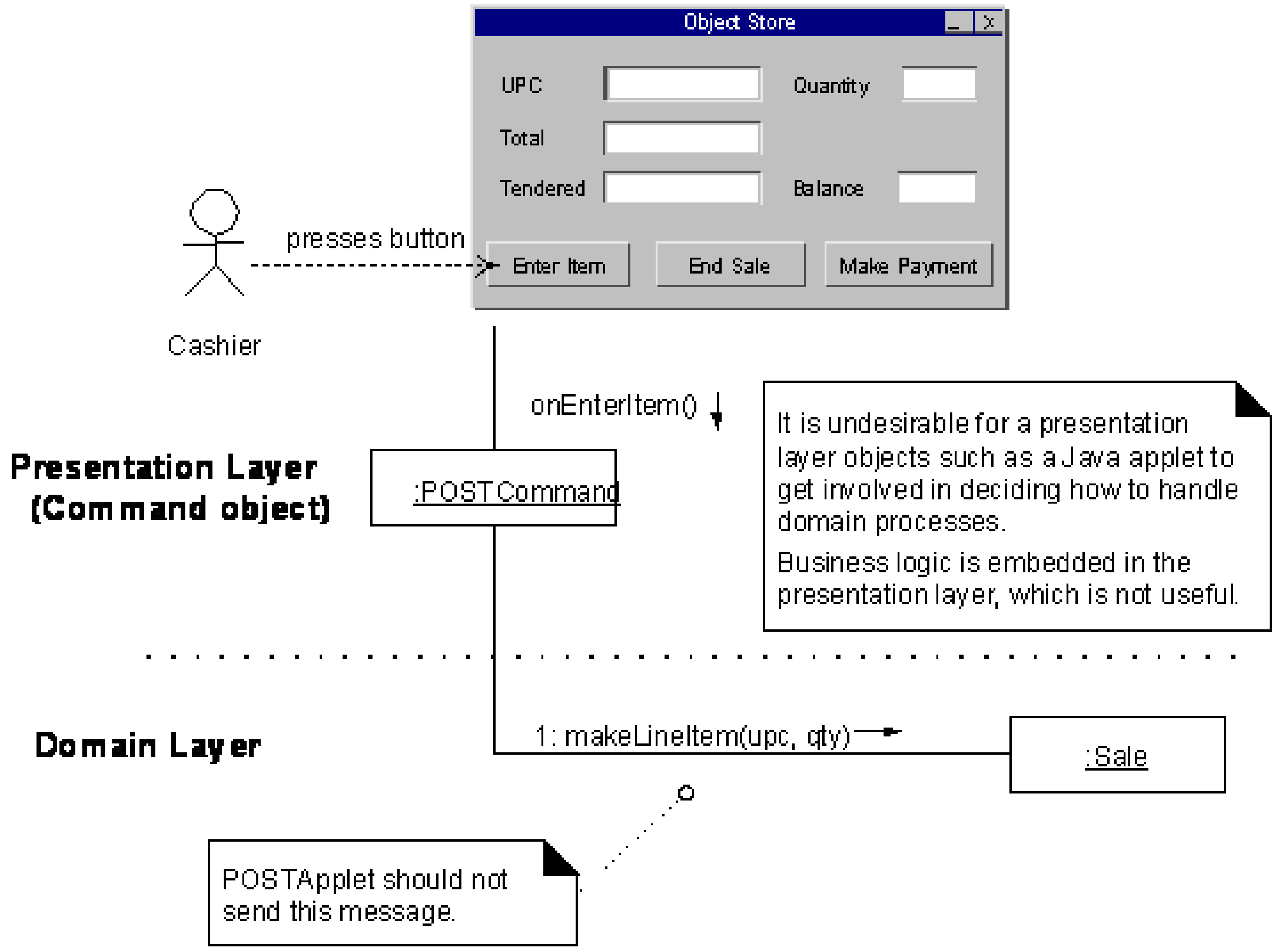
- System events in Buy Items use case
  - enterItem()
  - endSale()
  - makePayment()

