COMP 3711

OOD

GRASP AND PATTERNS

Larman Chapter 17

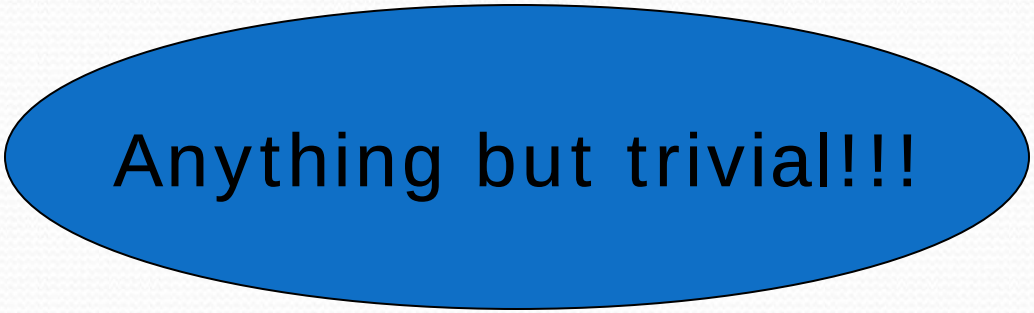
Changing Hats

Analysis

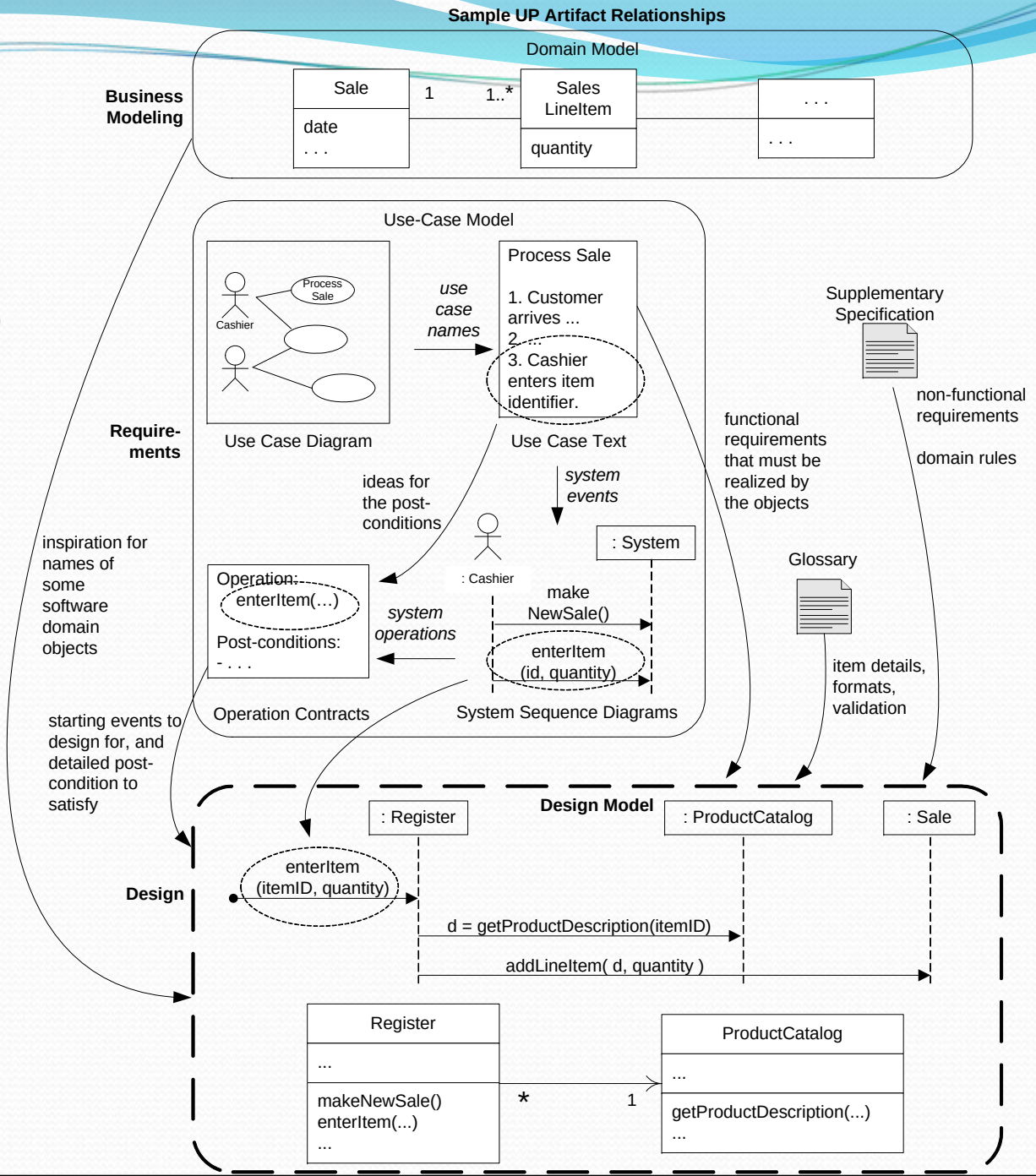
Design

Object Design

"After identifying your requirements and creating a domain model, then add methods to the software classes, and define the messaging between the objects to fulfill the requirements"



OOD Design Artifacts



Recipe

RDD

GRASP

GoF Design Patterns

RDD - Responsibility Driven Design

- Think of software objects as having responsibilities $\longrightarrow$ what they do
- Responsibilities are related to the obligations or behaviour of an object in terms of its role (its is abstraction)
- Methods fulfill responsibilities
- RDD – a general *Metaphore* of a community of collaborating responsible objects

Two Types Of Responsibilities

- Doing Responsibilities
  - Creating an object or doing a calculation
  - Initiating action in other objects
  - Controlling and coordinating activities in other objects
- Knowing Responsibilities
  - Knowing about private encapsulated data
  - Knowing about related objects
  - Knowing about things that can be derived or calculated

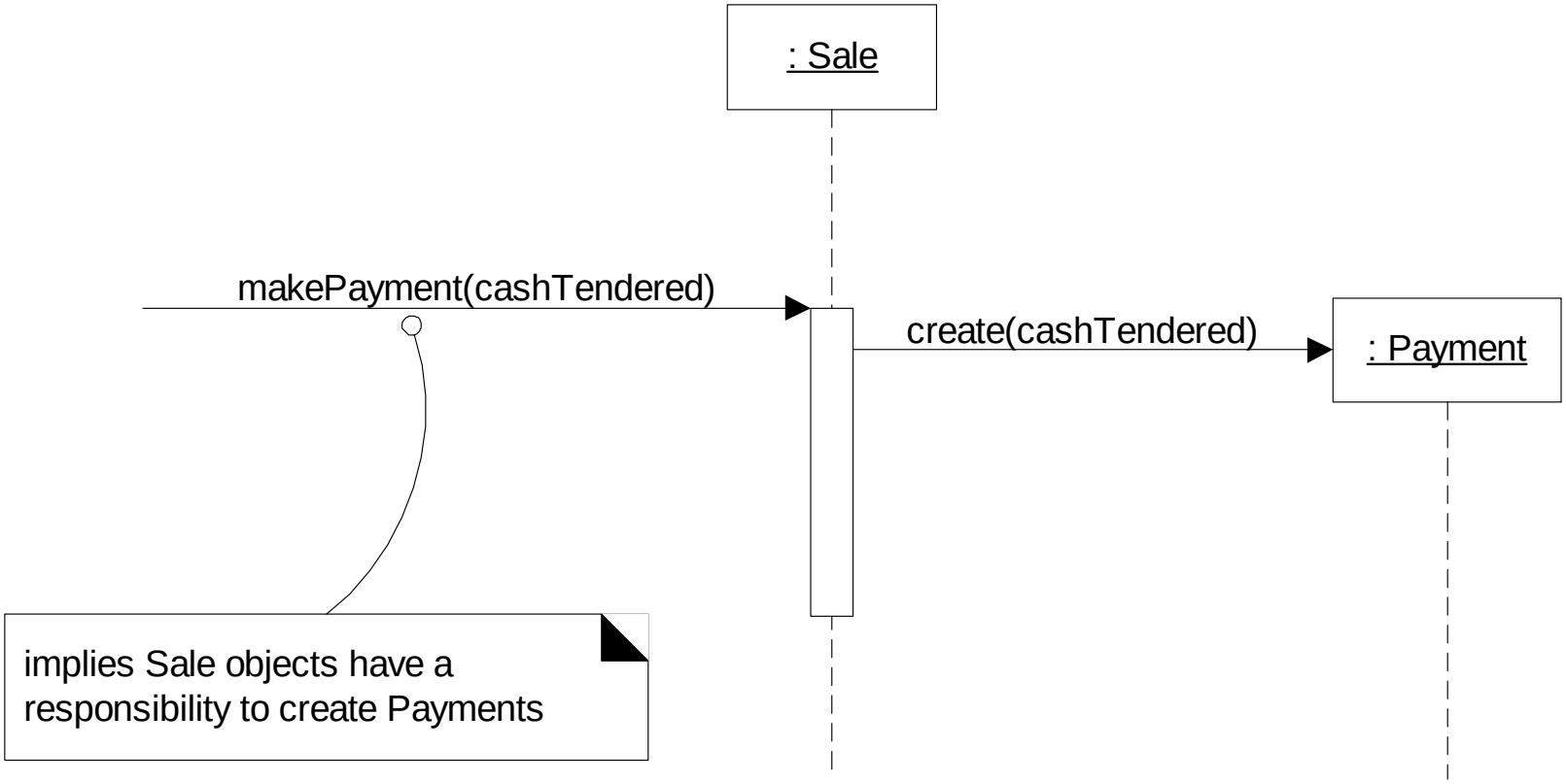
Design - Think Object

- Assigning responsibilities
- Granularity of responsibility influences how it is assigned
- What methods belong where?
- How objects should interact?

Responsibilities - Interaction Diagrams

- Show objects and the messages between
- UML Interaction Diagrams include:
  - Sequence diagrams
    - Have time on the Y axis
  - Collaboration diagrams
    - Focus is more on way the objects interact
- Record assignment of responsibilities

Sequence Diagram Example



1. Consider the object responsibilities
2. Realize the responsibilities as methods
3. Assign responsibilities in UML Interaction Diagram

GRASP

- G)eneral (R)esponsibility (A)ssignment (S)oftware (P)atterns (or Principles)
- Learning aid for OO Design with Responsibilities
- Key: Understand how to apply GRASP for OOD
- GRASP defines ***nine*** basic OOD principles

OO DESIGN - PATTERNS

- A named description of a problem and solution that can be applied to new contexts
- Typically an *existing named* and *well-known* long repeating problem/solution pair - not new ideas
- GRASP Patterns name and codify widely used basic principles
- There are nine GRASP Patterns

Nine **GRASP** Principles

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

Important to grasp the first 5 principles

See inside front textbook cover

OO DESIGN - GoF PATTERNS

- A software engineering book - “Design Patterns”, mostly coded in C++ and Smalltalk , was introduced in 1994 by the Gang-of-Four, covering 23 patterns with 15 commonly used.

