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 !→ 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)

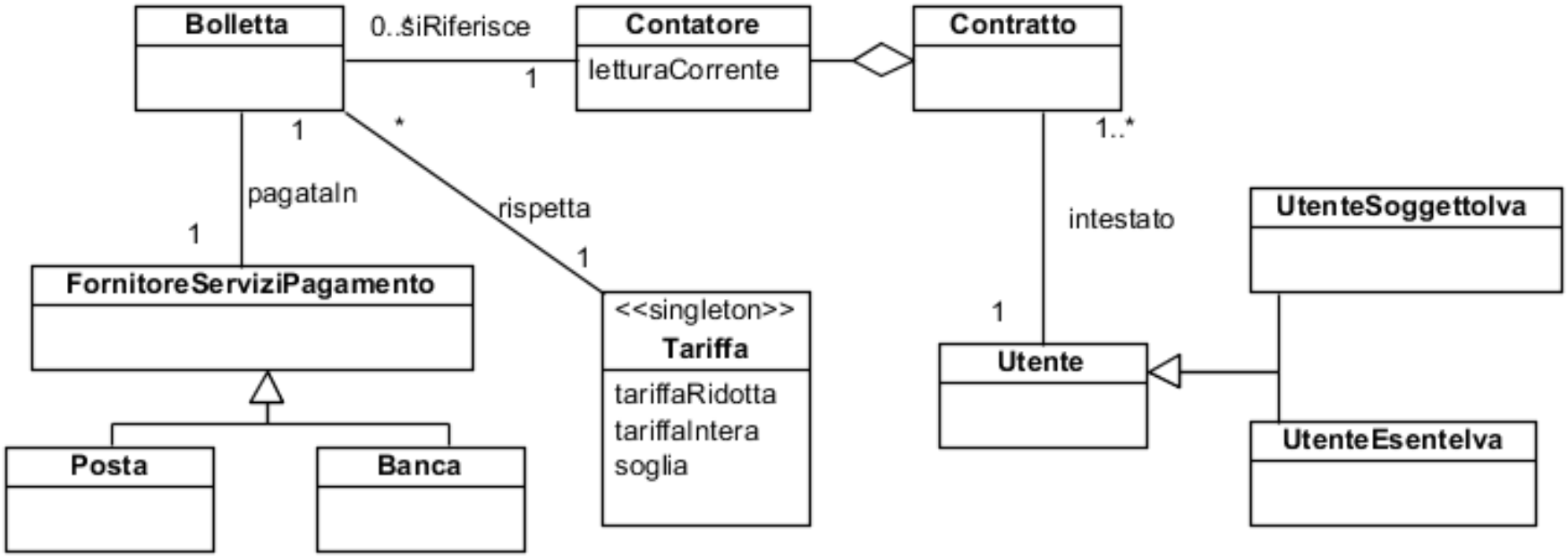
- GRASP
  - General Responsibility Assignment Software Patterns
  - First four:
    - Creator
    - Information Expert
    - High Cohesion
    - Low Coupling

- HOMEWORK

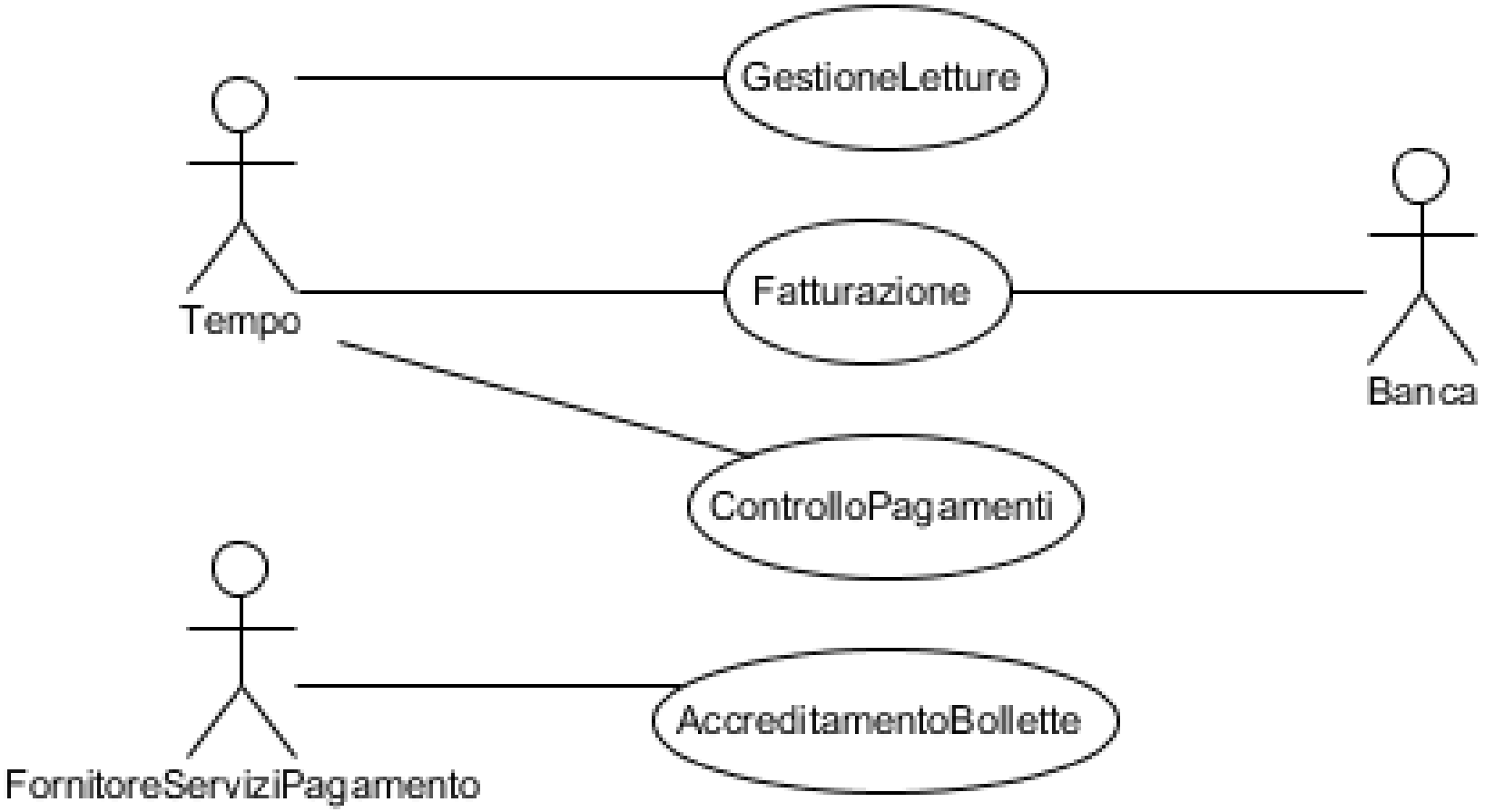
HOMEWORK

- Il sistema di fatturazione di una società di distribuzione di energia elettrica si basa sull'interazione tra diversi soggetti. Questo documento, destinato all'attenzione del responsabile sviluppo software, presenta un'analisi del problema basata su informazioni ottenute da interviste effettuate negli ultimi due mesi.
- La società di distribuzione (Società) eroga energia elettrica a utenti collegati all'impianto di distribuzione mediante un allacciamento controllato da un dispositivo elettronico di misura (contatore) in grado di registrare il consumo di energia, di fornire a richiesta della centrale di bassa potenza il valore della lettura corrente dei consumi, di segnalare eventuali malfunzionamenti nell'impianto elettrico installato presso l'utenza. La centrale di bassa potenza periodicamente raccoglie le letture e le invia al sistema di fatturazione della Società che provvede al calcolo dell'importo relativo al consumo registrato dal contatore e all'emissione di una bolletta o fattura. Il calcolo dell'importo è effettuato considerando il regime fiscale applicato a ogni utente (esente IVA o soggetto a IVA) che prevede, quando applicabile, un'imposizione determinata da un'aliquota (aliquota IVA, attualmente pari al 20%). Inoltre, nel rispetto della normativa vigente, il calcolo dell'importo deve considerare una soglia (cosiddetta di consumo sociale) sotto la quale deve essere applicata una tariffa ridotta (costo unitario sociale). Per consumi che superano questa soglia, l'importo deve essere calcolato applicando la tariffa piena (costo unitario). Una volta emessa, la bolletta è spedita al domicilio dell'utente che può provvedere al pagamento presso un qualsiasi ufficio postale (Posta). Nel caso in cui l'utente abbia domiciliato le bollette presso un istituto bancario, la bolletta è inviata anche all'istituto (Banca) che provvede automaticamente al suo pagamento alla data di scadenza indicata. In questo caso, la bolletta inviata all'utente è stampata in modo che non possa essere pagata presso un ufficio postale. Un utente può essere intestatario di più contratti, ognuno associato a un contatore. Periodicamente, la Società provvede alla verifica dei pagamenti delle bollette, mediante un processo di accreditamento. I pagamenti effettuati presso la Banca e la Posta sono incrociati con le bollette emesse. In casi di morosità grave, la Società si riserva il diritto di sospendere l'erogazione dell'energia elettrica.

DOMAIN


Bolletta
0..*
siRiferisce
1
Contatore
letturaCorrente
Contratto
1
*
1..*
pagataIn
1
rispetta
1
intestato
UtenteSoggettoIva
FornitoreServiziPagamento
<<singleton>>
Tariffa
tariffaRidotta
tariffaIntera
soglia
1
Utente
Posta
Banca
UtenteEsenteIva

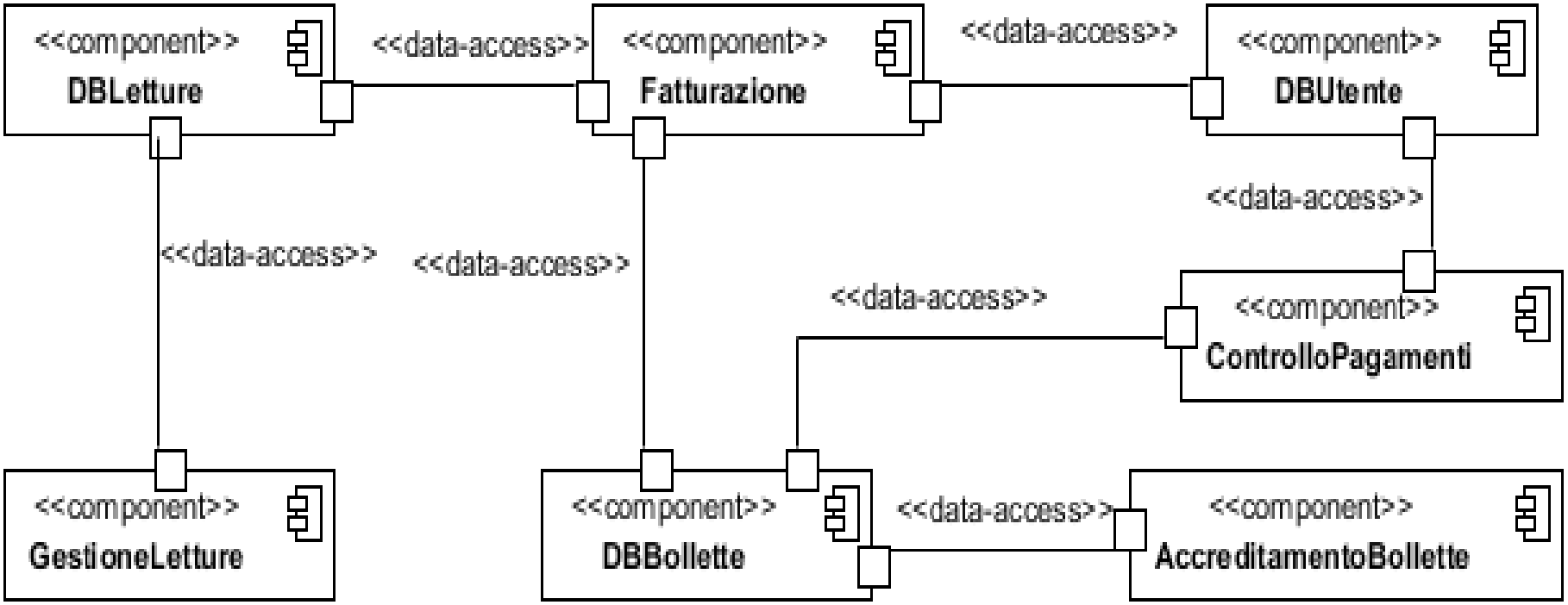
USE CASES


GestioneLetture
Tempo
Fatturazione
Banca
ControlloPagamenti
FornitoreServiziPagamento
AccreditamentoBollette

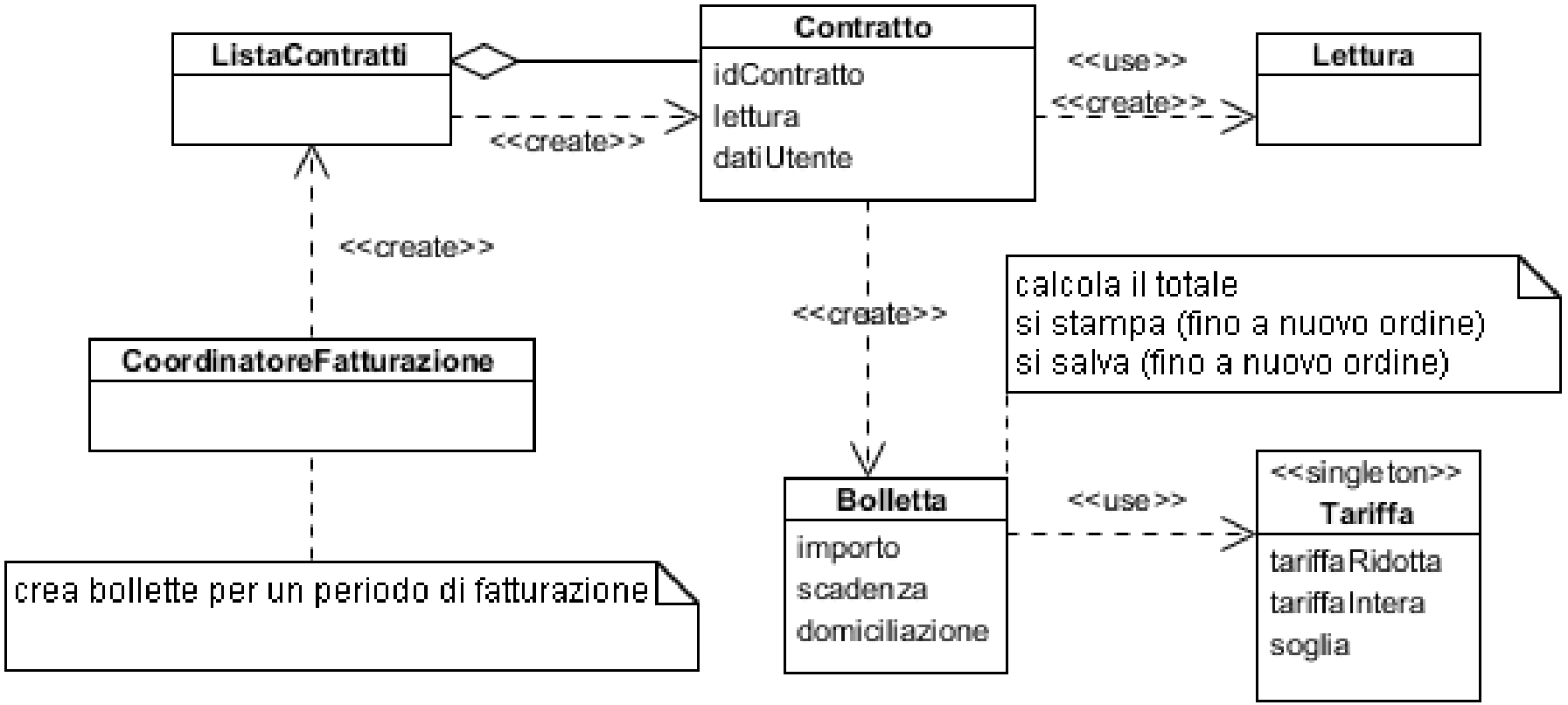
USE CASES (cont'd)

- *GestioneLetture*: periodicamente, il sistema preleva dai Contatori le letture del consumo e le registra in una base di dati, DBLetture.
- *Fatturazione*: periodicamente, il sistema stampa, per l'invio all'Utente, usando dati prelevati dal DBUtente, le bollette relative all'ultima lettura, e le registra nel DBBollette; nel caso di bollette domiciliate le invia pure elettronicamente alla Banca, per il pagamento.
- *AccreditamentoBollette*: Il sistema riceve elettronicamente dalla Banca e dalla Posta i pagamenti delle bollette, e li registra nel DBBollette.
- *ControlloPagamenti*: periodicamente, estraendo le informazioni dal DBBollette, individua per i clienti morosi e stampa i solleciti, usando i dati del DBUtenti.

C&C


<<component>>
DBLetture
<<data-access>>
<<component>>
Fatturazione
<<data-access>>
<<component>>
DBUtente
<<data-access>>
<<data-access>>
<<data-access>>
<<component>>
ControlloPagamenti
<<component>>
GestioneLetture
<<component>>
DBBollette
<<data-access>>
<<component>>
AccreditamentoBollette

Design of Fatturazione: first draft, applying first 4 GRASP



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

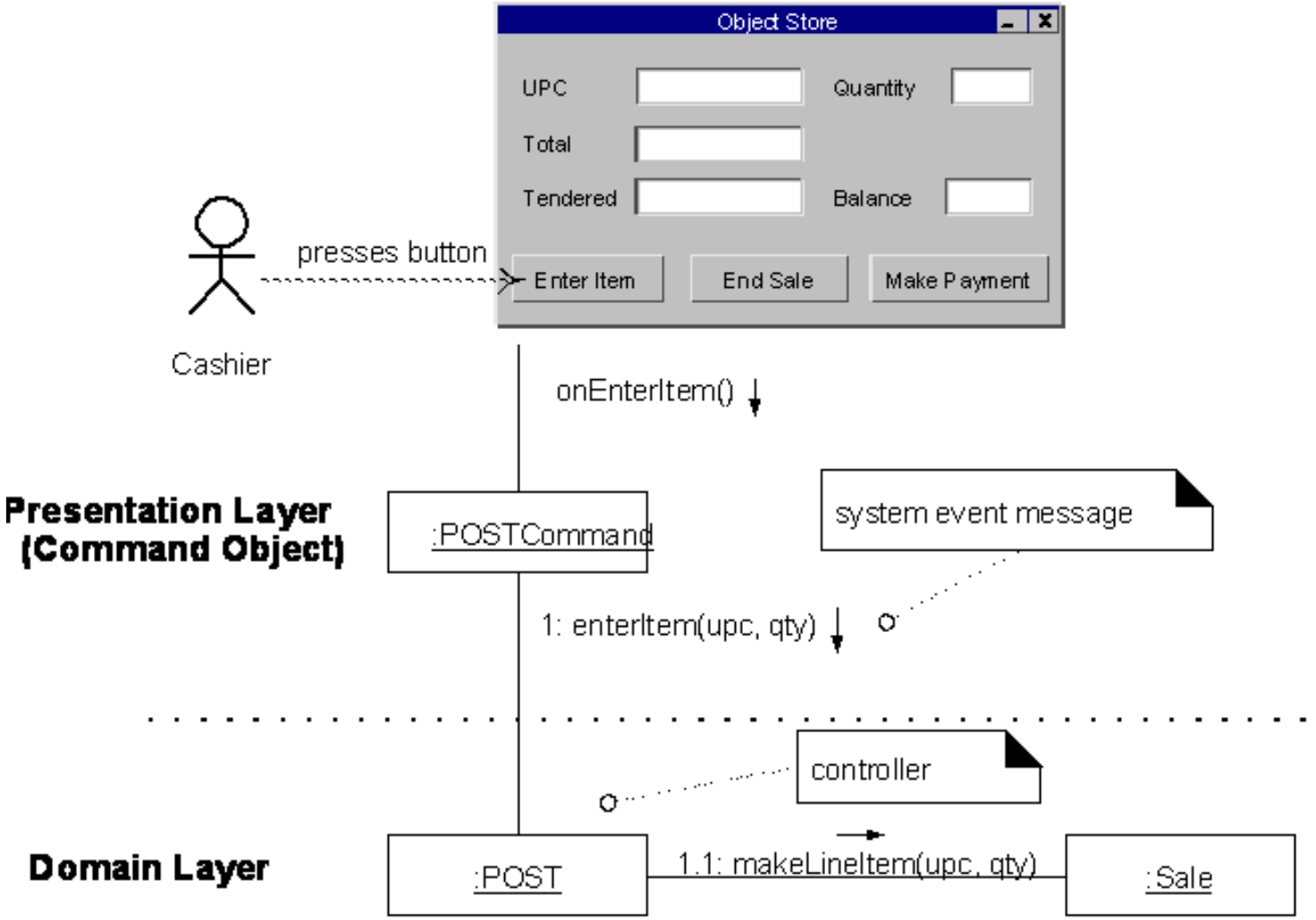
- Assign the responsibility for receiving or handling a system event message to a controller class that:
  - Represents the overall system, device, or subsystem
    - This class is called façade controller.
  - Represents a use case scenario within which the s. e. occurs
    - Often this class is named <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()