Good design
- presentation layer decoupled from problem domain

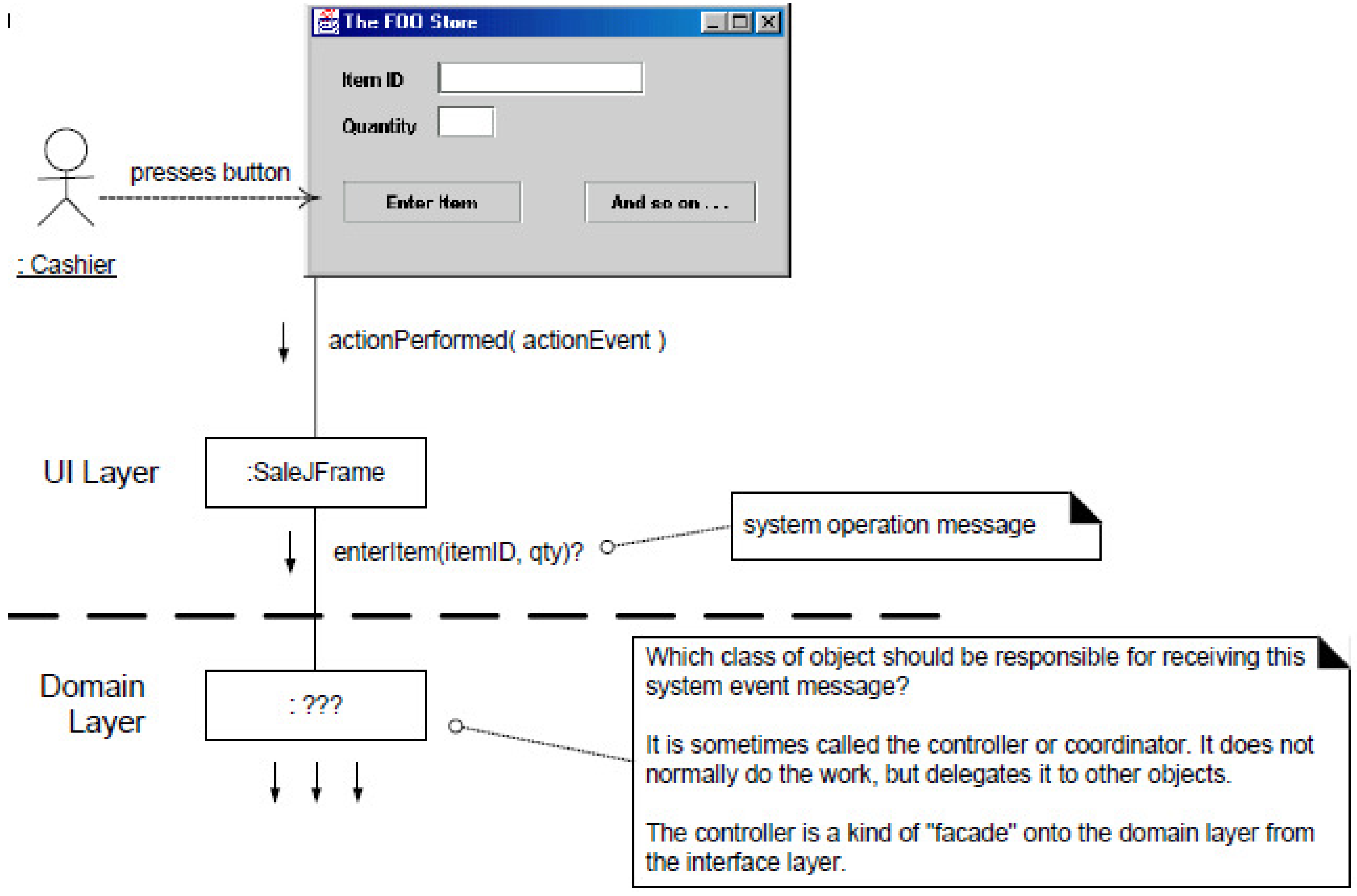


Bad design

– presentation layer coupled to problem domain

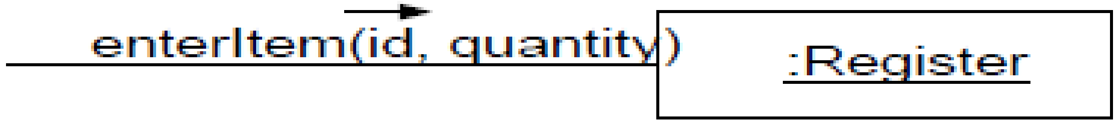

Object Store
UPC
Quantity
Total
Tendered
Balance
Enter Item
End Sale
Make Payment
presses button
Cashier
onEnterItem()
Presentation Layer
(Command object)
:POSTCommand
It is undesirable for a presentation layer objects such as a Java applet to get involved in deciding how to handle domain processes.
Business logic is embedded in the presentation layer, which is not useful.
Domain Layer
1: makeLineItem(upc, qty)
:Sale
POSTApplet should not send this message.

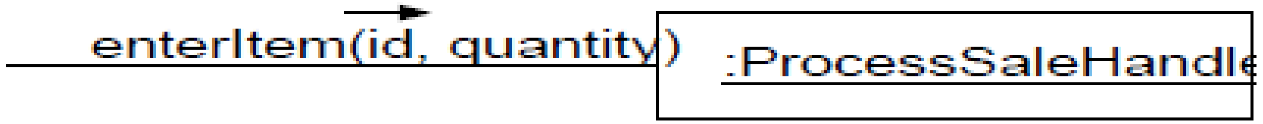
