Object-Oriented Analysis and Design

Session 6: Development Process

CASE STUDY: Point-of-Sale

- A Point Of Sale (POS) is a computerized application used (in part) to record sales and handle payments. Typically used in a retail store.
- Includes hardware and software.
- Interfaces to service applications, e.g., tax calculator and inventory control.
- Fault tolerant, e.g., remote services failure.
- Support varied client-side terminals and interfaces.
- Provide flexibility and customization -- different business rule processing, e.g., when a new sale is initiated.

Stages in a development cycle:

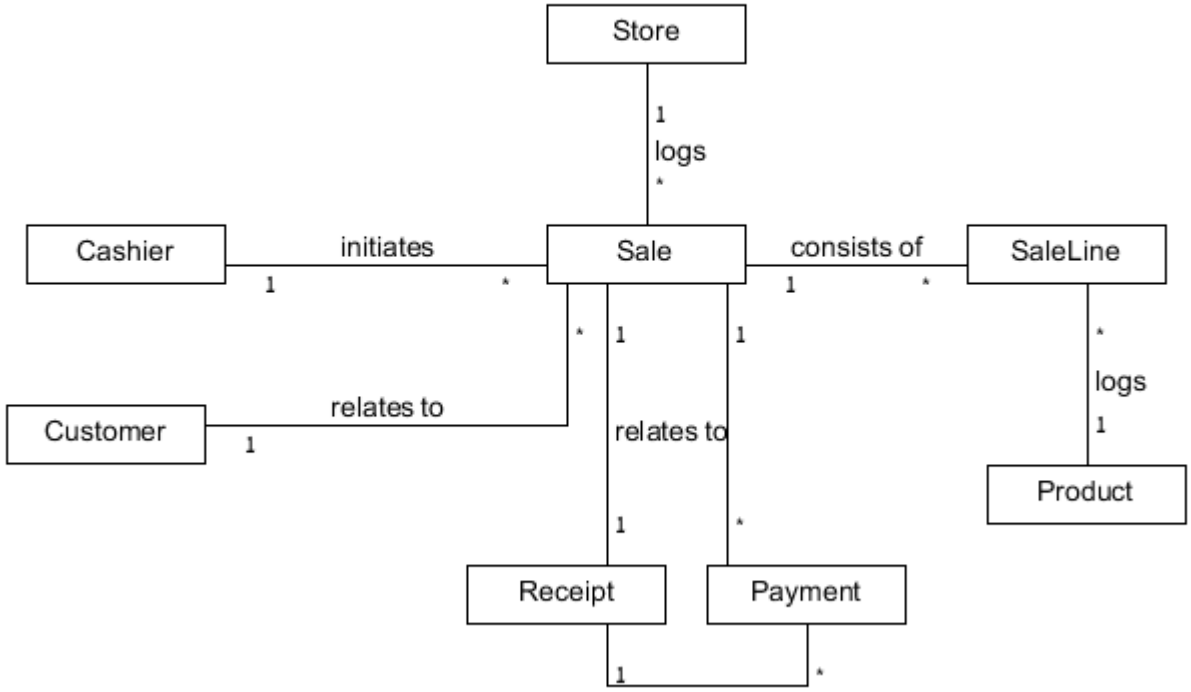
I. Understanding requirements – use cases.
II. Static modeling – Conceptual class diagram.
III. Identify operations and create their contracts.
IV. Assign responsibilities and design interaction diagrams.
V. Design level class diagrams, implementation and testing.

The stages ***are not*** performed in sequence.

Back to the User Stories

- As a cashier, I want to initiate a new sale, so that I can distinguish between the customers
- As a cashier, I want to register the sold products through their item identifier, so that I do not enter the wrong price
- As a cashier, I want to see the item description, so I can verify the item
- As a cashier, I want to see the item price so I can verify the price
- As a cashier, I want to see the running total, so I can inform the customer
- As a cashier, I want to obtain the tax-inclusive total, so that the customer pays the right amount
- As a customer, I want to see the item description and its price in a single line, so I can check the sale
- As a customer, I want to use various payment means
- As a customer, I want to receive a receipt for my purchase
- As a store, I want to send the sale logs to the inventory system, so that I can update and analyze the stock levels
- As a store, I want to send the payment information to the external accounting system, so that I comply with tax regulations

US-based Conceptual Model



I. Understanding requirements – Use Cases

- A use case is a *narrative document* – ***a story*** -- that describes the sequence of events of an actor using a system to complete a process.
- A use case describes a typical interaction between an external entity (the *actor* – person, system, organization) to the system.
- A use case describes multiple *scenarios*.
- A *scenario* is a single path through the use case.
- A use case depends on partial understanding of the requirements.
- Use cases can be found by looking for their initiator actors.

I. Understanding requirements – Use Cases

Example: Use case Process Sale, in a *detailed format*.

Use Case UC1: Process Sale

Primary Actor: Cashier
Stakeholders and Interests:
- Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer shortages are deducted from his/her salary.
- Salesperson: Wants sales commissions updated.
- Customer: Wants purchase and fast service with minimal effort. Wants proof of purchase to support returns.
- Company: Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory.
- Government Tax Agencies: Want to collect tax from every sale. May be multiple agencies, such as national, state, and county.
- Payment Authorization Service: Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.

I. Understanding requirements – Use Cases

Preconditions: Cashier is identified and authenticated.
Success Guarantee (Postconditions): Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.

Main Success Scenario (or Basic Flow):

1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.

Cashier repeats steps 3-4 until indicates done.

5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
8. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).
9. System presents receipt.
10. Customer leaves with receipt and goods (if any).

I. Understanding requirements – Use Cases

Extensions (or Alternative Flows):

*a. At any time, System fails:

To support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered from any step of the scenario.

1. Cashier restarts System, logs in, and requests recovery of prior state.
2. System reconstructs prior state.
   2a. System detects anomalies preventing recovery:
      1. System signals error to the Cashier, records the error, and enters a clean state.
      2. Cashier starts a new sale.

3a. Invalid identifier:

1. System signals error and rejects entry. 3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):
1. Cashier can enter item category identifier and the quantity.

3-6a: Customer asks Cashier to remove an item from the purchase:

1. Cashier enters item identifier for removal from sale.
2. System displays updated running total.

3-6b. Customer tells Cashier to cancel sale:

1. Cashier cancels sale on System.

3-6c. Cashier suspends the sale:

1. System records sale so that it is available for retrieval on any POS terminal. 4a.

I. Understanding requirements – Use Cases

4a. The system generated item price is not wanted (e.g., Customer complained about something and is offered a lower price):

1. Cashier enters override price.
2. System presents new price.

5a. System detects failure to communicate with external tax calculation system service:

1. System restarts the service on the POS node, and continues. 1a. System detects that the service does not restart.
    1. System signals error.
    2. Cashier may manually calculate and enter the tax, or cancel the sale.

5b. Customer says they are eligible for a discount (e.g., employee, preferred customer):

1. Cashier signals discount request.
2. Cashier enters Customer identification.
3. System presents discount total, based on discount rules.

5c. Customer says they have credit in their account, to apply to the sale:

1. Cashier signals credit request.
2. Cashier enters Customer identification.
3. Systems applies credit up to price=0, and reduces remaining credit.

6a. Customer says they intended to pay by cash but don't have enough cash:

1a. Customer uses an alternate payment method.
1b. Customer tells Cashier to cancel sale. Cashier cancels sale on System.

I. Understanding requirements – Use Cases

7a. Paying by cash:

1. Cashier enters the cash amount tendered.
2. System presents the balance due, and releases the cash drawer.
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. System records the cash payment.

7b. Paying by credit:

1. Customer enters their credit account information.
2. System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.
   2a. System detects failure to collaborate with external system:
      1. System signals error to Cashier.
      2. Cashier asks Customer for alternate payment.
3. System receives payment approval and signals approval to Cashier.
   3a. System receives payment denial:
      1. System signals denial to Cashier.
      2. Cashier asks Customer for alternate payment.
4. System records the credit payment, which includes the payment approval.
5. System presents credit payment signature input mechanism.
6. Cashier asks Customer for a credit payment signature. Customer enters signature.

I. Understanding requirements – Use Cases

7c. Paying by check...
7d. Paying by debit...
7e. Customer presents coupons:
    1. Before handling payment, Cashier records each coupon and System reduces price as appropriate. System records the used coupons for accounting reasons.
    1a. Coupon entered is not for any purchased item:
        1. System signals error to Cashier. 9a.
There are product rebates:
    1. System presents the rebate forms and rebate receipts for each item with a rebate.
9b. Customer requests gift receipt (no prices visible): 1.
    Cashier requests gift receipt and System presents it.

I. Understanding requirements – Use Cases

Special Requirements:

- Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
- Credit authorization response within 30 seconds 90% of the time.
- Somehow, we want robust recovery when access to remote services such the inven tory system is failing.
- Language internationalization on the text displayed.
- Pluggable business rules to be insertable at steps 3 and 7.

Technology and Data Variations List:

3a. Item identifier entered by bar code laser scanner (if bar code is present) or key-board.

3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.

7a. Credit account information entered by card reader or keyboard.

7b. Credit payment signature captured on paper receipt. But within two years, we pre-dict many customers will want digital signature capture.

I. Understanding requirements – Use Cases

Frequency of Occurrence: Could be nearly continuous.

Open Issues:

- What are the tax law variations?
- Explore the remote service recovery issue.
- What customization is needed for different businesses?
- Must a cashier take their cash drawer when they log out?
- Can the customer directly use the card reader, or does the cashier have to do it?

II. Static modeling -- Conceptual class diagram.

- Described by class diagrams.
- Iterative development.

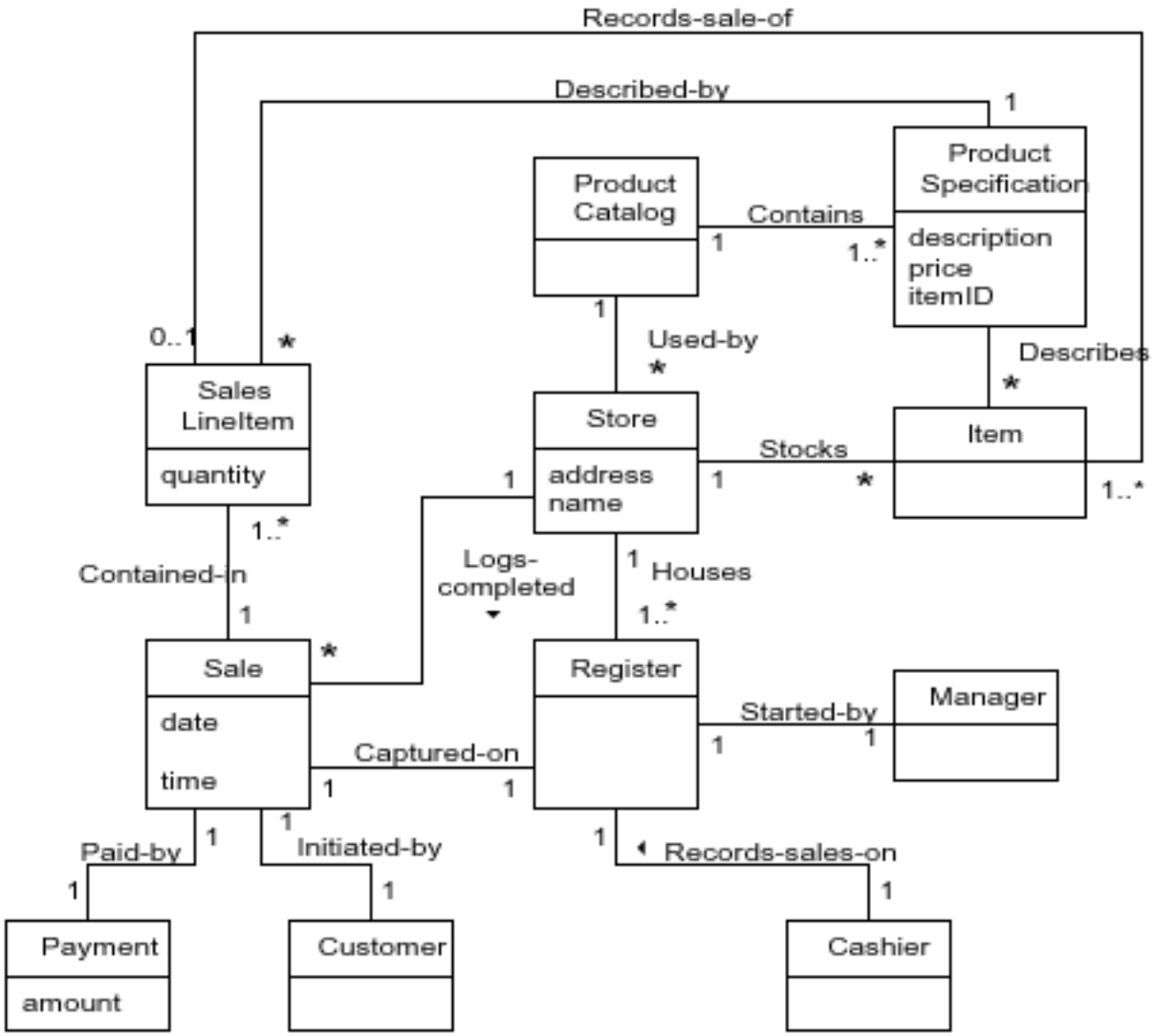
In the analysis stage:

- White diagrams: Classes, Associations, Multiplicity constraints.
- Add attributes, Qualifiers.
- Advanced features: Generalization (sub-typing), Aggregation, Association classes, packaging.

In the design stage -- add:

- Navigability (directions to associations).
- Types for attributes.
- Methods.