- Dealt with recurring solutions to common problems in software design.

- GoF – Gang-Of-Four are Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

Patterns – GoF Ideas

- Favours programming to interfaces over implementation

- Favours Black-box over White-box resue
  - Black-box: objects obtain references to other objects through interfaces dynamic ally at run-time without visibility of the details of the composed objects
  - White-box: GoF – a visible detail of inheritance by sub-classes from the super-classes

Top Five Patterns

- Let's take a look a the top five GRASP Design Patterns:
  - Information Expert
  - Creator
  - Low Coupling
  - High Cohesion
  - Controller

Information Expert Pattern

- In order to fulfill its responsibility, a responsibility needs information about other objects, object's own state and what's around the object, what's derived from the object, etc.

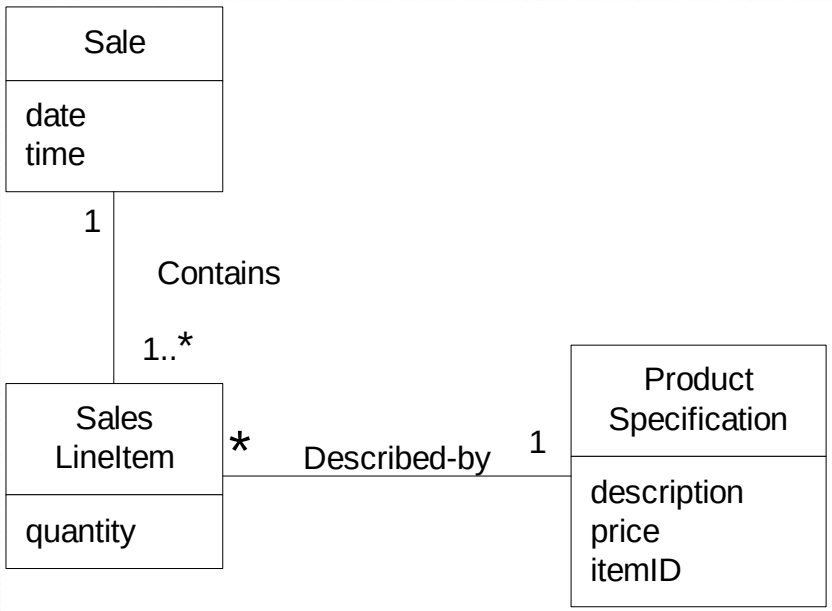
- Assign responsibility to the class that has the information necessary to fulfill the responsibility

Information Expert Pattern

- Design Model or Domain Model?
  - Look first in the design model
  - If not, look to modifying a class or classes in the domain model to be included in the design model
- This is actually part of the process of creating the design model

Example – Information Expert

- Before assigning responsibility, it's critical that the responsibility be clearly:
  - **DEFINED**
  - **STATED**
- In the POS system, the grand total for a sale must be known
  - What class should be responsible for the Grand Total?



Sale Class ??

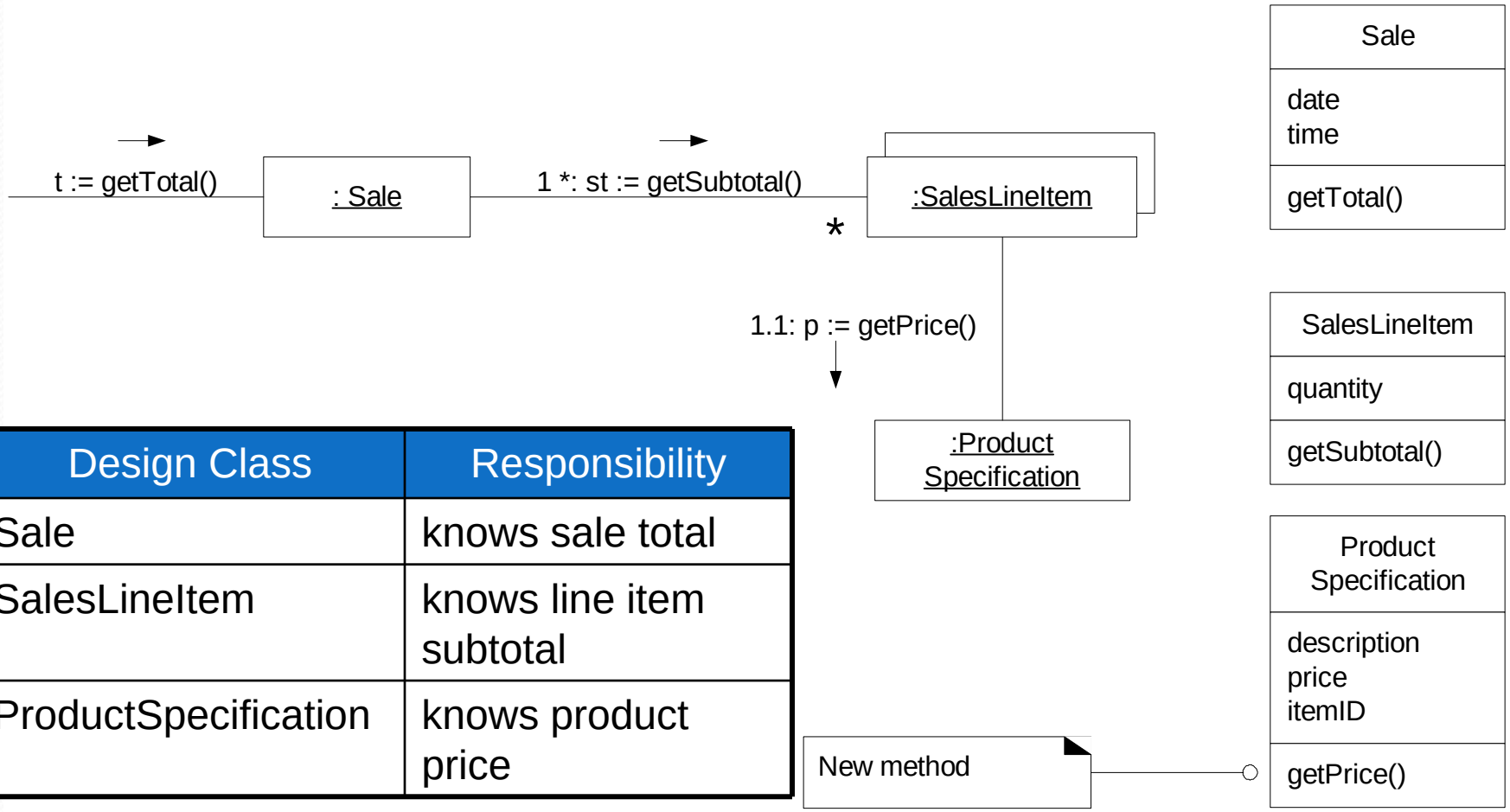
- Contains all of the line items
- Line items contain the quantity attribute and have access to the unit price
- Sale has attributes pertaining to the sale like date and time

Common sense to make the *Sale* class responsible for the *total*.

Reality/Programming Check

- The Sale class has a method that returns the current total for the sale
- The SalesLineItem class has a method that returns the subtotal for that line item
- The ProductSpecification class has a method that returns the unit price for an item
- These methods will be added to the classes in the design model

Collaboration Diagram Illustrating Information Expert Pattern



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

Place responsibility with object that has information needed to fulfill it.

Discussion

- Fulfillment of responsibility often requires collaboration among several classes
- Software object often **does** things that are in fact **done to** its real world equivalent
- In the human world, the people with the information needed to **do** the job are often given the responsibility for doing the job

When NOT to use Expert

- Usually if it causes coupling or cohesion problems
- Some services are best centralized
  - Especially things like database access
- Not good to have a class doing too much

Benefits

Information Encapsulation

- Supports low coupling (good)

Distributes Behavior

- Encourages cohesive lightweight classes
- High cohesion(good)

Creator Pattern

- If an instance of a class is to be created, which class should create it?
- A commonly doing responsibility in OO
- Need to minimize coupling
  - Pick a ***creator*** that will be coupled to the ***createe*** in any event
  - Benefit is that coupling is not increased

Creator ➔ Createe Relationships

Assign class B the responsibility to create an instance of class A if one or more of the following is true:

- B aggregates A objects
- B contains A objects
- B records instances of A objects
- B closely uses A objects
- B has initialization data that will be passed to A when it is created