But then: What object should be the controller for enterItem?


The FOO Store
Item ID
Quantity
Enter Item
And so on . . .
presses button
: Cashier
actionPerformed(actionEvent)
UI Layer
:SaleJFrame
enterItem(itemID, qty)?
system operation message
Domain Layer
: ???
Which class of object should be responsible for receiving this system event message?
It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects.
The controller is a kind of "facade" onto the domain layer from the interface layer.

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.

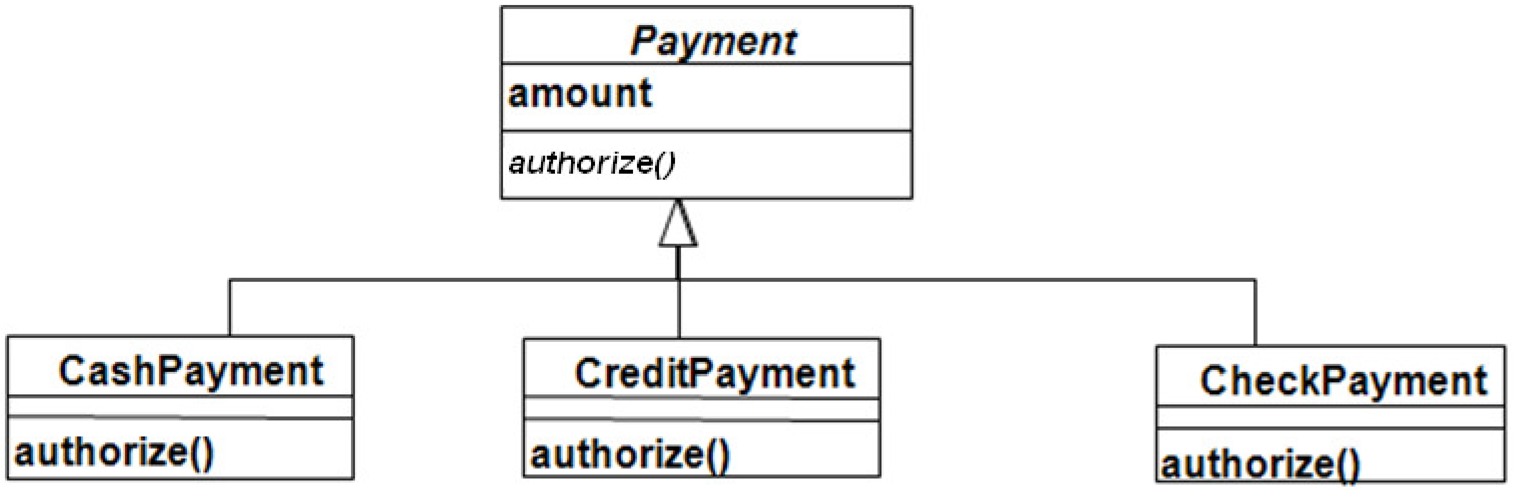
The nine GRASP Patterns

- 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.