II. Static modeling -- Conceptual class diagram.



III. Identify operations and create their contracts

Steps:

1. Identify *system events* that trigger *system operations*.

2. Create contracts for the identified operations.

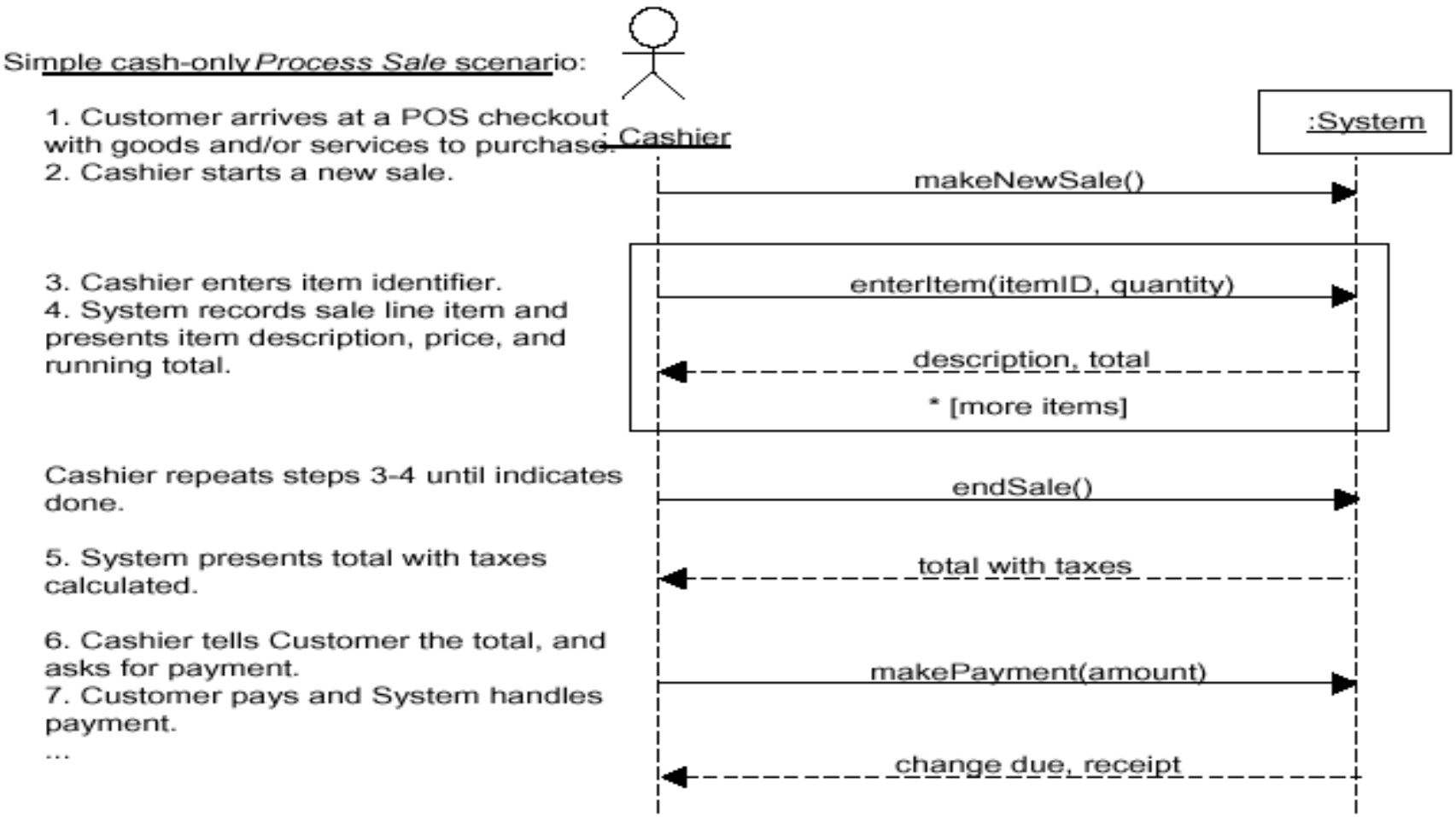
III. Identify operations and create their contracts

System events:

- The system events and their associated operations are identified by observing the course of events in use cases.
- Within a use case – follow the actions of actors that **directly** interact with the system.
- For the simple **Cash-only** scenario of the **Process Sale** use case, the cashier is the only actor that directly interacts with the system. It generates the events:
  - *makeNewSale*()
  - *enterItem*(*itemID*, *quantity*)
  - *endSale*()
  - *makePayment*(*amount*)

III. Identify operations and create their contracts

System events: The events can be described in a *System Sequence Diagram*:



III. Identify operations and create their contracts

System sequence diagrams are important!

1. They serve for connecting the application logic layer to other layers:
   - If the actor is human – connection to the presentation layer.
   - If the actor is an external service – interface.

2. They serve for implementing the use case scenarios.

III. Identify operations and create their contracts

A *contract* is a document that describes what an operation commits to achieve. A contract is expressed with:

- **pre-conditions** – Assumptions about the state of the system at the beginning of the operation.
- **post-conditions** – State of the system after the operation has finished:
  - Instance creation and deletion.
  - Attribute modification.
  - Links formed and broken.

Goal: Create contracts for complex system operations.

Contracts are used to derive interaction diagrams for the operations.

III. Identify operations and create their contracts

Create contracts:

Contract CO2: enterItem

Operation: Cross References:	enterItem(itemID : ItemID, quantity : integer) Use Cases: Process Sale There is a sale underway.
Preconditions:	
Postconditions:	- A SalesLineItem instance sli was created (instance creation). - sli was associated with the current Sale (association formed). -sli.quantity became quantity (attribute modification). - sli was associated with a ProductSpecification, based on itemID match (association formed).

Post conditions specify *what* must happen during a system operation, but not *how*!

III. Identify operations and create their contracts

Create contracts – The POS example:

System Operations of Process Sale

Contract CO1: makeNewSale

Operation: Cross	makeNewSale()
References:	Use Cases: Process Sale
Preconditions:	none
Postconditions:	- A Sale instance s was created (instance creation). - s was associated with the Register (association formed). - Attributes of s were initialized.

III. Identify operations and create their contracts

Create contracts – The POS example:

Contract CO2: enterItem

Operation: Cross	enterItem(itemID : ItemID, quantity : integer) Use
References:	Cases: Process Sale There is a sale underway.
Preconditions:	
Postconditions:	- A SalesLineItem instance sli was created (instance cre ation). - sli was associated with the current Sale (association formed). -sli.quantity became quantity (attribute modification). - sli was associated with a ProductSpecification, based on itemID match (association formed).

III. Identify operations and create their contracts

Create contracts – The POS example:

Contract CO3: endSale

Operation: Cross	endSaleQ
References:	Use Cases: Process Sale
Preconditions:	There is a sale underway.
Postconditions:	- *Sale.isComplete became true (attribute modification).*

III. Identify operations and create their contracts

Create contracts – The POS example:

Contract CO4: makePayment

Operation: Cross References:	makePayment(amount: Money) Use Cases: Process Sale There is a sale
Preconditions:	underway.
Postconditions:	- *A Payment instance p was created (instance creation).* - *p.amountTendered became amount (attribute modification).* - *p was associated with the current Sale (association formed).* - *The current Sale was associated with the Store (associa tion formed); (to add it to the historical log of completed sales)*

III. Identify operations and create their contracts

Create contracts:

1. Contracts might imply changes to the static domain model:

   For example, add a boolean *isComplete* attribute to the ***Sale*** class, to mark the end of a sale.

2. Contracts for operations can be expressed in OCL.

   Preconditions and postconditions are written as OCL constraints in the context of the contract operation.

IV. Assign responsibilities and Design interaction diagrams

Two interwoven tasks:

1. Impose *responsibilities* – assign operations to classes.

2. Design *operations* – sequence/collaboration diagrams.

IV. Assign responsibilities and Design interaction diagrams

Input to this stage: A contract built for a system operation.

Output of this stage: An interaction diagram.

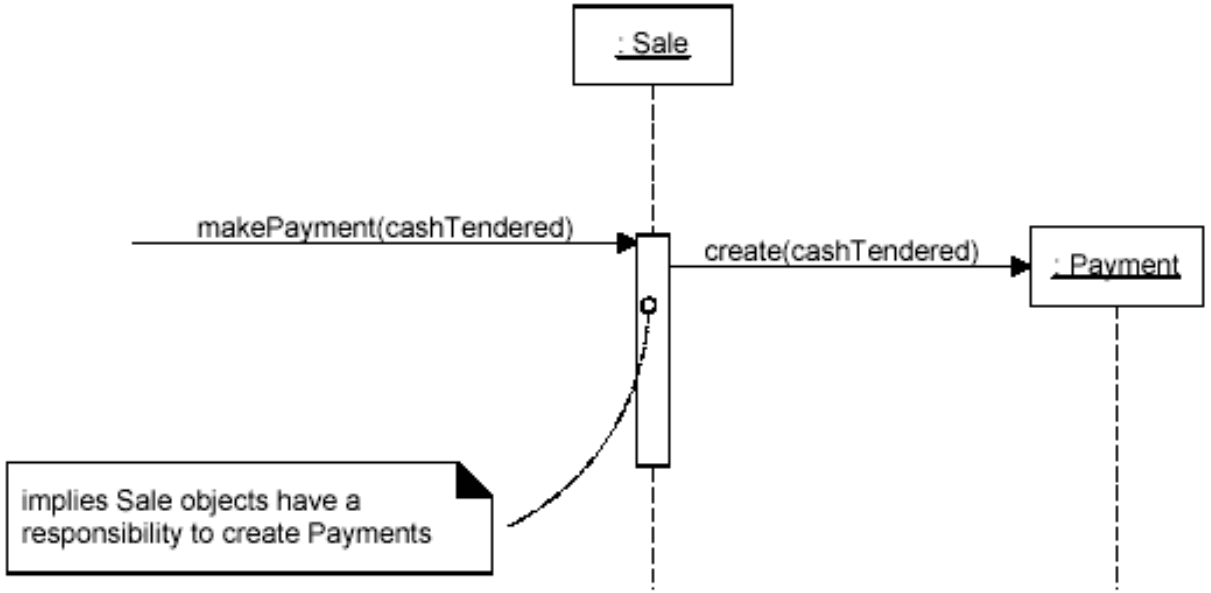
The design is based on Principles (Patterns) for Assigning Responsibilities –

GRASP (General Responsibility Assignment Software Patterns).

IV. Assign responsibilities and Design interaction diagrams

Responsibility assignment:

Determine the class whose objects are responsible for performing an operation. The operation becomes a method of that class.



IV. Assign responsibilities and Design interaction diagrams

Responsibility assignment - Based on **patterns**

A **pattern** is a *named problem/solution* pair that can be applied in new context, with advice on how to apply it in novel situations and discussion of its trade-offs.

- Patterns exist in *various paradigms*.
- In *Software Engineering* they are *idioms* for software creation.
- They are *expressed* in narrative documents, programming code and more quasi formal languages like UML diagrams.

IV. Assign responsibilities and Design interaction diagrams

Responsibility assignment - Based on **patterns**

Major GRASP patterns:

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

They address very basic, common questions and fundamental design issues.

IV. Assign responsibilities and Design interaction diagrams

Responsibility assignment - Based on **patterns**

GRASP patterns are introduced as required for the design of interaction diagrams for the POS problem.

We consider 2 use cases:

- **Process Sale.**
- **Start-up**.

IV. Assign responsibilities and Design interaction diagrams

Reminder: The overall steps towards the development of interaction diagrams:

Use cases + conceptual model

→ Identify system events (operations) – use SSDs.

→ Build contracts

→ Interaction diagrams.

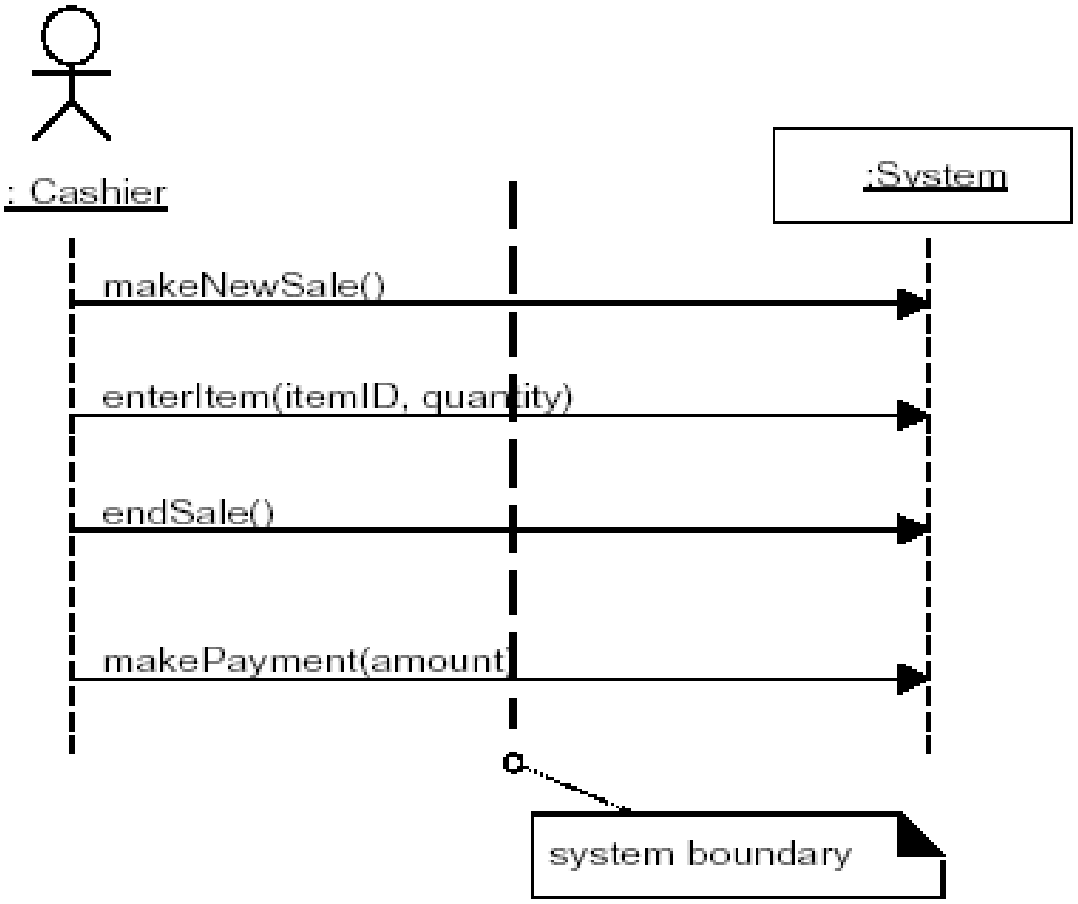
IV. Assign responsibilities and Design interaction diagrams

Importance of interaction diagrams:

- Express decisions about responsibilities for operations.
- Complement the conceptual model.
- The conceptual model and the interaction diagrams are the most important artifacts created in the object-oriented analysis and design.
- Good quality interaction diagrams express codified patterns, principles and idioms.

IV. Assign responsibilities and Design interaction diagrams

The events singled out for the Process Sale use case:



IV. Assign responsibilities and Design interaction diagrams

First question:

Which class is responsible for each operation?

An advice is given by the *Controller pattern*.

Its problem is:

"Who should be responsible for handling an input system event?"

IV. Assign responsibilities and Design interaction diagrams

The Controller Pattern

Problem: Who should be responsible for handling a *system event*?

- The controller pattern provides an advice for selecting a controller for a system event – an operation triggered by an external actor.
- A **controller** is a domain layer object responsible for handling a system event. A controller defines a method for a system operation.

IV. Assign responsibilities and Design interaction diagrams

The Controller Pattern

Solution: Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:

- *Facade controller* -- Represents the overall system, device, or subsystem.
- *Use case controller* -- Represents a use case scenario within which the system event occurs.

  Common name: *<UseCaseName>Handler*.
- *Role controller* – Represents something that can be involved in the task.

IV. Assign responsibilities and Design interaction diagrams

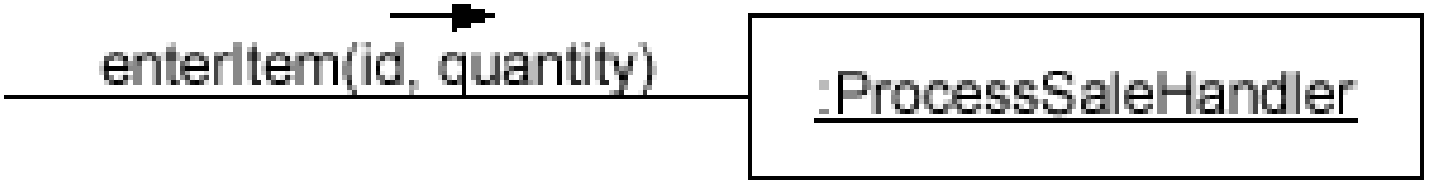
The Controller Pattern

For the *enterItem* event in the Process-sale use case:

- Possible facade controllers are:
  - *Register* – represents the overall system.
  - *Store* – represents the overall organization.
- Possible role controller is *Cashier* – represents something that can be involved in the task.
- Use case controller: *ProcessSaleHandler*– its instances represent sessions (conversations) with an actor.
  - In that case – all system events in the same use case are handled by the same controller.

IV. Assign responsibilities and Design interaction diagrams

The Controller Pattern – possible options for the *enterItem* event:



IV. Assign responsibilities and Design interaction diagrams

The Controller Pattern

A facade controller is a good option if there are only few system events.

