GRASP Patterns

Dr. M. Taimoor Khan
Taimoor.khan@nu.edu.pk

What are patterns?

- Design **principles** and **solutions** written in a structured format describing a problem and a solution
- A named problem/solution pair that can be applied in new contexts of software design.
- It is an advice from previous designers to help designers in new situations

Patterns

- Are not invented. They are harvested from existing solutions.
- Are given a name to aid in communications.
- Are documented in a rigorous fashion
- Sometimes conflict with each other. For example: you apply a patterns to solve one problem, but by doing so, you may introduce others.
  - This is called a contradiction, or side-effect.
  - These are the tradeoffs designers have to deal with!

GRASP

- Stands for General **Responsibility Assignment** Software Patterns (Principles)
- OOD: after identifying requirements, create domain model, define responsibilities and assign methods to classes
- define “messaging” (who sends the message)
- Patterns are a very important part in software design.

Characteristics of Good Patterns

- It solves a problem
- It is a proven concept
- The solution isn't obvious
- It describes a relationship
- The pattern has a significant human component (ease of use).

Why Apply GRASP Patterns?

Which class, in the general case is responsible?

- You want to assign a responsibility to a class
- You want to avoid or minimize additional dependencies
- You want to maximise cohesion and minimise coupling
- You want to increase reuse and decrease maintenance
- You want to maximise understandability
- …..etc.

9 GRASP PATTERNS

1. Creator
2. Information Expert
3. Low Coupling
4. Controller
5. High Cohesion
6. ...

Creator Pattern

Problem:

Assign responsibility for creating a new instance of some class?

Solution:

Determine which class should create instances of a class **based on** the **relationship** between potential creator classes and the class to be instantiated.

Creator Pattern

Who has responsibility to create an object?

By creator, assign class B responsibility of creating instance of class A if:

- **B *aggregates* A objects**
- **B *contains* A objects**
- **B *records* instances of A objects**
- **B *closely* uses A objects**
- **B *has the initializing data* for creating A objects**

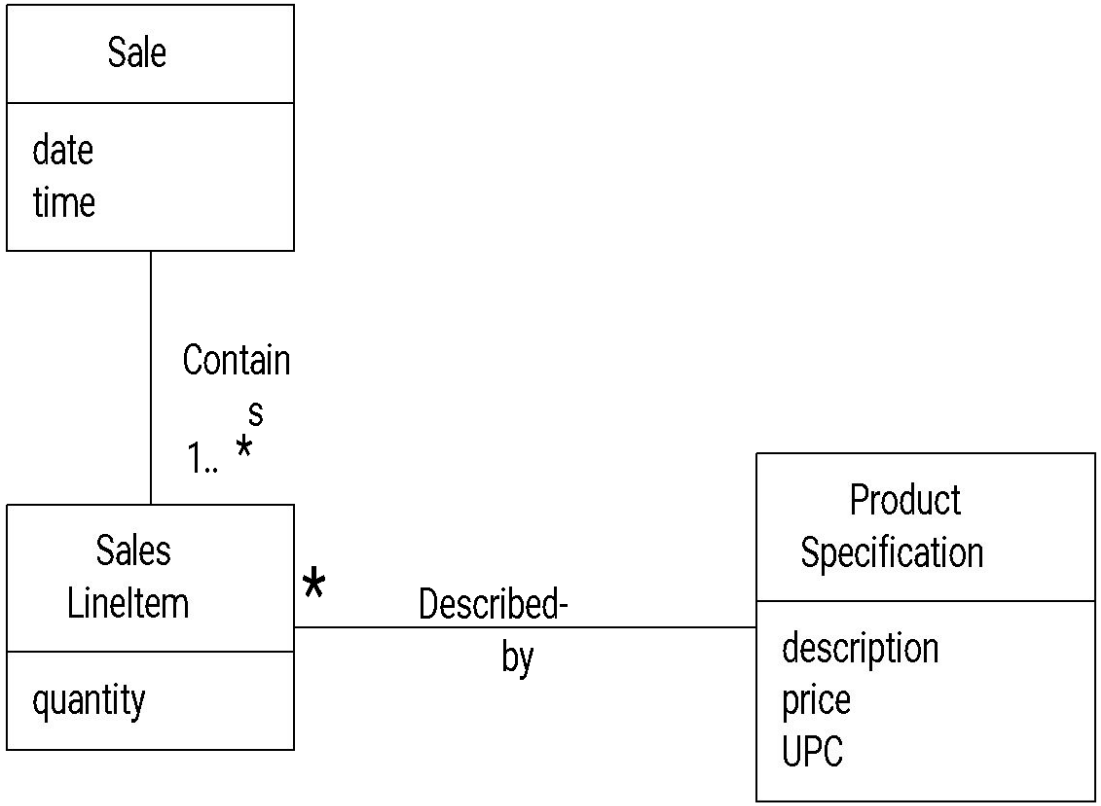
Where there is a choice, prefer

B *aggregates* or *contains* A objects

Creator Pattern : Example

Who is **responsible** for creating ***SalesLineItem*** objects?

Look for a class that aggregates or contains ***SalesLineItem*** objects.



Creator Pattern

- Promotes **low coupling** by making instances of a class responsible for creating objects they need to reference.
- By creating the objects themselves, they avoid being dependent on another class to create the object for them.

Grasp Patterns: Information Expert

Information Expert: Responsibility delegation principle/pattern

Problem: A system will have hundreds of classes. How do I begin to assign responsibilities to them?

Solution: Assign responsibility to the **Information Expert** – the class that has the information necessary to fulfill the responsibility.

Grasp Patterns

Mechanics:

Step 1: Clearly state the responsibility

Step 2: Look for classes that have the information we need to fulfill the responsibility.

Step 3: Domain Model or Design Model?

Step 4: Sketch out some interaction diagrams.

Step 5: Update the class diagram.