Picking a Creator

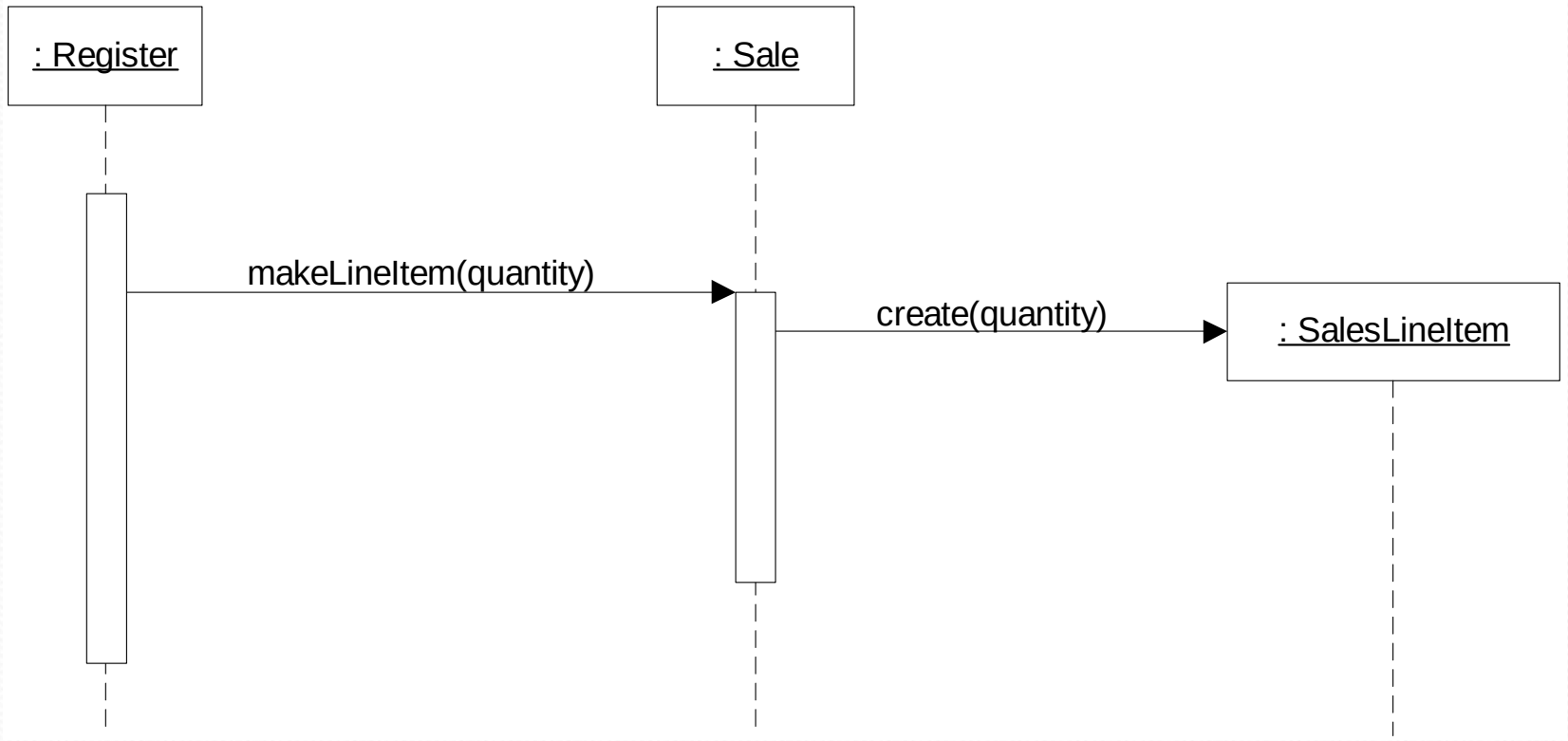
- Any class that has any one of these relationships with a class to be created is a potential creator
- If more than one class has one of these relationships, pick one of the ones that aggregates or contains as the creator
- If there are no classes that meet this criterion, then system coupling is increased

LGR – Low Representational Gap

- When assigning responsibilities, look to the Domain Model for inspiration and apply LGR

Sequence Diagram Illustrating Creator Pattern

Who should be responsible for creating a *SalesLineItem* instance?



When **NOT** to Use Creator

- If creation is a complex process, best to have a factory class to do the creation
  - Example, when using recycled instances (conditionally creating an instance from one of a family of similar classes) , the factory pattern is more appropriate

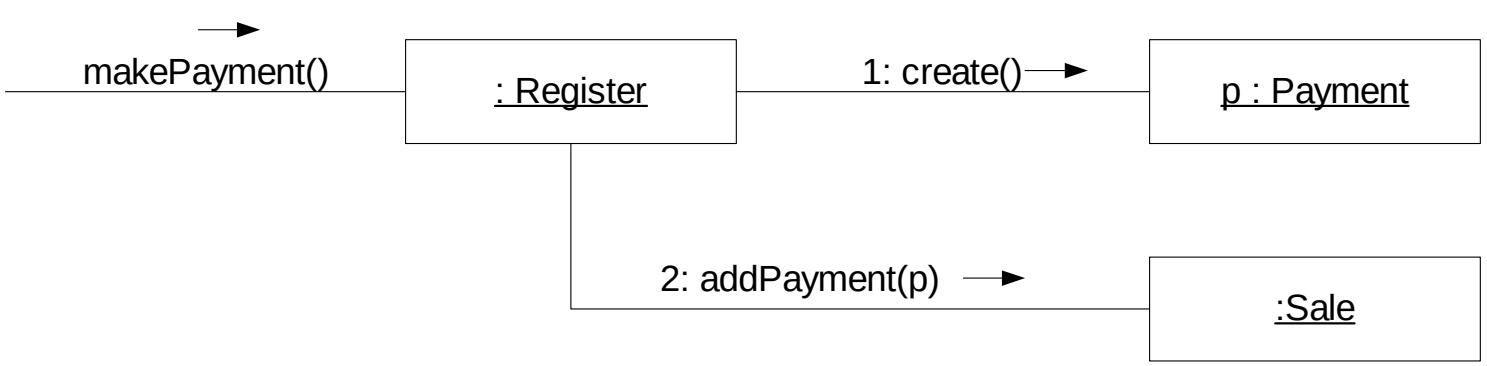
Low Coupling Pattern

Assign a responsibility so that coupling remains low.

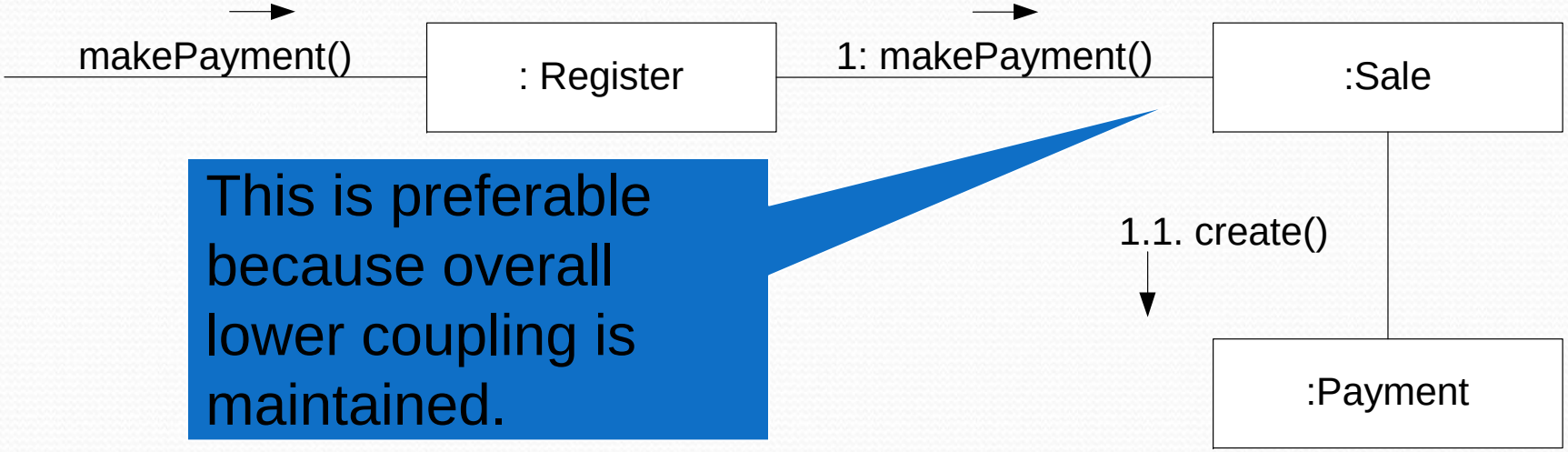
- Coupling is a measure of how strongly one element:
  - Is connected to
  - Has knowledge of
  - Relies on other elements
- Benefits:
  - Classes are more independent
  - Changes can be localized
  - Components can be understood in isolation
  - Reuse is more feasible

What class should be responsible for creating a *Payment* instance and associate it with the *Sale*?

Register Creates Payment



Sale Creates Payment



This is preferable because overall lower coupling is maintained.

Coupling in Code

- Attribute, instance variable, or data member
- Message, method invocation, or function call
- Instances as arguments, local variable, or return values in methods
- Inheritance/Generalization
- Implementation of an interface

When is Low Coupling **BAD**

- When taken to the extreme
  - Results in a few multipurpose bloated classes that do everything with a large number of passive data classes
- There is no absolute measure of coupling
  - A system does not become too highly coupled in a day
  - Usually the result of a long series of short sighted decisions

High Cohesion Pattern

- How to keep objects focused, understandable and manageable that support low coupling?

- Assign responsibilities so that cohesion remains high

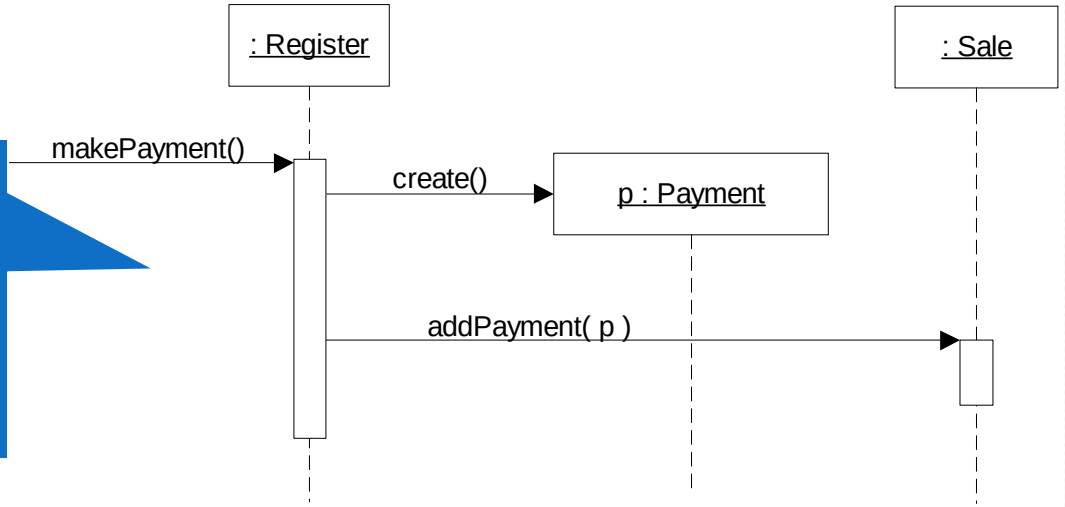
High Cohesion Pattern

- A measure of how strongly related and focused the responsibilities of an element are:
  - Many objects, each of which does a small amount of work in one particular area
  - Relatively fine grain of abstraction

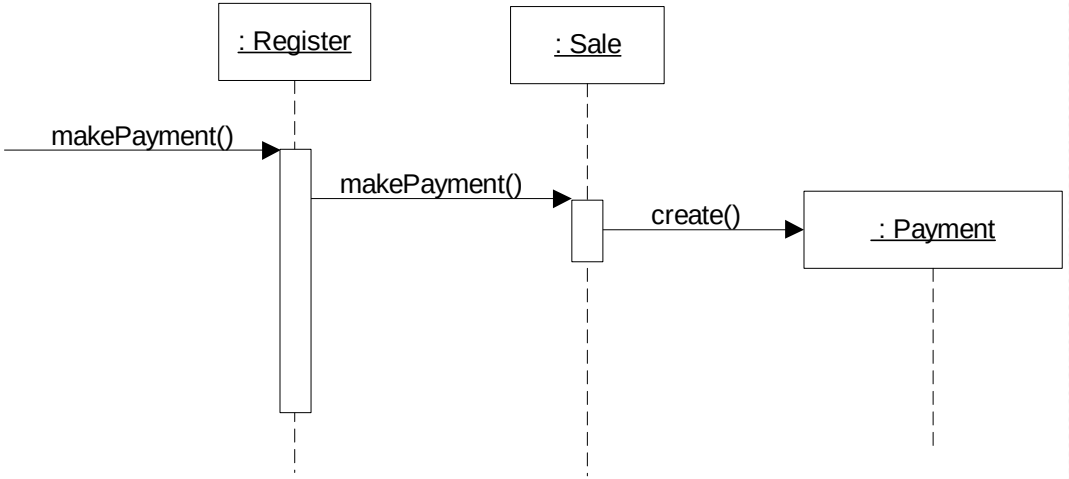
What class should be responsible for creating a (Cash) *Payment* instance and associate it with the *Sale*?

Sequence Diagram Showing Poor Cohesion

Not good that Register is given too many responsibilities. It should delegate work to other classes.



Sequence Diagram Showing Good Cohesion



Benefits

- Clarity and ease of comprehension
- Maintenance and enhancement are simplified
- Low coupling often supported
- Reuse more likely
  - Cohesive class usually has very specific purpose

When is High Cohesion **BAD**

- There are very rare circumstances where grouping of large numbers of responsibilities is warranted
  - Example: In distributed systems where high cohesion may cause problems because of performance penalty of remote calls over the network.

Controller responsibilities

- How to connect the UI layer to the Domain layer (application logic)?
- What first object beyond the UI layer receives and coordinates (i.e. control) as system operation?
- Who should be responsible for handling a system input event?

Controller responsibilities

- Important NOT to give controller too much responsibility
- Controller delegates all jobs, just coordinates
- In the UP there are:
  1. Boundary classes: UI (presentation layer) abstractions
  2. Controller classes: use-case handlers
  3. Entity classes: application independent (typically persistent) domain software objects
- Not a domain object

Controller Pattern

Assign responsibility for receiving or handling a system event message to a particular class.

- Controllers responsibilities range from events for a whole system to events for a single use case scenario
- Not a user interface object
- User interface object should pass external events to a controller for handling
  - Example: endSale(), spellCheck()

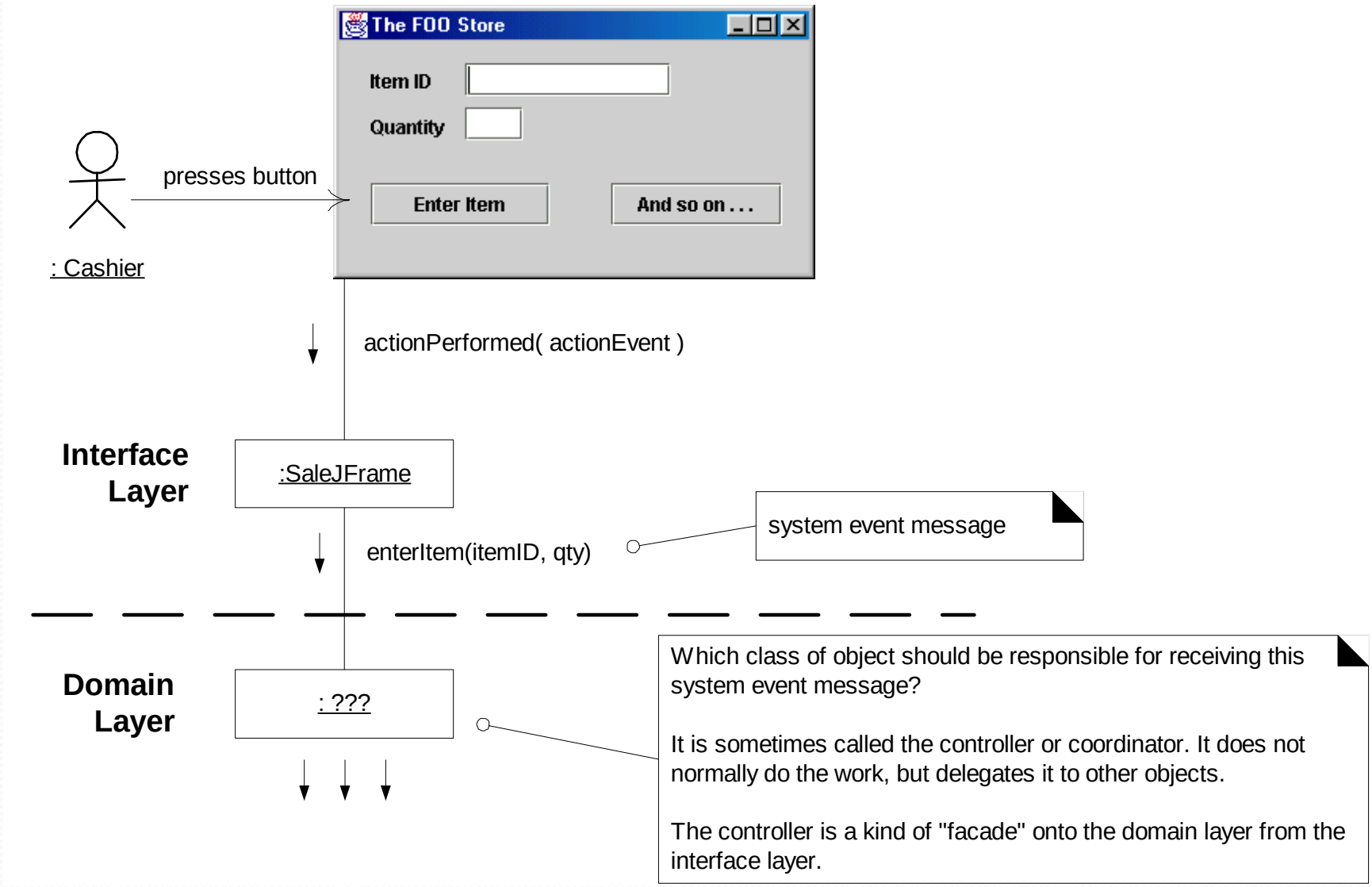
Facade Controllers

- Only one controller for large area of responsibility
- Sometimes the bloated controller is performing tasks that should be delegated
- Controller has many attributes and maintains significant system state information
- Suitable where there are not too many systems events or when UI cannot redirect system event messages to alternate controller
- Tend to be built incrementally

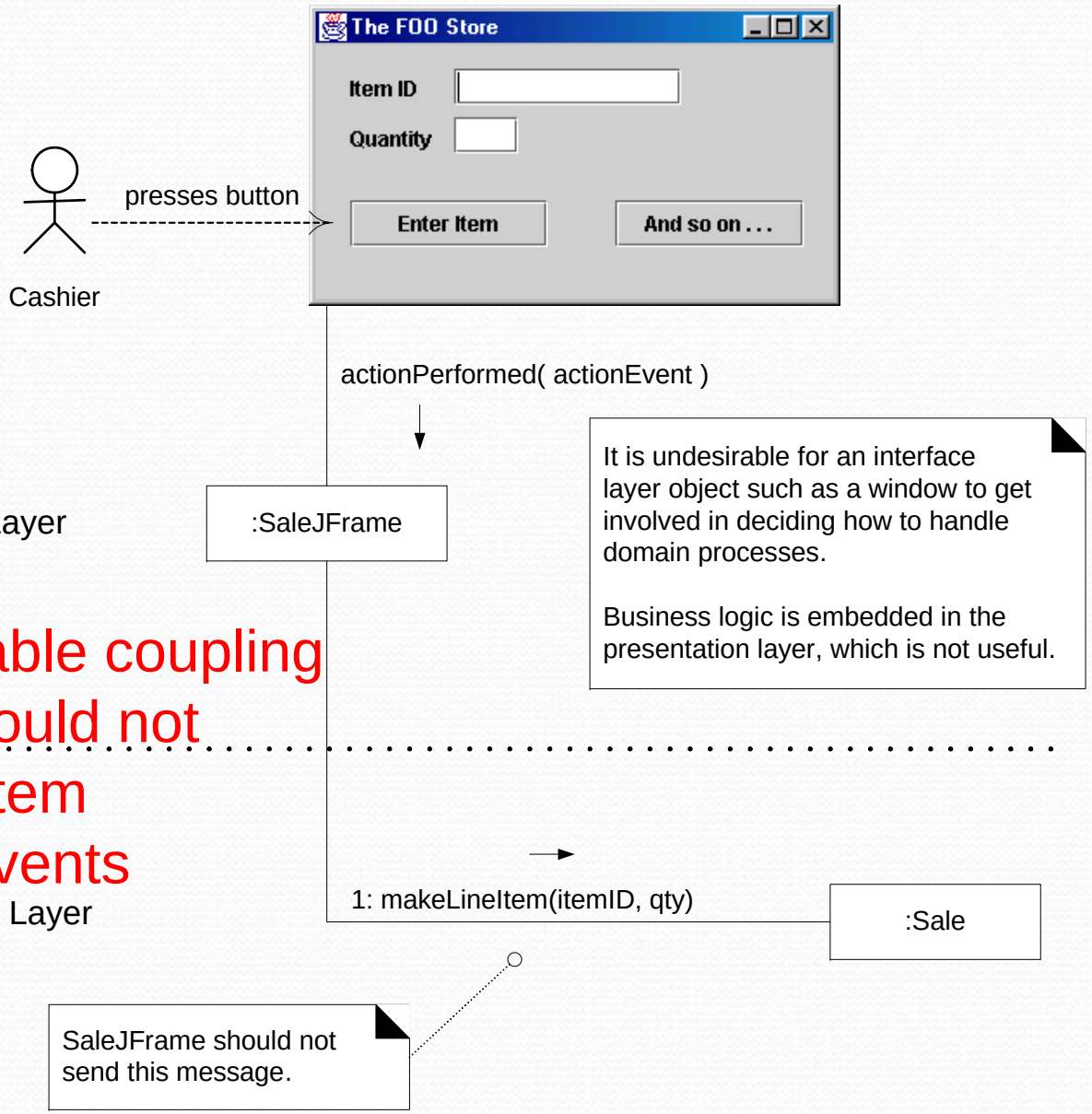
Use Case Controllers

- Use Case Controllers - Add more controllers (one for each user case)
- Delegates the fulfillment of each system operation responsibility to other objects
- Applicable when there are many system events across different processes which factors their handling into manageable separate classes