Otherwise – the façade controller takes too many Responsibilities

→ ***It becomes incohesive*.**

IV. Assign responsibilities and Design interaction diagrams

The Controller Pattern

Decision among various options suggested by a pattern:

Use the general criteria:

- ***High cohesion.***
- ***Low coupling.***

IV. Assign responsibilities and Design interaction diagrams

The Controller Pattern

For the events of the *Process Sale* use case:

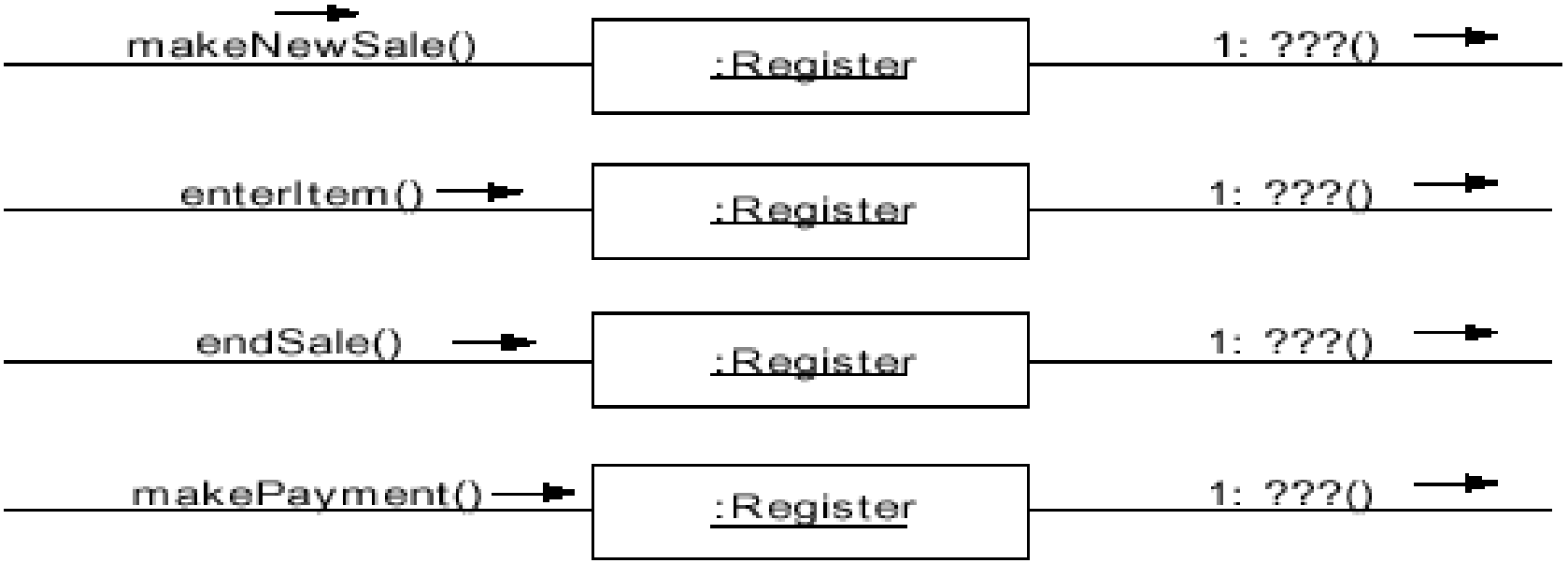
Possible façade controllers are:

Register, Store.

Based on cohesion → select *Register*.

IV. Assign responsibilities and Design interaction diagrams

The Controller Pattern



IV. Assign responsibilities and Design interaction diagrams

Design interaction diagrams –

Use contracts built for the system events that are singled out for a use case.

In each contract – use the post conditions section!

For the Process Sale use case --

4 interaction diagrams for the events:

makeNewSale, enterItem, endSale, makePayment.

IV. Assign responsibilities and Design interaction diagrams

Design interaction diagrams –

System events:

1. ***makeNewSale.***
2. ***enterItem.***
3. ***endSale.***
4. **Sale – getTotal:** Presentation requirement.
5. ***makePayment.***
6. **Payment – getBalance:** Presentation requirement.
7. ***startUp***.

IV. Assign responsibilities and Design interaction diagrams

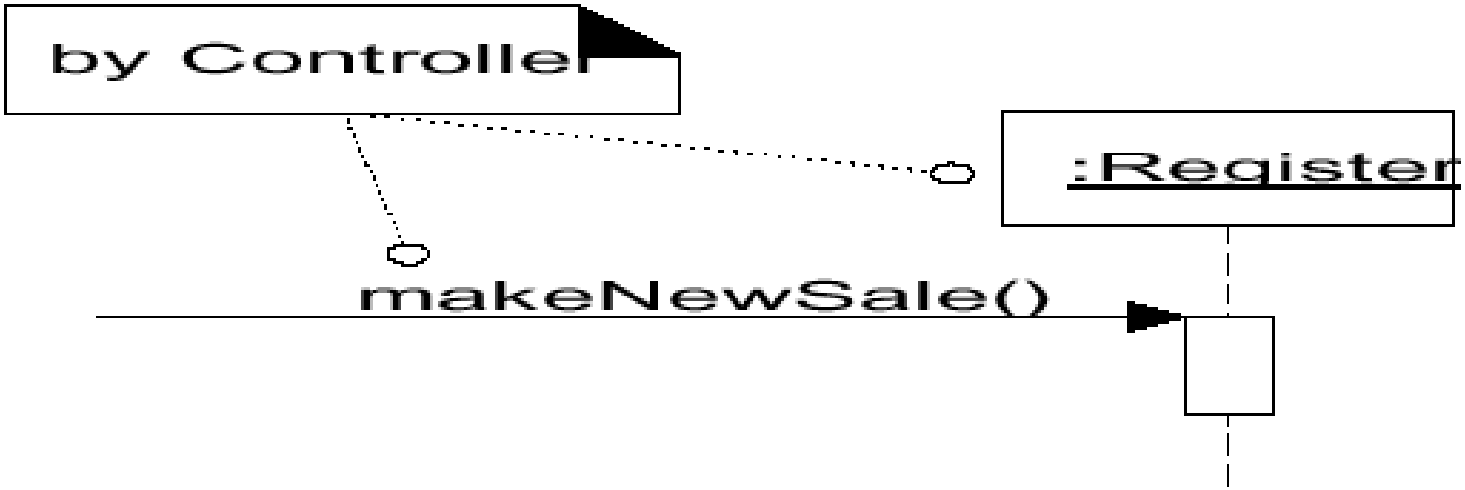
Contract CO1: makeNewSale

Operation: Cross	makeNewSale()
References:	Use Cases: Process Sale
Preconditions:	none
Postconditions:	- A Sale instance s was created (instance creation). - s was associated with the Register (association formed). - Attributes of s were initialized.

IV. Assign responsibilities and Design interaction diagrams

Interaction diagram for *makeNewSale*:

First decision: By the *Controller* pattern – assign responsibility on the *Register* class.



IV. Assign responsibilities and Design interaction diagrams

Interaction diagram for *makeNewSale*:

Post condition -- instance creation:

"A *Sale* instance was created".

Use the *Creator* pattern for determining the object that triggers the *Sale* constructor – *the Creator*.

IV. Assign responsibilities and Design interaction diagrams

The Creator pattern

Problem: Who should be responsible for creating a new instance of some class?

Solution: Assign class B the responsibility to create an instance of class A if one of the following holds:

- B *aggregates* objects of A.
- B *contains* objects of A.
- B *records* instances of objects of A.
- B closely *uses* objects of A.
- B has *initializing data* that will be passed to A when it is created (B is an *Expert* with respect to creating A).

IV. Assign responsibilities and Design interaction diagrams

The Creator pattern -- In the *makeNewSale* event:

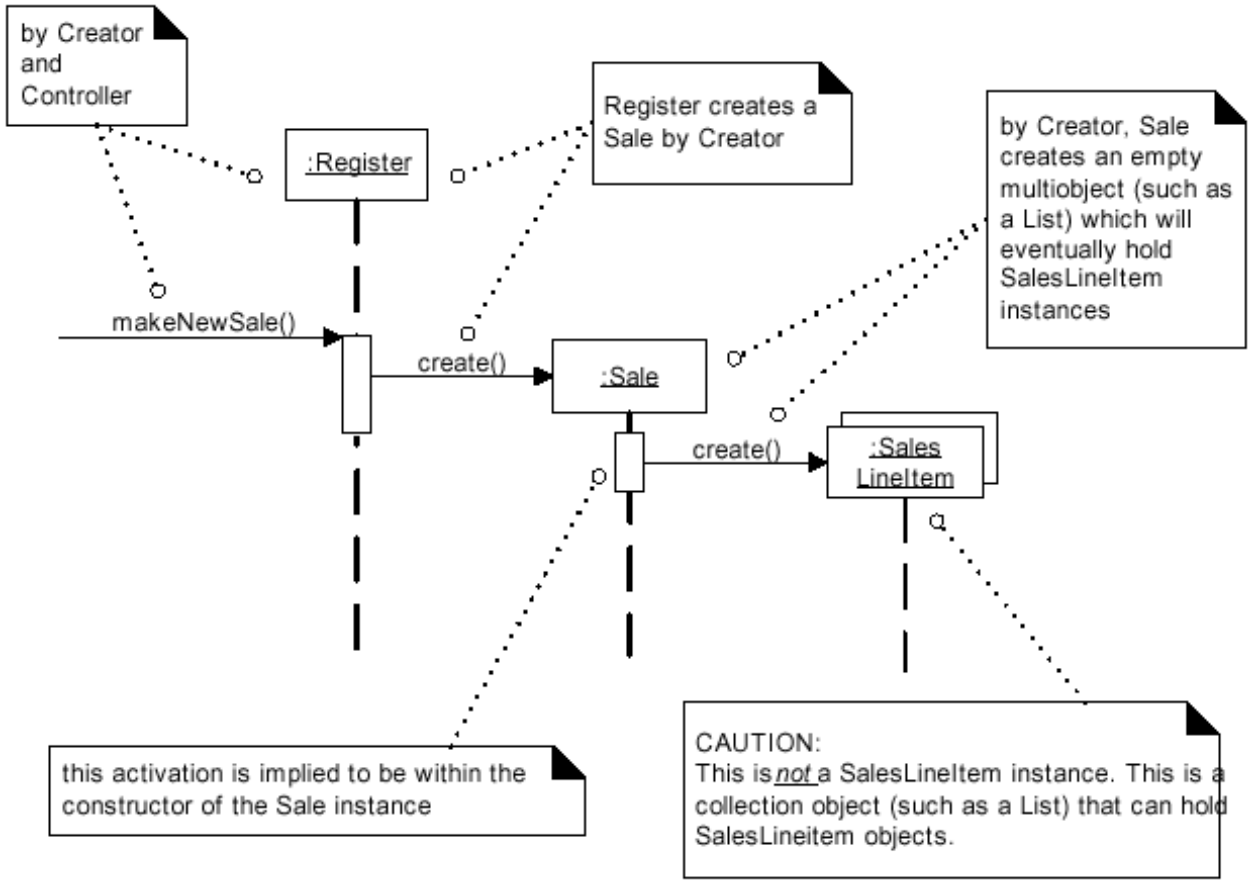
- *Sale* can be a creator for *SalesLineItem* – A *Sale* instance *contains* *SalesLineItem* objects.
- Register can be a creator for Sale – Register *records* and is *associated with* Sale objects.

IV. Assign responsibilities and Design interaction diagrams

Interaction diagram for *makeNewSale*:

Sale is a container for *SaleLineItem* objects

→ A *Sale* Constructor must create a container (a collection) for *SaleLineItems* objects.



IV. Assign responsibilities and Design interaction diagrams

Design interaction diagrams –

System events:

√ 1. ***makeNewSale.***

2. ***enterItem.***

3. ***endSale.***

4. **Sale – getTotal:** Presentation requirement.

5. ***makePayment.***

6. **Payment – getBalance:** Presentation requirement.

7. ***startUp***.

IV. Assign responsibilities and Design interaction diagrams

Contract CO2: enterItem

Operation: Cross References:	enterItem(itemID : ItemID, quantity : integer) Use Cases: Process Sale There is an underway sale.
Preconditions:	
Postconditions:	- A SalesLineItem instance sli was created (instance creation). - sli was associated with the current Sale (association formed). - sli.quantity became quantity (attribute modification). - sli was associated with a ProductSpecification, based on itemID match (association formed).

IV. Assign responsibilities and Design interaction diagrams

Interaction diagram for *enterItem*:

3 first post conditions:

- A *SalesLineItem* instance *sli* was created (instance creation).
- - *sli* was associated with the current *Sale* (association formed).
- - *sli.quantity* became quantity (attribute modification).

→

creation, initialization, association (linking) of a *SalesLineItem* object.

IV. Assign responsibilities and Design interaction diagrams

Interaction diagram for *enterItem*:

Creator for a *SalesLineItem* object.

- *Creator* -- Sale is the creator. It *contains* SalesLineItems objects.
- *Association over time* – the new object is stored in the collection of SalesLineItem objects contained in Sale.
- *Quantity*: The new SalesLineItem has quantity. Must be passed as a parameter.

IV. Assign responsibilities and Design interaction diagrams

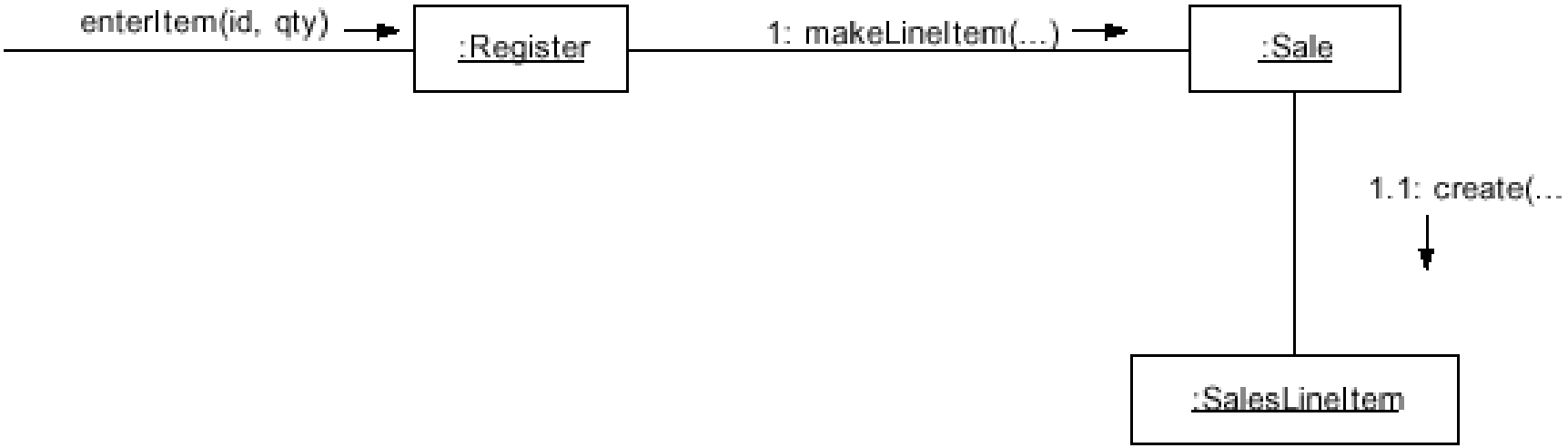
Interaction diagram for *enterItem*:

Creator for a *SalesLineItem* object.

- *Register* → *Sale*: *makeLineItem* message.
- *Sale* creates a *SalesLineItem* and stores it in its permanent collection.

IV. Assign responsibilities and Design interaction diagrams

Interaction diagram for *enterItem – so far*:



IV. Assign responsibilities and Design interaction diagrams

Interaction diagram for *enterItem*:

Parameters to *makeLineItem*:

First parameter: *quantity* – recoded by *SalesLineItem*.

4th postcondition:

sli was associated with a *ProductSpecification*, based on itemID match.

→

Second parameter: *productSpecification* – matches the ID code.

IV. Assign responsibilities and Design interaction diagrams

Interaction diagram for *enterItem*:

Finding a *productSpecification* based on an ID code.

Task: Retrieve a *ProductSpecification* based on an ID code match.