GRASP -Information Expert

- **Discussion Points:**

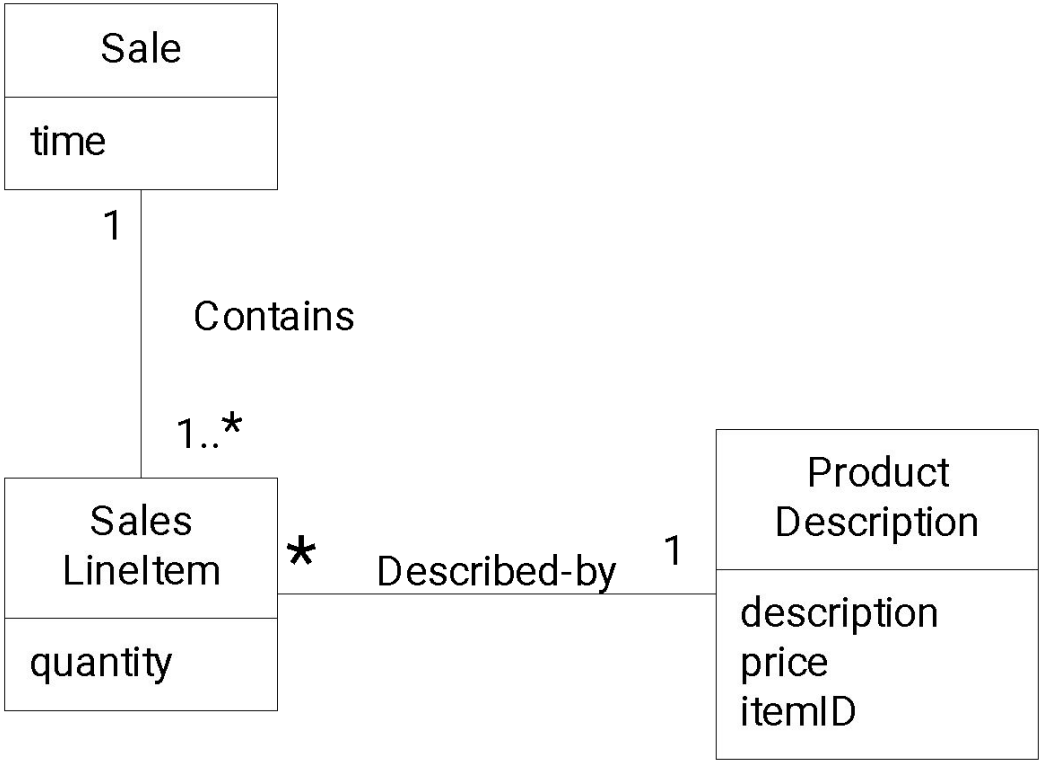
  – Throughout the process of assigning a responsibility, you may discover many lower-level responsibilities

- **Contradictions:**

  – Sometimes application of the Expert Pattern is undesirable or may have conflict with other responsibilities