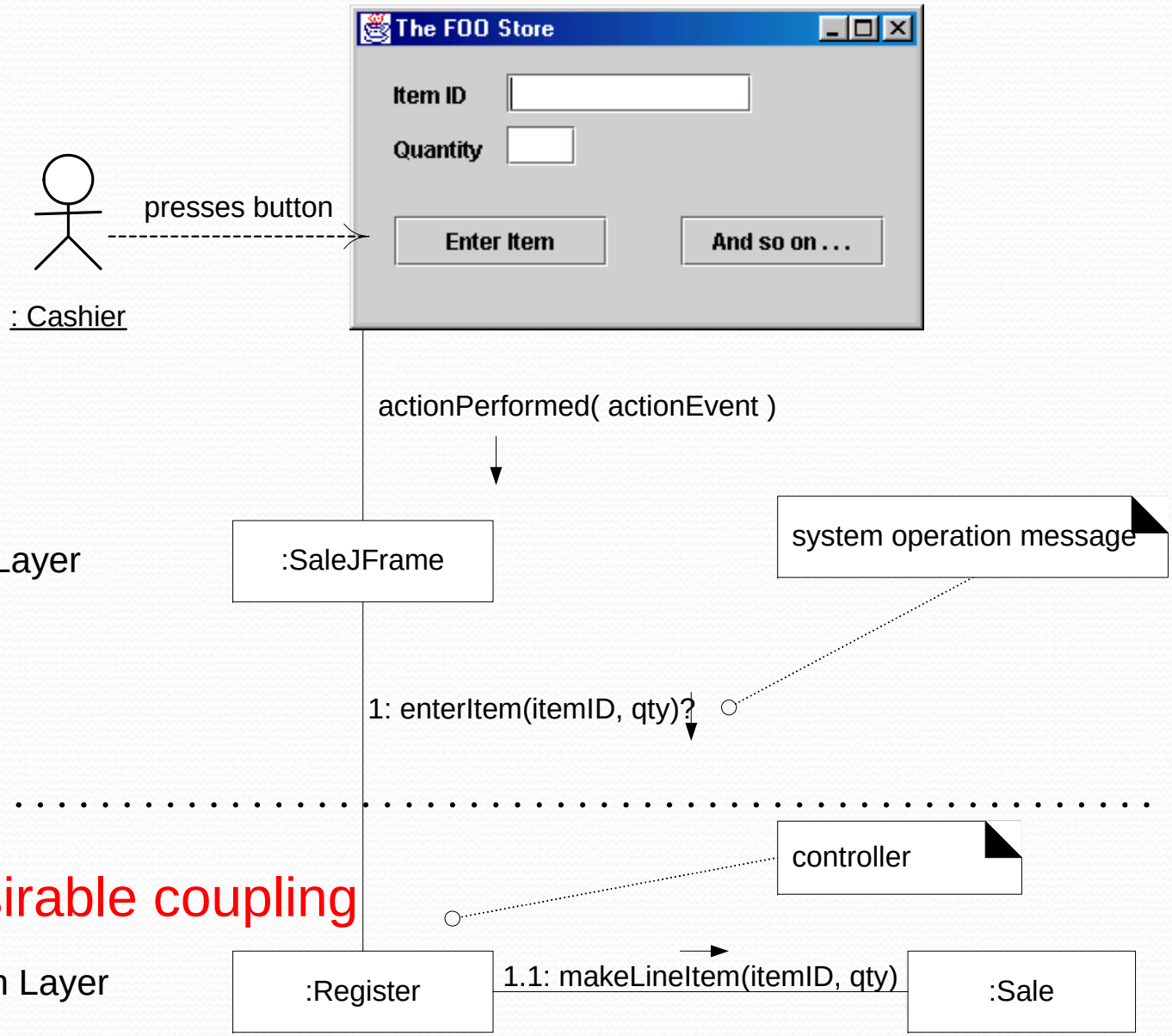
Controller ?? for *enterItem()*



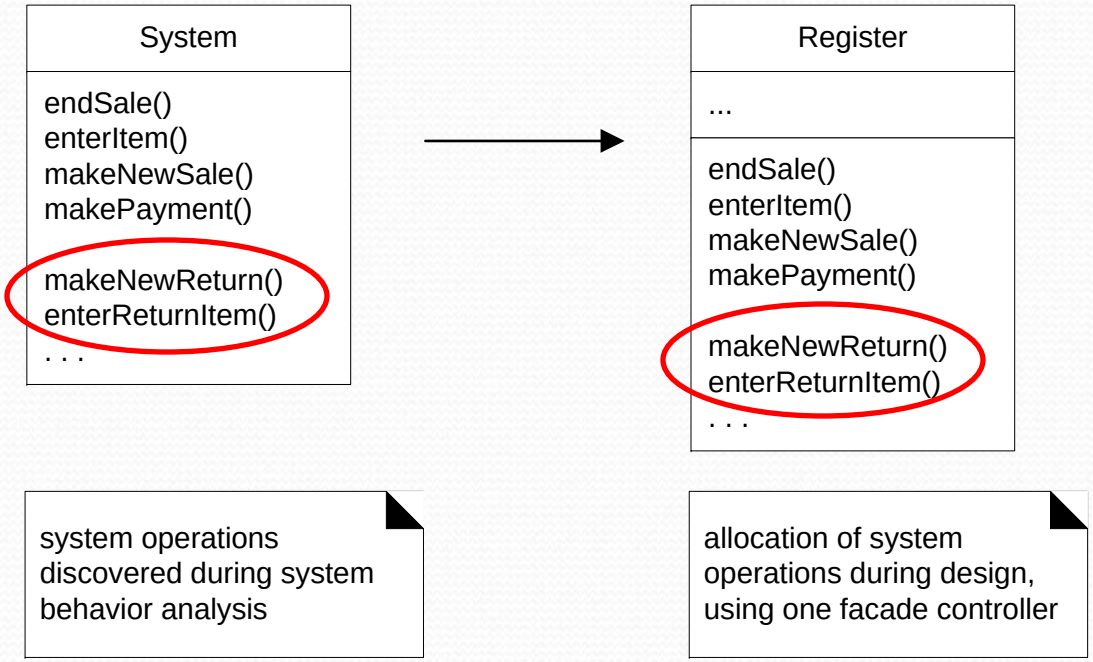
Controller for *enterItem()*



Controller for *enterItem()*



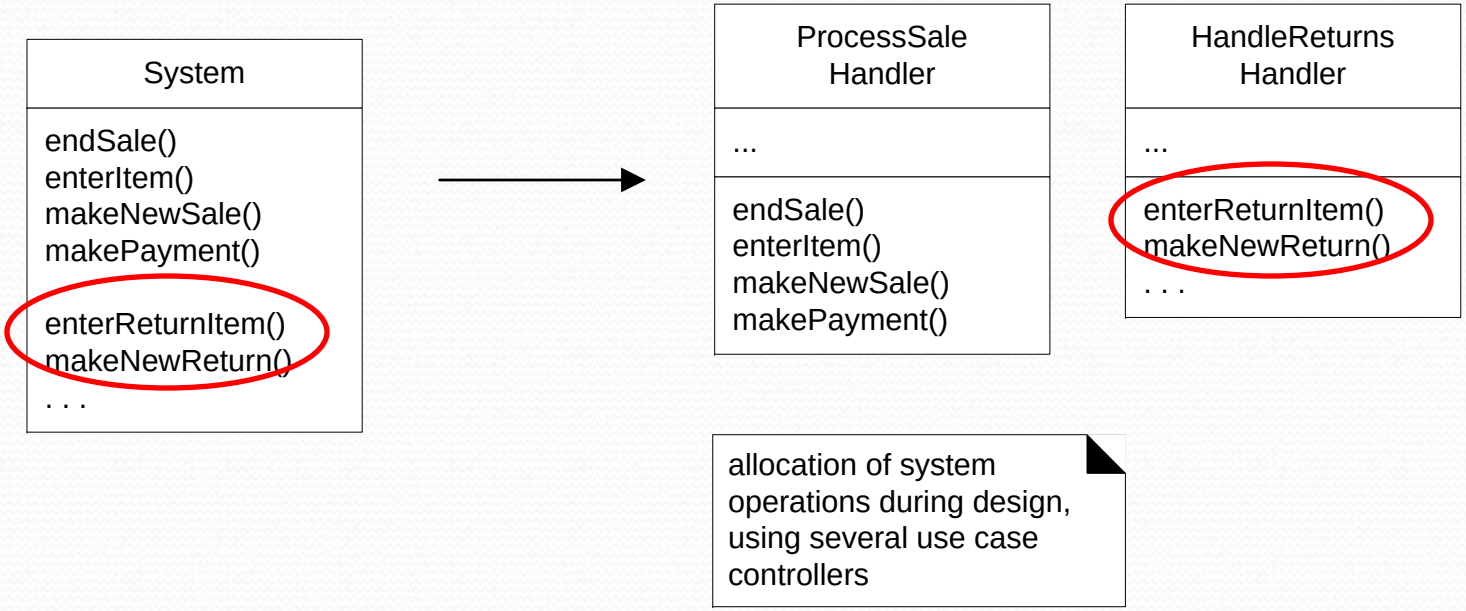
Facade Controller



Less desirable design

A facade controller handles “overall” system events

Use Case Controllers



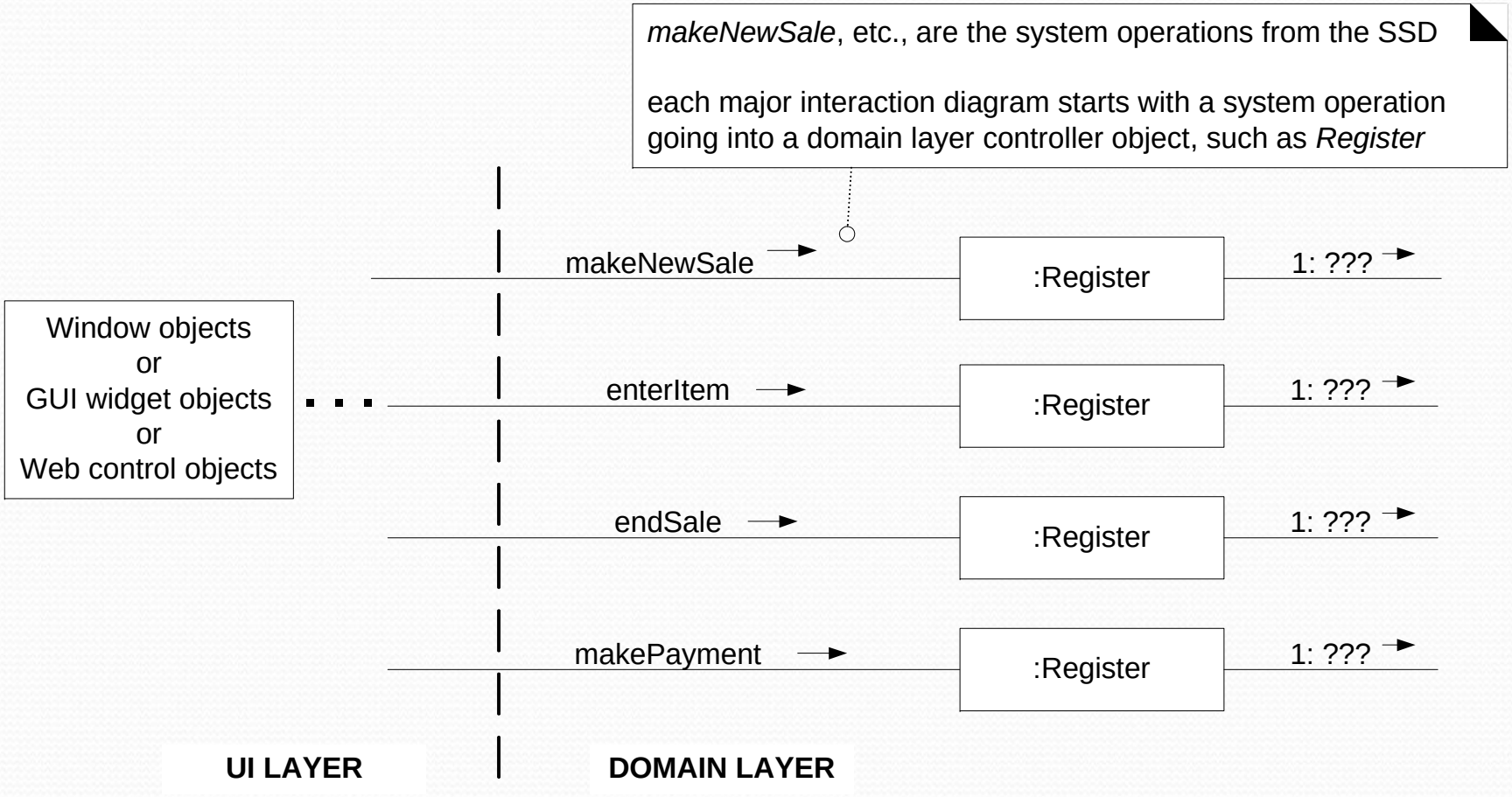
More desirable design

Individual use case controller handles events of each use case scenario

Top Five Patterns - Summary

- Information Expert
- Creator
- Low Coupling
- High Cohesion
- Controller

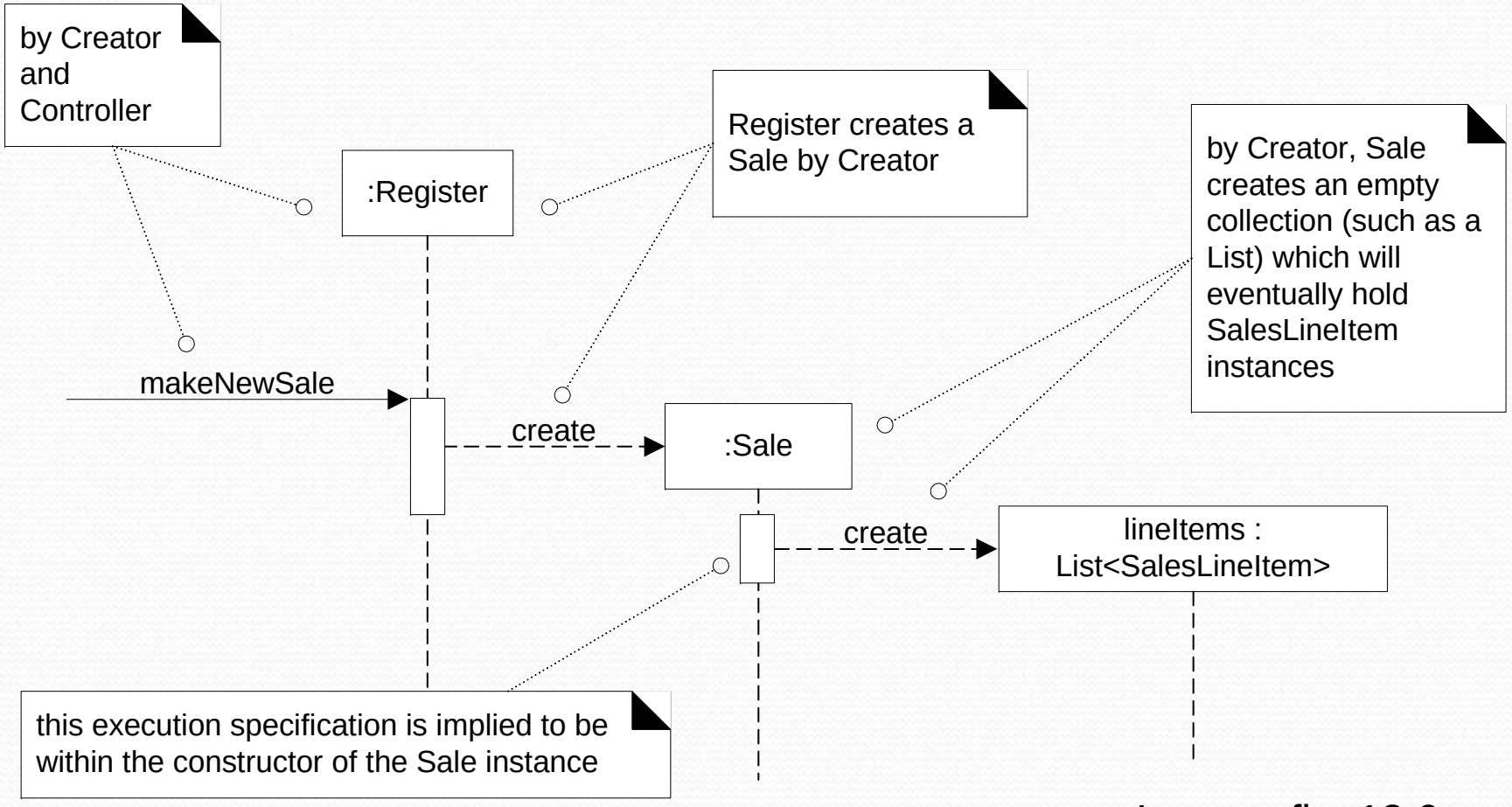
NextGen POS Iterations



Larman fig. 18.2

Creators – makeNewSale

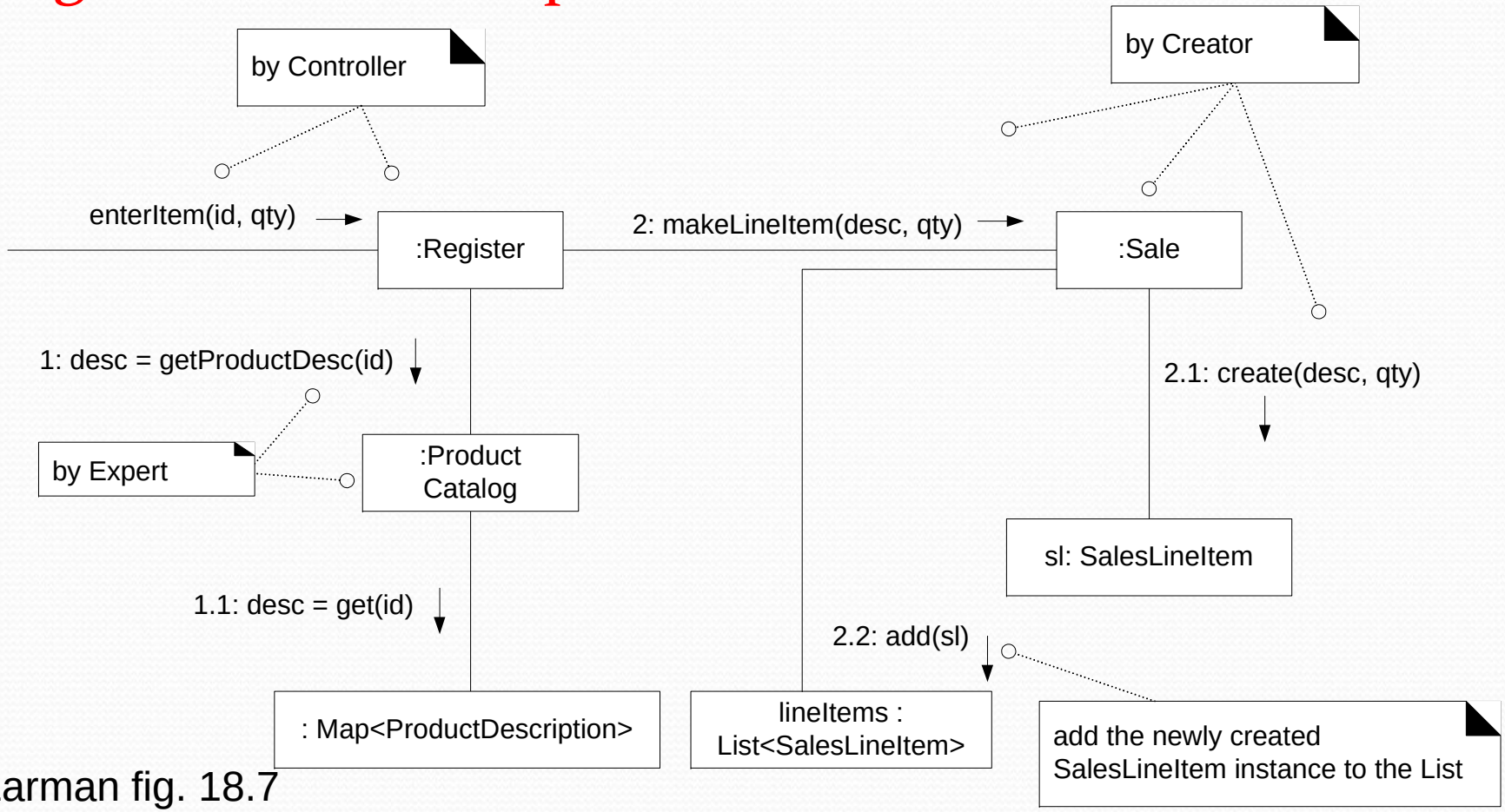
- Register creates the Sale, the Sale creates an empty collection for SalesLineItem(e.g. Java List)



Larman fig. 18.6

Controller / Creators – enterItem

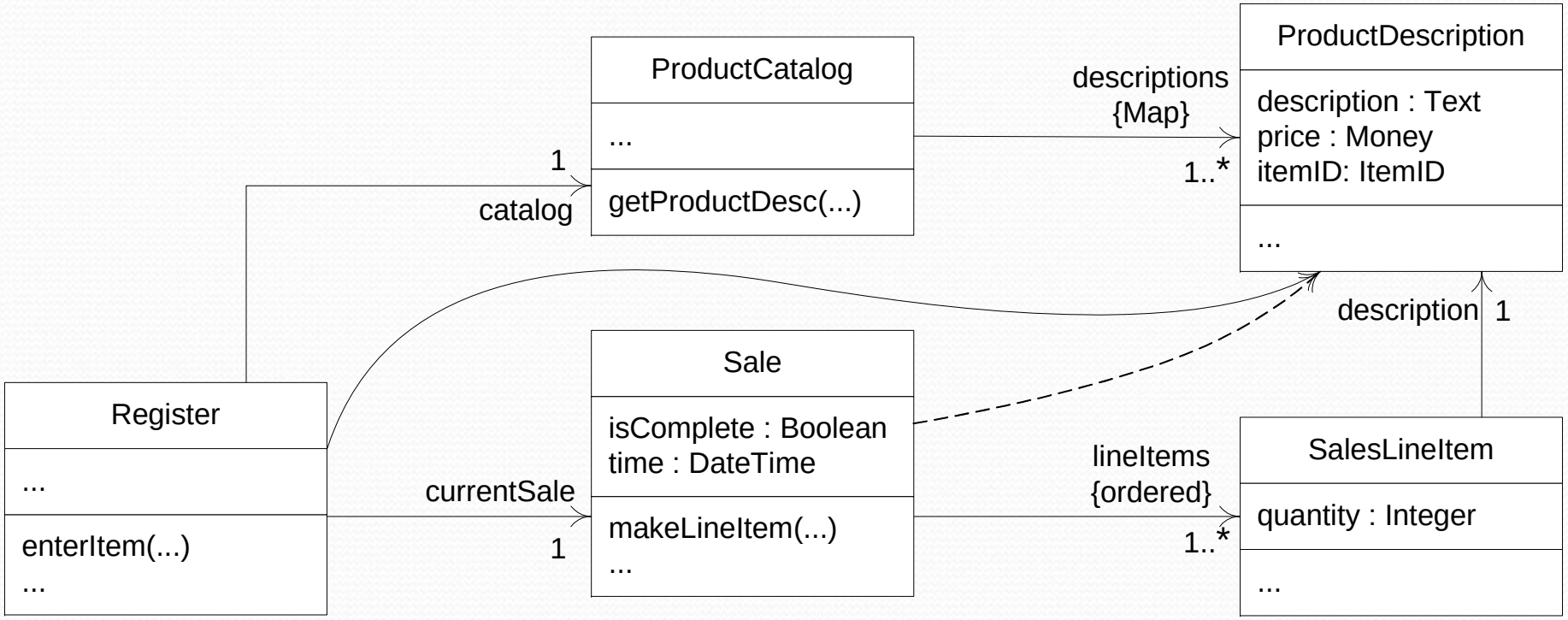
- Creation, initialization and association of a SalesLineItem with visibility to ProductCatalog to getProductDescription



Larman fig. 18.7

Controller / Creators – enterItem

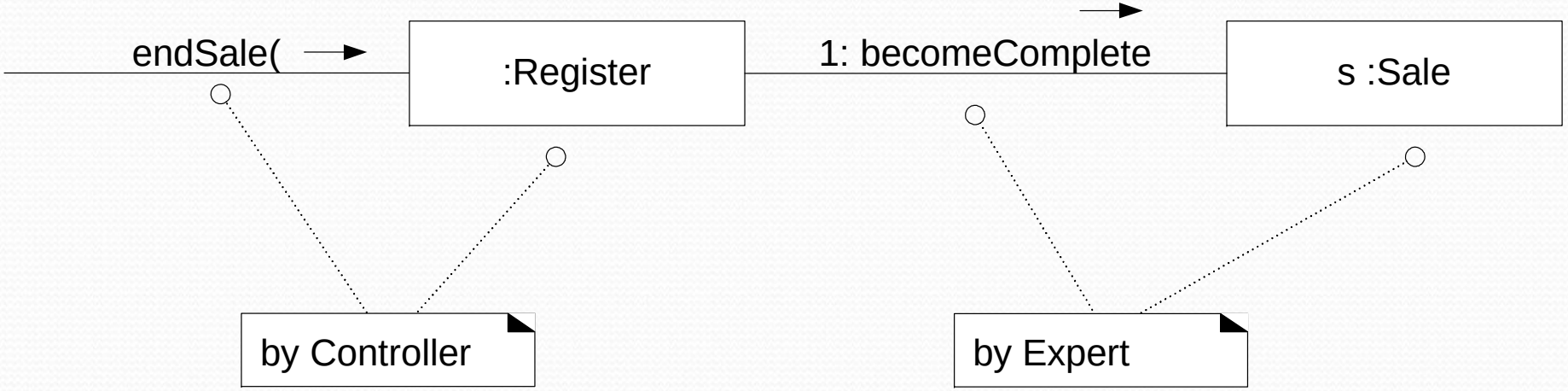
- Partial DCD – static view of neterItem



Larman fig. 18.8

Controller / Expert – endSale

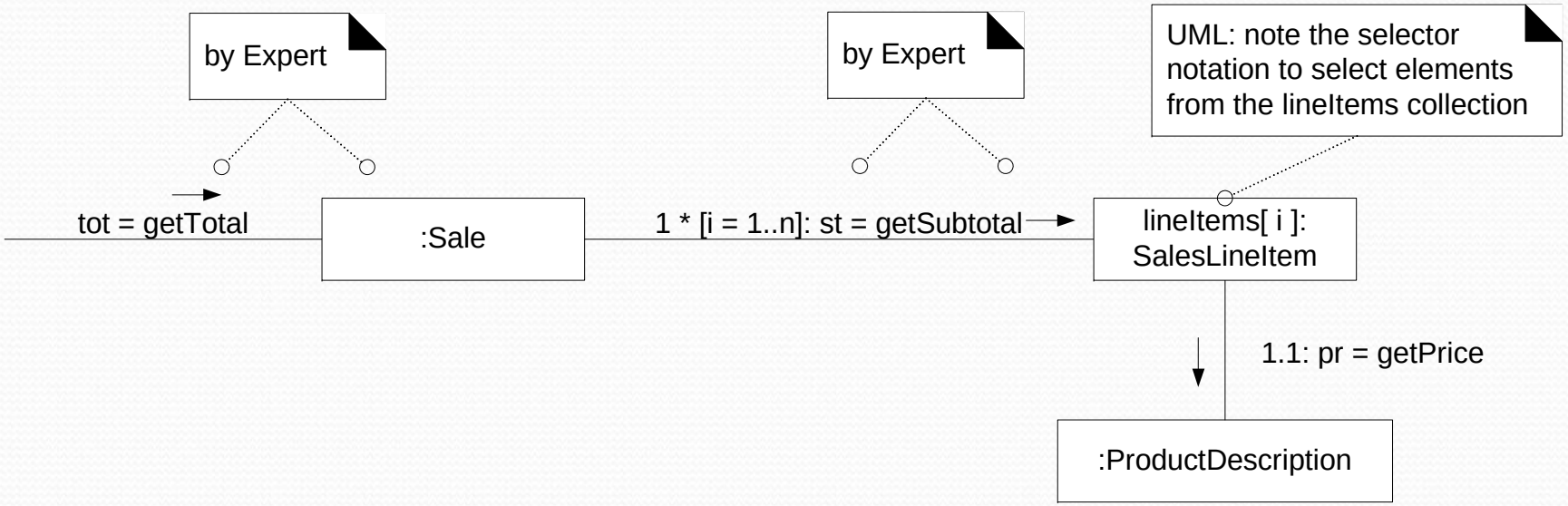
- Register continues to be the controller for the system operation message of endSale



Larman fig. 18.19

Expert – getTotal

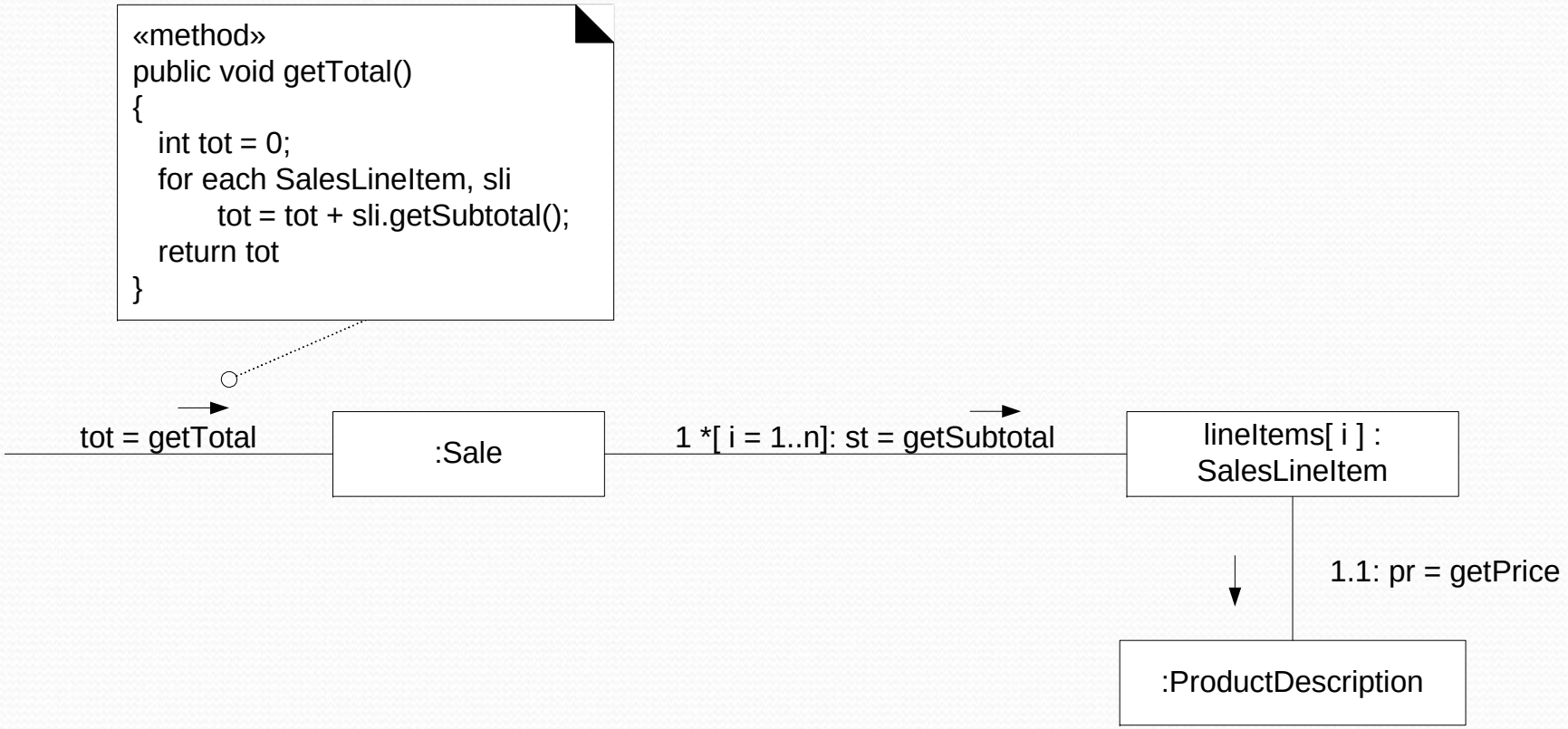
- Sale is responsible for knowing its total and requies ProductDescription, SalesLineITem, Sale(i.e. all the SalesLneITems in the current Sale)



Larman fig. 18.10

getTotal Method

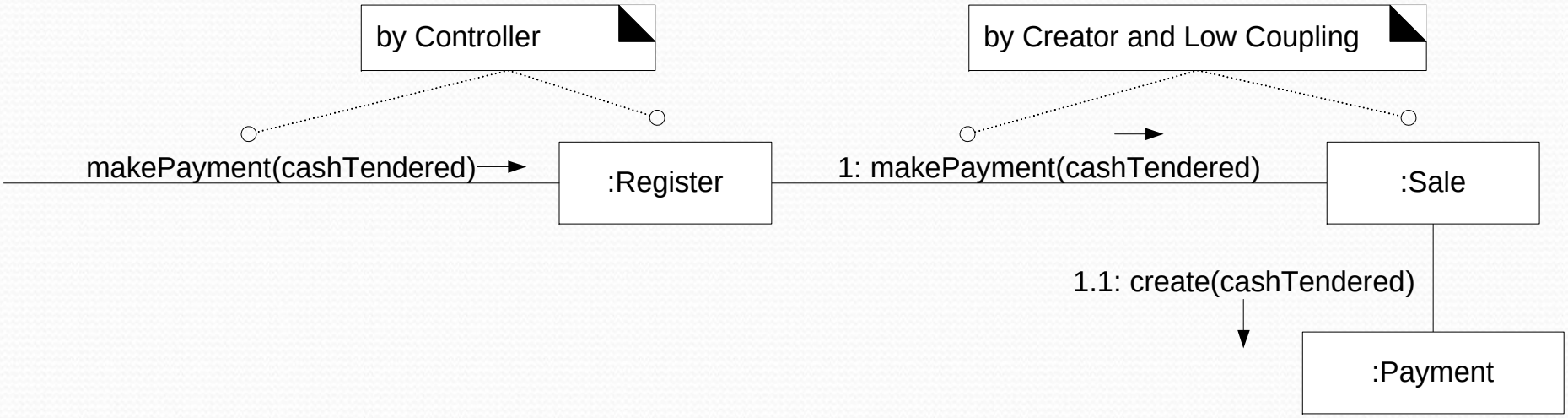
- Sale is responsible for knowing its total and requies ProductDescription, SalesLineITem, Sale



Larman fig. 18.12

Controller / Creator – makePayment

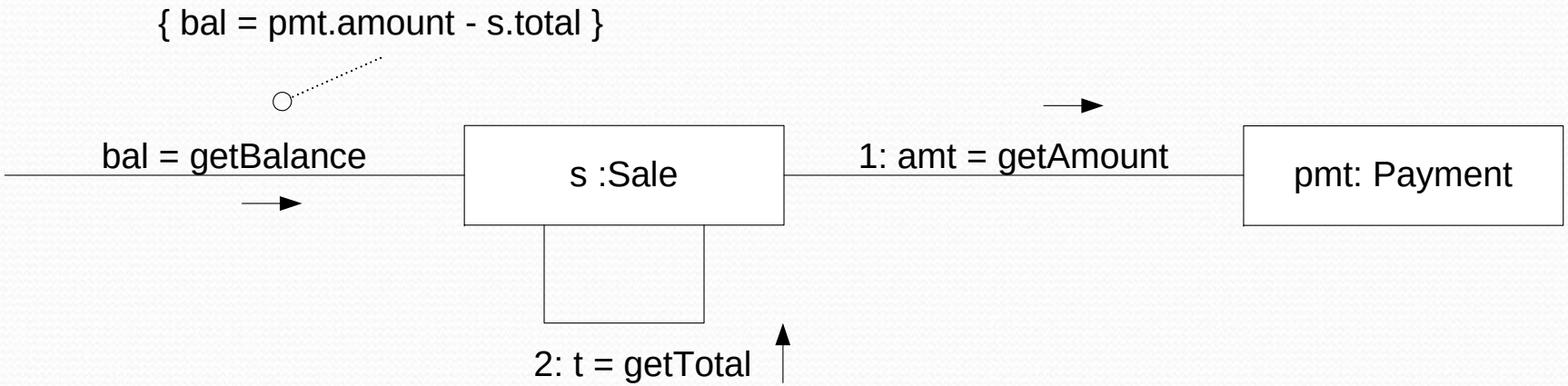
- Sales create the Payment for better cohesion and low coupling in the Register



Larman fig. 18.13

Expert – makePayment

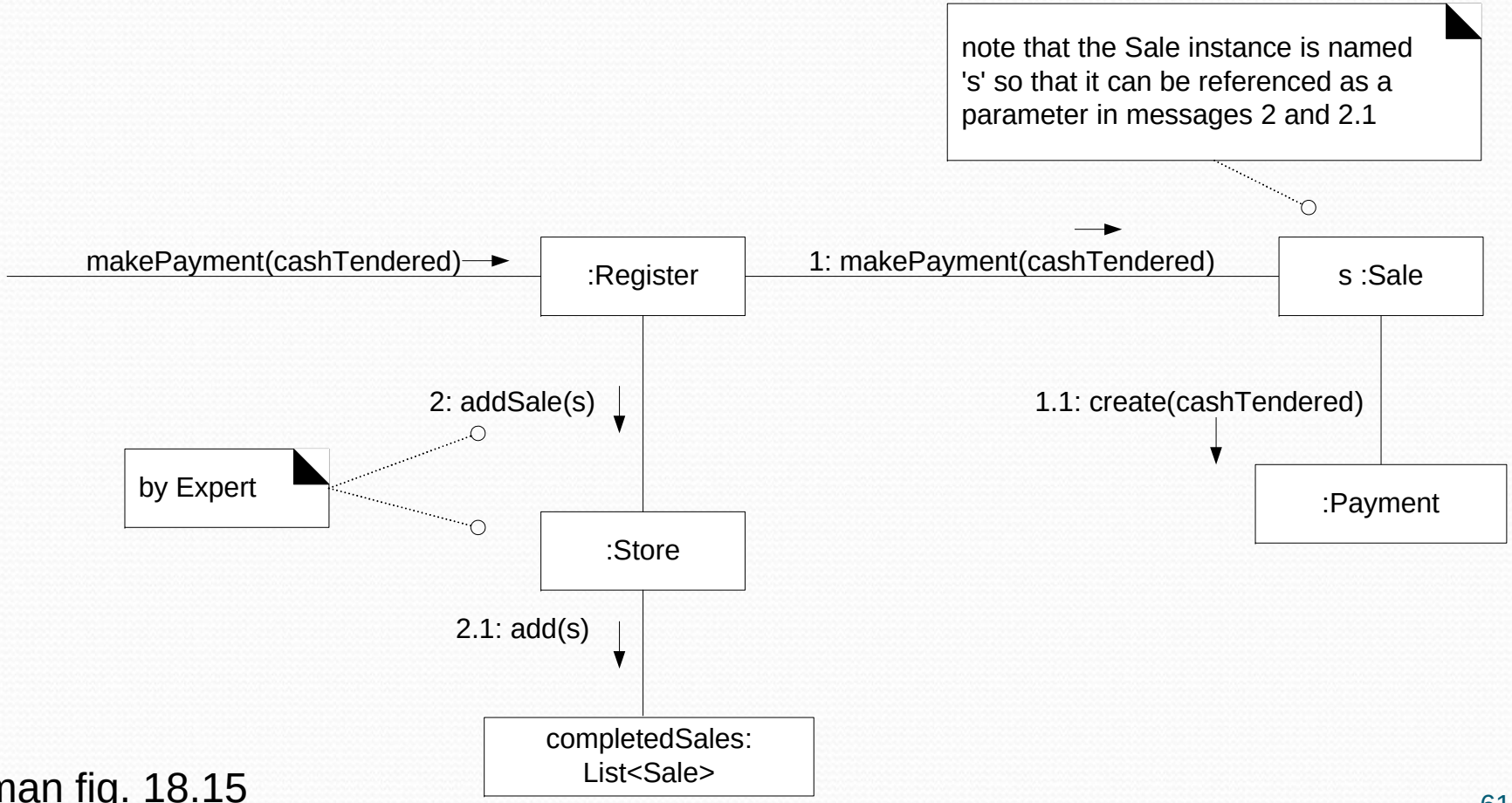
- Sale knows the balance (i.e. the sales total and payment tendered) from its visibility to Payment



Larman fig. 18.163

Expert – Log Completed Sale

- Use Store to keep list of completedSales per Operation Contract postconditions



Larman fig. 18.15