Question: Who is responsible for knowing the *ProductSpecification* based on an ID code match?

→ Use the **Expert pattern.**

IV. Assign responsibilities and Design interaction diagrams

The Expert pattern

Problem: What is the most basic principle by which responsibilities are assigned in object-oriented design?

Solution: Assign a responsibility to the ***information expert*** – the class that has the information necessary to fulfill the responsibility.

IV. Assign responsibilities and Design interaction diagrams

The Expert pattern – in the POS example:

"who knows about the grand total of a sale?"

To answer – consider the classes in the conceptual model.

- A *Sale* total needs subtotals of all *SalesLineItems*.
- Who knows about all *SalesLineItems*

  → answer = ***Sale***.

Expert advice should be combined with

→ **Low coupling** and **High cohesion**.

IV. Assign responsibilities and Design interaction diagrams

Interaction diagram for *enterItem*:

By the **Expert** pattern:

- Look for an object that knows about all *ProductSpecifications*.
  → By the class diagram: *ProductCatalog*.
- Who should send the *find-specification* message to the *ProductCatalog*?
  → By visibility considerations: *Register*.

Explanation: *ProductCatalog* and *Register* are (singletons) created once in the *Start Up* use case.

IV. Assign responsibilities and Design interaction diagrams

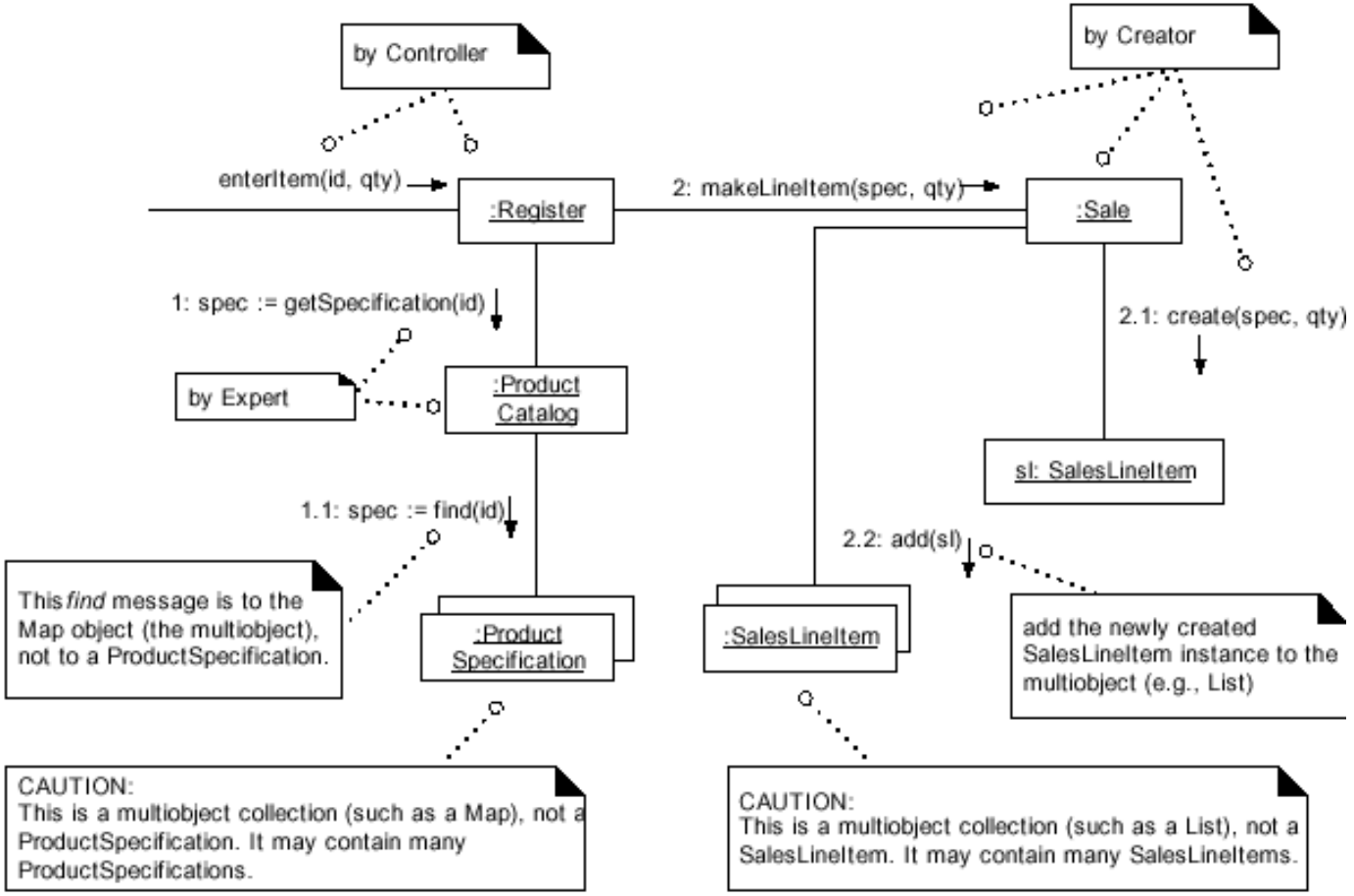
Interaction diagram for *enterItem*:

Summary of decisions:

- *Register* sends a *specification* message to *ProductCatalog* (visibility argument).
- *ProductCatalog* sends a *find* message to the collection (object) of *ProductSpecifications* that it contains.

IV. Assign responsibilities and Design interaction diagrams

Interaction diagram for *enterItem*:



IV. Assign responsibilities and Design interaction diagrams

Design interaction diagrams –

System events:

√ **1. *makeNewSale.***

√ **2. *enterItem.***

3. *endSale.*

4. Sale – getTotal: Presentation requirement.

5. *makePayment.*

6. Payment – getBalance: Presentation requirement.

7. *startUp*.

IV. Assign responsibilities and Design interaction diagrams

Interaction diagram for *endSale*:

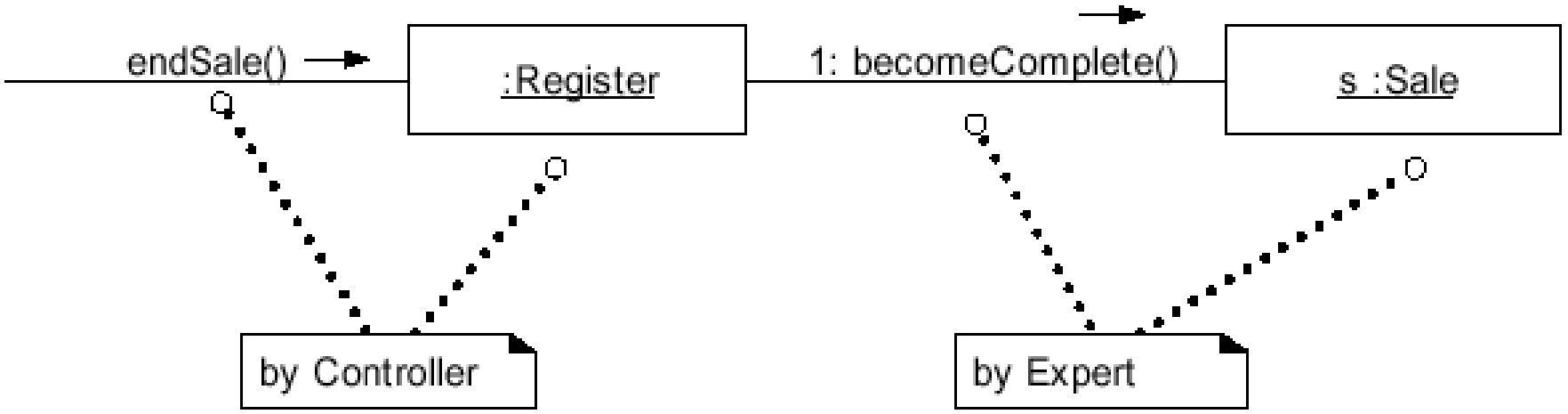
Contract CO3: endSale

Operation: Cross	endSale()
References:	Use Cases: Process Sale
Preconditions:	There is an underway sale.
Postconditions:	Sale.isComplete became true (attribute modification).

IV. Assign responsibilities and Design interaction diagrams

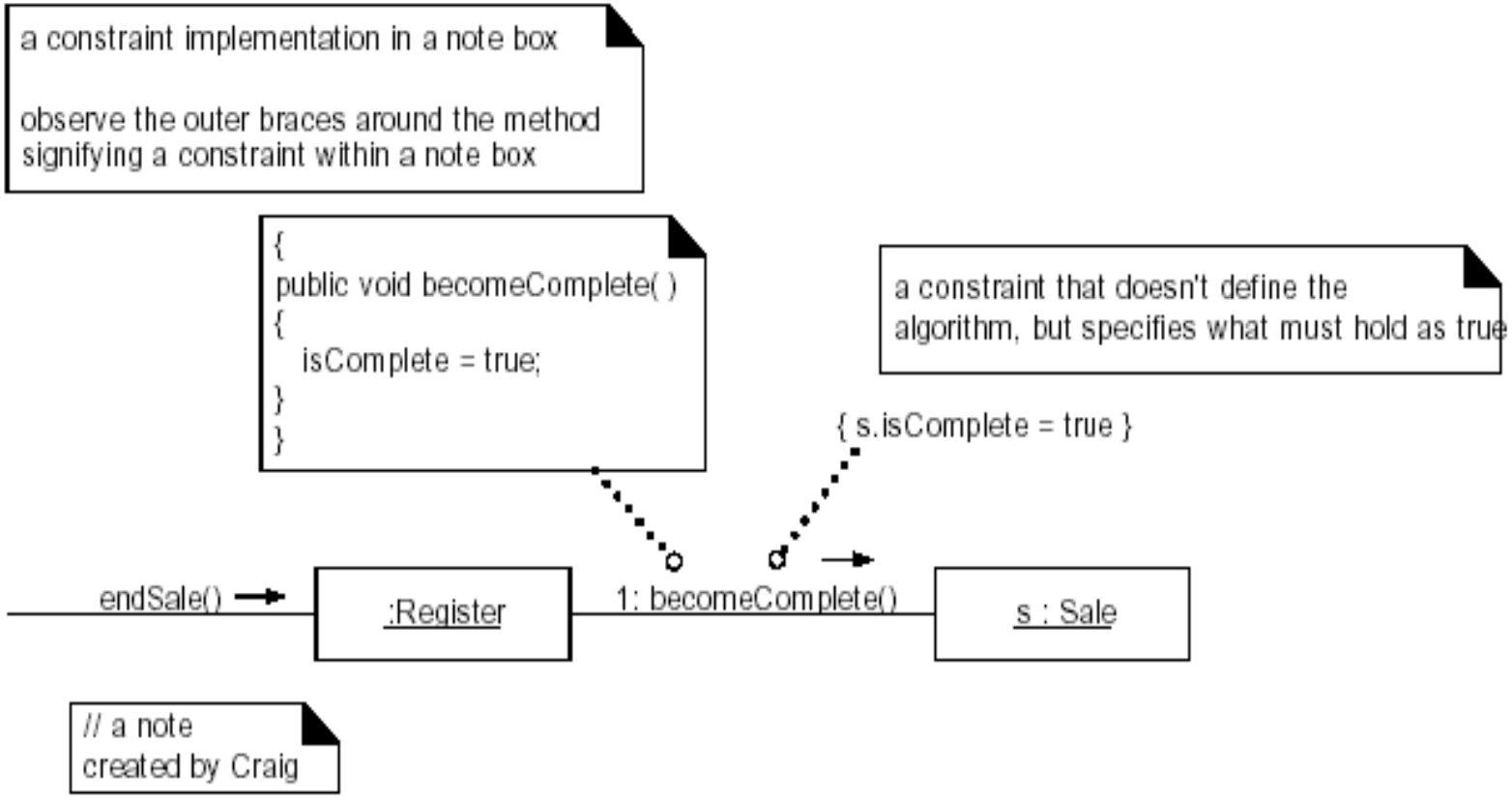
Interaction diagram for *endSale*:

Controller: *Register*.



IV. Assign responsibilities and Design interaction diagrams

Interaction diagram for *endSale*:



IV. Assign responsibilities and Design interaction diagrams

Design interaction diagrams –

System events:

√ **1. *makeNewSale.***

√ **2. *enterItem.***

√ **3. *endSale.***

4. Sale – getTotal: Presentation requirement.

5. *makePayment.*

6. Payment – getBalance: Presentation requirement.

7. *startUp*.

IV. Assign responsibilities and Design interaction diagrams

Calculating the Sale Total:

The *endSale* event is associated with computation of the sale total.

The *Process Sale* use case:

Main Success Scenario:

1. Customer arrives ...
2. Cashier tells System to create a new sale.
3. Cashier enters item identifier.
4. System records sale line item and ...

Cashier repeats steps 3-4 until indicates done.

5. System presents total with taxes calculated.

IV. Assign responsibilities and Design interaction diagrams

Calculating the Sale Total:

The *presentation* is not handled in designing the domain layer – *Model-View separation* principle.

But – the domain layer should provide the **Sale-Total service**.

Who will trigger this operation – probably the presentation layer.

→ *getTotal* can be viewed as a regular system operation (event).

IV. Assign responsibilities and Design interaction diagrams

Calculating the Sale Total:

- **Responsibility:** *Sale* – by **Expert**.
- **Summary of needed information:**

  total of sale = sum of *subtotals*,
  taken over all *SalesLineItem* objects
  contained in *Sale*.

  subtotal of a *SalesLineItem* = *SalesLineItem.quantity* * *ProductSpecification.price*.

IV. Assign responsibilities and Design interaction diagrams

Calculating the Sale Total:

- Information required and **Expert** classes:

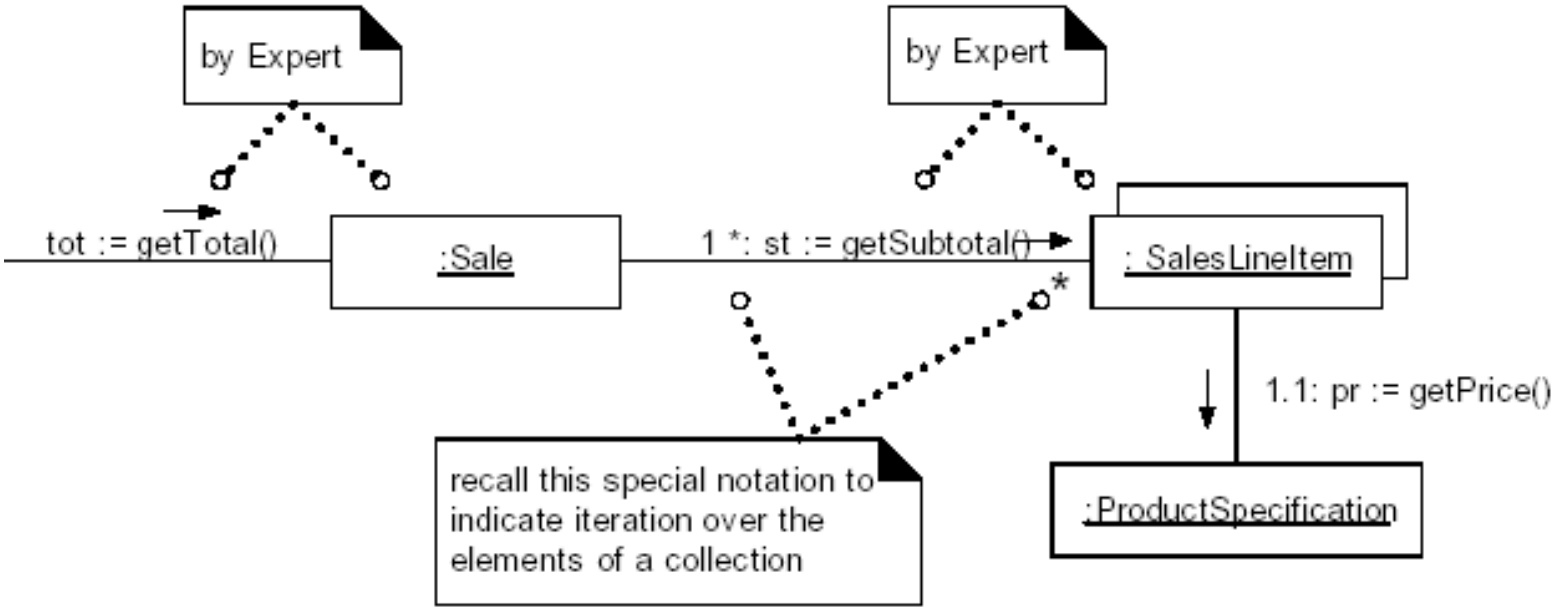
Information Required for Sale Total	Information Expert
ProductSpecification.price	*ProductSpecification*
SalesLineItem. quantity	*SalesLineItem*
all the *SalesLineItems* in the current Sale	*Sale*

IV. Assign responsibilities and Design interaction diagrams

Calculating the Sale Total:

→ *SalesLineItem* – responsible for *subtotal*() – since it knows about *quantity*, and is associated with *productSpecification*.