Fig. 17.14

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



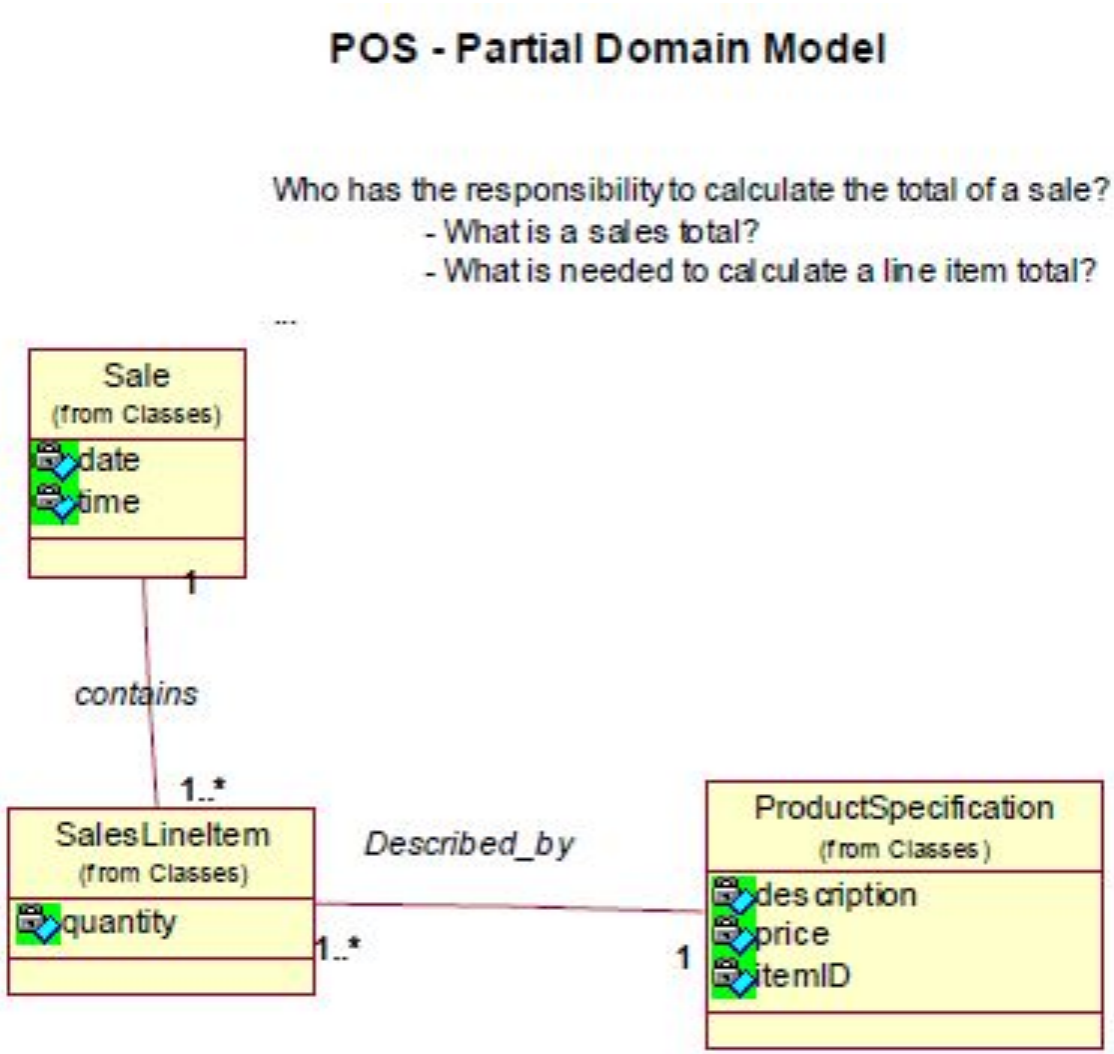
Domain Model Figure

Expert : Example

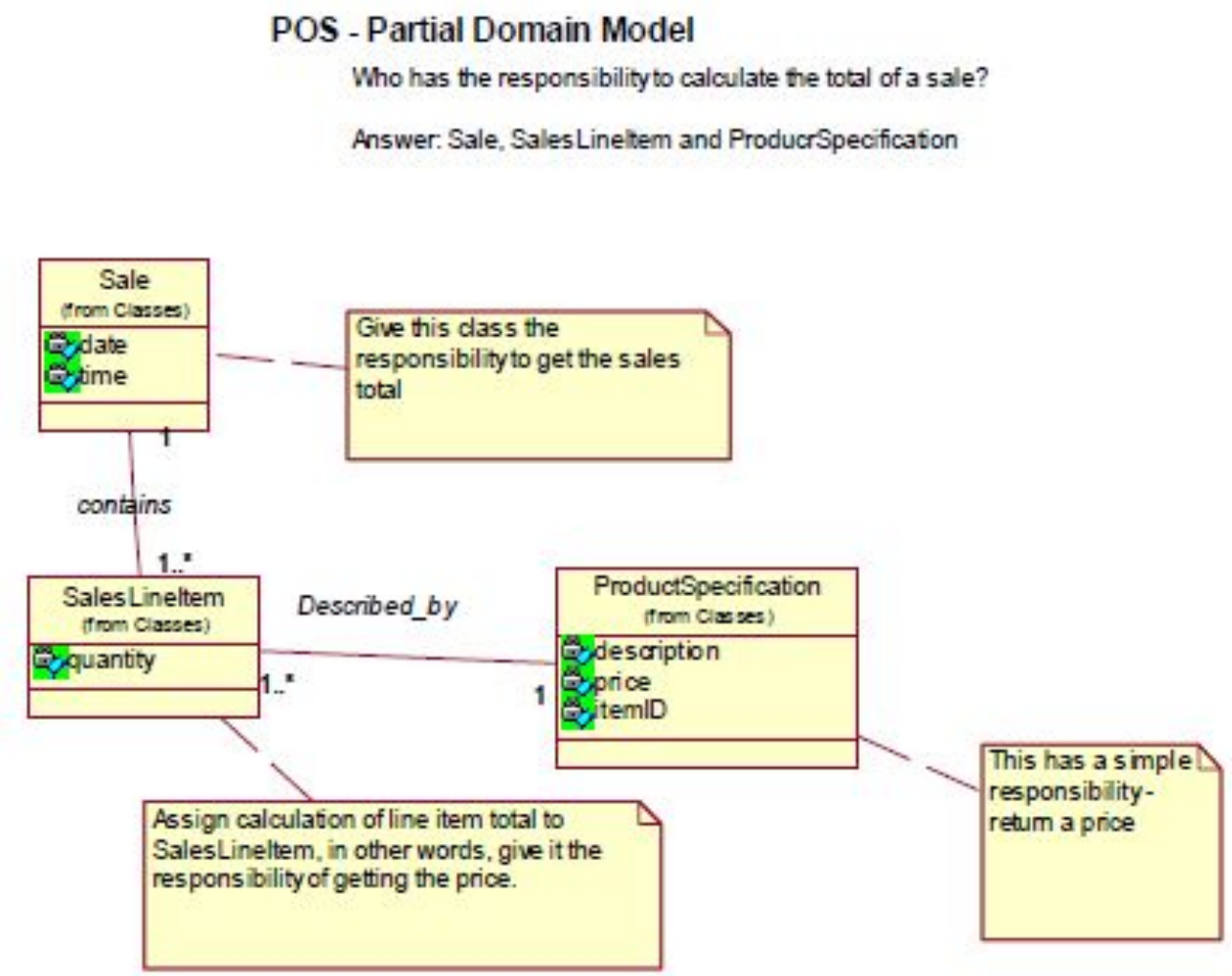
Hence responsibilities assign to the 3 classes.

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