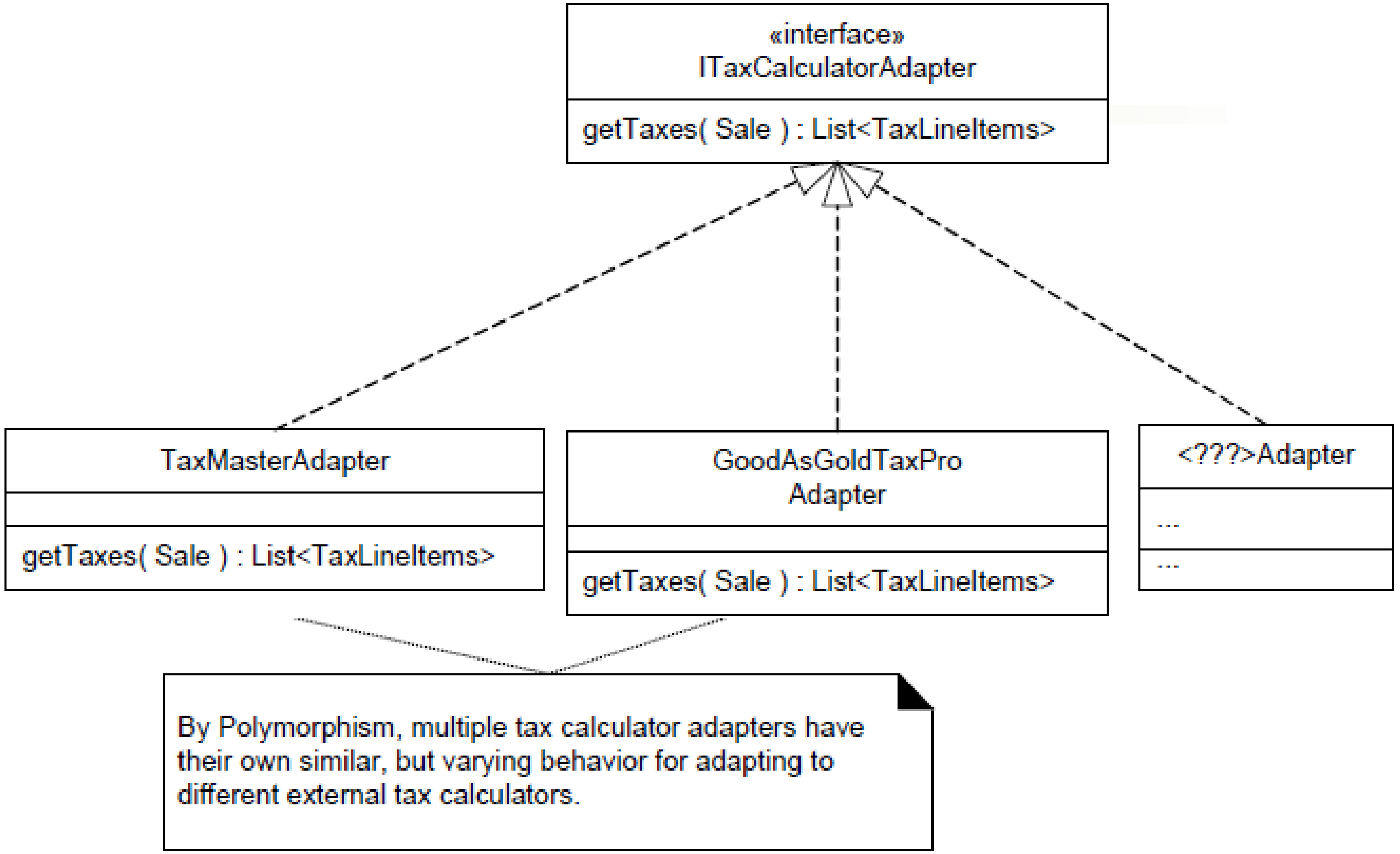
Broader use of polymorphism

- 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?’”