→ *Sale* --- responsible for *total* ()



IV. Assign responsibilities and Design interaction diagrams

Design interaction diagrams –

System events:

√ **1. *makeNewSale.***

√ **2. *enterItem.***

√ **3. *endSale.***

√ **4. Sale – getTotal:** Presentation requirement.

5. *makePayment.*

6. Payment – getBalance: Presentation requirement.

7. *startUp*.

IV. Assign responsibilities and Design interaction diagrams

Contract CO4: makePayment

Operation: Cross References:	makePayment(amount: Money) Use Cases: Process Sale
Preconditions:	There is an underway sale.
Postconditions:	- A Payment instance p was created (instance creation). - p.amountTendered became amount (attribute modification). - p was associated with the current Sale (association formed). - The current Sale was associated with the Store (association formed); (to add it to the historical log of completed sales).

IV. Assign responsibilities and Design interaction diagrams

Interaction diagram for *makePayment*:

- Choose the **Controller class**: *Register* – same controller for all system events in a single use case.
- **Post condition**: "A Payment was created".

  Question: "Who sends the create message to class Payment?"

  → Use the **Creator** pattern.

  → Use the **Expert** pattern.

  Based on: "Who knows, aggregates or records payments?"

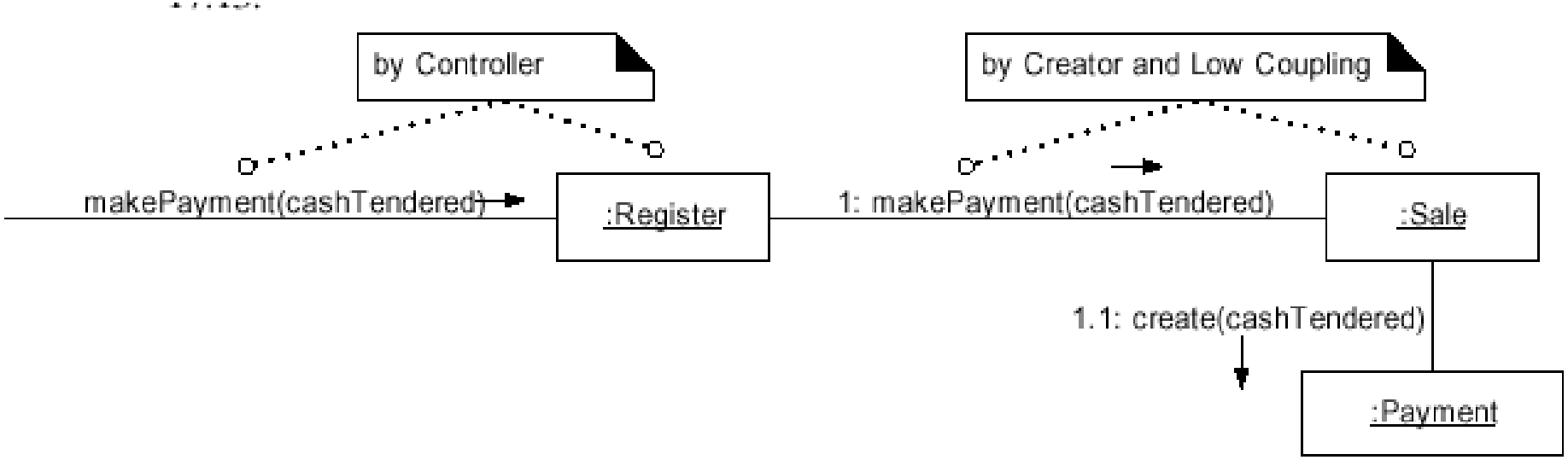
  → 2 candidates: *Register, Sale.*

IV. Assign responsibilities and Design interaction diagrams

Interaction diagram for *makePayment*:

2 candidates for *Payment* creation: *Register, Sale*.

Based on: *High cohesion.* *Low coupling.* → select *Sale*.



IV. Assign responsibilities and Design interaction diagrams

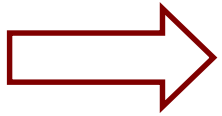
Interaction diagram for *makePayment*:

- **Post condition**: "The Sale was associated with the Store, to add it to the historical log of completed sales".
- **Question:** "Who knows about logged sales?"

  "Who is responsible for doing the logging?"

  → Use the **Expert** pattern.

  → Consulting the conceptual schema:

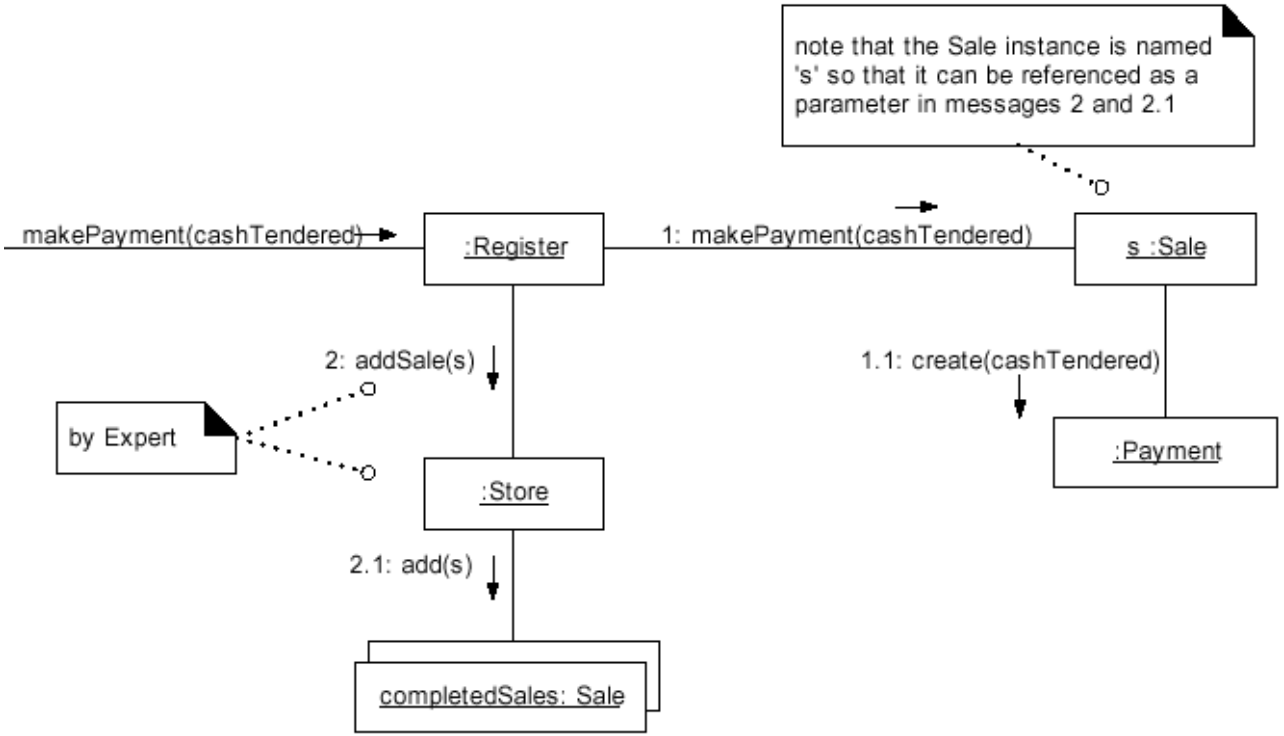


select ***Store.***

IV. Assign responsibilities and Design interaction diagrams

Interaction diagram for *makePayment*:

- **Decision:** *Register* sends the logging message to *Store*.
  Store does the logging.



IV. Assign responsibilities and Design interaction diagrams

Design interaction diagrams –

System events:

√ **1.** ***makeNewSale.***

√ **2.** ***enterItem.***

√ **3.** ***endSale.***

√ **4. Sale – getTotal:** Presentation requirement.

√ **5.** ***makePayment.***

6. Payment – getBalance: Presentation requirement.

7. ***startUp***.

IV. Assign responsibilities and Design interaction diagrams

Calculating the balance of the payment:

By the *Process Sale* use case –

The *makePayment* event is associated with

- Balance computation,
- receipt printing,
- balance display.

IV. Assign responsibilities and Design interaction diagrams

Calculating the balance of the payment :

The *presentation* is not handled in designing the domain layer – *Model-View separation* principle.

But – the domain layer should provide the ***getBalance* service**.

Who will trigger this operation – probably the presentation layer.

→ *getBalance* can be viewed as a regular system operation (event).

IV. Assign responsibilities and Design interaction diagrams

Calculating the balance of the payment :

- **Question:** "Who knows about balance?"

  Needed information:

  – **Sale total;**

  – **Cash tendered**.

  By **Expert:**

  → Candidate classes: *Sale*,

  Payment.

IV. Assign responsibilities and Design interaction diagrams

Calculating the balance of the payment :

- **Considerations:**

1. Choosing Payment →

   Payment-*Sale* visibility (*Payment* asks *Sale* for the total).

2. Choosing Sale →

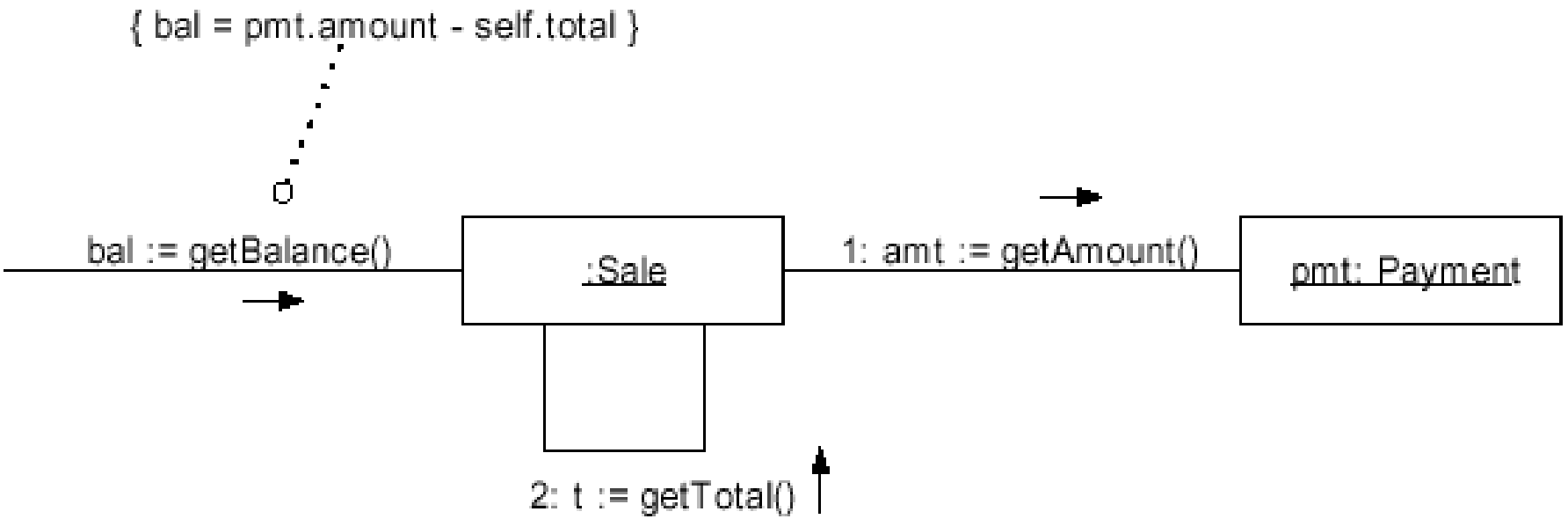
   Sale-*Payment* visibility (*Sale* asks *Payment* for cash-tendered).

   BUT: Already exists (*Sale* is *Payment's* creator)!

IV. Assign responsibilities and Design interaction diagrams

Calculating the balance of the payment :

- **Decision**: Assign the getBalance responsibility to *Sale*.
- **Argumentation:** Supports low-coupling.



IV. Assign responsibilities and Design interaction diagrams

Design interaction diagrams –

System events:

√ **1.** ***makeNewSale.***

√ **2.** ***enterItem.***

√ **3.** ***endSale.***

√ **4. Sale – getTotal:** Presentation requirement.

√ **5.** ***makePayment.***

√ **6. Payment – getBalance:** Presentation requirement.

7. ***startUp***.

IV. Assign responsibilities and Design interaction diagrams

Application Start Up – system initialization

- The start up of an application represents its **initialization**.
- **Design idiom (pattern)**:
  - Create an ***initial domain object***.
  - Send a run message to the ***initial domain object***, or to any of the domain objects that it creates.
- The initial domain object is **responsible** for the creation of other problem domain objects and their initialization.

IV. Assign responsibilities and Design interaction diagrams

Application Start Up – system initialization

- How to **choose** the initial domain object?
  - A class that represents the entire logical information system.
  - A class that represents the overall business or organization.
- Use the **High Cohesion** and **Low Coupling** criteria for choosing.

IV. Assign responsibilities and Design interaction diagrams

Application Start Up – system initialization

- Who **creates** and **runs** the application?
  - Creation of the initial domain object – Message sent by an **object of the presentation** (interface) layer.
  - Running (controlling) the application – Either the **initial domain object** or the **presentation layer**.

IV. Assign responsibilities and Design interaction diagrams

Application Start Up for the POS system:

- **Initial domain object:** Choose between
  - *Register* – entire logical information system.
  - *Store* – overall business.

  → Decision based on high cohesion: *Store*.
- Who creates and runs in POS:
  - Creation -- A presentation object like a GUI object.
  - Running -- A presentation object.

IV. Assign responsibilities and Design interaction diagrams

Application Start Up for the POS system:

- **Persistent object decisions – database objects**:

  A persistent object persists throughout the life of the system.

  Stored persistently: files, databases.
- Decision on persistent objects –

  In **POS**: *ProductSpecification*.
- Decision on a mediating object –

  In **POS**: *ProductCatalog*.

IV. Assign responsibilities and Design interaction diagrams

Application Start Up for the POS system:

- The **implementation of persistent classes** in a database requires a mapping that accounts for:
  - Attributes. Key attributes. Attribute domains.
  - Associations. Association classes.
  - Multiplicity constraints.
  - Abstractions – sub-typing, aggregation.