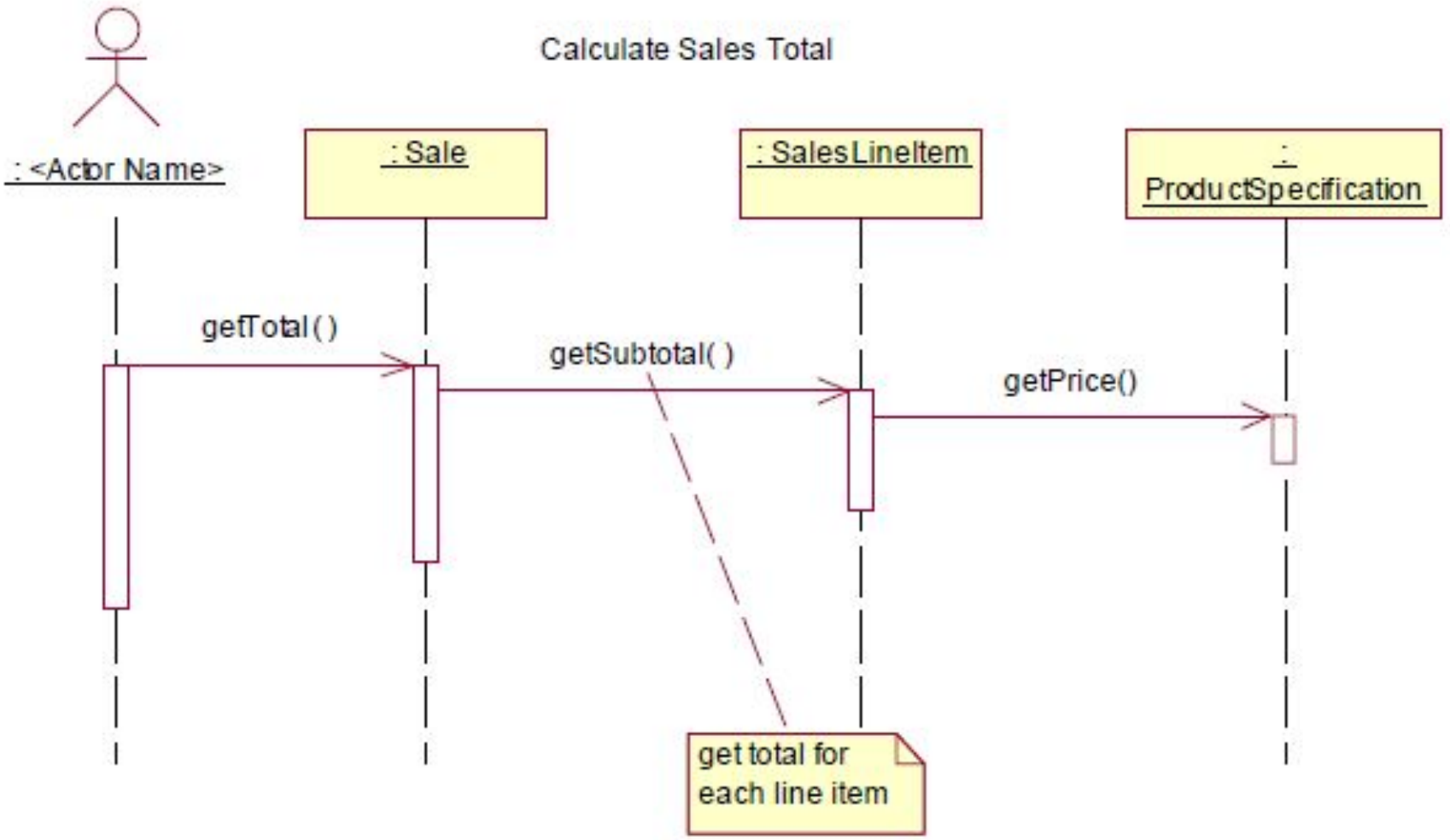
GRASP -Information Expert



GRASP -Information Expert



GRASP -Information Expert



Expert Pattern

- Maintains encapsulation of information (private data members).
- Promotes low coupling
- Promotes highly cohesive classes
- Can cause a class to become excessively complex

Additional Note (Coupling)…

- Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. These elements include classes, subsystems, systems and so on.
- A class with high coupling relies on many other classes. Highly coupled classes suffer the following problems:
  - Forced local changes because of changes in related classes
  - Harder to understand in isolation
  - Harder to reuse because its use requires the additional presence of its dependants.

Additional Note (Coupling)…

- Low coupling supports the design of classes that are more independent, which **reduces** the **impact of change**.
- Inheritance is a strong form of coupling. Any decision to create an inheritance relationship between two classes should be considered carefully.
- Is no coupling between classes a good thing?
  - An object-oriented system is a system of collaborating objects. Some moderate degree of coupling between classes is normal and necessary .

Grasp Patterns: Low Coupling

- Evaluatives and supports:
  - lower dependency between the classes,
  - change in one class having lower impact on other classes,
  - higher reuse potential.

GRASP: Controller

Problem:

To assign responsibility for handling a system event?

Solution:

Add an event class to decouple the event source(s) from the objects that actually handle the events

GRASP: Controller

The Controller pattern provides guidance for generally acceptable choices.

Assign the responsibility for handling a system event message to a class representing one of these choices:

1. The business or overall organization (a **facade** controller).
2. The overall "system" (a **facade** controller).
3. An animate thing in the domain that would perform the work (a role controller).
4. An artificial class (Pure Fabrication representing the use (a **use case** controller).

GRASP: Controller

Benefits:

Increased **potential for reuse**. Using a controller object keeps external event sources and internal event handlers independent of each other's type and behaviour.

Use Case controller ensures that the system operations occurs in legal sequence, or to be able to reason about the current state of activity and operations within the use case.

Controller : Example

System events in Buy Items use case

enterItem()

endSale()

makePayment()

who has the responsibility for ***enterItem()***?

Fig. 17.21

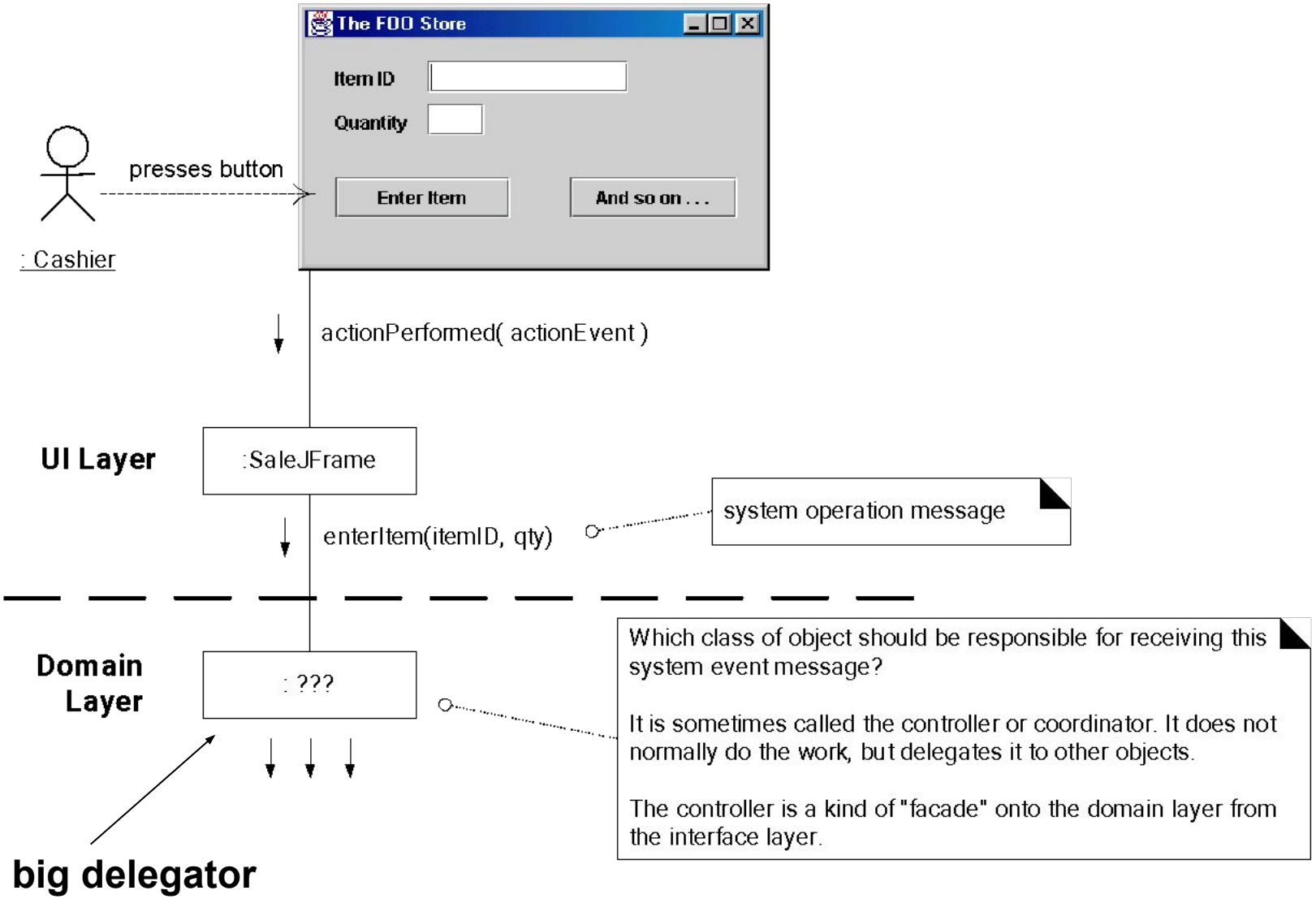

The FOO Store
Item ID
Quantity
Enter Item
And so on . . .
presses button
: Cashier
actionPerformed(actionEvent)
UI Layer
:SaleJFrame
enterItem(itemID, qty)
system operation message
Domain Layer
: ???
Which class of object should be responsible for receiving this system event message?
It is sometimes 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.
big delegator

