Software Design Patterns

- An overlap exists between architectural patterns and implementation-level patterns
- Just as certain basic concepts are behind system-level architectural patterns, so also certain basic concepts underlie software design patterns
- Look now at these basic concepts as described by Larman

Software Design Patterns

The idea of software design patterns is attributed to software development visionaties like Kent Beck and Ward Cunningham in the late 80s

The book

- Design Patterns by Gamma, Helm, Johnson, Vlissides (1995)

is the original “bible” for software design patterns.

The authors are called the Gang of Four, and the 23 patterns defined in the book the Gang of Four or GoF patterns

GRASP
(General Responsibility Assignment Software Patterns/Principles)

Larman develops a set of patterns that are really more general underlying principles, and several of these underlie more specific software design patterns. As well as his own GRASP patterns, he relates his general principles to several GoF patterns.

The following overview and examples are taken from the Larman textbook.

GRASP
(General Responsibility Assignment Software Patterns/Principles)

- GRASP: 5 basic patterns
  - Information Expert
  - Creator
  - Controller
  - Low Coupling (evaluation pattern)
  - High Cohesion (evaluation pattern)

  Plus:
  - Polymorphism,
  - Pure fabrication
  - Indirection
  - Protected Variation

Plus GOF: Adaptor, Factory, Singleton, Strategy, Facade

Assigning Responsibilities

Core design activity:

- The identification of objects and responsibilities and providing a solution in terms of an interaction diagram
- this is the creative part where the choices lead to good or bad designs

cf. origins of O-O methodology; Smalltalk, Alan Kay

Responsibility Driven Design(RDD)

- Objects have responsibilities, an abstraction of what they do
- UML definition of responsibility:

  "a contract or obligation of a classifies"

Responsibilities, in general, of two kinds:

- Knowing
- Doing

Responsibility

- Doing responsibilities, include
  - Creating object or doing calculation
  - Initiating actions in other objects
  - Controlling and co-ordinating actiovities in other objects
- Knowing responsibilities, include knowing about:
  - Private encapsulated data
  - Related objects
  - Things it can derive or calculate

Assigning responsibility

- have an overview of responsibility for system operations (e.g. via contracts)
- need solution in terms of co-operating objects to achieve these
- design is
  - identification of suitable objects
  - assignment of responsibility

Responsibility Driven Design(RDD)

- RDD is a general metaphor for thinking about object-oriented design
- Involves thinking of software objects as similar to people with responsibilities who collaborate with other people to get work done.

- Results in viewing an OO design as a

community of collaborating responsible objects

How to assign responsibility?

Goal: to make a system that is easy to understand, maintain, reuse, extend …

- Exploration of design options, e.g. CRC cards
  - CRC (Class-Responsibility-Collaborator) cards
  - team brainstorming around table using index cards
  - Card: Class name, its Responsibilities, its Collaborators
  - Due to Kent Beck, Ward Cunningham

- GRASP patterns
- ...

Patterns

- Pattern
  - name
  - problem being solved
  - solution
- codified solution principles, idioms
- no new ideas
- communication benefit:
  - vocabulary; abstraction

General info on software design patterns separately; here deal with Larman's take on design patterns

Patterns for responsibility

- GRASP (General Responsibility Assignment Software Patterns/Principles)
- GRASP: 5 basic patterns
  - Information Expert
  - Creator
  - Controller
  - Low Coupling (evaluation pattern)
  - High Cohesion (evaluation pattern)

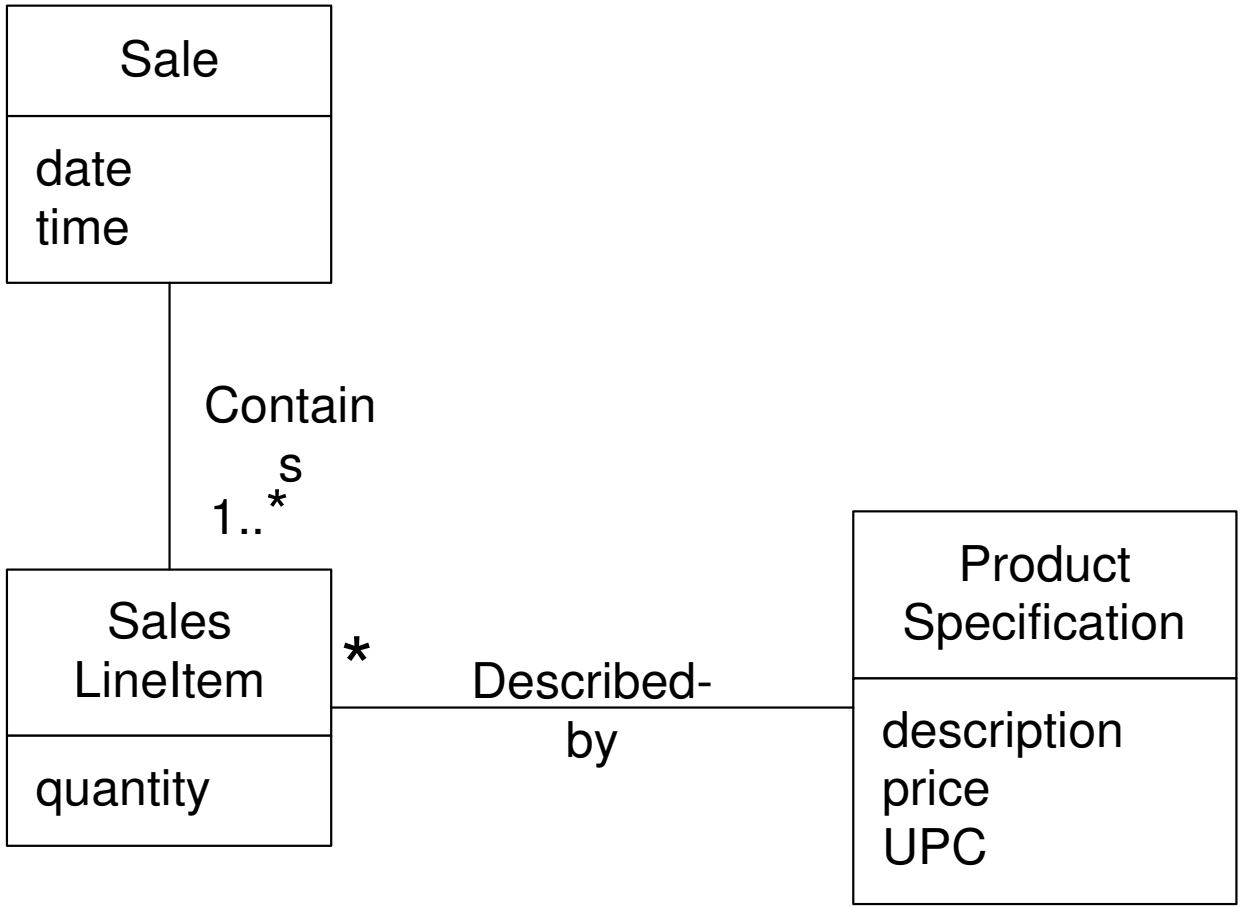
  Plus: Polymorphism, Pure fabrication, Indirection, Protected Variation

  Plus GOF: Adaptor, Factory, Singleton, Strategy, Facade

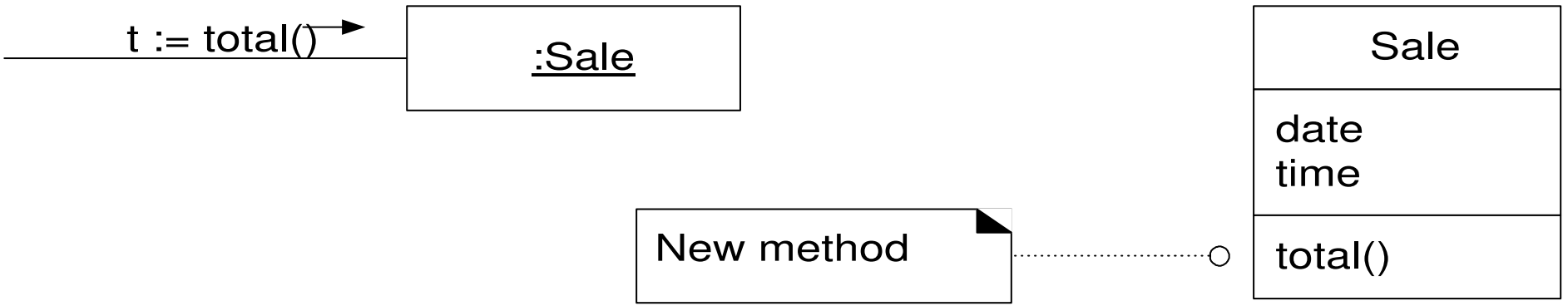
Information Expert

- Assign responsibility to class that has information necessary to fulfill responsibility
- Partial expert
  - needs collaboration
- Benefits
  - encapsulation; low coupling
  - distributes behaviour across classes with required info
- Most used pattern
- `Do It Myself' strategy --- Peter Coad

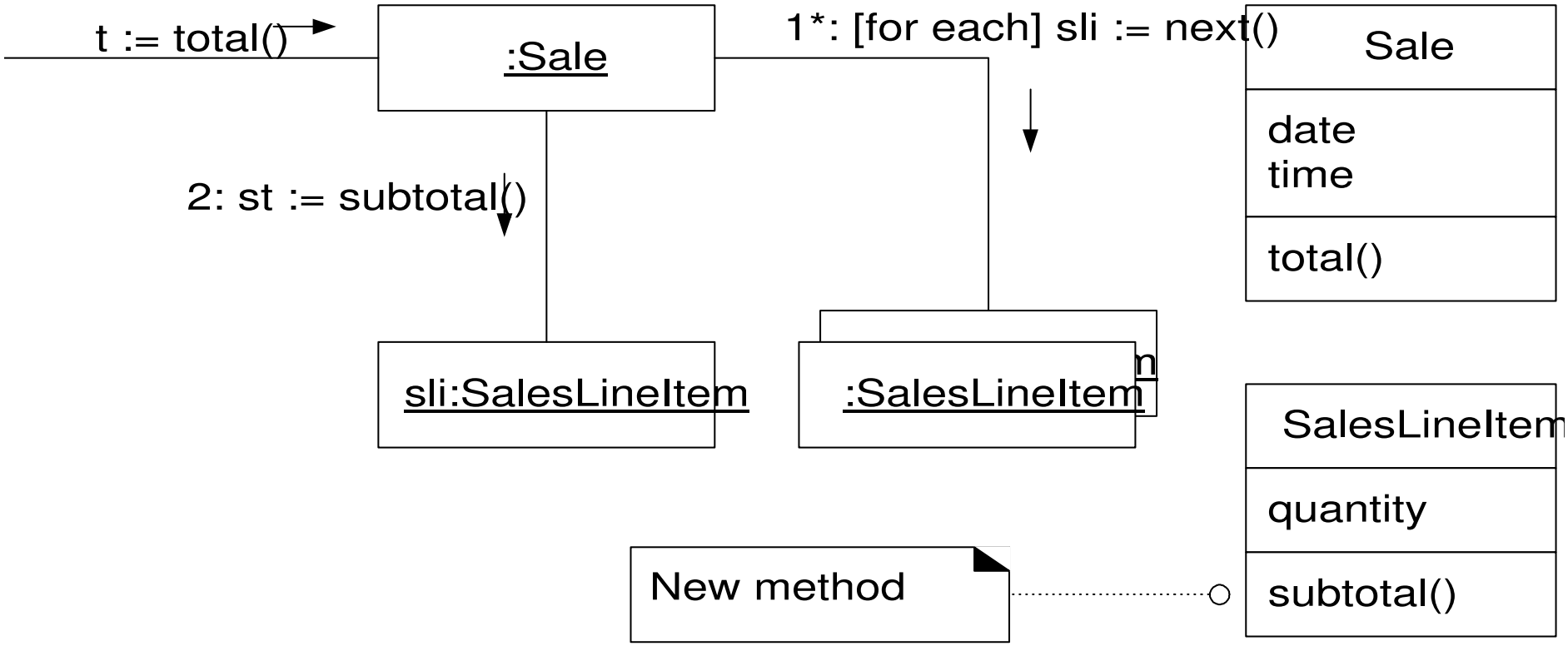
Part of Conceptual model

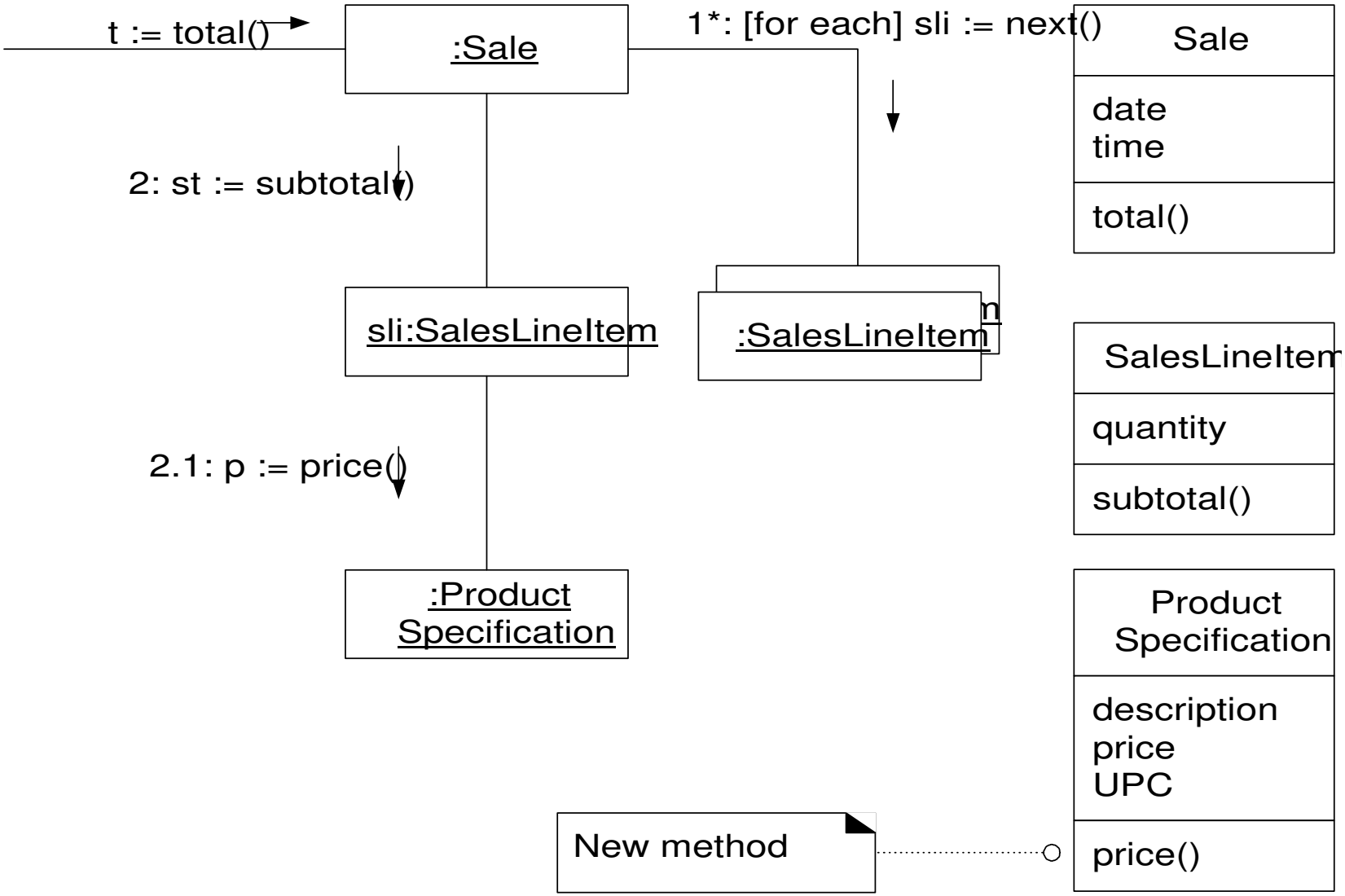


Who should be responsible for knowing the grand total of the sale?


t := total()
:Sale
Sale
date
time
total()
New method

Interaction(Communication) Diagram ...





Product Specification is the expert on price

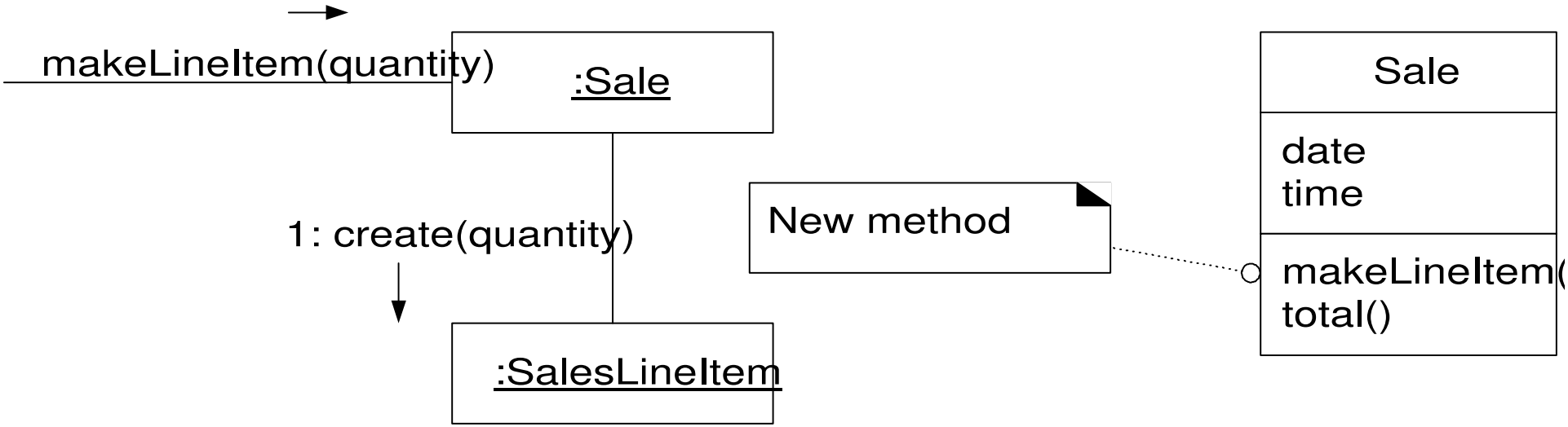
Assigning responsibility by Expert Pattern

Class	Responsibility
Sale	Knows sale total
SalesLineItem	Knows line item subtotal
ProductSpecification	Knows product price

Creator

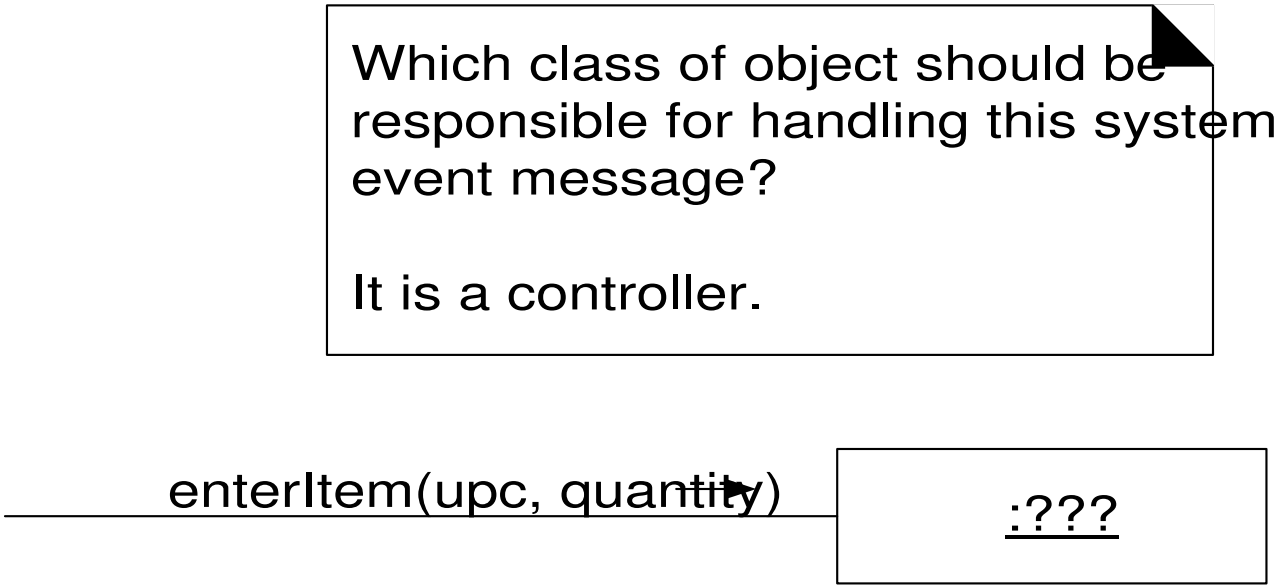
- Class B has responsibility to create instance of class A if
  - B contains A objects
  - B aggregates A objects
  - B has the initializing data for A
  - B records instances of A objects
  - B closely uses A objects

Assigning Responsibility based on Creator Pattern



Controller

- Assign responsibility for handling input system events to a class that
  - represents the overall ‘system’
  - represents the overall business or organisation (façade controller)
  - represents an active entity in the real-world that does this (role controller)
  - represents an artificial handler of all system events of a use case (use case controller)



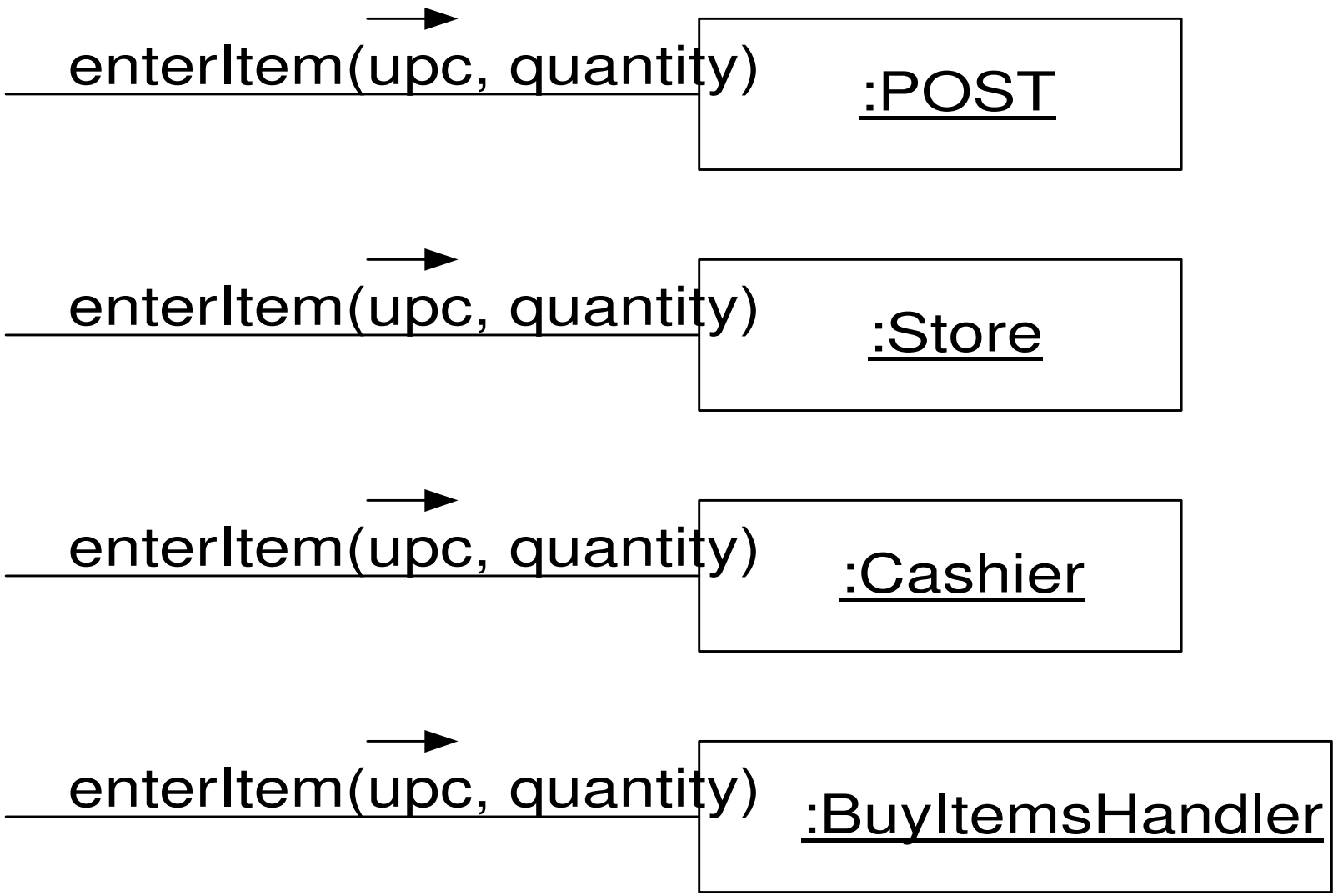
Controller Pattern choices:

System --- POST

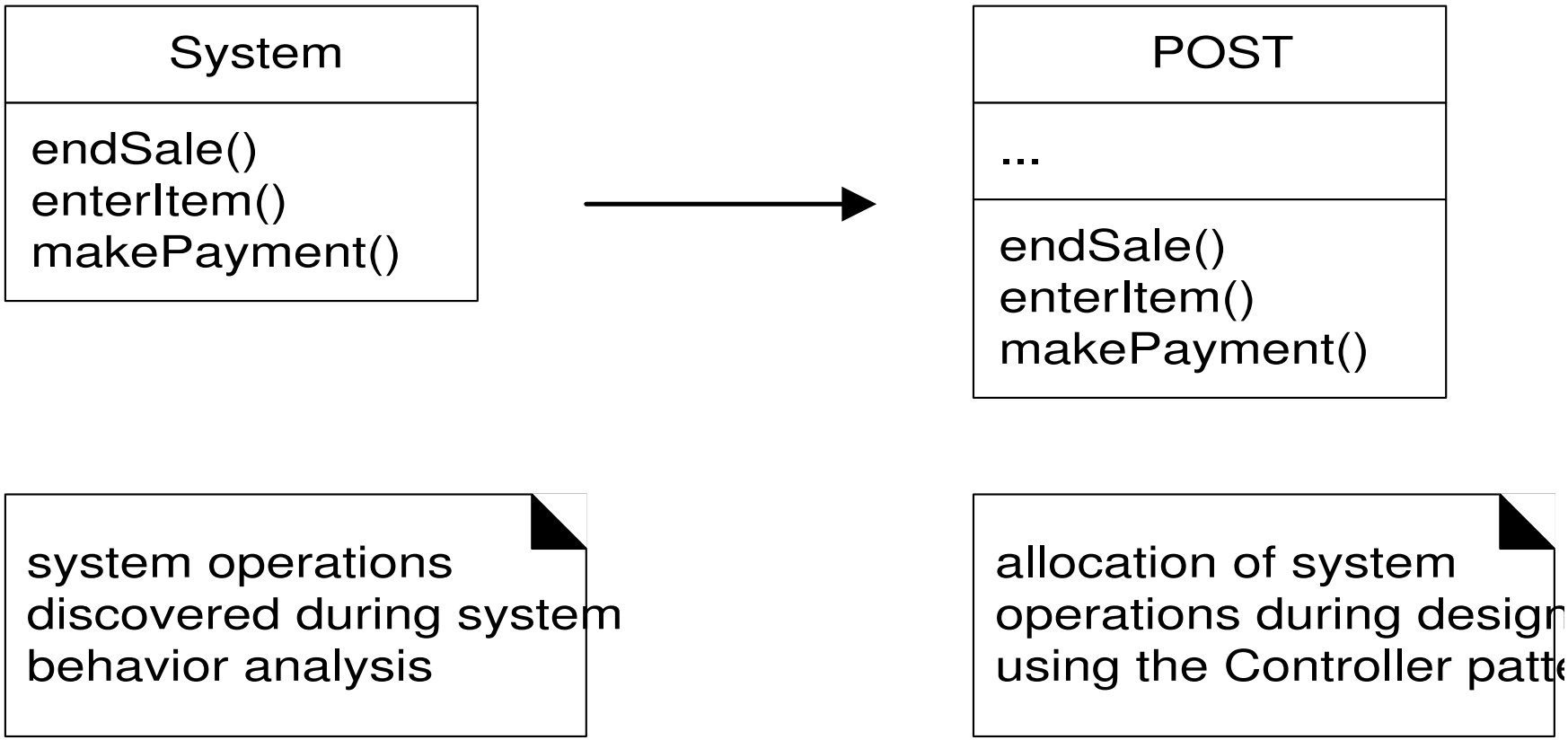
Business or organisation --- Store

Real-world controller --- Cashier

Use Case handler --- BuyItemsHandler


enterItem(upc, quantity)
:POST
enterItem(upc, quantity)
:Store
enterItem(upc, quantity)
:Cashier
enterItem(upc, quantity)
:BuyItemsHandler

Assigning System operations to one or more controller classes



Controller issues ...

- UI objects and presentation layer should not have responsibility for fulfilling system events
- problem of bloated controllers
  - single controller class handling all system events
- role-controllers (e.g. cashier object for handling makepayment) can be incohesive and avoid delegation

High Cohesion (evaluation pattern)

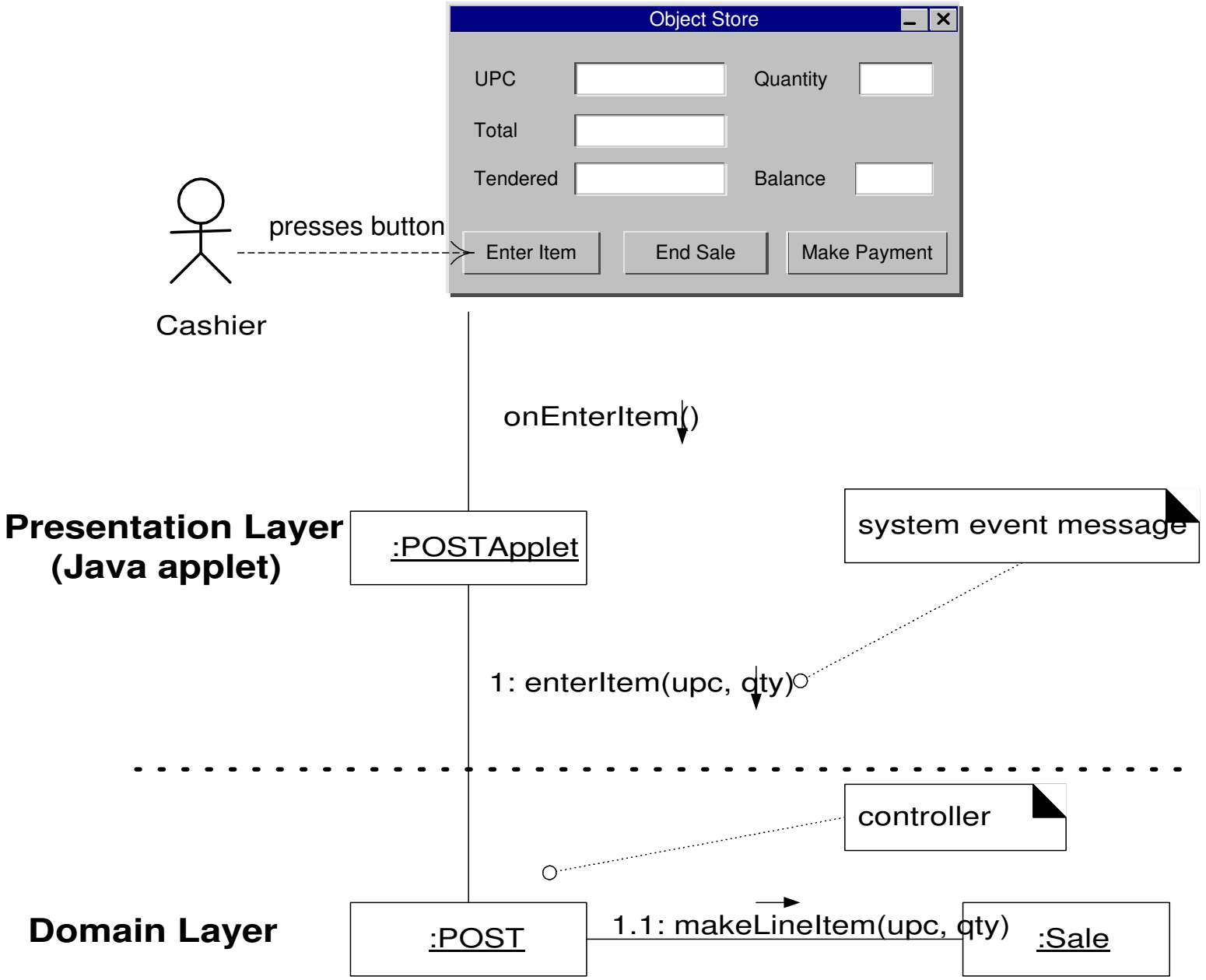
- “when the elements of a class (component) all work together to provide some well-bounded behaviour
- bad if a class is solely responsible for lots of different functions
- Assign responsibility to keep cohesion high

Low Coupling (evaluation pattern)

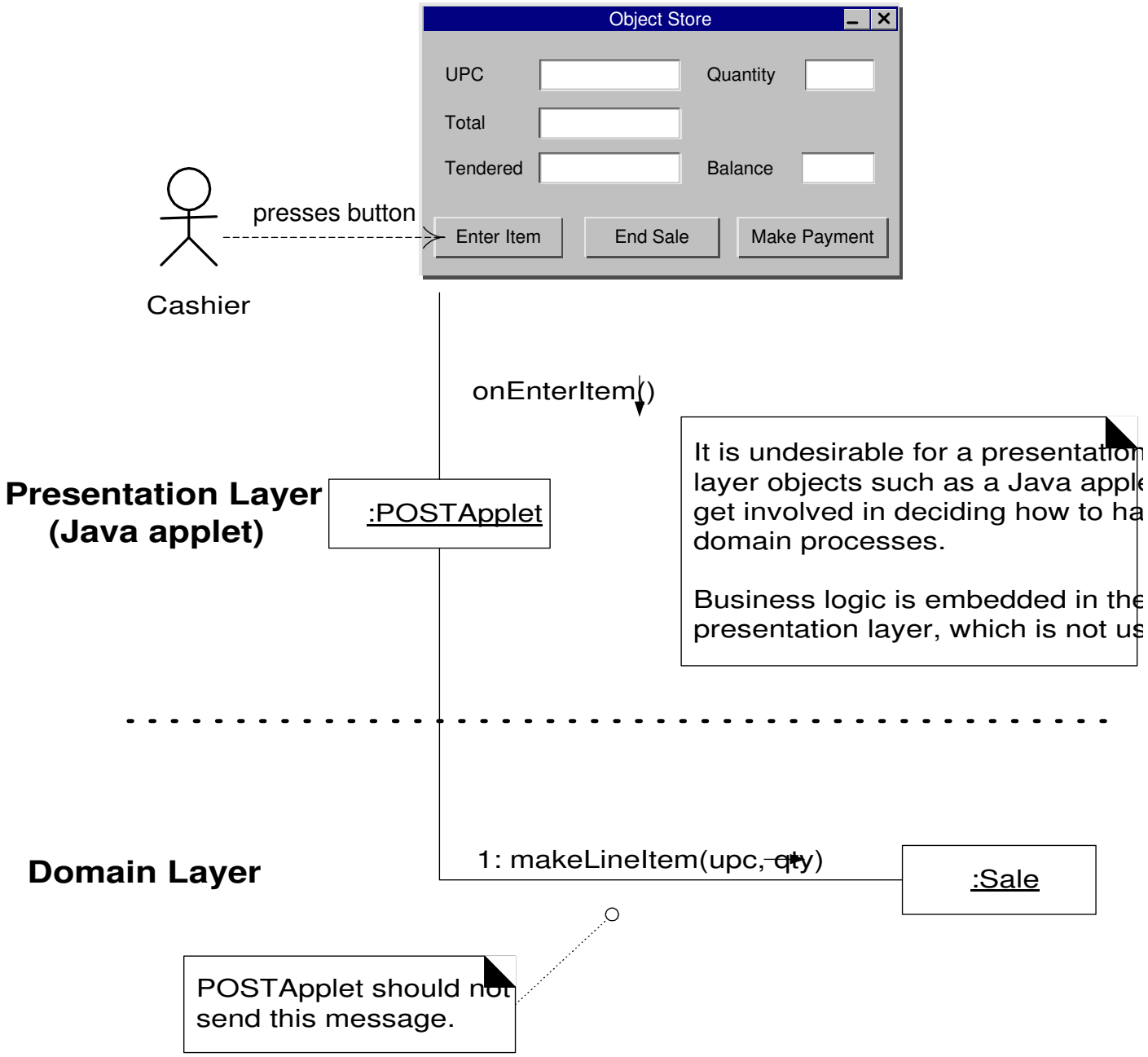
- Assign responsibility to keep coupling low
- Coupling examples:
  - A connected to B
  - A has knowledge of B
  - A relies on B
- Problems of high coupling
  - changes ripple; difficult to understand alone, to reuse
- But excessive de-coupling: large, complex

Coupling

- typeA has attribute or method that refers to an instance of typeB or typeB
- typeA is a direct or indirect subclass of typeB
- typeA is an interface; typeB implements the interface



Desirable coupling ….


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