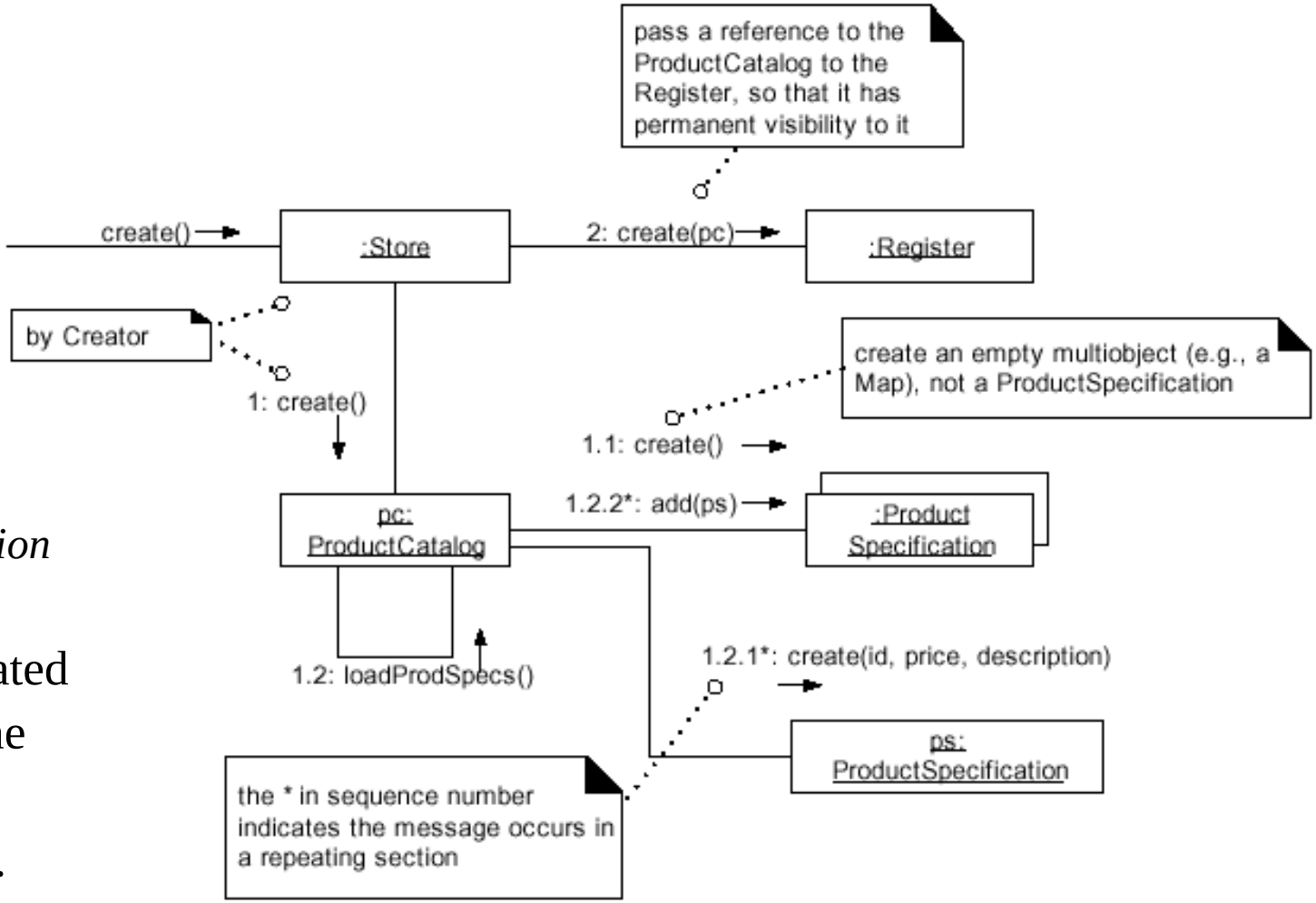
IV. Assign responsibilities and Design interaction diagrams

The POS Start Up contract – *Store* create:

- A *Store*, *Register*, *ProductCatalog* and *ProductSpecifications* need to be created.
- The *ProductCatalog* needs to be associated with *ProductSpecifications*.
- *Store* needs to be associated with *ProductCatalog*.
- *Store* needs to be associated with *Register*.
- *Register* needs to be associated with *ProductCatalog*.

IV. Assign responsibilities and Design interaction diagrams

Simplifying assumption: *ProductSpecification* Instances are "magically" created in memory by the Singleton *ProductCatalog*.



IV. Assign responsibilities and Design interaction diagrams

Design interaction diagrams –

System events:

√ **1.** ***makeNewSale.***

√ **2.** ***enterItem.***

√ **3.** ***endSale.***

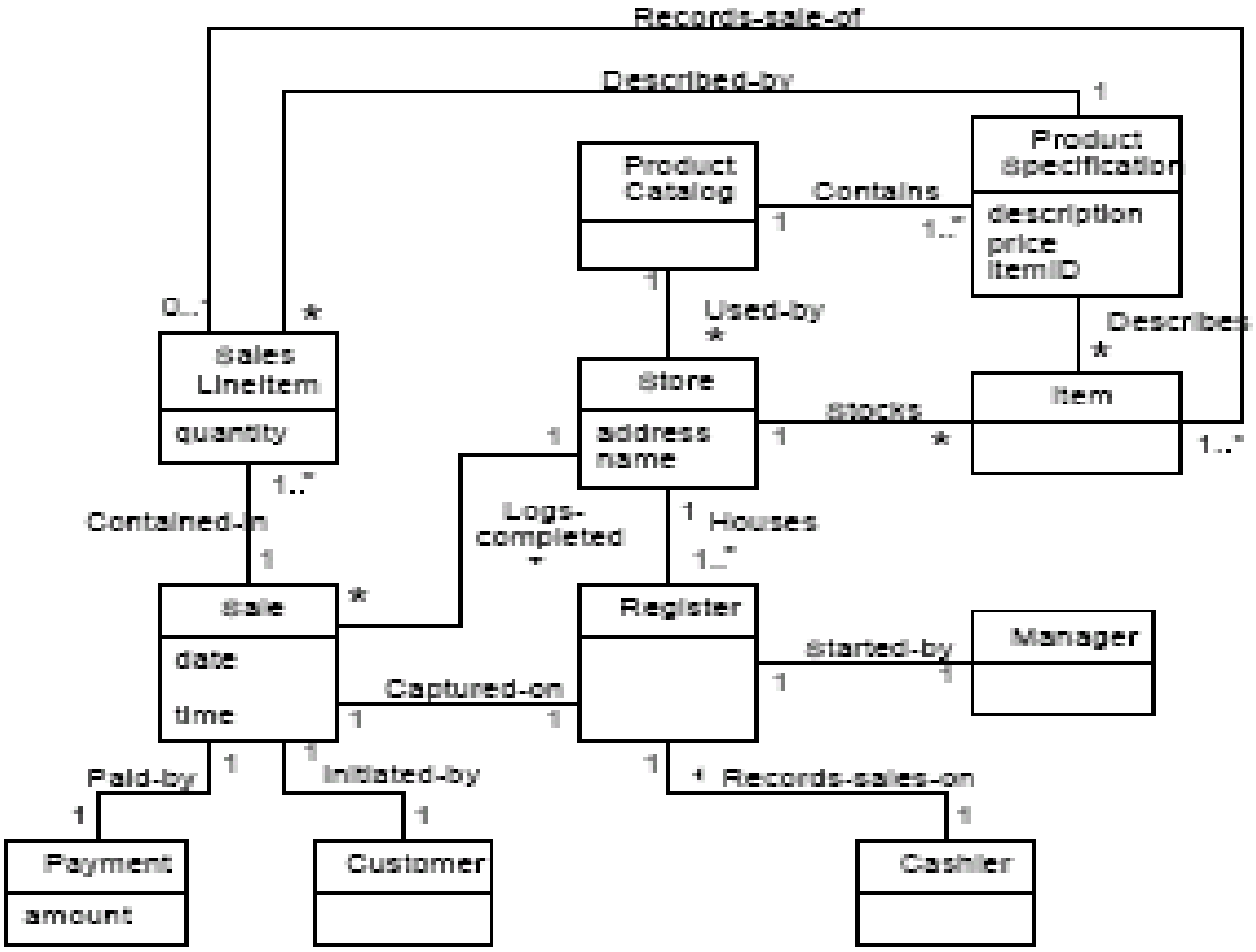
√ **4. Sale – getTotal:** Presentation requirement.

√ **5.** ***makePayment.***

√ **6. Payment – getBalance:** Presentation requirement.

√ **7.** ***startUp***.

V. Design level class diagrams – implementation and testing.



V. Design Class Diagrams – Implementation

This stage is responsible for mapping design artifacts to code.

Two steps:

1. Develop Design Class Diagrams (DCDs).
2. Implement.

Design class diagrams are developed in parallel to the development of interaction diagrams!

V. Design Class Diagrams – Implementation

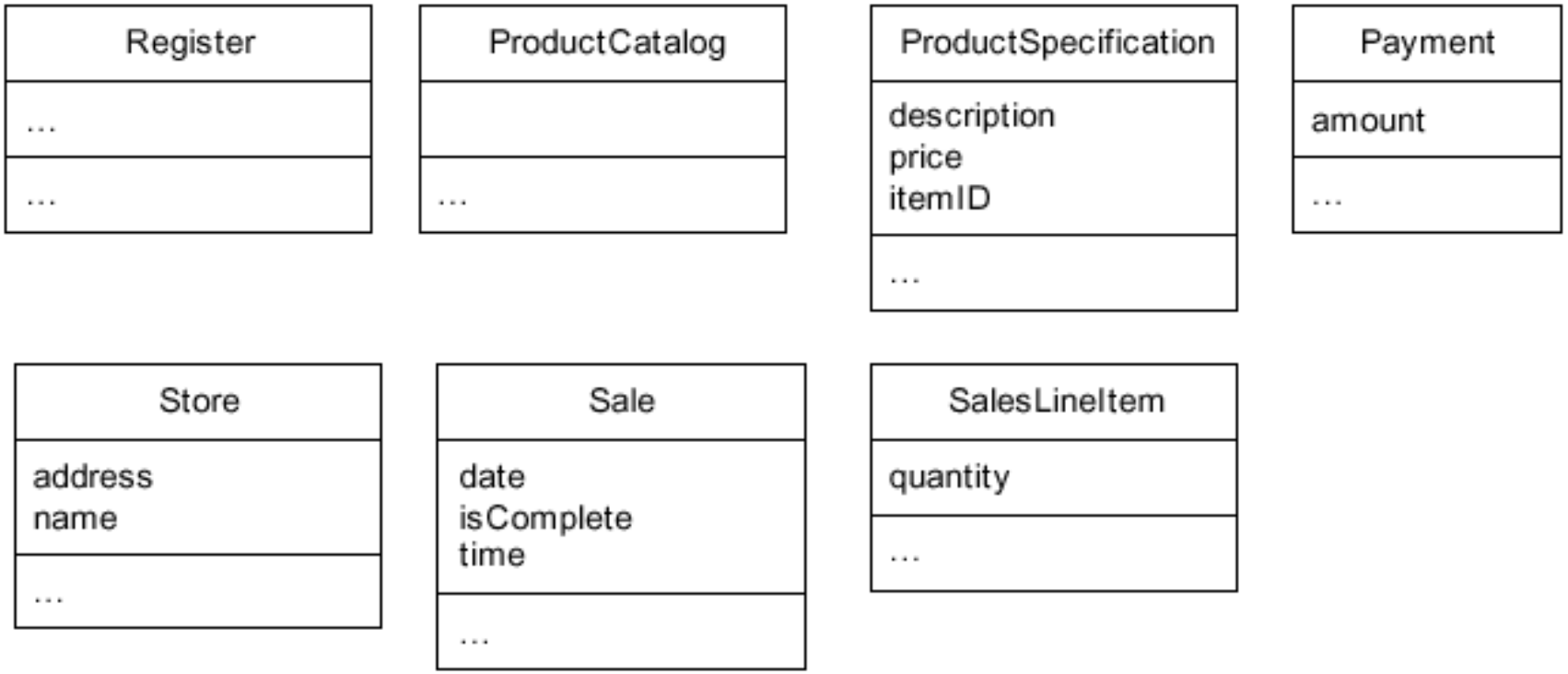
A Design Class Diagram (DCD) provides specification for a software class or interface. It includes:

- Classes, associations, typed attributes, data-types.
- Interfaces – with operations.
- Methods – with signatures.
- Navigability – permanent (attribute) visibility.
- Dependencies – temporary visibilities.

V. Design Class Diagrams – Implementation

Developing Design Class Diagrams (DCDs):

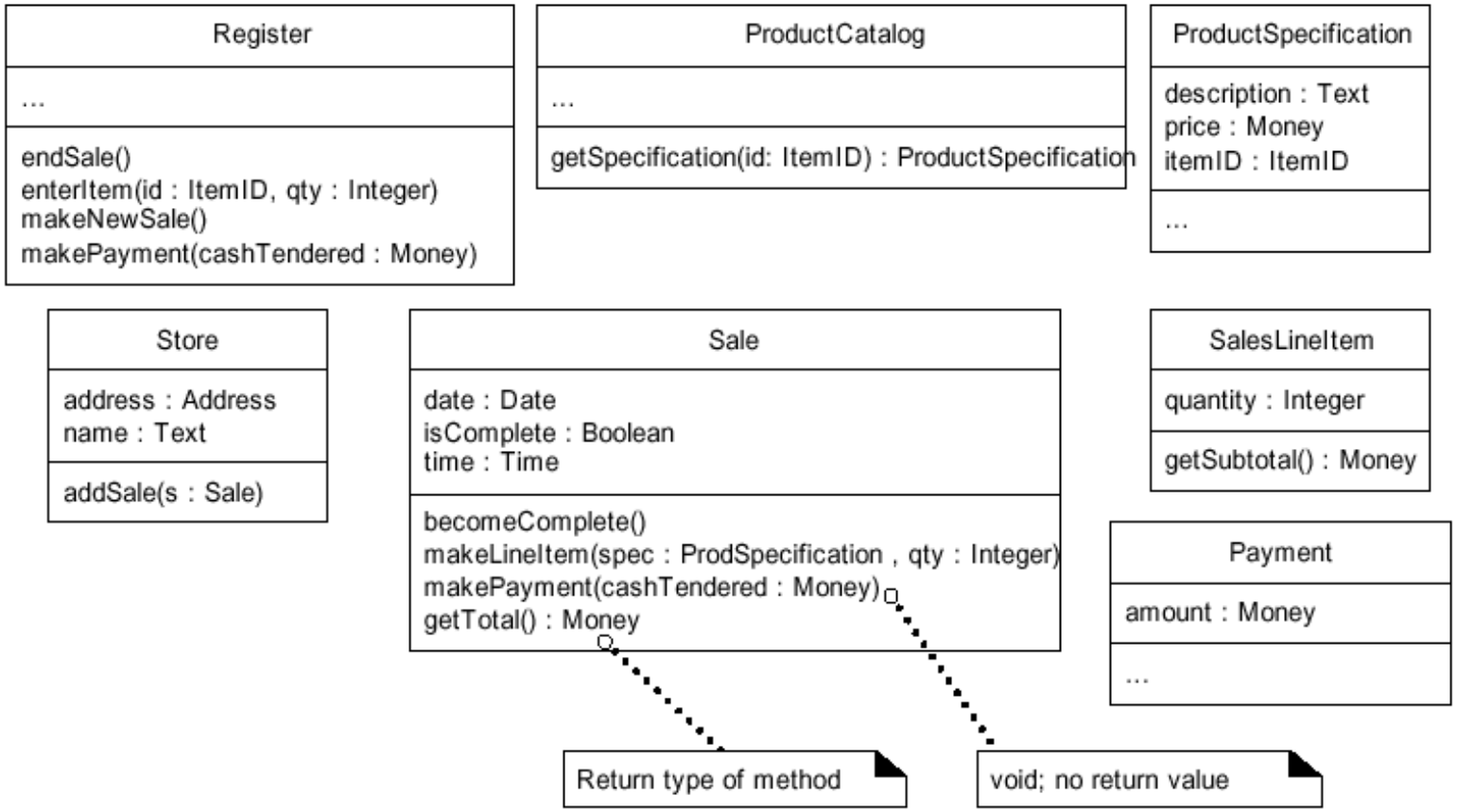
1. Identify classes.