Fig. 17.23

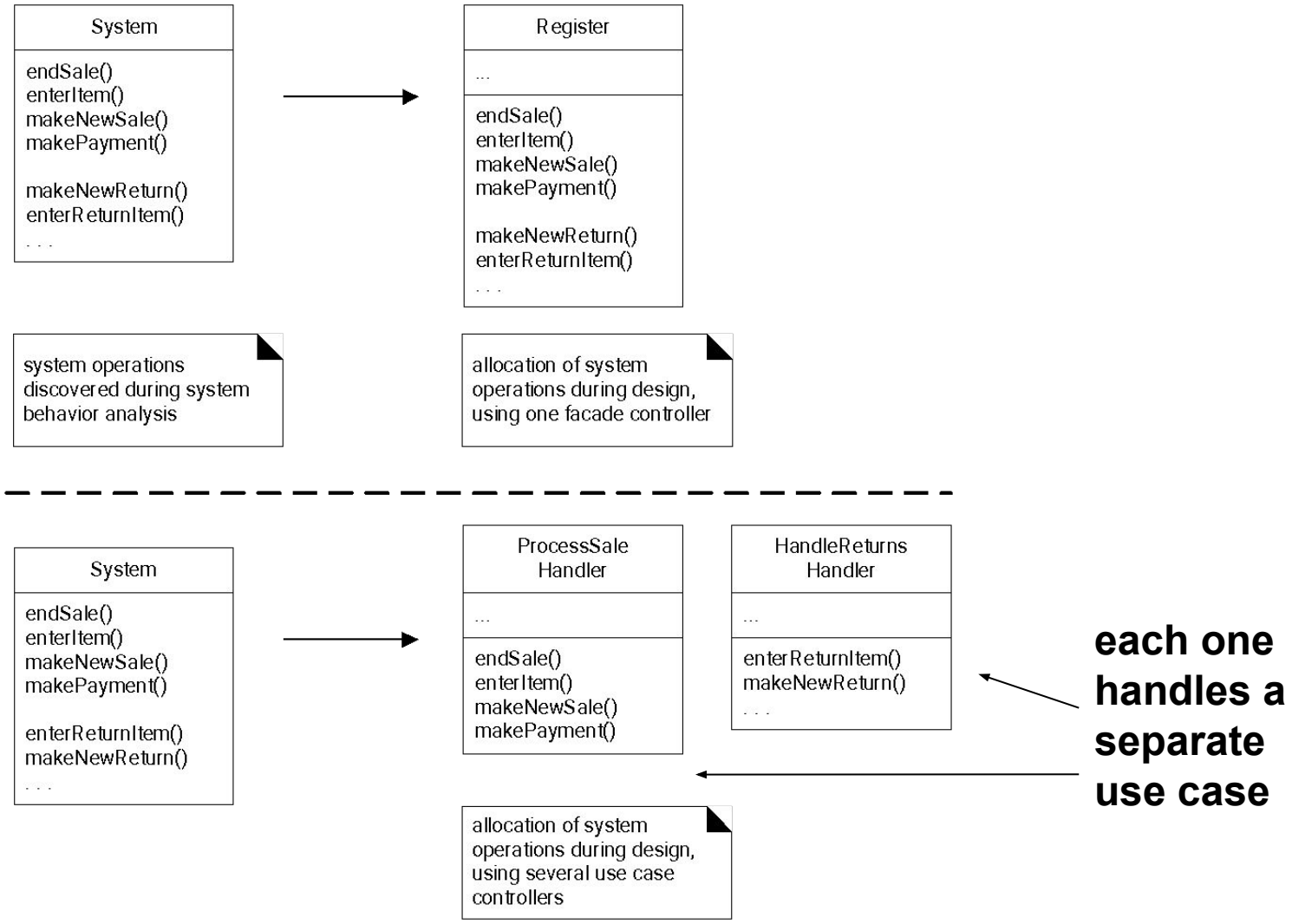

System
endSale()
enterItem()
makeNewSale()
makePayment()
makeNewReturn()
enterReturnItem()
. . .
Register
...
endSale()
enterItem()
makeNewSale()
makePayment()
makeNewReturn()
enterReturnItem()
. . .
system operations discovered during system behavior analysis
allocation of system operations during design, using one facade controller
System
endSale()
enterItem()
makeNewSale()
makePayment()
enterReturnItem()
makeNewReturn()
. . .
ProcessSale Handler
...
endSale()
enterItem()
makeNewSale()
makePayment()
HandleReturns Handler
...
enterReturnItem()
makeNewReturn()
. . .
each one handles a separate use case
allocation of system operations during design, using several use case controllers

Fig. 17.24

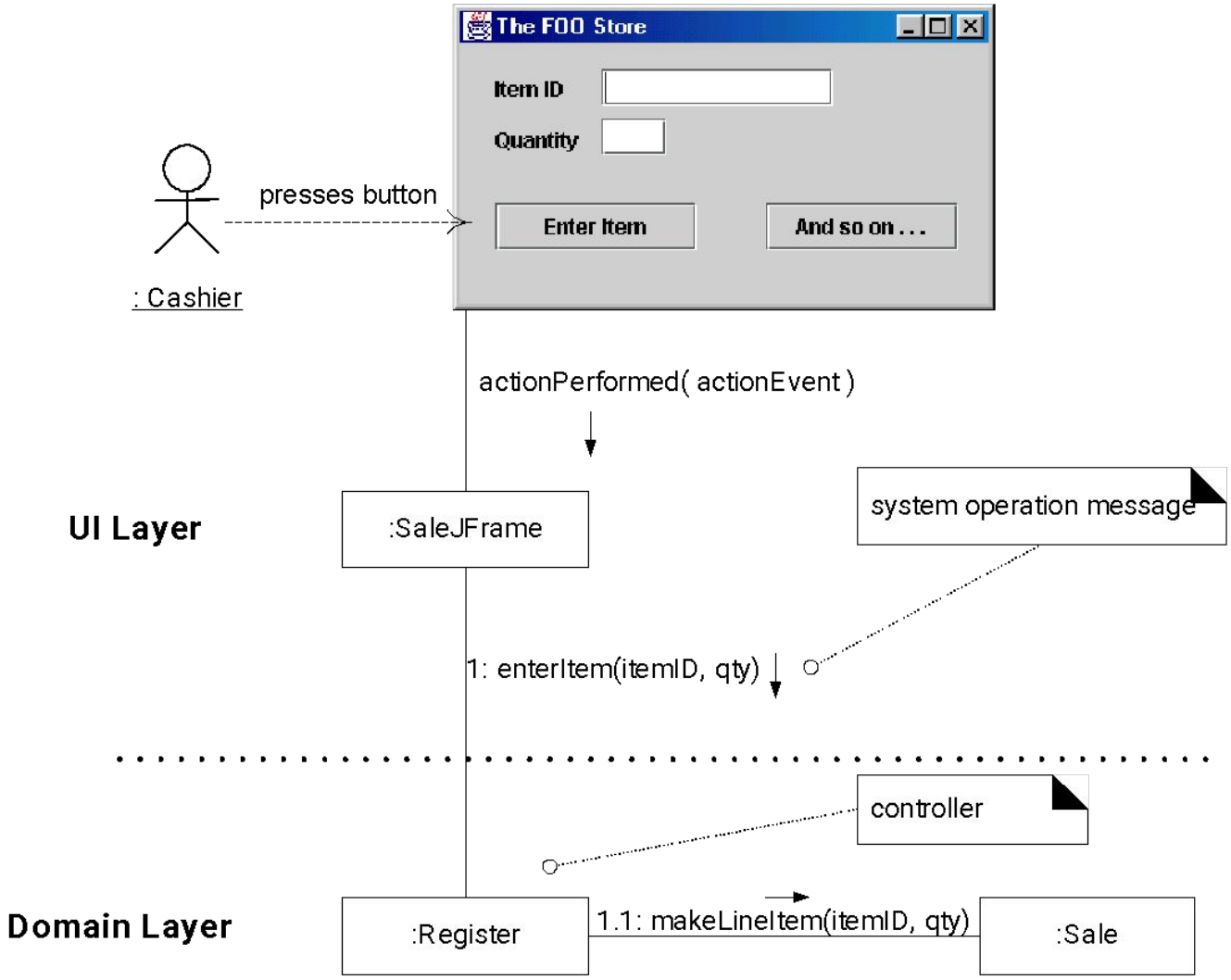
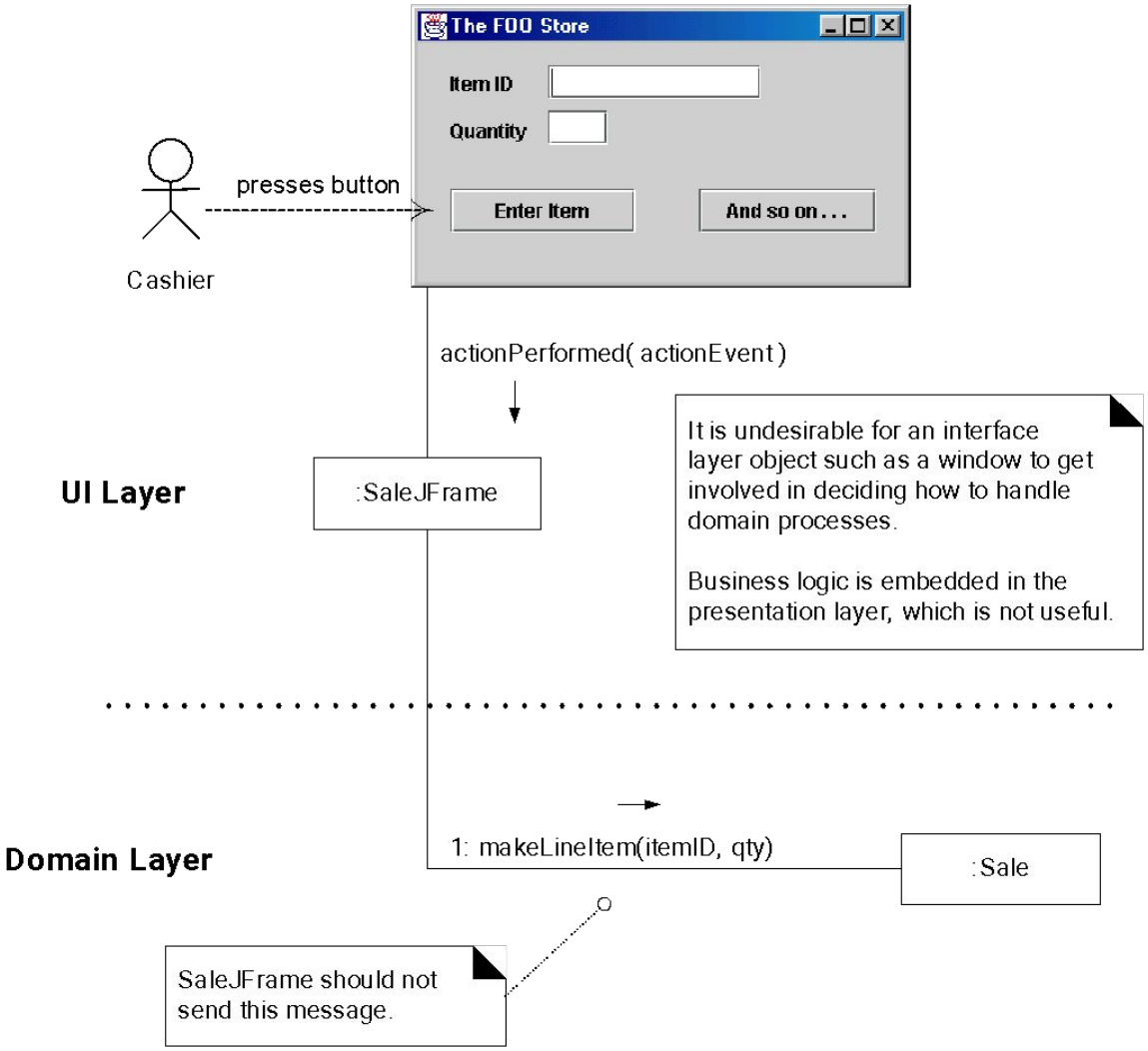
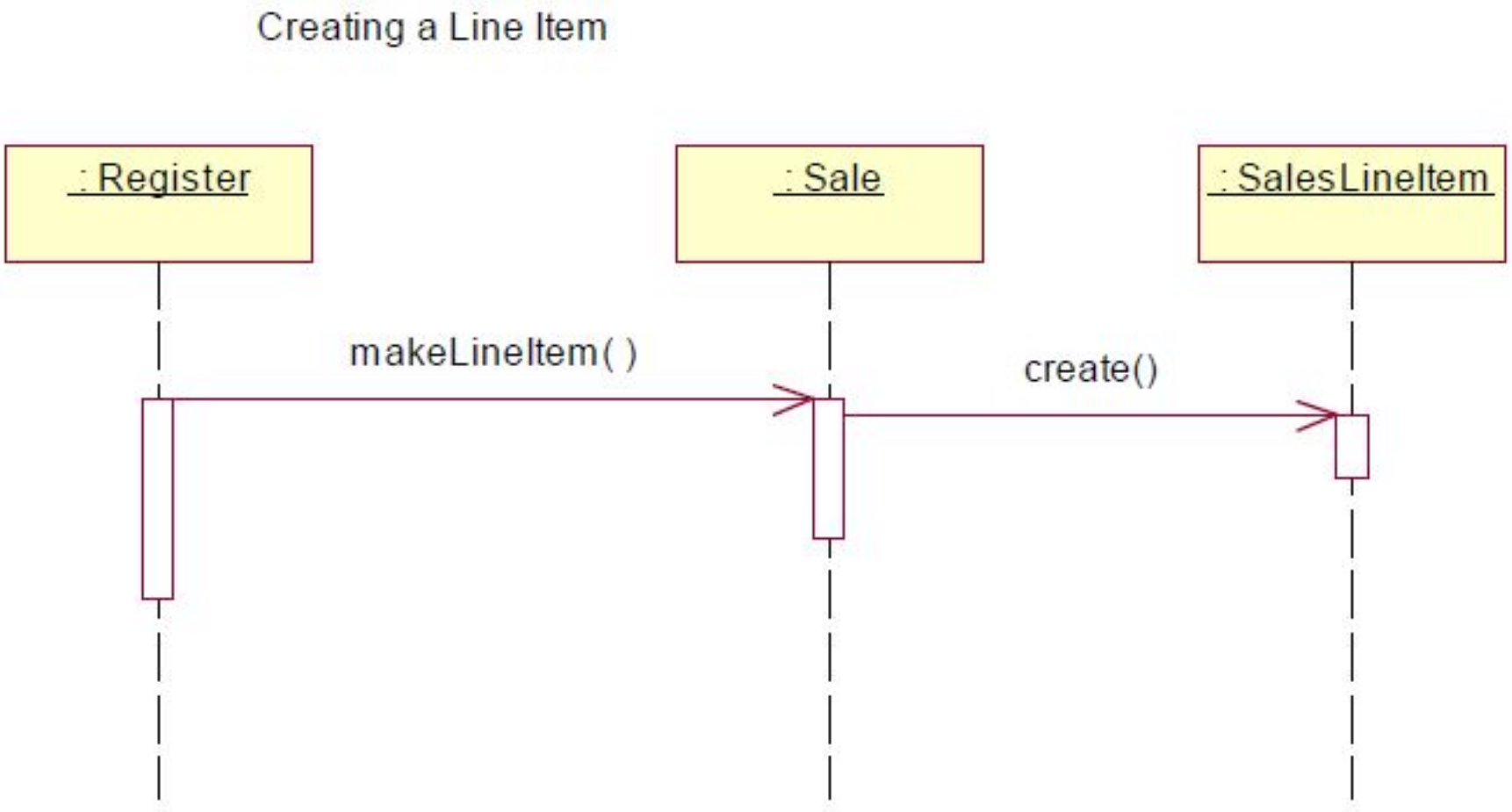

The FOO Store
Item ID
Quantity
Enter Item
And so on . . .
presses button
: Cashier
actionPerformed(actionEvent)
UI Layer
:SaleJFrame
system operation message
1: enterItem(itemID, qty)
controller
Domain Layer
:Register
1.1: makeLineItem(itemID, qty)
:Sale

Fig. 17.25


The FOO Store
Item ID
Quantity
Enter Item
And so on . . .
presses button
Cashier
actionPerformed(actionEvent)
UI Layer
:SaleJFrame
It is undesirable for an interface layer object such as a window 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(itemID, qty)
:Sale
SaleJFrame should not send this message.

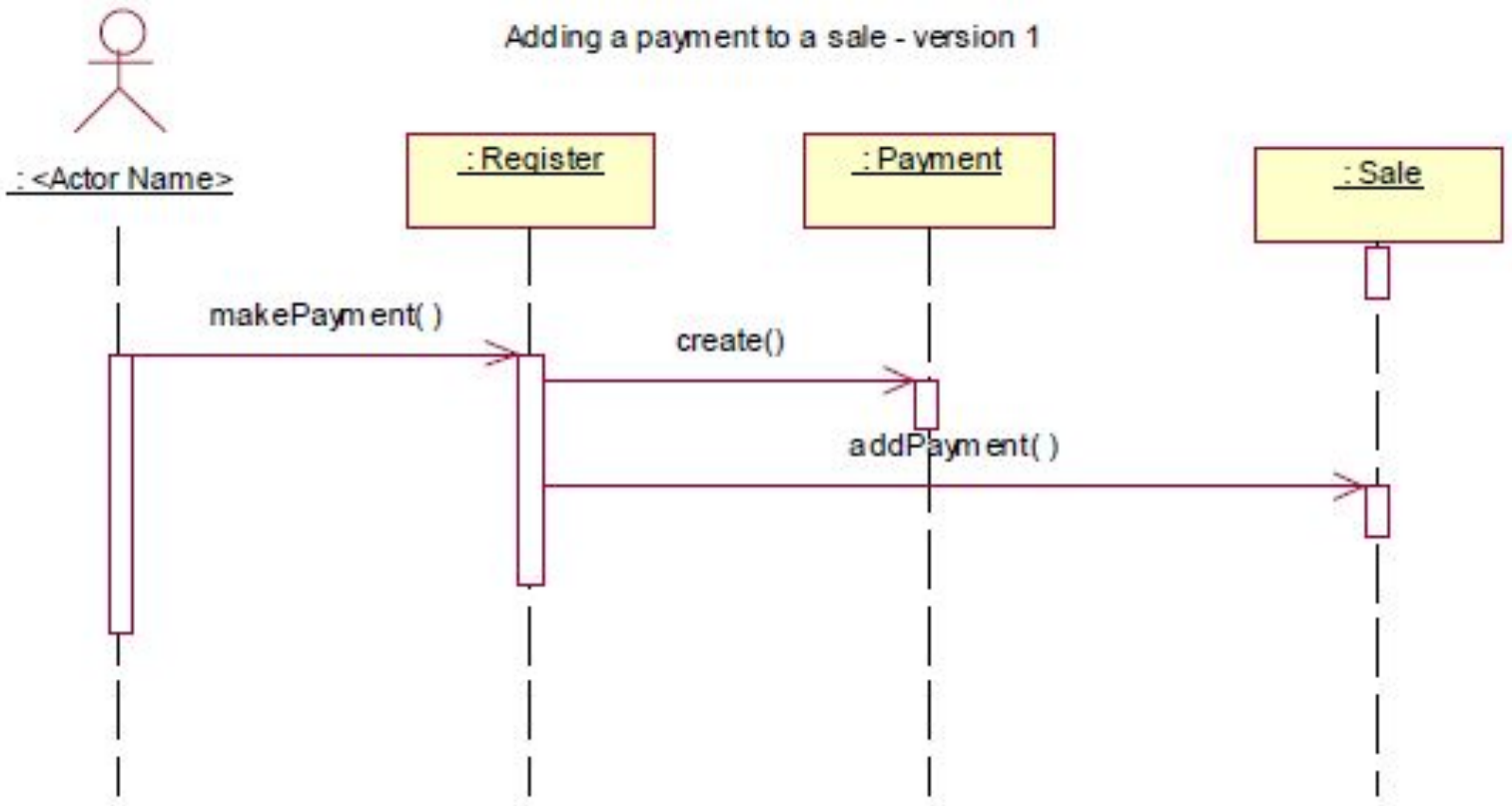
GRASP –Creator Pattern



Grasp Patterns

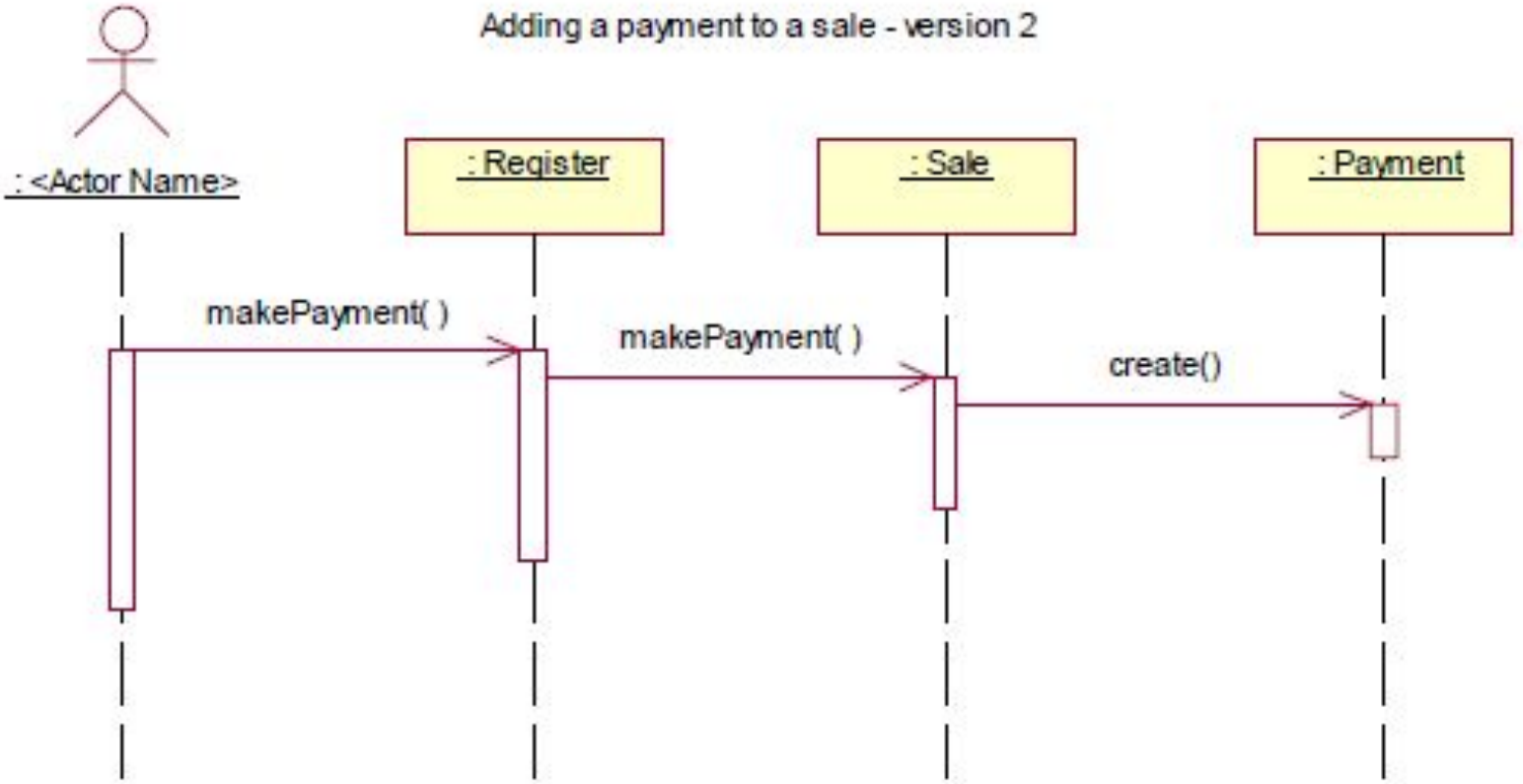
- **High Cohesion**: evaluative pattern; keep objects appropriately focused, manageable and understandable.
  - responsibilities are strongly related
- **Low Cohesion**: hard to comprehend, hard to reuse, hard to maintain and adverse to change

GRASP –Low Coupling



Add a payment to a sale - version 1. Note Register's responsibilities

GRASP –Low Coupling



This version let's Sale create a Payment - as opposed to Register creating one. Which version supports lower coupling? Why?

GRASP –High Cohesion

Concept –Cohesion:

- Cohesion is a measure of “relatedness”
- High Cohesion says elements are strongly related to one another
- Low Cohesion says elements are not strongly related to one another

Low Cohesion examples:

- System level : ATM with a use case (function) called “Teller Reports”
- Class level: A Student class with a method called “getDrivingRecord().
- Method level: Methods with the word “And” or “Or” in them.
- Also applies to subsystem (package) level, component level, etc.
- –Designs with low cohesion are difficult to maintain and reuse.
- –One of the fundamental goals of an effective design is to achieve high cohesion with low coupling

GRASP –High Cohesion

Problem: How do you keep complexity manageable?

Solution : Assign responsibility so that cohesion remains high.

Mechanics : Look for classes with too-few or disconnected methods.

- Look for methods that do too much (hint: method name)
- Rework your design as needed.

Contradictions:

- High Cohesion and low coupling are competing forces.

Assigning Responsibilities –Other Sources

- CRC Cards:
- –Another popular technique to assigning responsibilities to classes is to use CRC cards CRC = **Class: Responsibility: Collaboration**
- –Introduced by Kent Beck and Ward Cunnigham