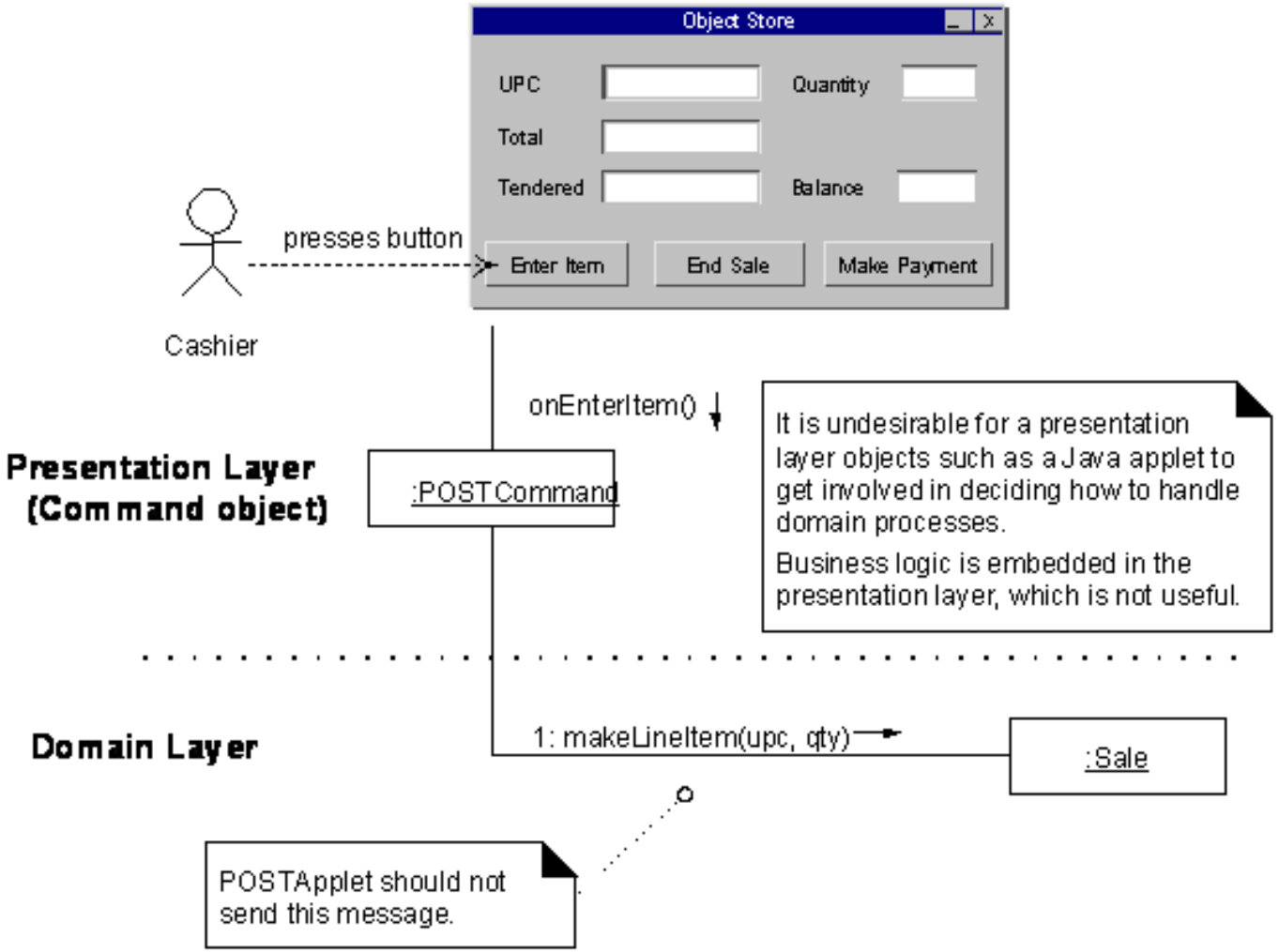
Good design

- presentation layer decoupled from problem domain

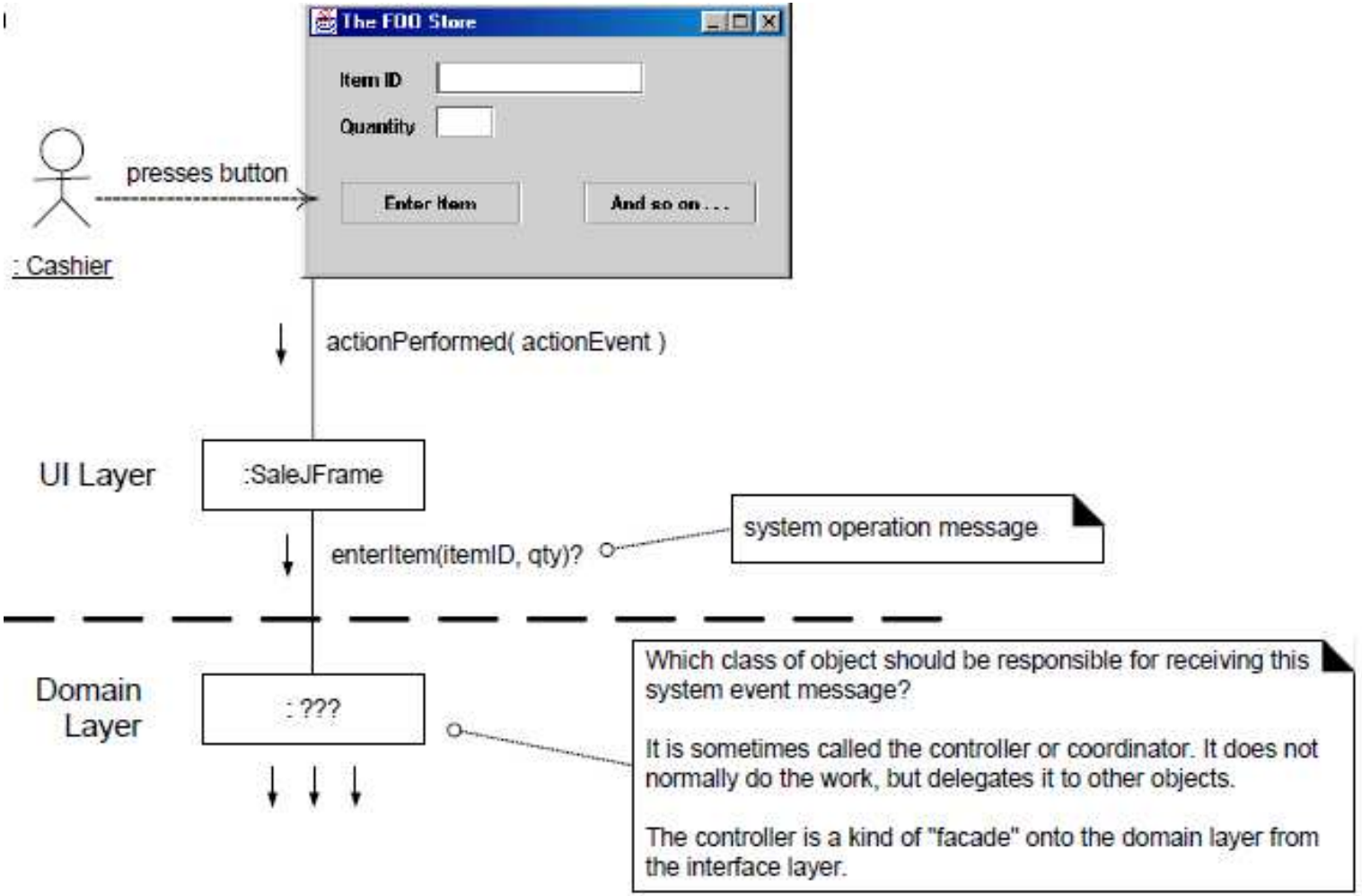


Bad design

– presentation layer coupled to problem domain

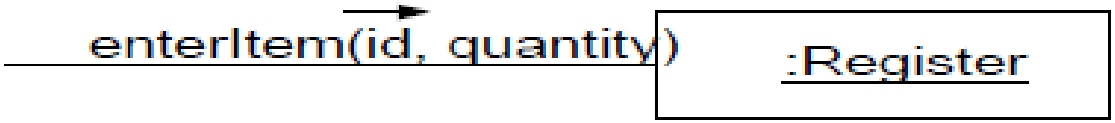


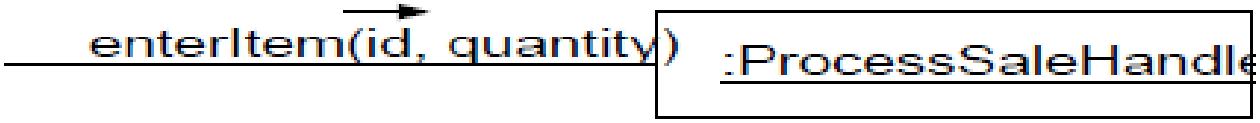
But then: What object should be the controller for enterItem?



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

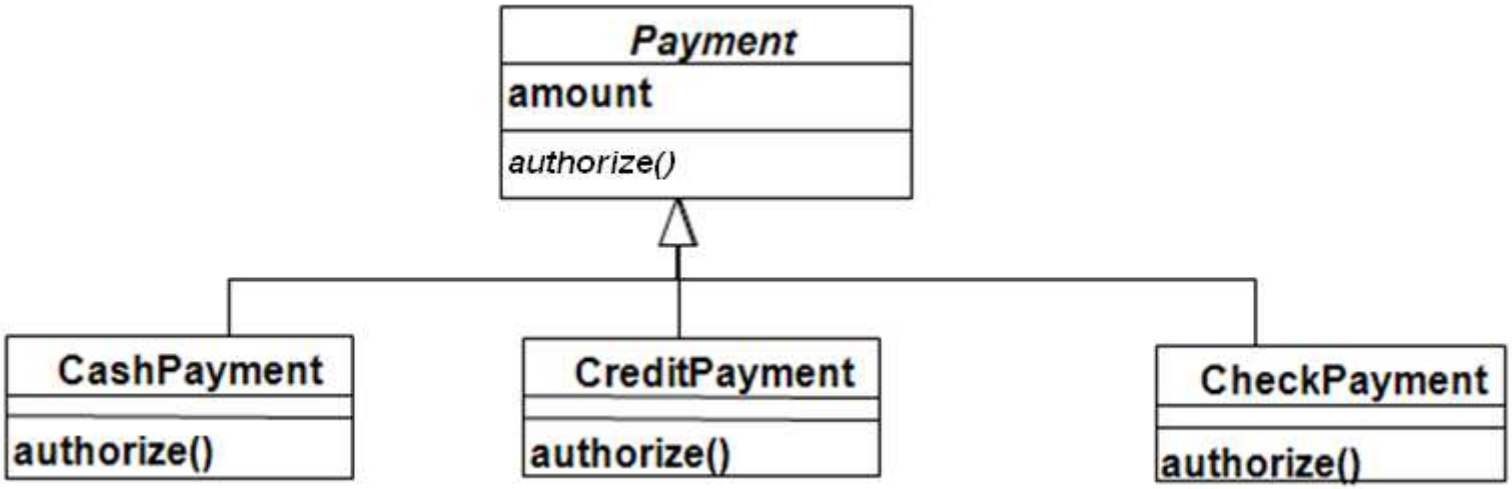
Def of polymorphism

- 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

- 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.

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)

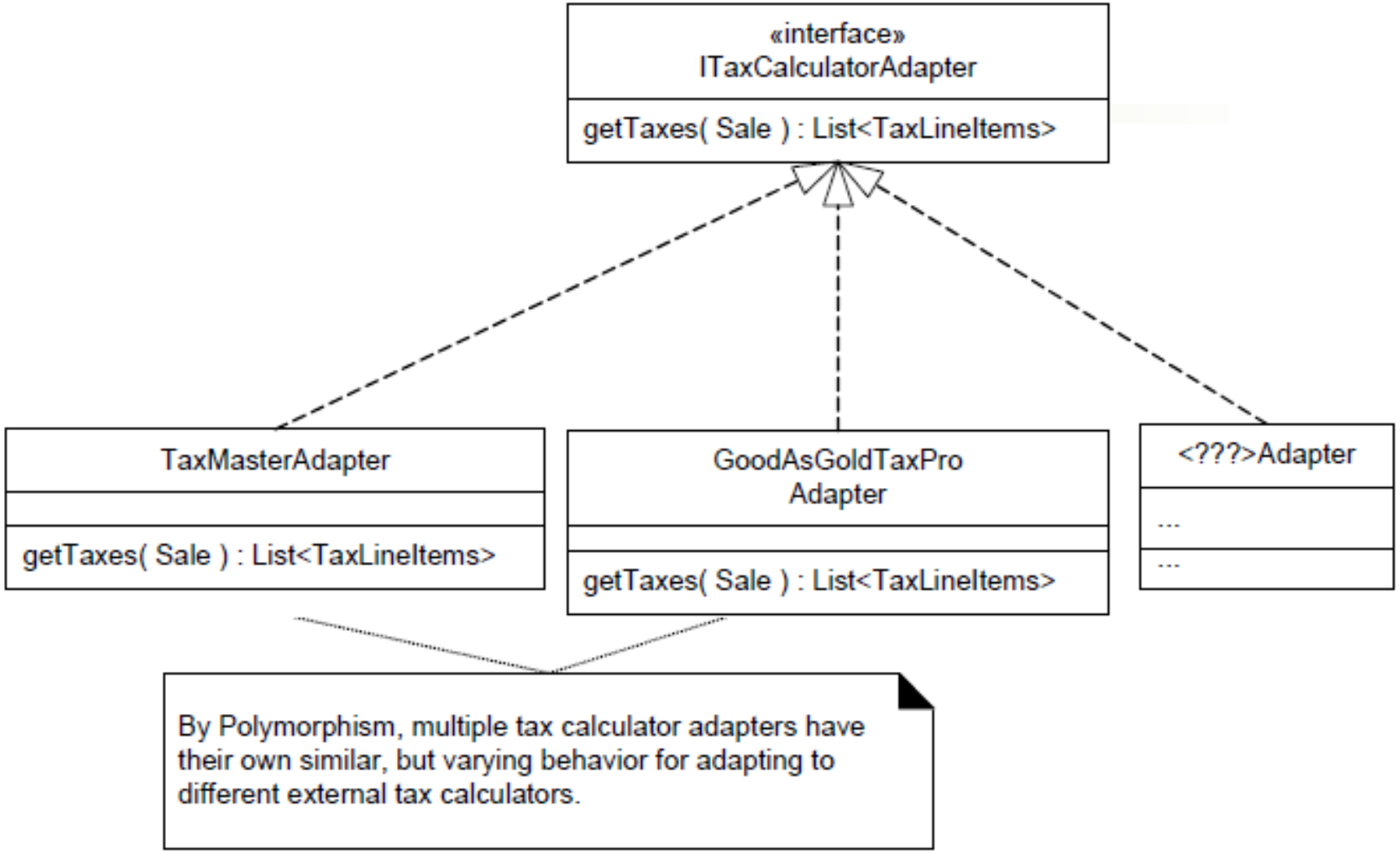
Broader use of polymorphism

- 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 does not 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

- Save / Sale indirection
- 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

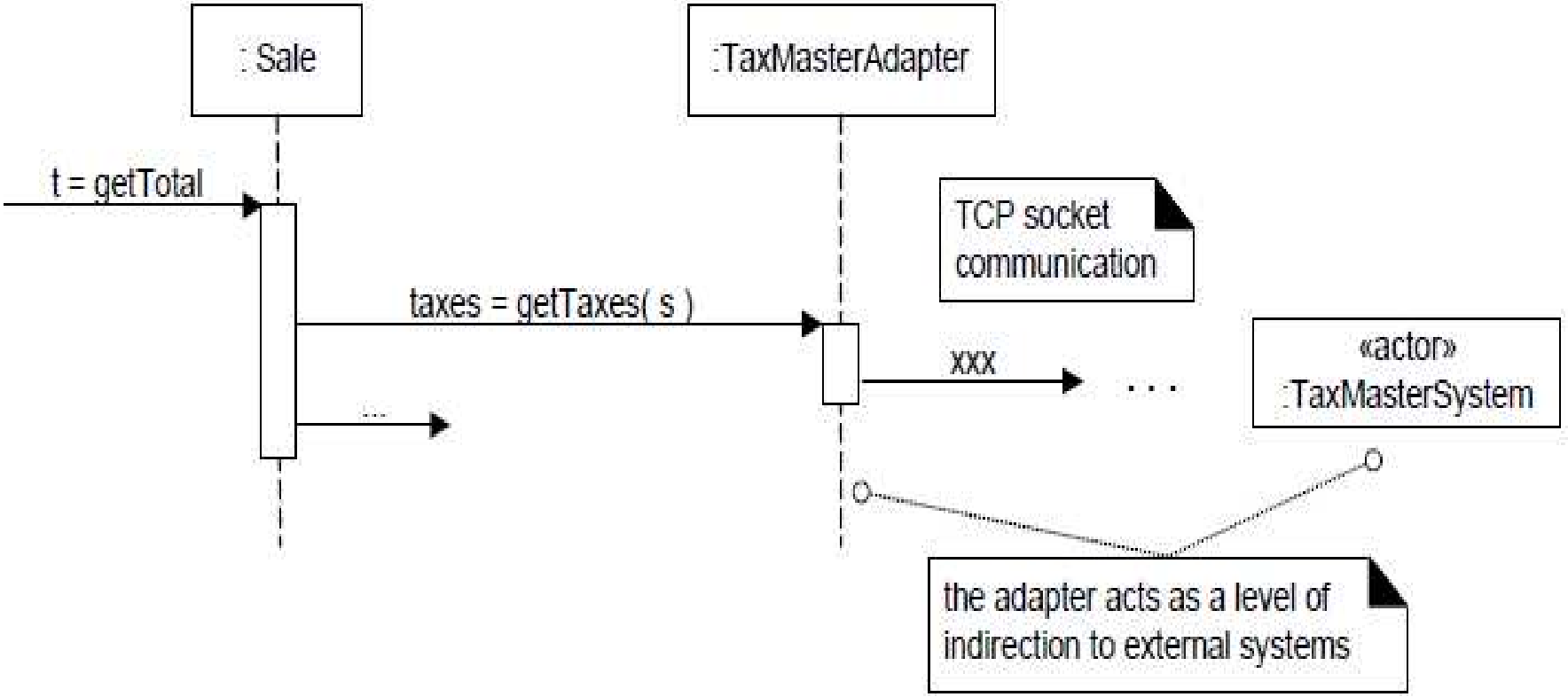
Indirection

- 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.

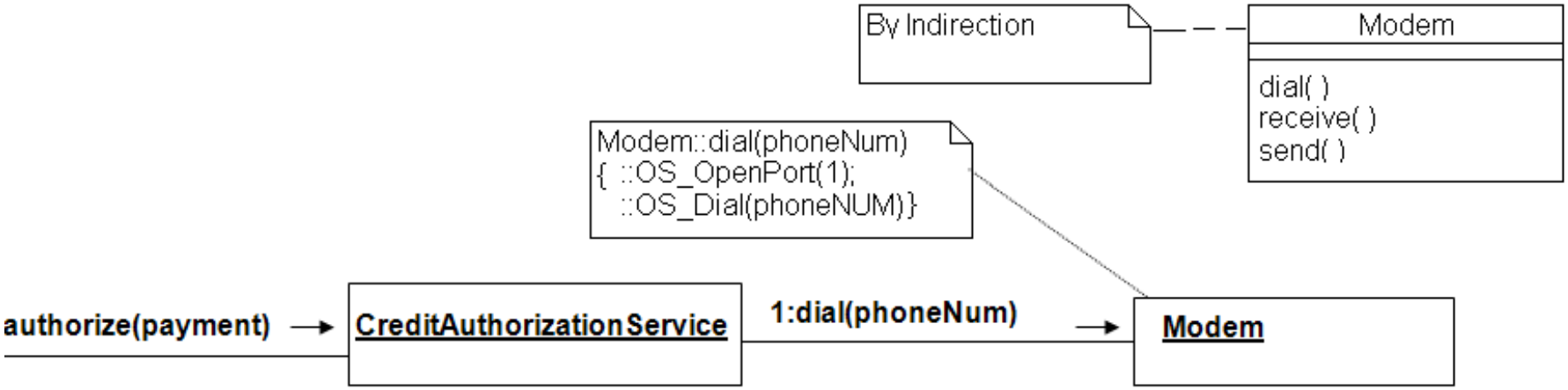
Example : PersistentStorage

- 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.

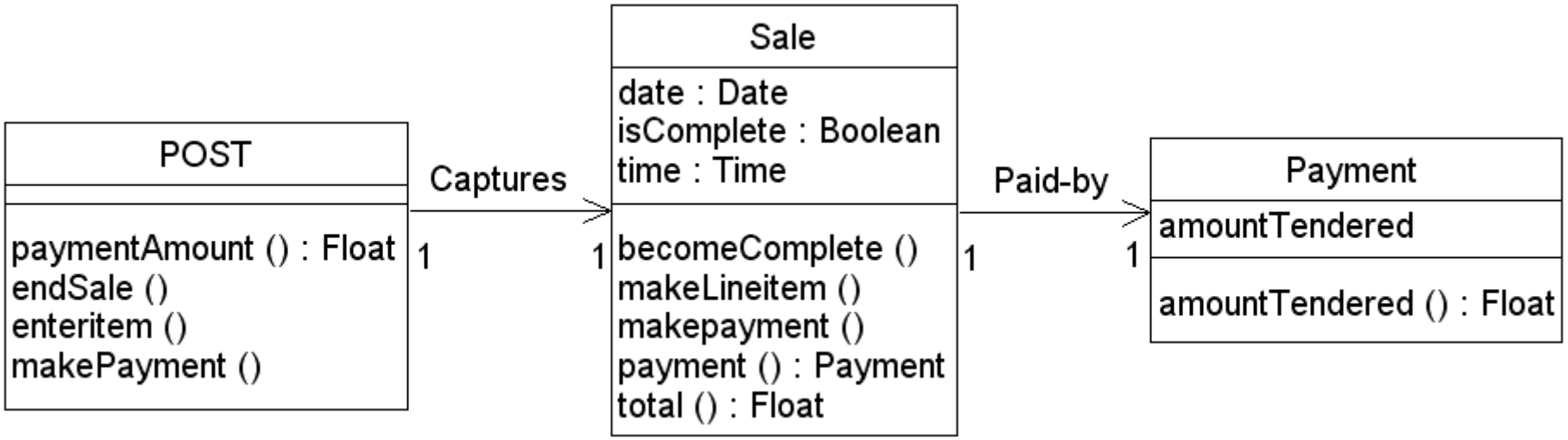
Protected Variation: Law of Demeter

- 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

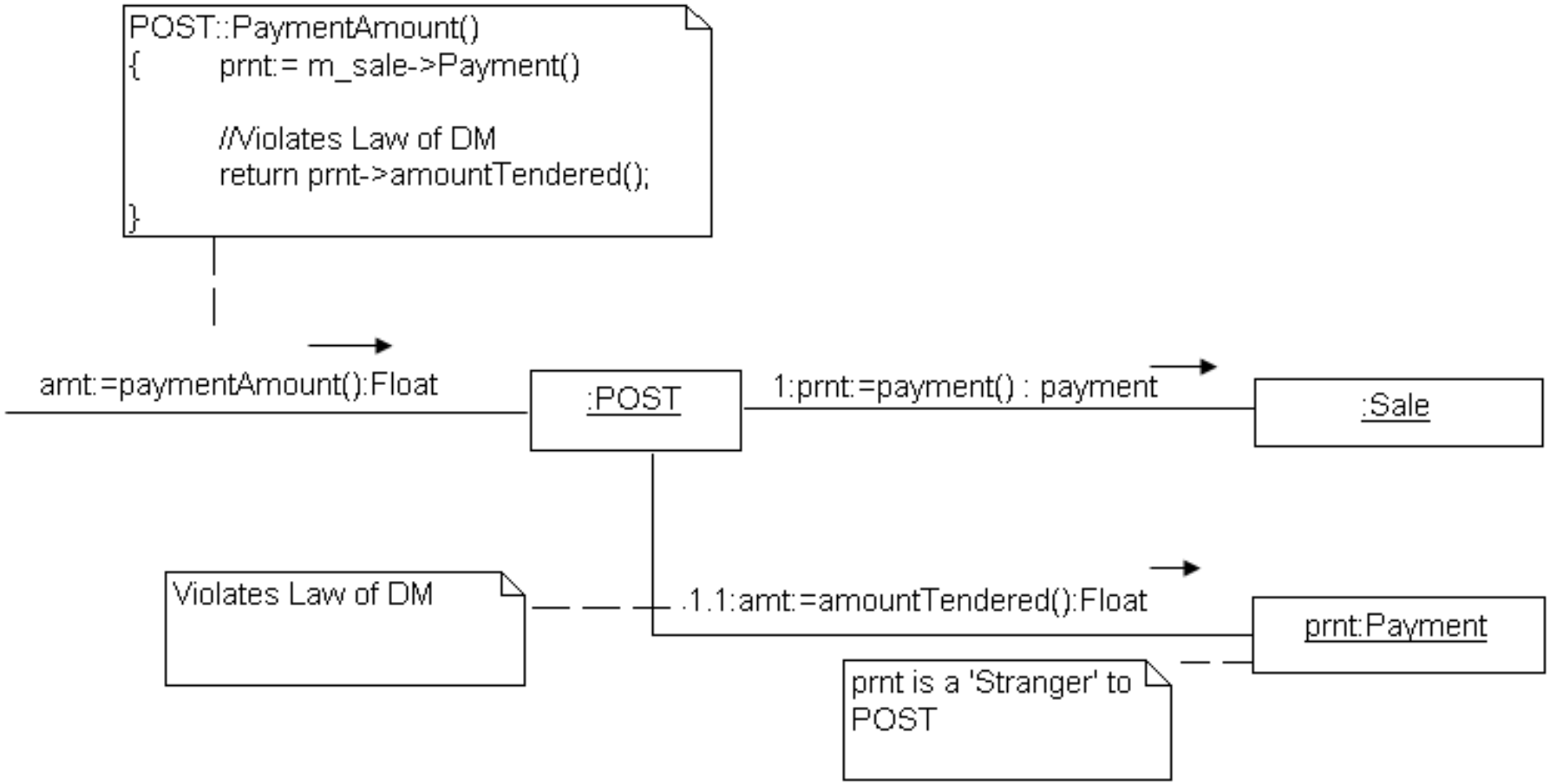
Protected Variation: Law of Demeter

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

Law of Demeter : Example

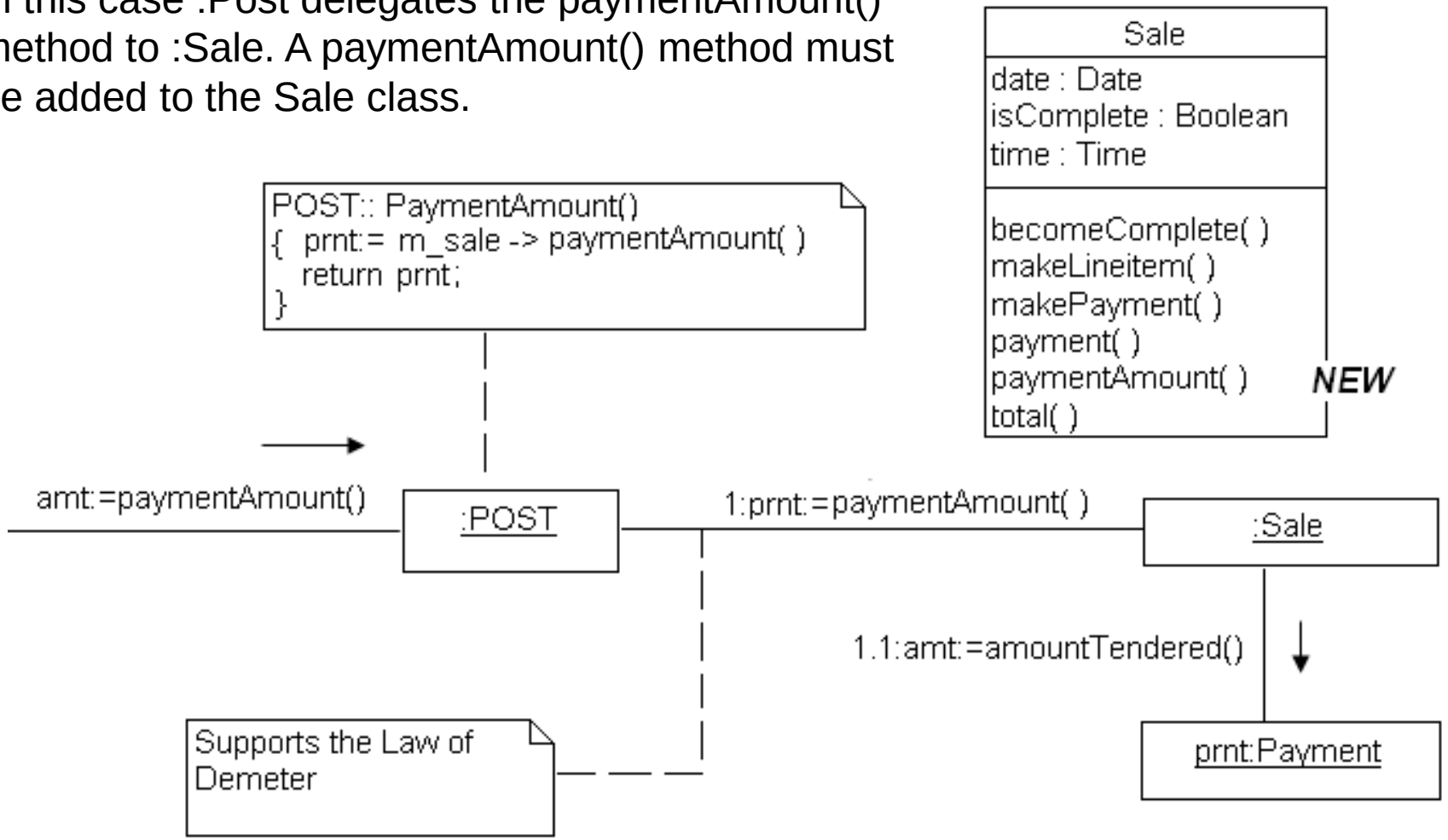


Violating Law of Demeter: Example



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