V. Design Class Diagrams – Implementation

Developing Design Class Diagrams (DCDs):

2,3. Add typed attributes and methods with signatures.



V. Design Class Diagrams – Implementation

Developing Design Class Diagrams (DCDs):

4,5. Add navigability and dependency.

Visibility Between Objects – the ability of one object to see or have reference to another.

Four kinds of visibility:

1. Attribute visibility – Permanent.
2. Parameter visibility – Temporary.
3. Local visibility – Temporary.
4. Global visibility – Permanent.

V. Design Class Diagrams – Implementation

1. Attribute visibility – B is a reference attribute of A.

   Example: Sale -> SalesLineItem.

2. Parameter visibility – B is a parameter of a method of A.

   Example: Register sends to Sale: makeLineItem(spec, qty).

3. Local visibility – B is declared as a local object in a methods of A.

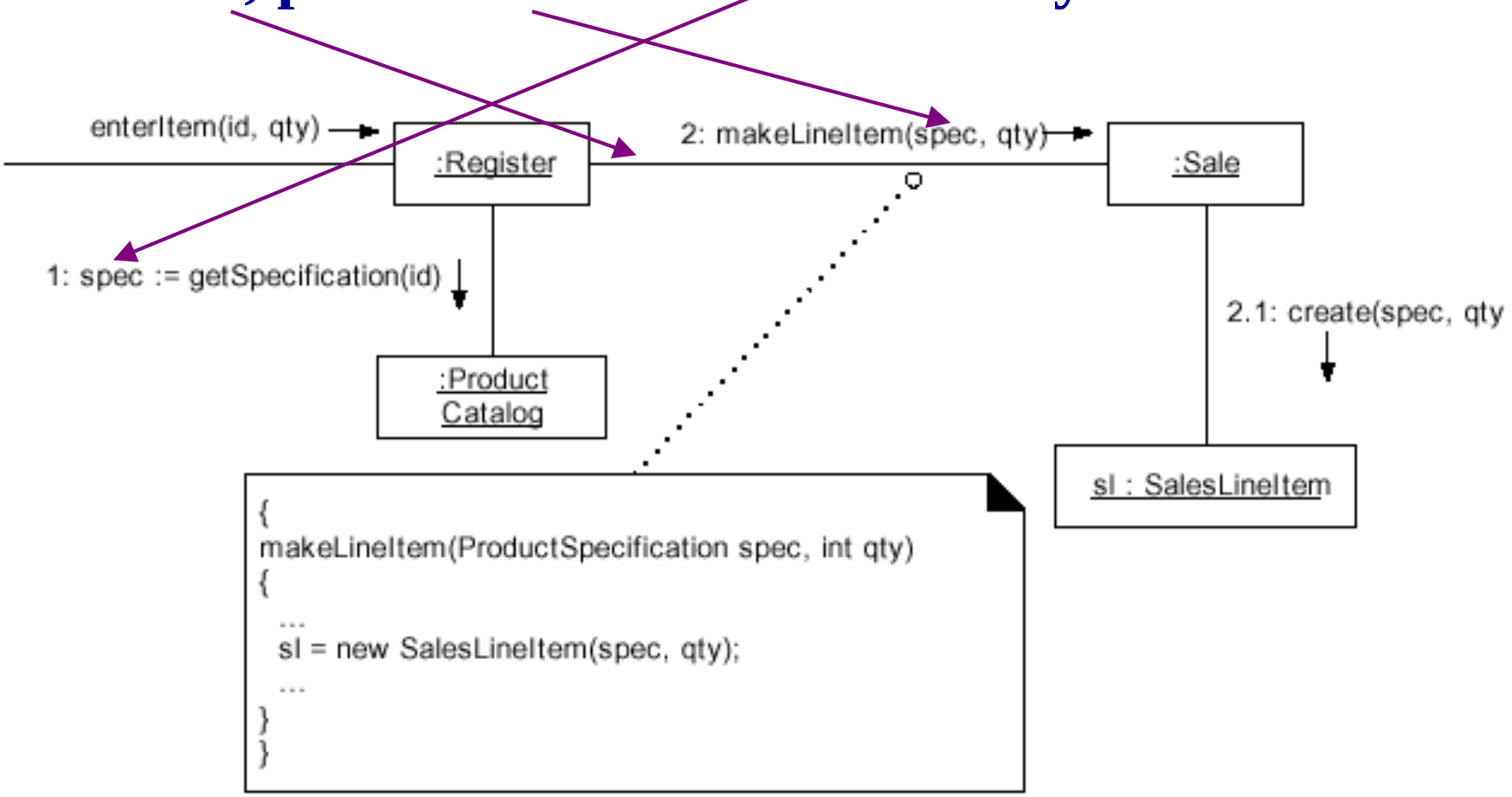
   Example: register -> ProductSpecification.

4. Global visibility – B is globally visible.

   Example: B has static methods.

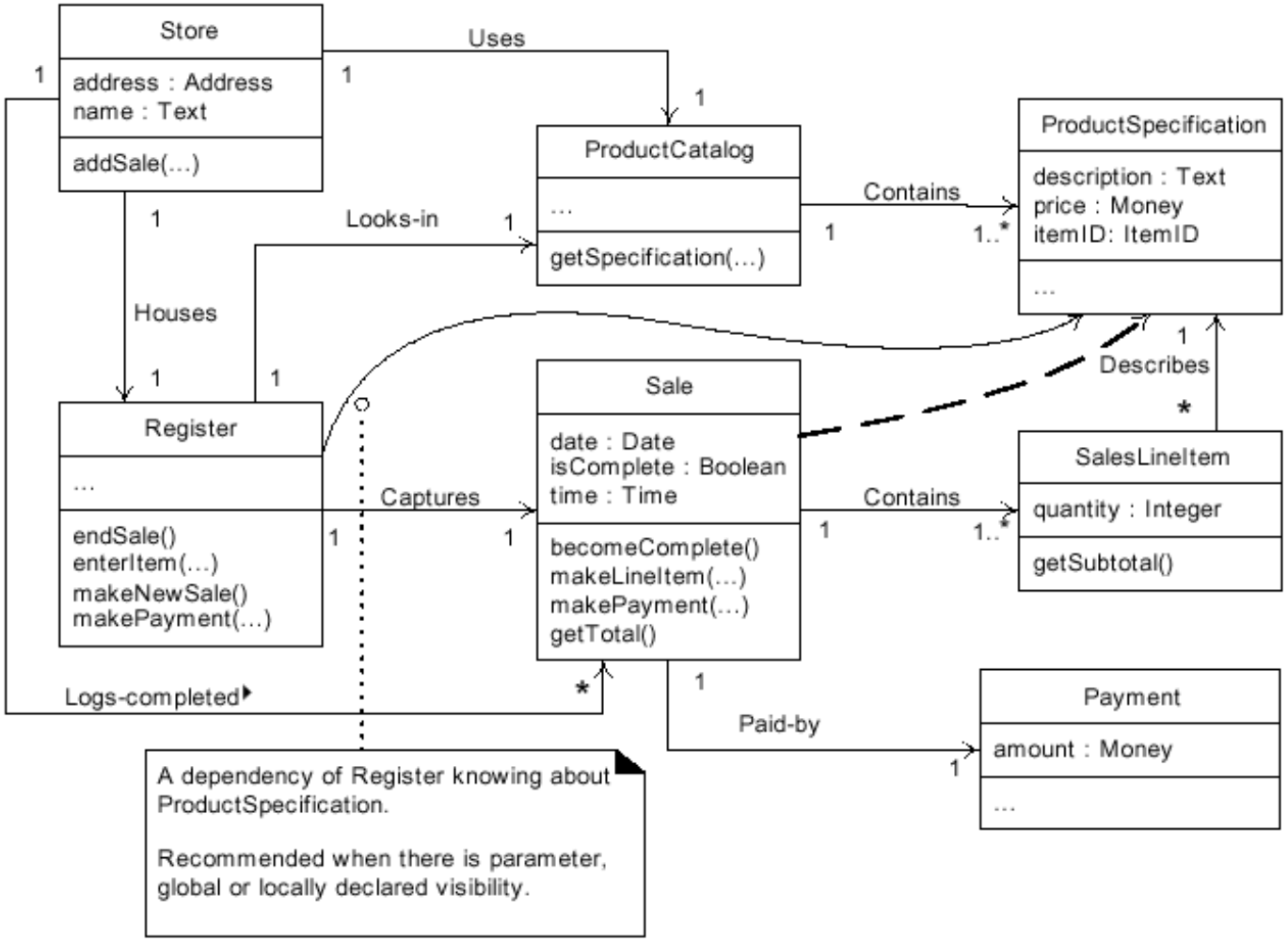
V. Design Class Diagrams – Implementation

Attribute, parameter and local visibility:



V. Design Class Diagrams – Implementation

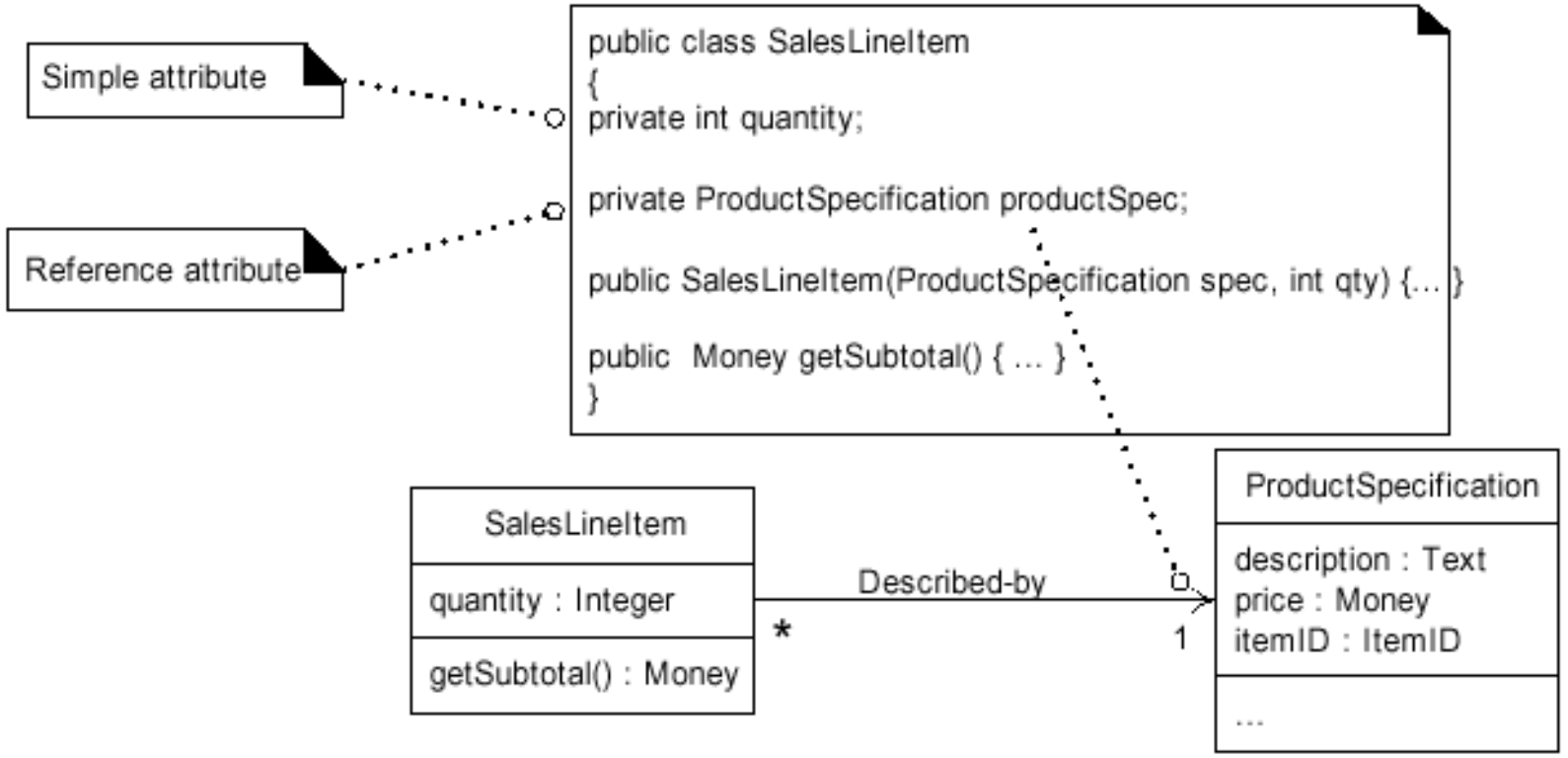
Design Class Diagrams with navigability and dependency.



V. Design Class Diagrams – Implementation

Implement – based on the DCD and the interaction Diagrams.

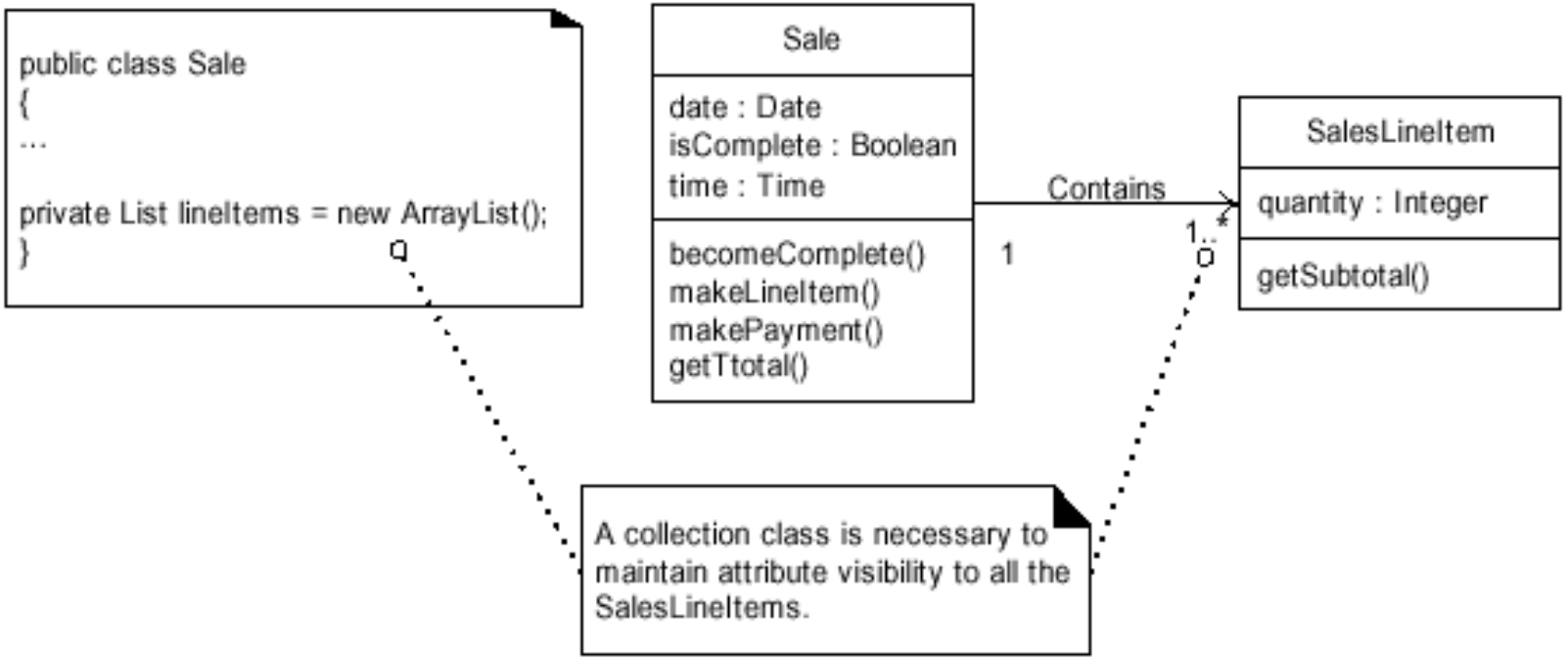
1. Adding reference attributes:



V. Design Class Diagrams – Implementation

Implement – based on the DCD and the interaction Diagrams.

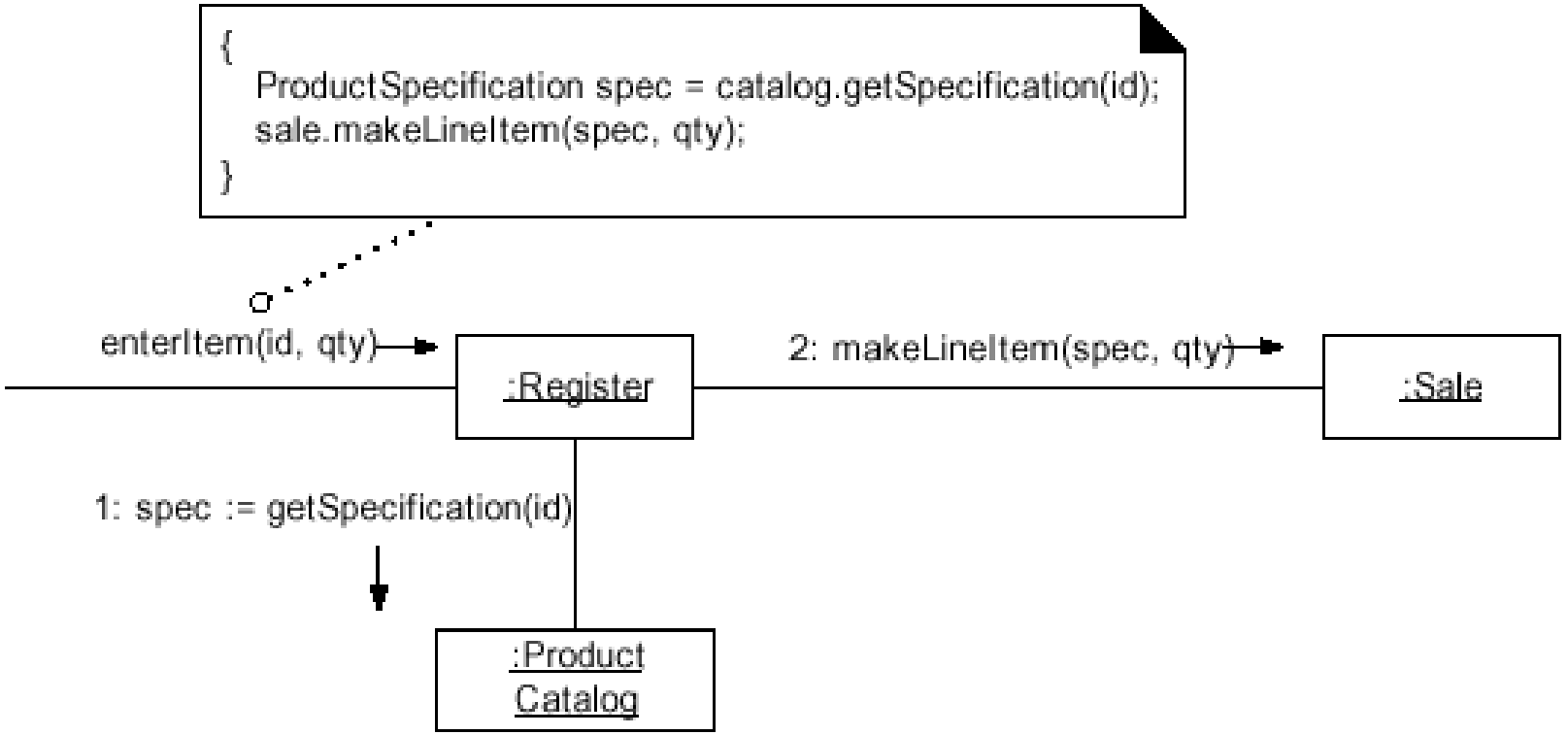
1. Adding collection attributes:



V. Design Class Diagrams – Implementation

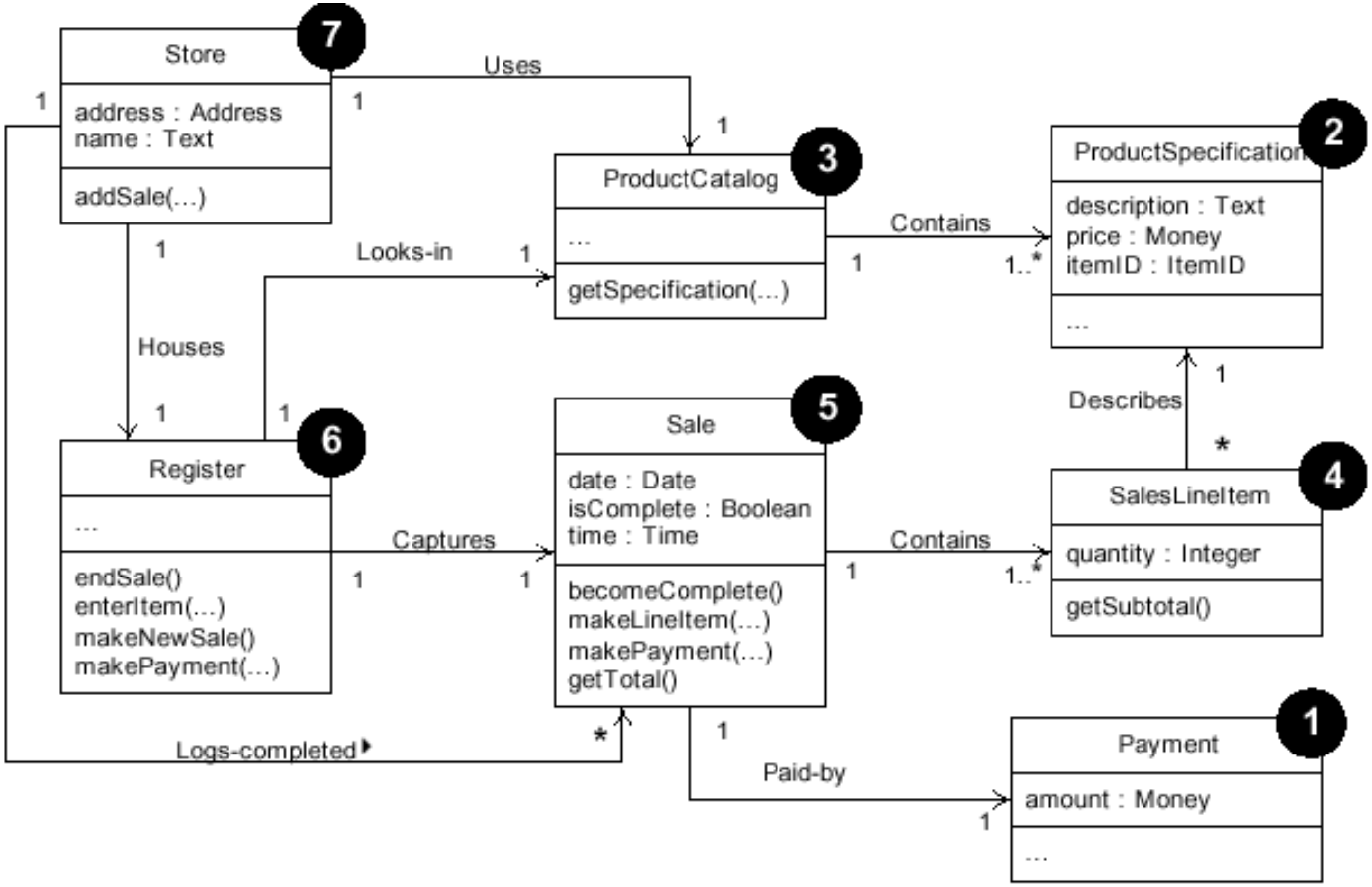
Implement – based on the DCD and the interaction Diagrams.

2. Adding methods:



V. Design Class Diagrams – Implementation

Implement – based on the DCD and the interaction Diagrams. Possible test-and-implement ordering:



V. Design Class Diagrams – Implementation

Possible POS implementation (1st cycle):

Class Payment

```
public class Payment {
    private Money amount;
    public Payment( Money cashTendered ){ amount = cashTendered; }
    public Money getAmount() { return amount; } }
```

V. Design Class Diagrams – Implementation

Possible POS implementation (1st cycle):

Class ProductCatalog

```
public class ProductCatalog {
    private Map productSpecifications = new HashMap();


 public ProductCatalog() {
     // sample data
     ItemID id1 = new ItemID( 100 );
     ItemID id2 = new ItemID( 200 );
     Money price = new Money( 3 );

     ProductSpecification ps;
     ps = new ProductSpecification( id1, price, "product 1" );
     productSpecifications.put( id1, ps );
 ps = new ProductSpecification( id2, price, "product 2" );
 ProductSpecifications.put( id2, ps ); }

 public ProductSpecification getSpecification( ItemID id ) {
     return (ProductSpecification)productSpecifications.get( id );
 }
}
```

V. Design Class Diagrams – Implementation

Possible POS implementation (1st cycle):

Class Register

```
public class Register {
    private ProductCatalog catalog;
    private Sale sale;

    public Register( ProductCatalog catalog ) {
        this.catalog = catalog; }

    public void endSale0 {
        sale.becomeComplete();
     }

    public void enterItem( ItemID id, int quantity ) {
        ProductSpecification spec = catalog.getSpecification( id );

        sale.makeLineItem( spec, quantity ); }

    public void makeNewSale() {
        sale = new Sale(); }

    public void makePayment( Money cashTendered ) {
        sale.makePayment( cashTendered ); }

}
```

V. Design Class Diagrams – Implementation

Possible POS implementation (1st cycle):

Class ProductSpecification

```
public class ProductSpecification {
    private ItemID id;
    private Money price;
    private String description;

    public ProductSpecification
        ( ItemID id. Money price. String description ) {
        this.id = id;
        this.price = price;
        this.description = description; }

    public ItemID getItemIDO { return id;}

    public Money getPrice() { return price; }

    public String getDescription() { return description; }
}
```

V. Design Class Diagrams – Implementation

Possible POS implementation (1st cycle):

Class Sale

```
public class Sale
{
   private List lineItems = new ArrayList();
   private Date date = new Date();
   private boolean isComplete = false;
   private Payment payment;

   public Money getBalance() {
      return payment.getAmount().minus( getTotal() ); }

   public void becomeComplete() { isComplete = true; }

   public boolean isComplete() { return isComplete; }

   public void makeLineItem
       ( ProductSpecification spec, int quantity ) {
      lineItems.add( new SalesLineItem( spec, quantity ) ); }

   public Money getTotal()
   {
      Money total = new Money();
      Iterator i = lineItems.iterator( ) ;
      while ( i.hasNext() )
      {
       SalesLineItem sli = (SalesLineItem) i.next();
       total.add( sli.getSubtotal() );
      }
      return total; }
   public void makePayment( Money cashTendered )
   {
   payment = new Payment( cashTendered ); } }
```

V. Design Class Diagrams – Implementation

Possible POS implementation (1st cycle):

Class SalesLineItem

```
public class SalesLineItem {
   private int quantity;
   private ProductSpecification productSpec;

   public SalesLineItem (ProductSpecification spec, int quantity )
   {
      this.productSpec = spec;
      this.quantity = quantity; }

   public Money getSubtotal() {
      return productSpec.getPrice().times( quantity );
   } }
```

V. Design Class Diagrams – Implementation

Possible POS implementation (1st cycle):

Class Store

```
public class Store
{
    private ProductCatalog catalog = new ProductCatalog();
    private Register register = new Register( catalog );

    public Register getRegister() { return register; } }
```

Persistency decisions

Persistency decisions involve objects that deserve persistent storage – e.g., files, databases.

In the POS problem – move *ProductSpecification* to the database. The *ProductCatalog* serves as a communication channel to the persistent objects.

The object → relational mapping poses many problems, due to the mismatch between the data-structures.

Using Packages to Organize the Domain Model

Package partitioning guidelines:

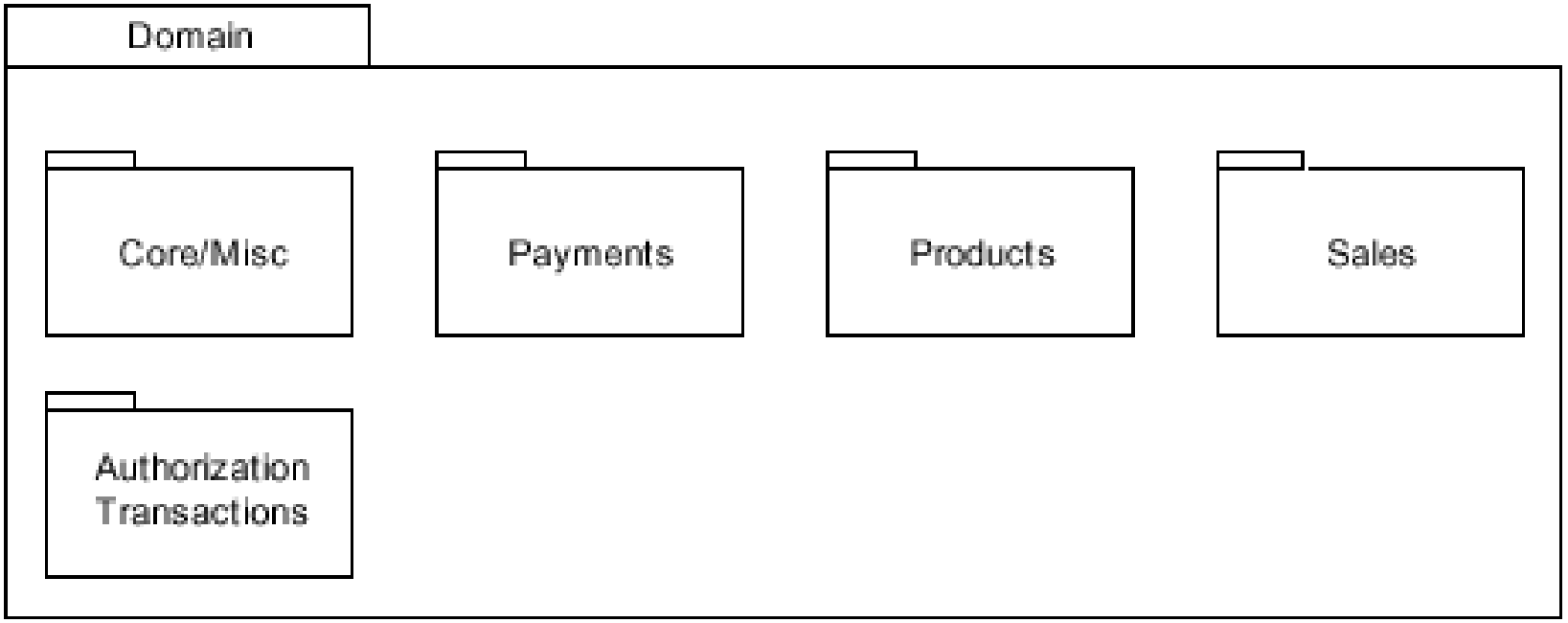
To partition the domain model into packages, place elements together that:

- are in the same subject area —closely related by concept or purpose
- are in a class hierarchy together
- participate in the same use cases
- are strongly associated

The overall domain layer is itself a package!

Using Packages to Organize the Domain Model

Package organization for the POS domain model:



Using Packages to Organize the Domain Model

Package dependencies: