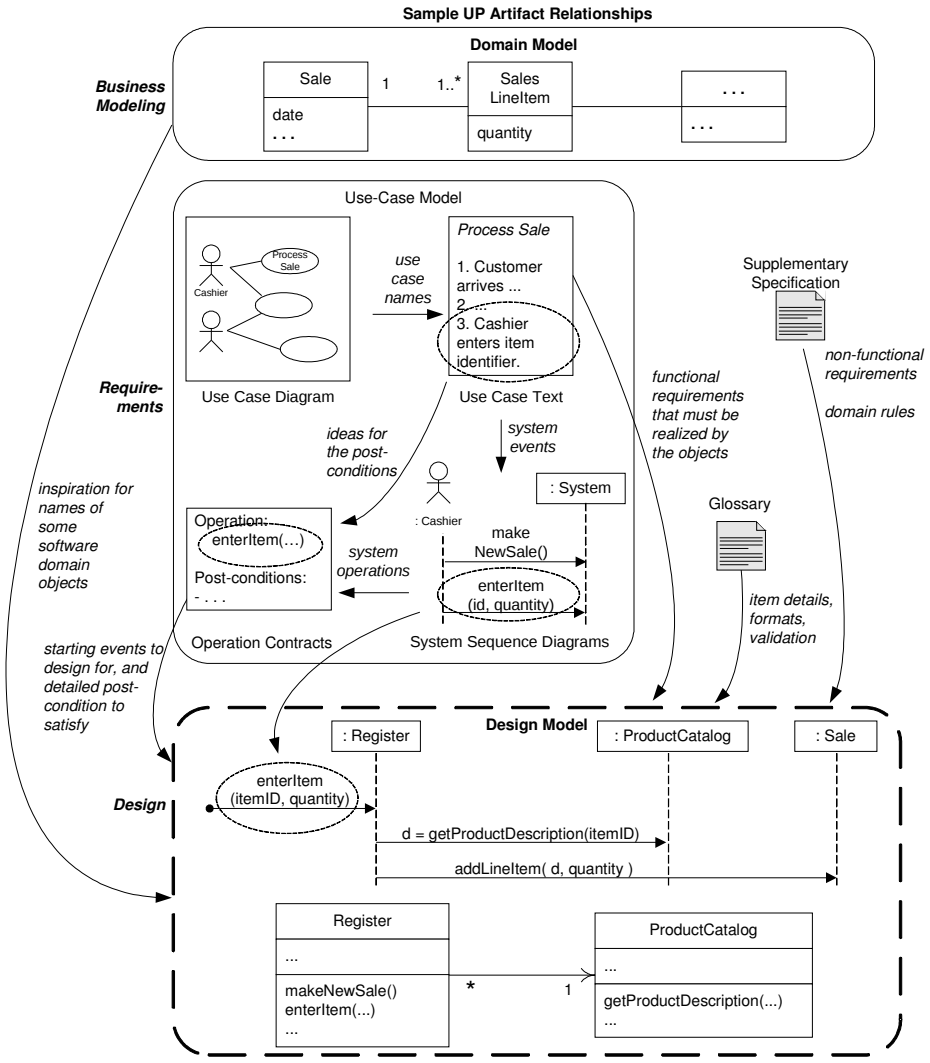
COMP 6471
Software Design Methodologies

Fall 2011

Dr Greg Butler

http://www.cs.concordia.ca/~gregb/home/comp6471-fall2011.html

Context



Larman, Figure 17.1

This diagram from Larman illustrates how the design model fits into the other UP artifacts we've looked at so far.

GRASP: Designing Objects with Responsibilities*

* General Responsibility Assignment Software Patterns

Responsibilities and Methods

- The focus of object design is to identify classes and objects, decide what methods belong where and how these objects should interact.
- Responsibilities are related to the obligations of an object in terms of its behaviour.
- Two types of responsibilities:
  - doing:
    - doing something itself (*e.g.* creating an object, performing a calculation)
    - initiating action in other objects.
    - controlling and coordinating activities in other objects.
  - knowing:
    - knowing about private encapsulated data.
    - knowing about related objects.
    - knowing about things it can derive or calculate.

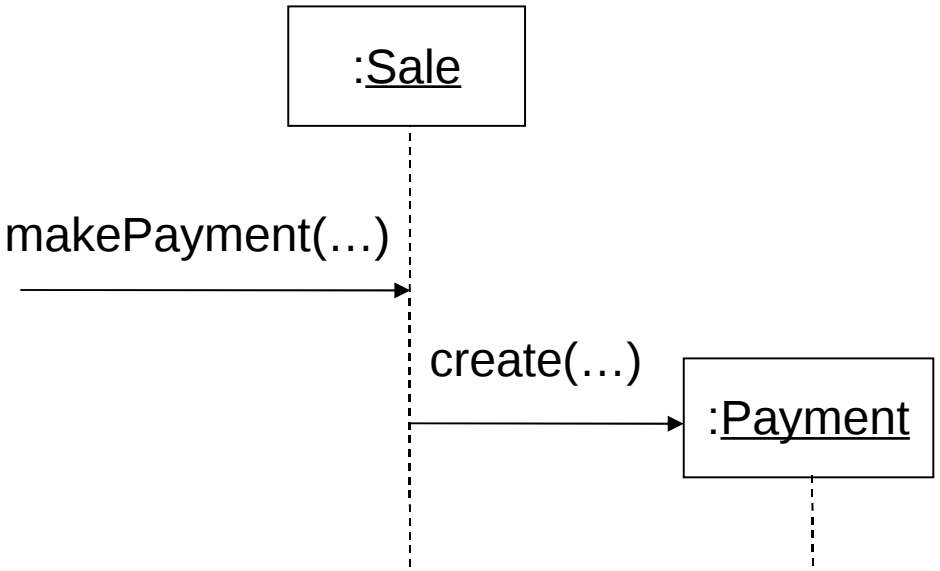
Responsibilities and Methods

- Responsibilities are assigned to classes during object design. For example, we may declare the following:
  - *"a Sale is responsible for creating SalesLineItems*" (doing)
  - *"a Sale is responsible for knowing its total"* (knowing)
- Responsibilities related to "knowing" can often be inferred from the Domain Model (because of the attributes and associations it illustrates).

Responsibilities and Methods

- The translation of responsibilities into classes and methods is influenced by the granularity of responsibility.
  - For example, *"provide access to relational databases"* may involve dozens of classes and hundreds of methods, whereas *"create a Sale"* may involve only one or two methods.
- A responsibility is not the same thing as a method, but methods are implemented to fulfill responsibilities.
- Methods either act alone, or collaborate with other methods and objects.

Responsibilities and Interaction Diagrams



- Within the UML artifacts, a common context where these responsibilities (implemented as methods) are considered is during the creation of interaction diagrams.
- Sale objects have been given the responsibility to create Payments, handled with the makePayment method.

Patterns

- We will emphasize principles (expressed in patterns) to guide choices in where to assign responsibilities.
- A pattern is a named description of a problem and a solution that can be applied to new contexts; it provides advice in how to apply it in varying circumstances. For example,
  - Pattern name: Information Expert
  - Problem: What is the most basic principle by which to assign responsibilities to objects?
  - Solution: Assign a responsibility to the class that has the information needed to fulfill it.

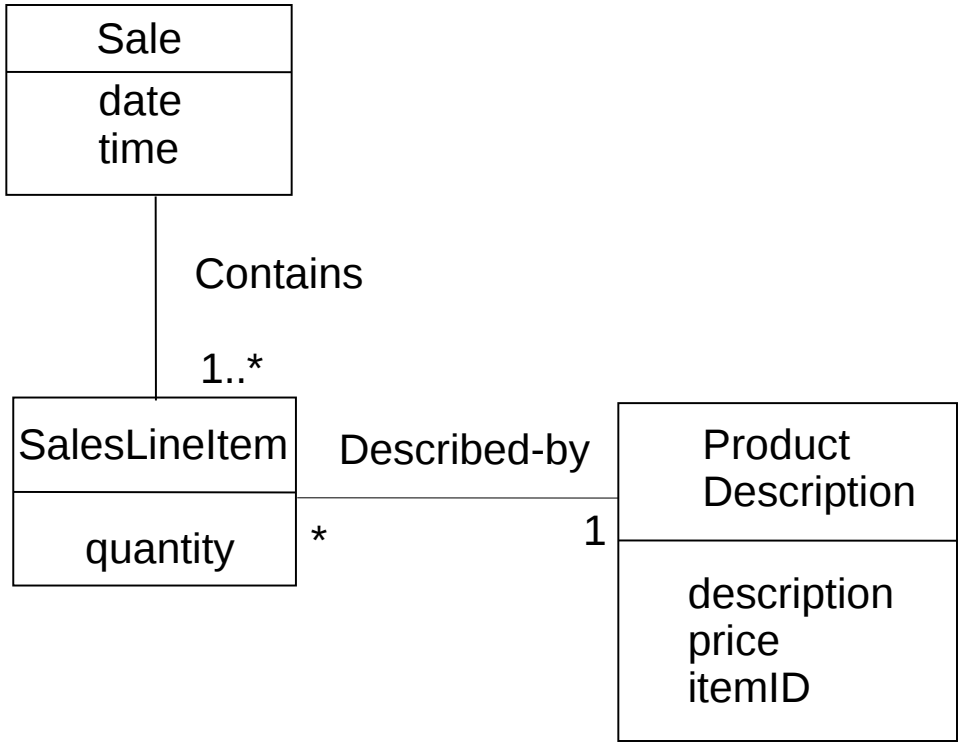
Information Expert (or Expert)

- Problem: What is a general principle of assigning responsibilities to objects?
- Solution: Assign a responsibility to the information expert — the class that has the information necessary to fulfill the responsibility.
- In the NextGen POS application, who should be responsible for knowing the grand total of a sale?
- Information Expert suggests that we should look for that class that has the information needed to determine the total.

Information Expert (or Expert)

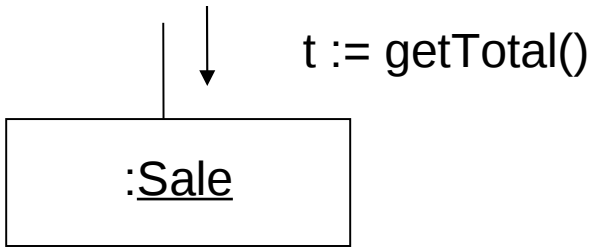
- Do we look in the Domain Model or the Design Model to analyze the classes that have the information needed?
- A: Both. Assume there is no or minimal Design Model. Look to the Domain Model for information experts.

Information Expert (or Expert)



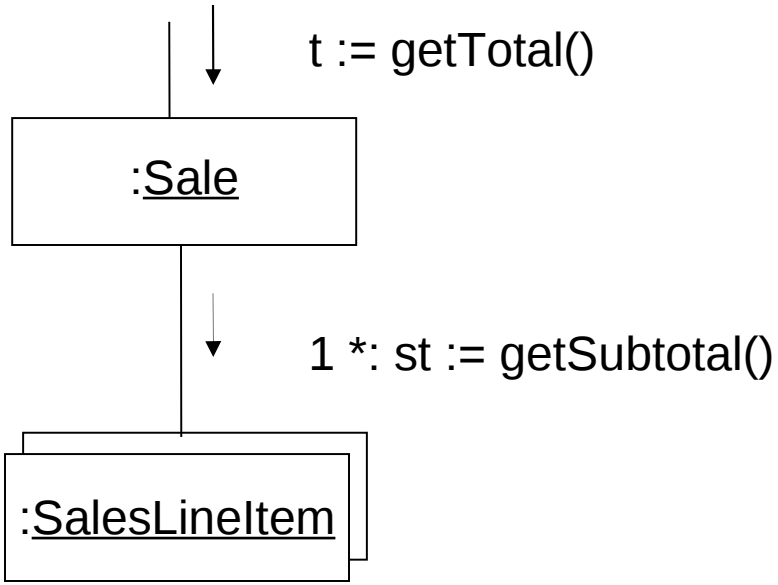
- It is necessary to know about all the SalesLineItem instances of a sale and the sum of the subtotals.
- A Sale instance contains these, *i.e.* it is an information expert for this responsibility.

Information Expert (or Expert)



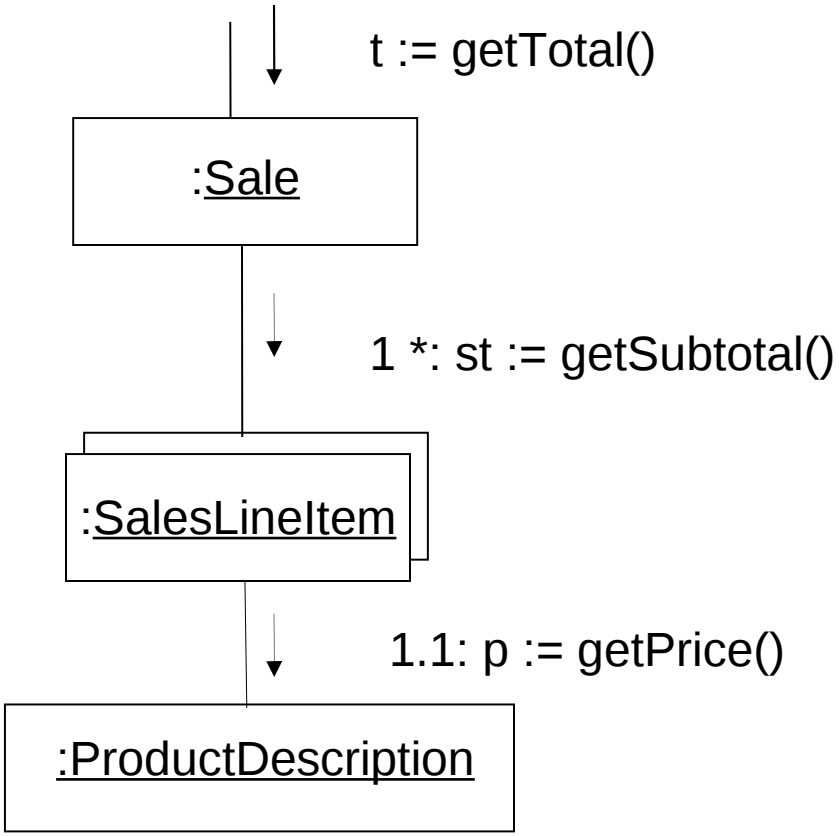
- This is a partial interaction diagram.

Information Expert (or Expert)



- What information is needed to determine the line item subtotal?
  - quantity and price.
- SalesLineItem should determine the subtotal.
- This means that Sale needs to send getSubtotal() messages to each of the SalesLineItems and sum the results.

Information Expert (or Expert)



- To fulfil the responsibility of knowing and answering its subtotal, a SalesLineItem needs to know the product price.
- The ProductDescription is the information expert on answering its price.

Information Expert (or Expert)

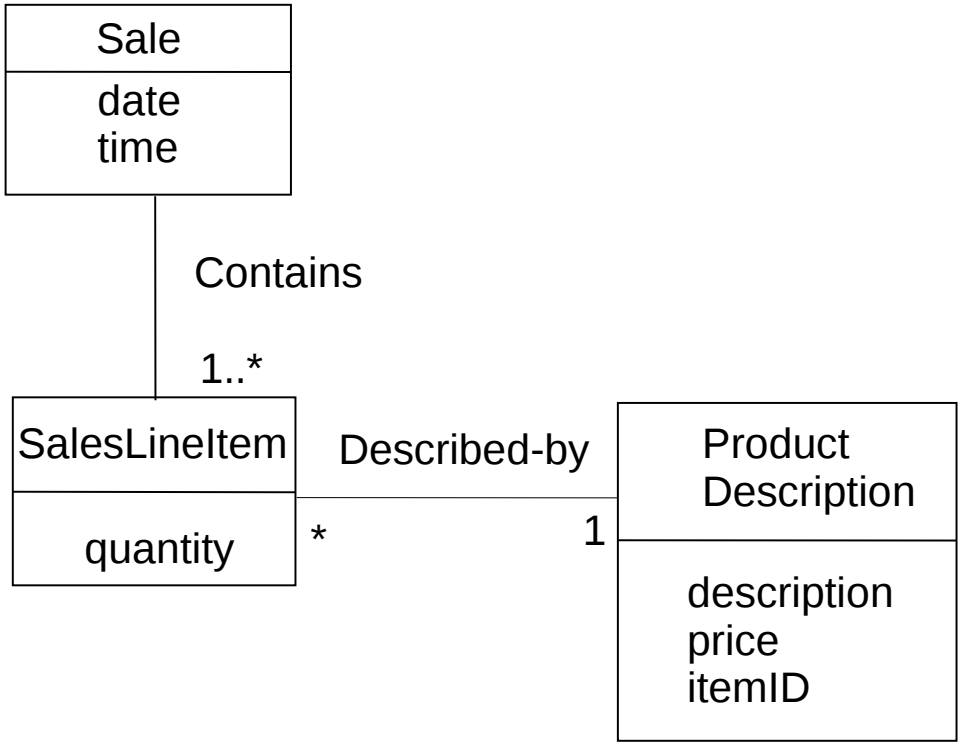
Class	Responsibility
Sale	Knows Sale total
SalesLineItem	Knows line item total
ProductDescription	Knows product price

- To fulfil the responsibility of knowing and answering the sale's total, three responsibilities were assigned to three design classes.
- The fulfillment of a responsibility often requires information that is spread across different classes of objects. This implies that there are many "partial experts" who will collaborate in the task.

Creator

- 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 at least one of the following is true:
  - B *aggregates* A objects.
  - B *contains* A objects.
  - B *records* instances of A objects.
  - B *has the initializing data* that will be passed to A when it is created (thus B is an Expert with respect to creating A).

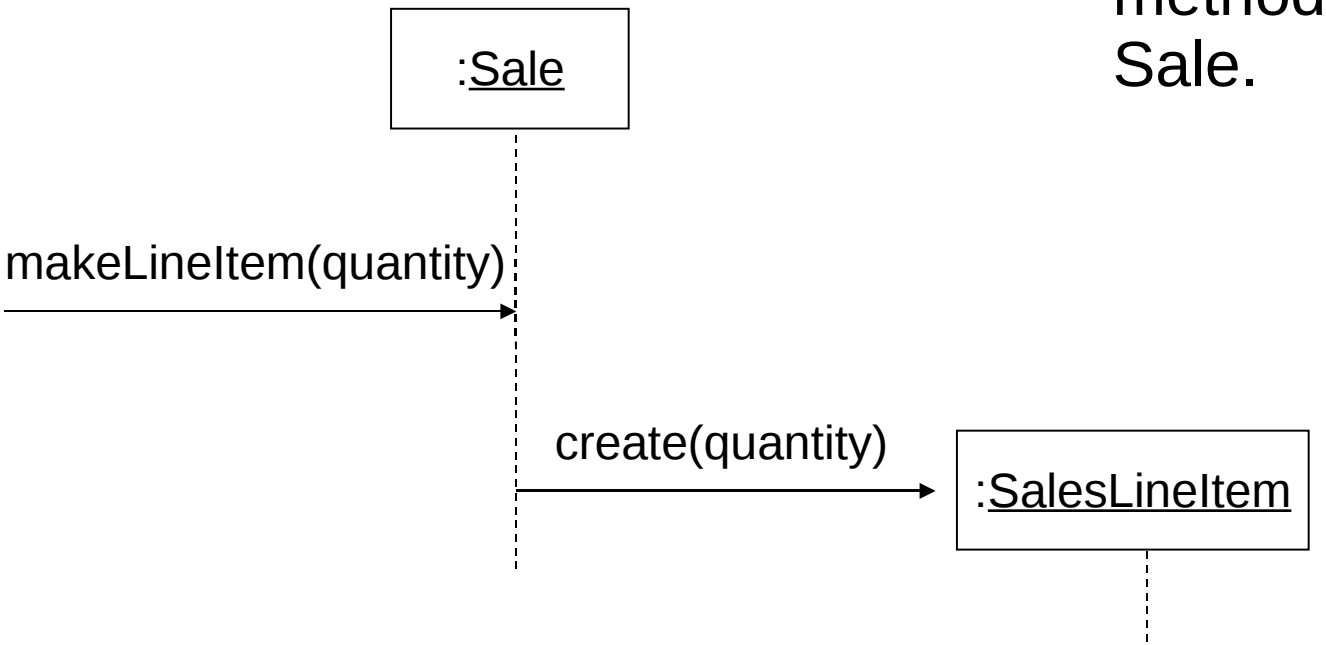
Creator



- In the POS application, who should be responsible for creating a SalesLineItem instance?
- Since a Sale contains many SalesLineItem objects, the Creator pattern suggests that Sale is a good candidate.

Creator

- This assignment of responsibilities requires that a makeLineItem method be defined in Sale.



Recall

GRASP = General Responsibility Assignment Software Patterns

Principles:

- A *responsibility* is basically a contract or obligation:

  A member of a given class must either **do** something specific or **know** something specific.

- A responsibility is not the same as a method.

  Simple responsibilities may map one-to-one, but a complex responsibility may involve many methods.

Recall

Information Expert:

- *problem:* What is the most basic principle by which to assign responsibilities to objects?
- *solution:* Assign a responsibility to the class that has the information needed to fulfill it.

Recall

Creator:

- *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 at least one of the following is true:
  - B *aggregates* A objects.
  - B *contains* A objects.
  - B *records* instances of A objects.
  - B *has the initializing data* that will be passed to A when it is created (thus B is an Expert with respect to creating A).

Simple and Complex Patterns

- If the GRASP design patterns don't look like anything new or surprising... Good! That's kind of the point. :-)

  More generally, any design pattern will look familiar to an experienced designer — that's what patterns **are**, namely a description of a common solution to a common problem.

- We'll see some much more interesting and complex patterns (including some of the so-called "Gang of Four" or "GoF" patterns) later on.

Pattern or Principle?

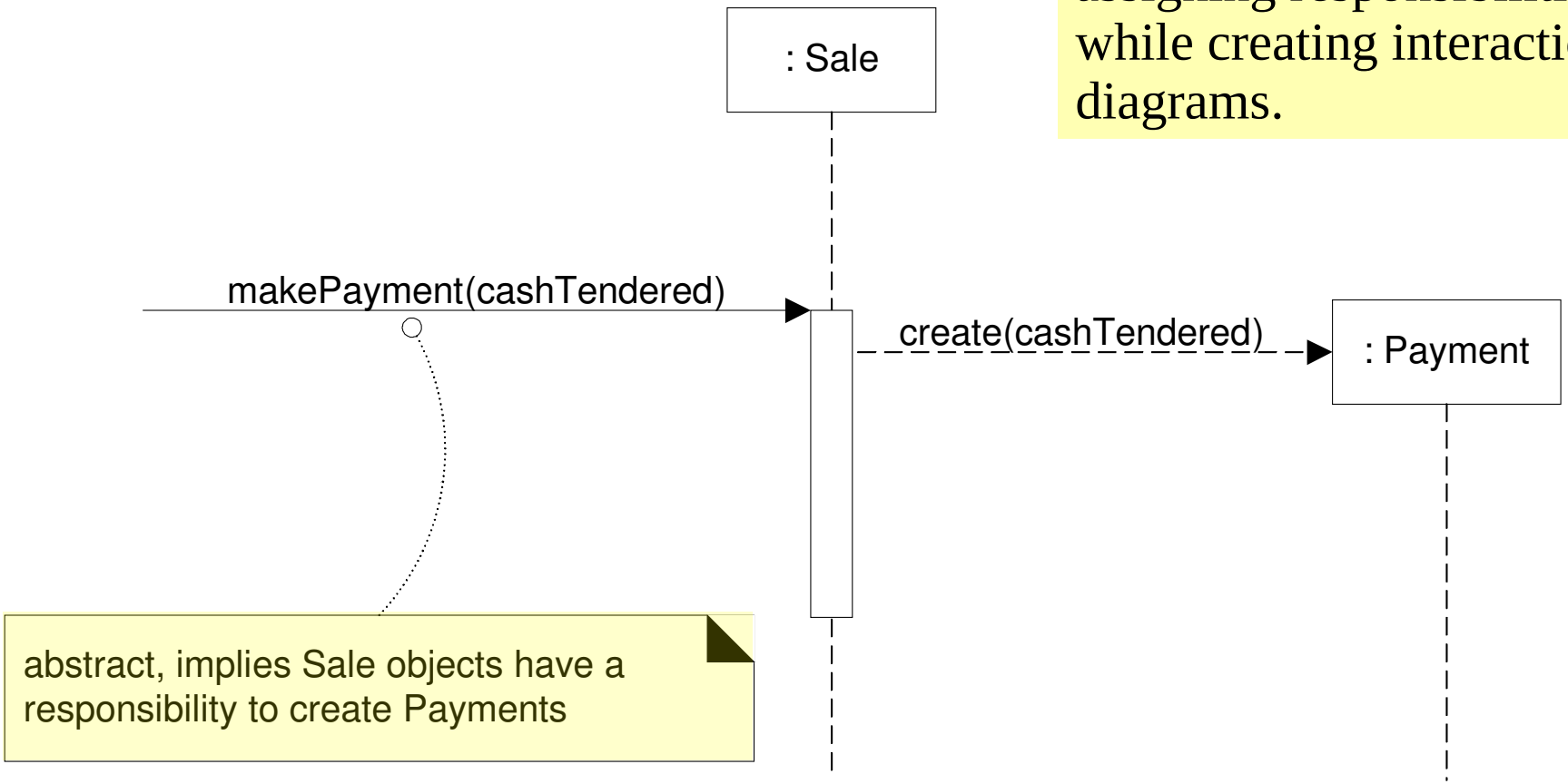
- The GRASP patterns can also be considered as general design principles.
- Is there a difference? The "Gang of Four" put it this way:

> One person's pattern is another person's primitive building block.

Whether you personally prefer to see it as a pattern or as a principle, the important thing is that you see it!

Assigning Responsibilities

A good time to think about assigning responsibilities is while creating interaction diagrams.



Larman, Figure 17.2

GRASP Patterns

- We've already seen two GRASP patterns:
  - Information Expert (or just Expert)
  - Creator
- There are seven more:
  - Low Coupling
  - High Cohesion
  - Controller
  - Polymorphism
  - Pure Fabrication
  - Indirection
  - Protected Variations

Low Coupling

- *Coupling* is a measure of how strongly one element is connected to, has knowledge of, or relies upon other elements.
- A class with high coupling depends on many other classes (libraries, tools).
- design problems caused by high coupling:
  - changes in related classes force local changes
  - harder to understand in isolation; need to understand other classes
  - harder to reuse because it requires additional presence of other classes

Low Coupling

- Problem: How to support low dependency, low change impact and increased reuse?
- Solution: Assign a responsibility so that coupling remains low.

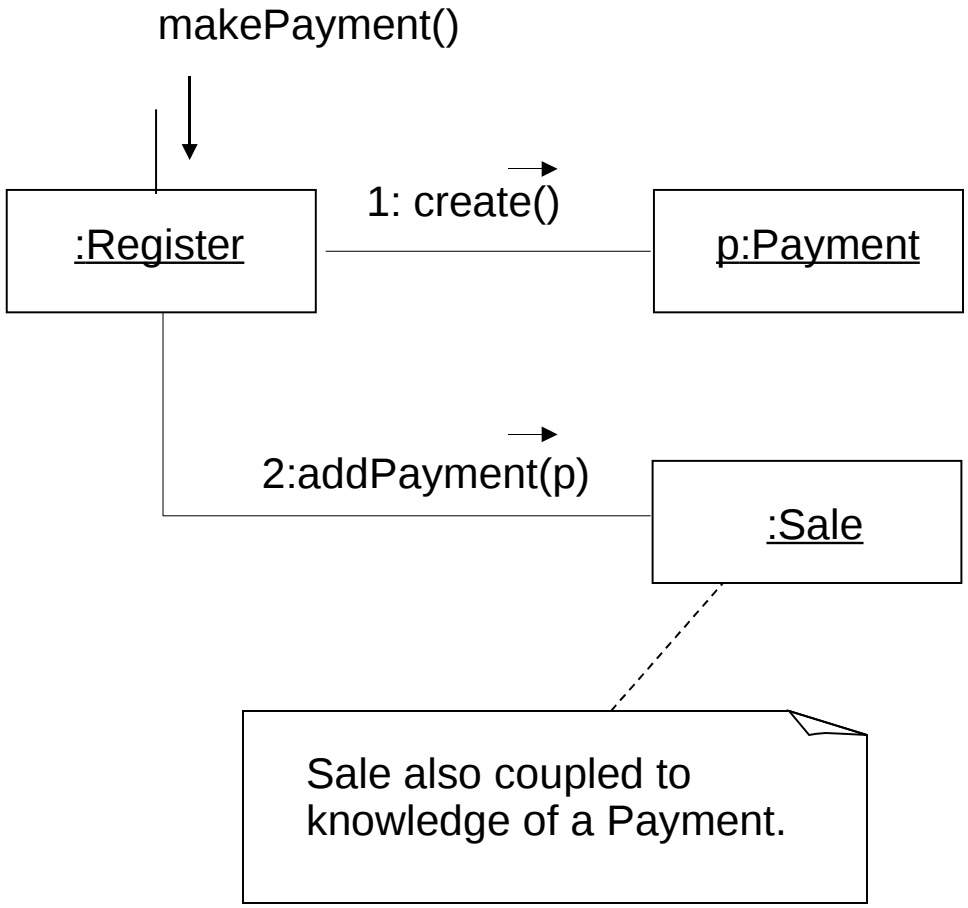
Low Coupling

- Assume we need to create a Payment instance and associate it with the Sale.
- What class should be responsible for this?

Low Coupling

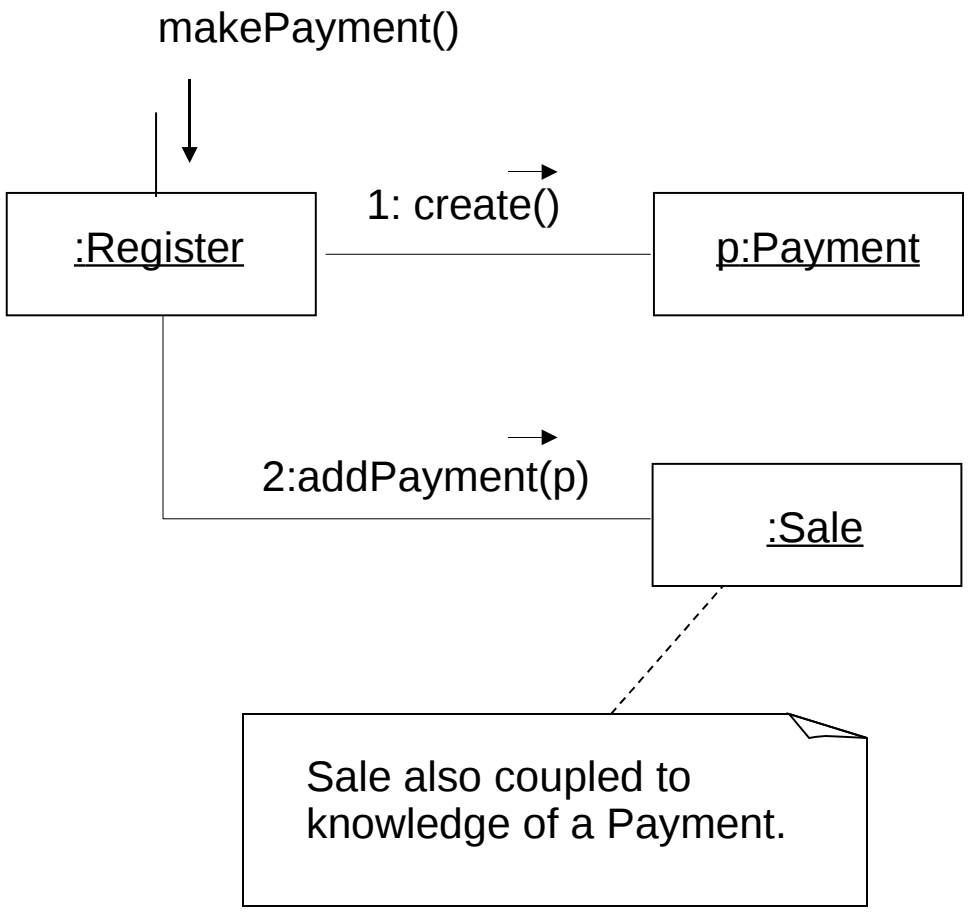
- Assume we need to create a Payment instance and associate it with the Sale.
- What class should be responsible for this?
- Creator suggests that Register is a candidate.

Low Coupling



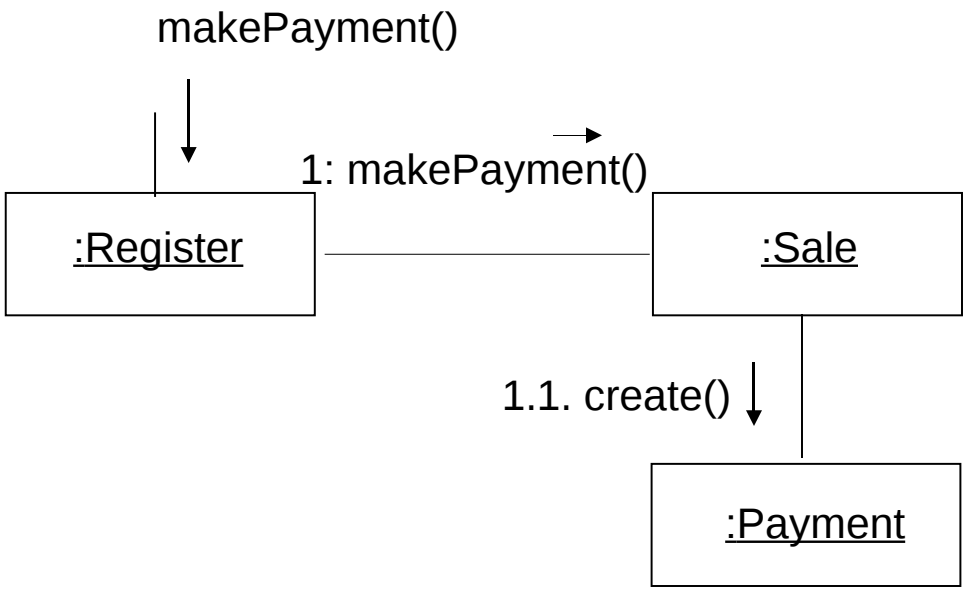
- Register could send an addPayment message to Sale, passing along the new Payment as a parameter.

Low Coupling



- Register could send an addPayment message to Sale, passing along the new Payment as a parameter.
- **BUT**: This assignment of responsibilities couples the Register class to knowledge of the Payment class.

Low Coupling



- An alternative solution is to have Sale create the Payment.
- Either way, Sale and Payment are coupled — but that's okay, because they have to be.
- ...but this design avoids unnecessary coupling between Register and Payment.

Low Coupling

- Some of the places where coupling occurs:
  - attributes: X has an attribute that refers to a Y instance.
  - methods: *e.g.* a parameter or a local variable of type Y is found in a method of X.
  - inheritance: X is a subclass of Y.
  - types: X implements interface Y.

  In general, any reference to Y inside X represents coupling.
- There is no specific measurement for coupling, but in general, classes that are generic and simple to reuse have low coupling.
- There will always be some coupling among objects. Otherwise, there would be no collaboration!

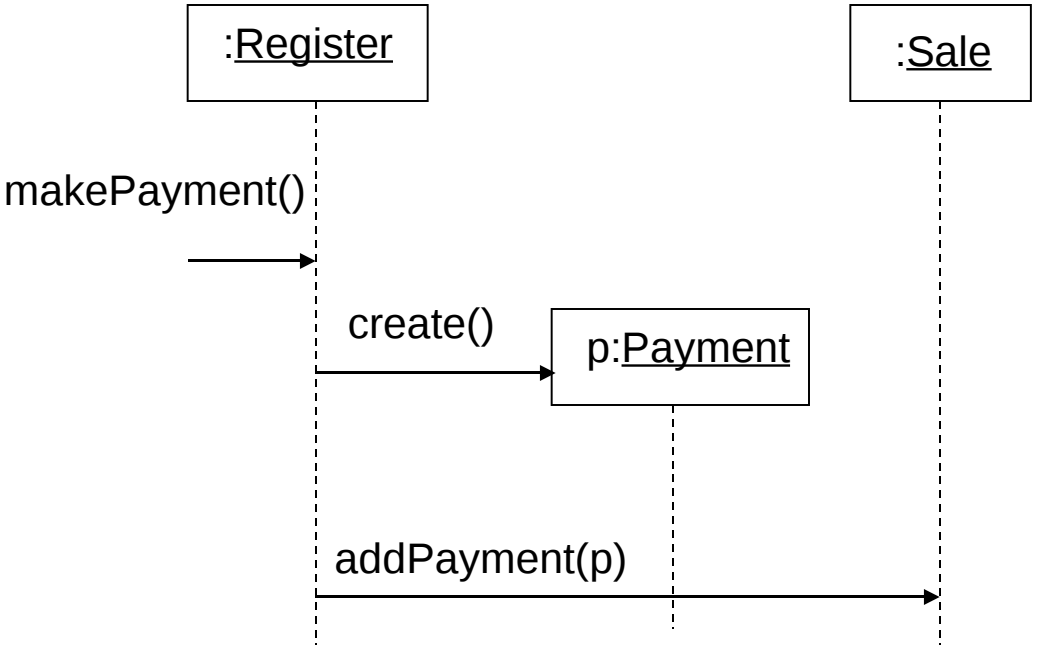
Low Coupling

- Note that high coupling isn't always a bad thing. For example, having references in your class to Java library classes isn't a problem, because those classes are always available (at least until you switch to C++ :-).
- Where high coupling becomes especially bad is when the coupled class is unstable in some way, *e.g.* one which is under active development and whose interface often changes.

High Cohesion

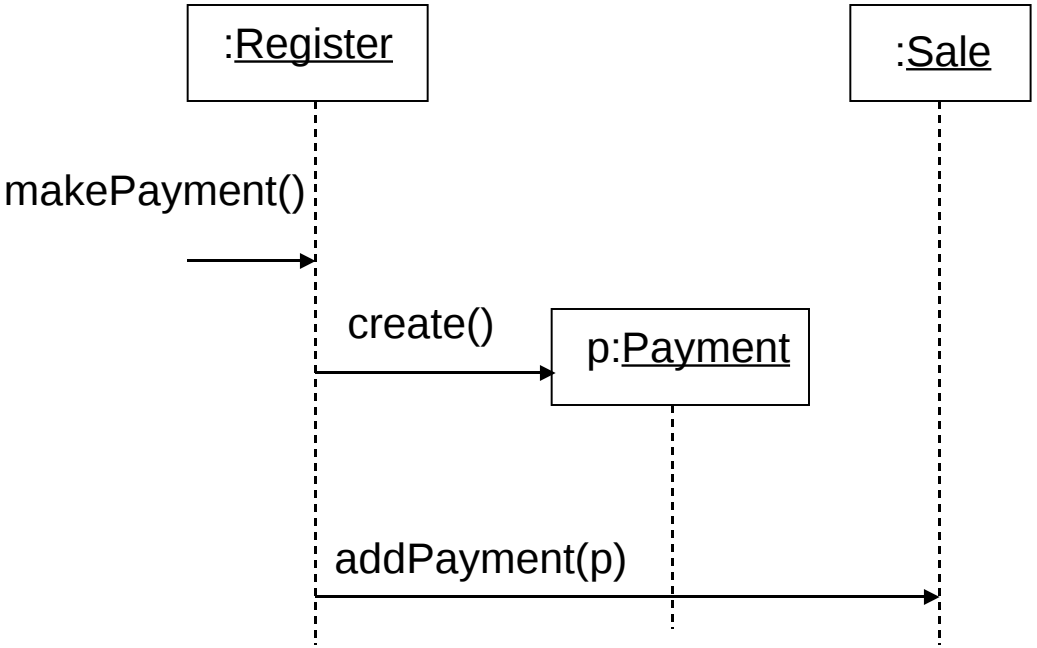
- *Cohesion* is a measure of how strongly related and focused the responsibilities of an element are.
- A class with low cohesion does many unrelated activities or does too much work.
- A design with low cohesion is fragile, *i.e.* easily affected by change.
  - Low-cohesion designs are difficult to understand, reuse, and maintain.
- Problem: How to keep complexity manageable?
- Solution: Assign a responsibility so that cohesion remains high.

High Cohesion



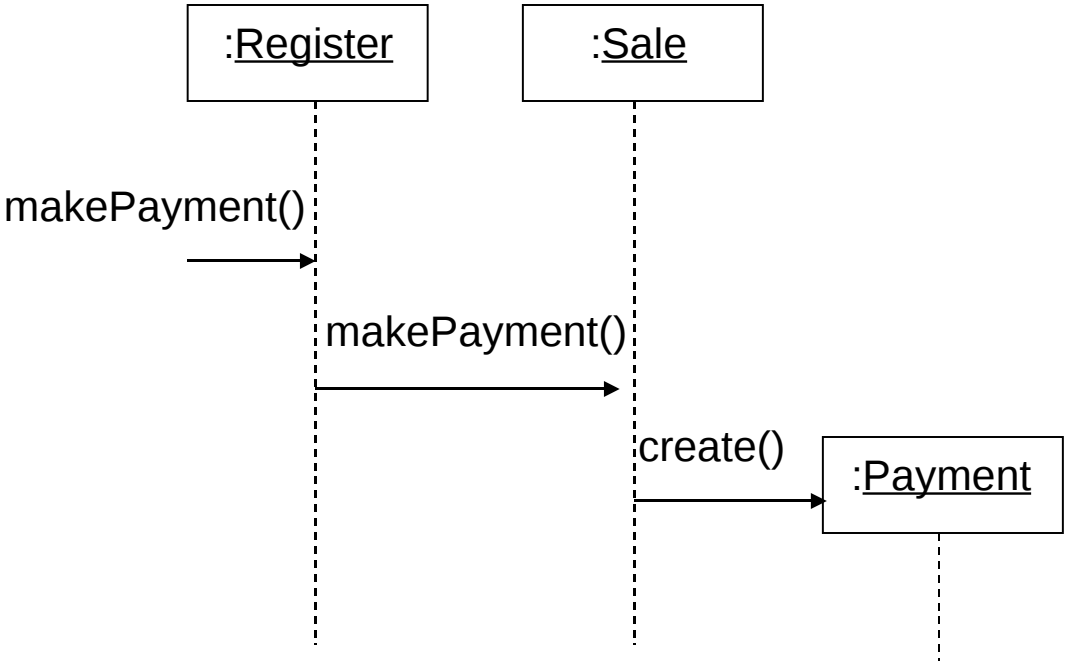
- Assume we need to create a Payment instance and associate it with Sale. What class should be responsible for this?
- Once again, Creator suggests that Register is a candidate.

High Cohesion



- Assume we need to create a Payment instance and associate it with Sale. What class should be responsible for this?
- Once again, Creator suggests that Register is a candidate.
- **BUT**: Register may become bloated if it is assigned more and more system operations.

High Cohesion



- An alternative design delegates the Payment creation responsibility to the Sale, which supports higher cohesion in the Register.
- This design supports high cohesion **and** low coupling.

High Cohesion

varying degrees of functional cohesion:

- very low cohesion: class responsible for many things in many different areas.

  e.g. a class responsible for interfacing with a data base and remote-procedure-calls

- low cohesion: class responsible for a complex task in a functional area.

  e.g. a class responsible for interacting with a relational database

- high cohesion: class has moderate responsibility in one functional area and collaborates with other classes to fulfill a task.

  e.g. a class responsible for one section of interfacing with a database.

High Cohesion

Rule of thumb:

A class with high cohesion has a relative low number of methods, with highly related functionality, and doesn't do much work itself.

Instead, it collaborates and delegates.

Modular Design

- The concept of *modular design* is much older than object-oriented programming, but it's still a good idea. :-)

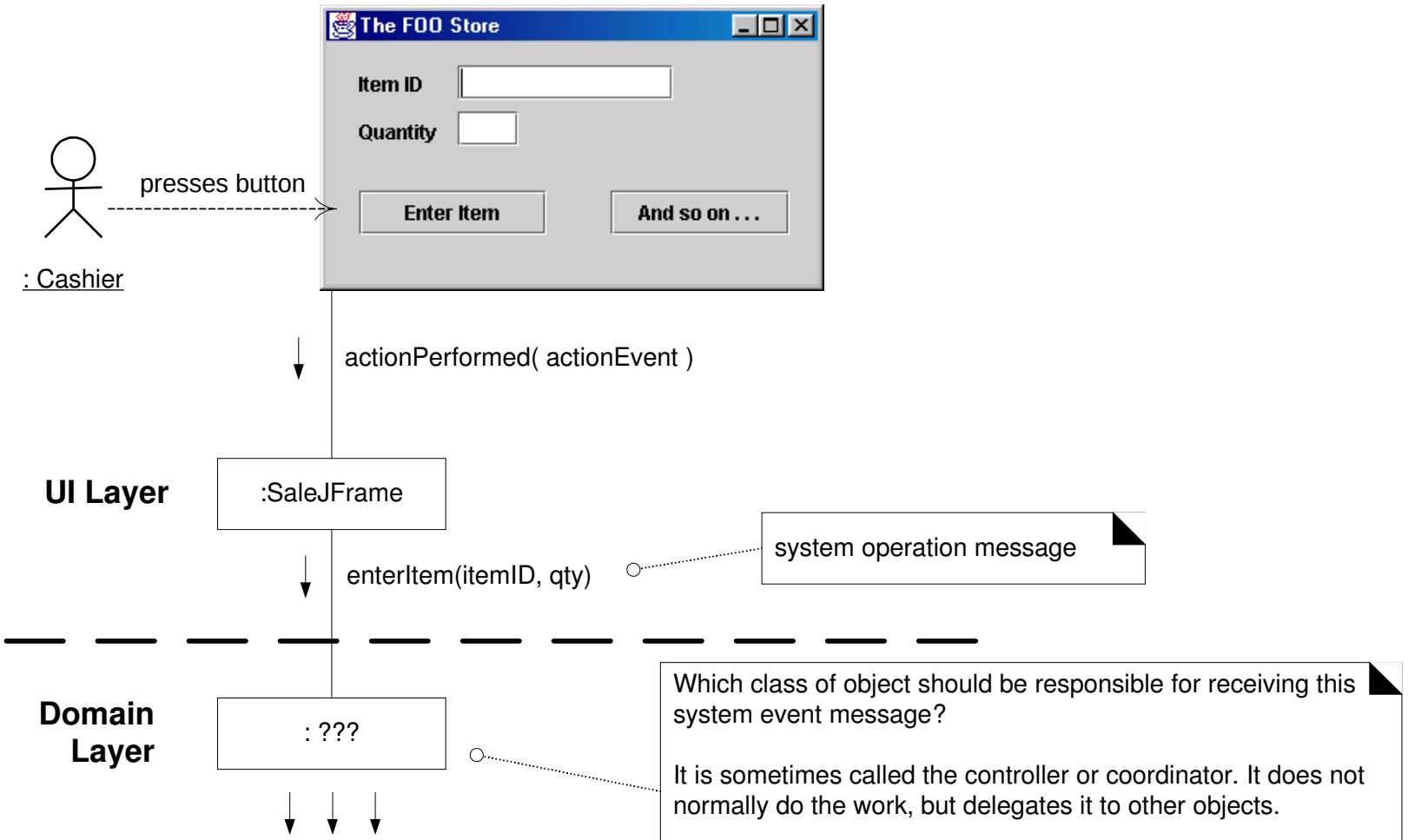
> Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules
>
> - Grady Booch, 1994.

- Note that low (or loose) coupling and high cohesion generally work together — each one helps the cause of the other.
- Likewise, high (or tight) coupling and low cohesion are often found together.

Controller

- *problem:* Beyond the UI layer, what first object should receive and coordinate a system operation?
- *solution:* Assign the responsibility to a class representing one of the following choices:
  - represents the overall system
  - represents a use case scenario in which the system event occurs
- some POS system event examples:
  - endSale(), enterItem(), makeNewSale(), makePayment()
- For an example with actual code, see pp. 309-311 in Larman.

Controller



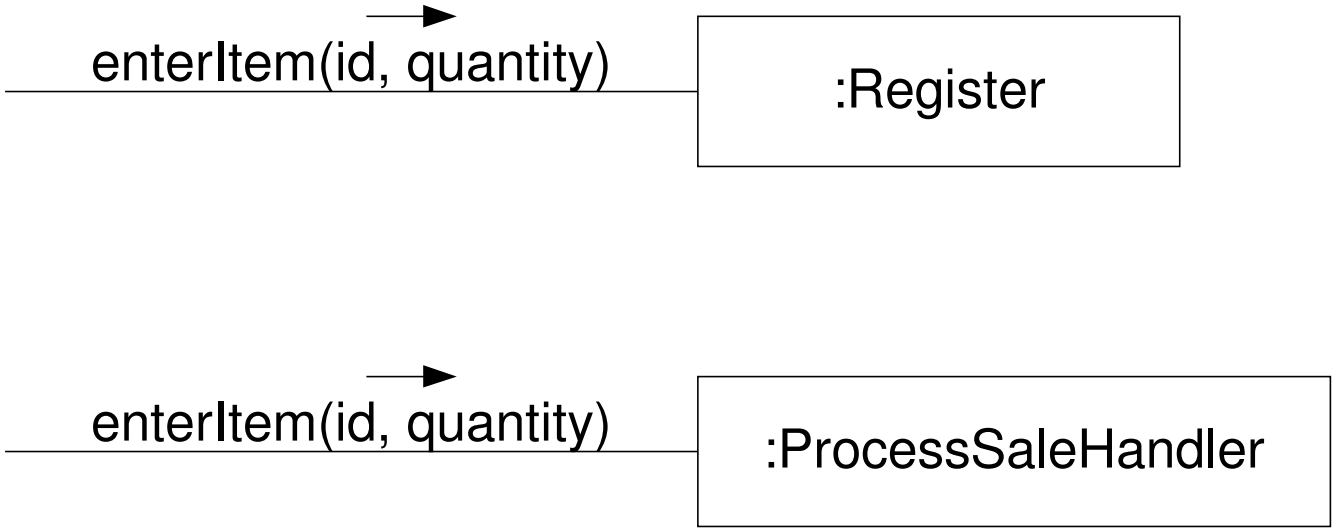
Larman, Figure 17.21

Controller

- A Controller is an object that is not part of the user interface, and which defines the method for the system operation.
- Note that classes such as "window", "view", "document", etc. look like controllers, but typically they don't handle system events — instead they're at a higher level of abstraction: they receive events and pass them to a controller.
- Controllers also (should) delegate almost all of their work. The main reason for not allowing a UI object to be a controller is to separate interface from implementation.

Controller

Which class should be the controller for enterItem()?

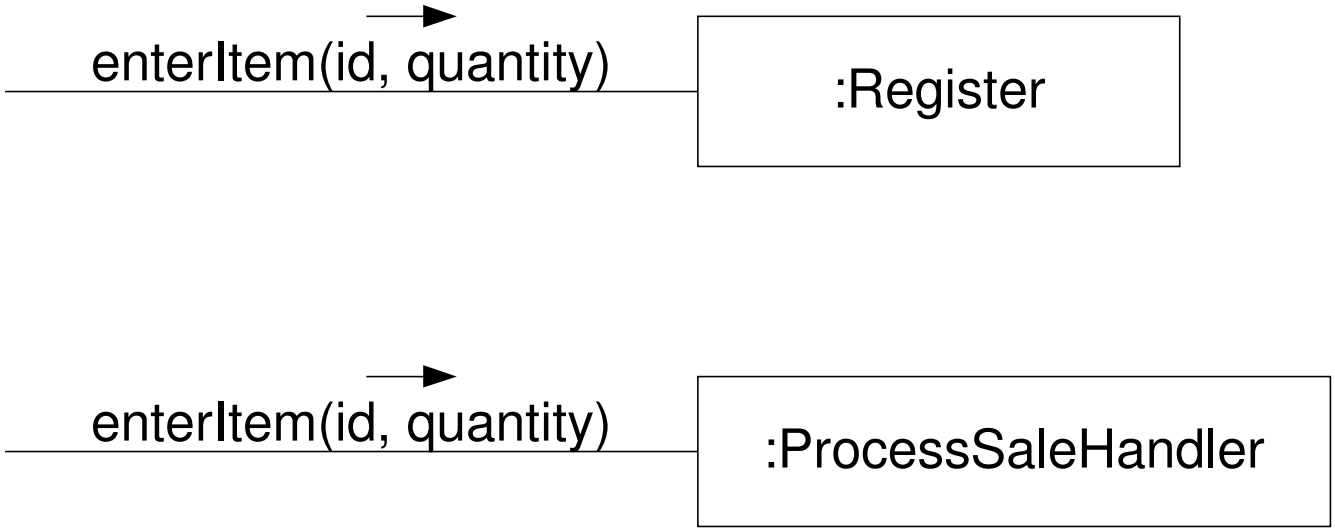


Larman, Figure 17.22

Controller

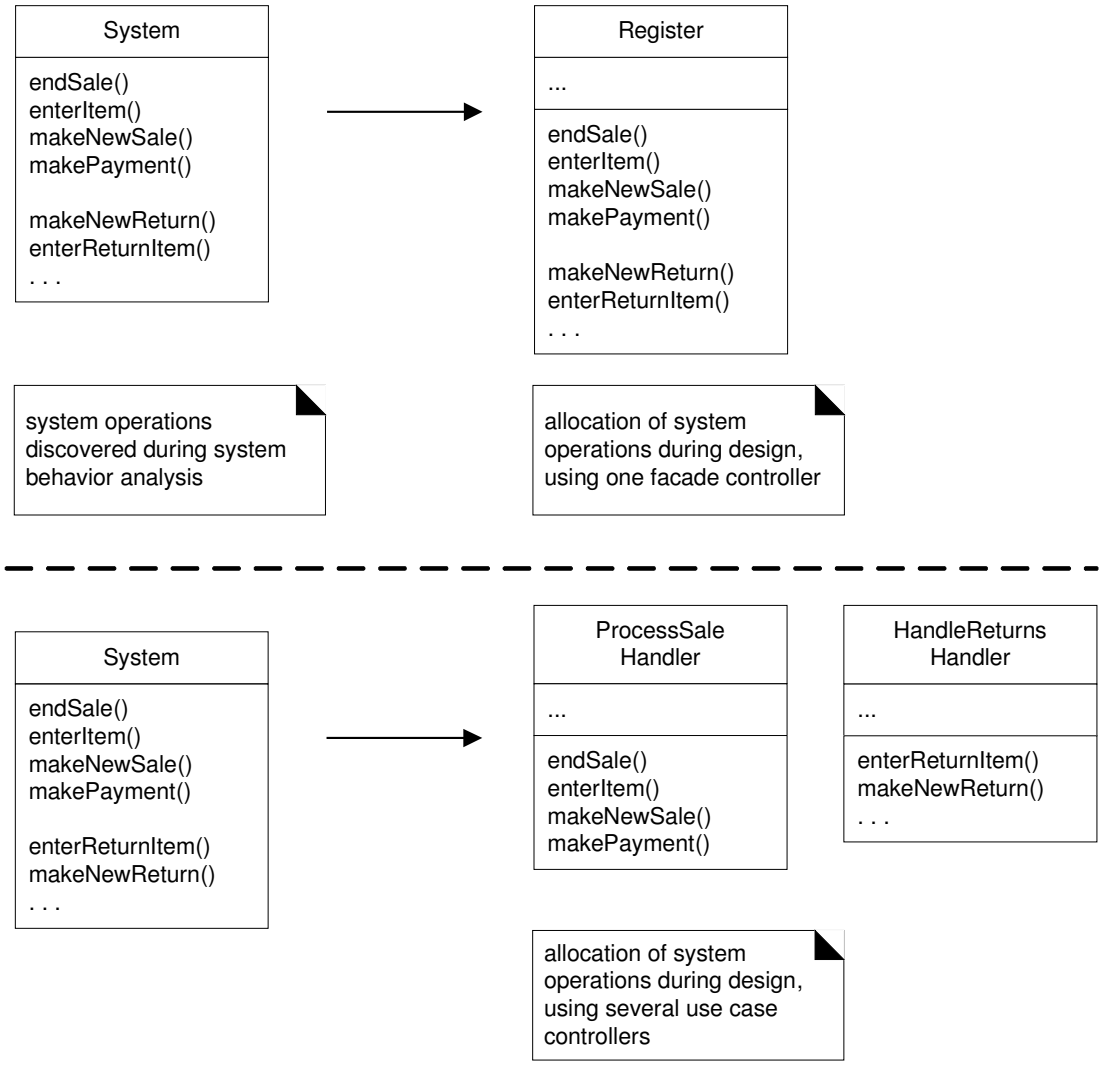
Which class should be the controller for enterItem()?

Both possibilities are reasonable. Which one is preferable depends on other factors, *e.g.* coupling and cohesion.



Larman, Figure 17.22

Controller



two possible designs

Larman, Figure 17.23

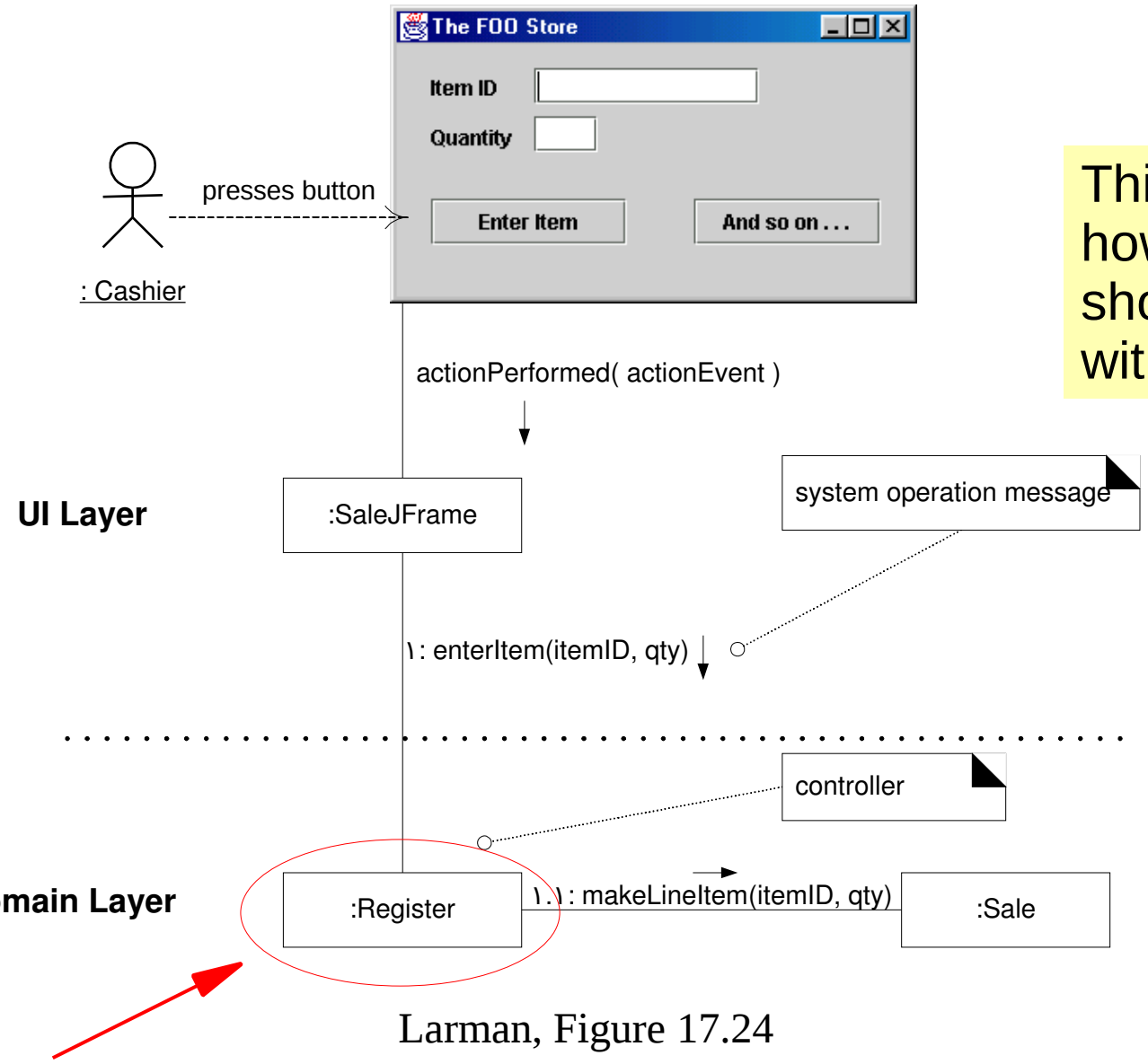
Controller

- Facade (system) controllers are generally best when there aren't too many event types.
- Use case controllers are invariably not domain objects, but instead are purely software objects that have no direct domain analogue (*cf.* the Pure Fabrication GRASP pattern).
- Use case controllers are generally best when a Facade controller would suffer from low cohesion or high coupling.
- Typically the same controller would be used for all events corresponding to different scenarios of the same use case. This makes it possible to maintain state.

Controller

- signs that a controller class is badly designed:
  - There is only one controller class in the system, it receives all event types, and there are many event types. This is a recipe for very low cohesion.
  - The controller itself performs most of the work needed to handle events, rather than delegating. This usually violates the Information Expert pattern (or principle :-), and also leads to low cohesion.
  - Many of the controller's attributes are duplicates of those in other classes.
- possible fixes:
  - Add more controllers if one is too big and too unfocused.
  - Redesign the controller to delegate as much as possible.

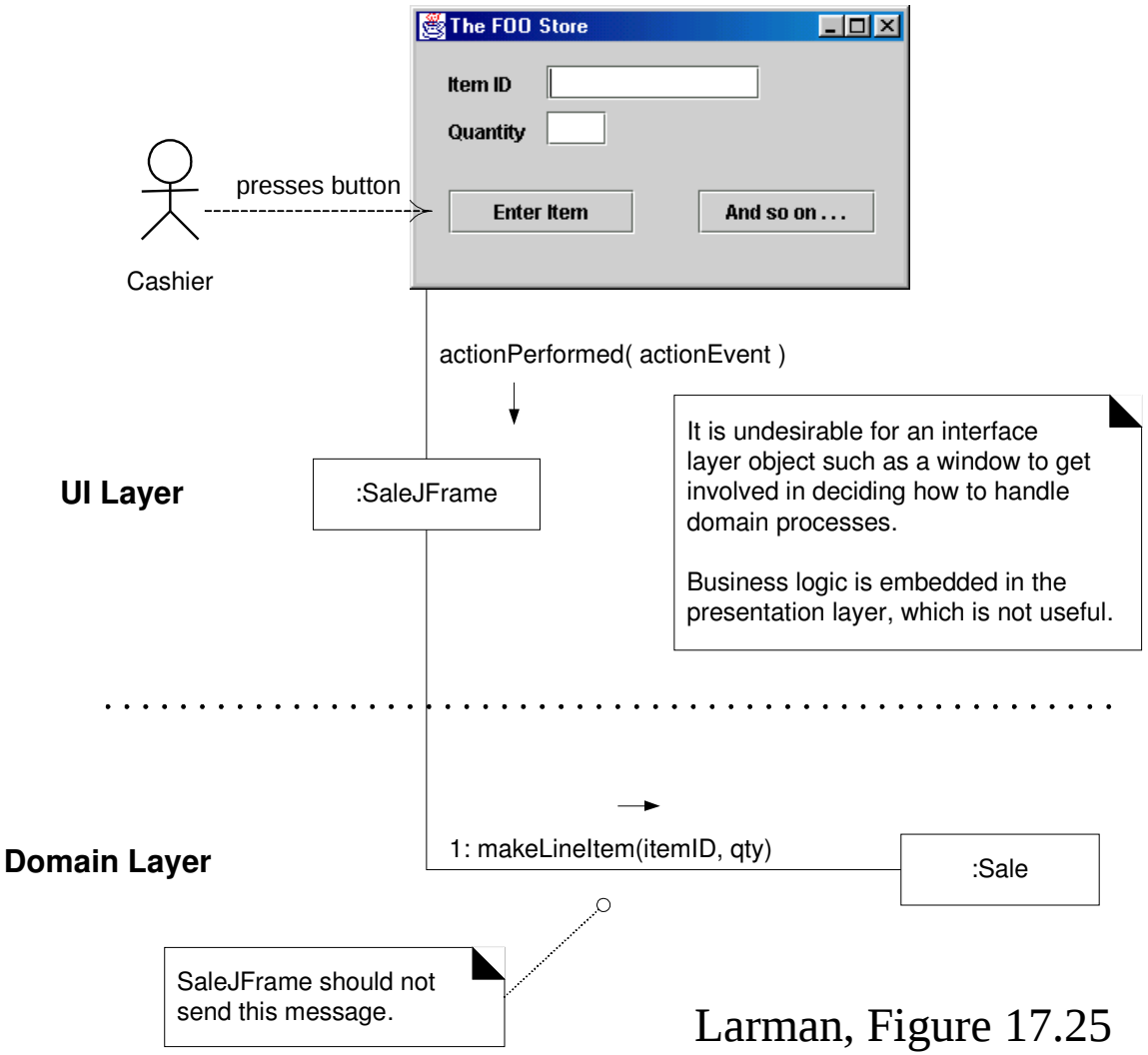
Controller



This example shows how the UI layer should communicate with the domain layer.

Larman, Figure 17.24

Controller



Larman, Figure 17.25

Don't do this!

What principles are being violated here?

Use Case Realizations

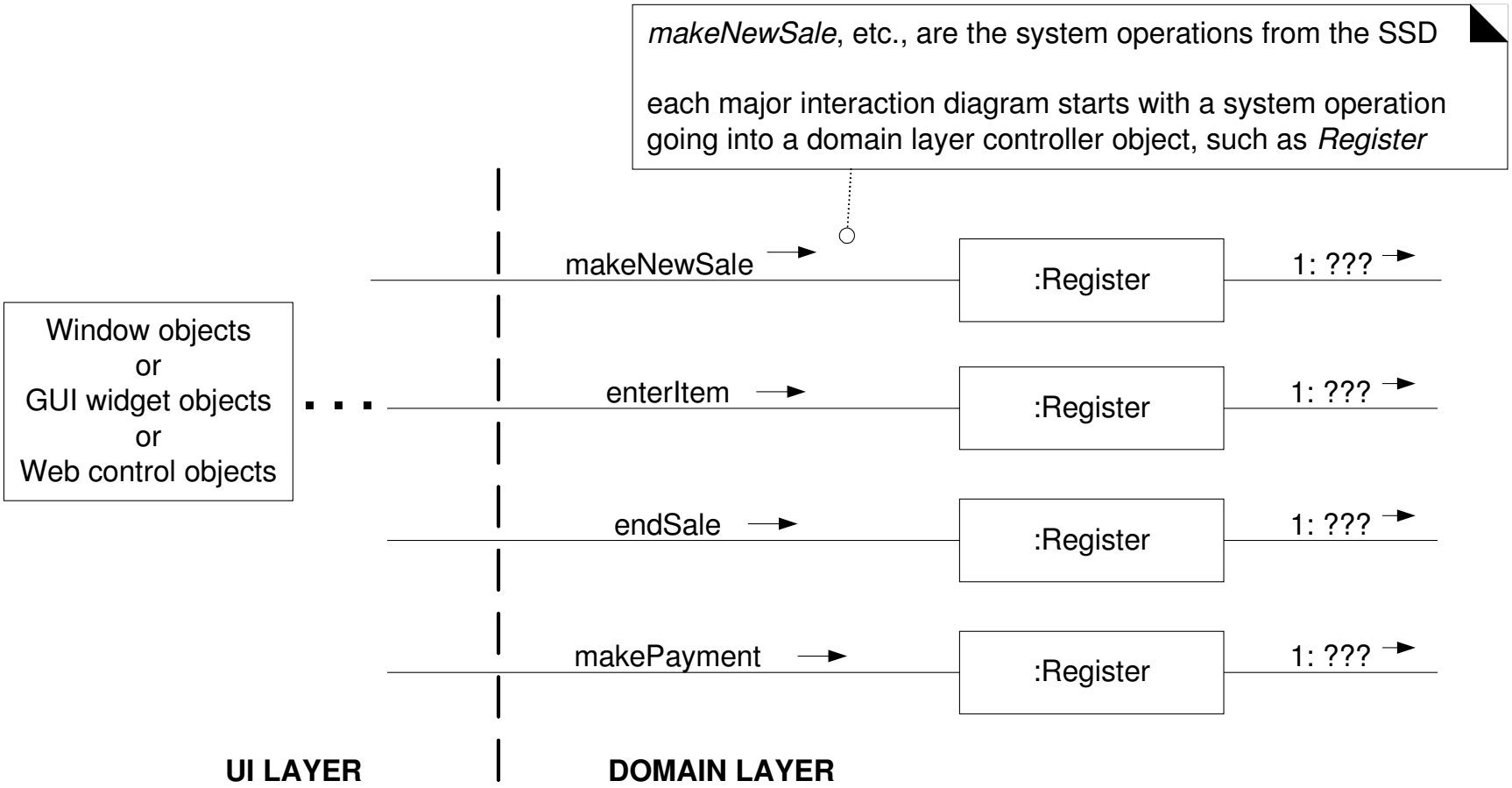
Use Case Realizations

- A use case realization describes the design for a given use case, in terms of collaborating objects.
- UML interaction diagrams are used to illustrate use case realizations.
- Each use case identifies a number of system events (operations). These are shown in system sequence diagrams.
- The system events become the starting messages that enter the Controllers for the domain, as shown in a domain layer interaction diagram. For example, for the POS system we have

System
makeNewSale() enterItem(itemID, quantity) endSale() makePayment()

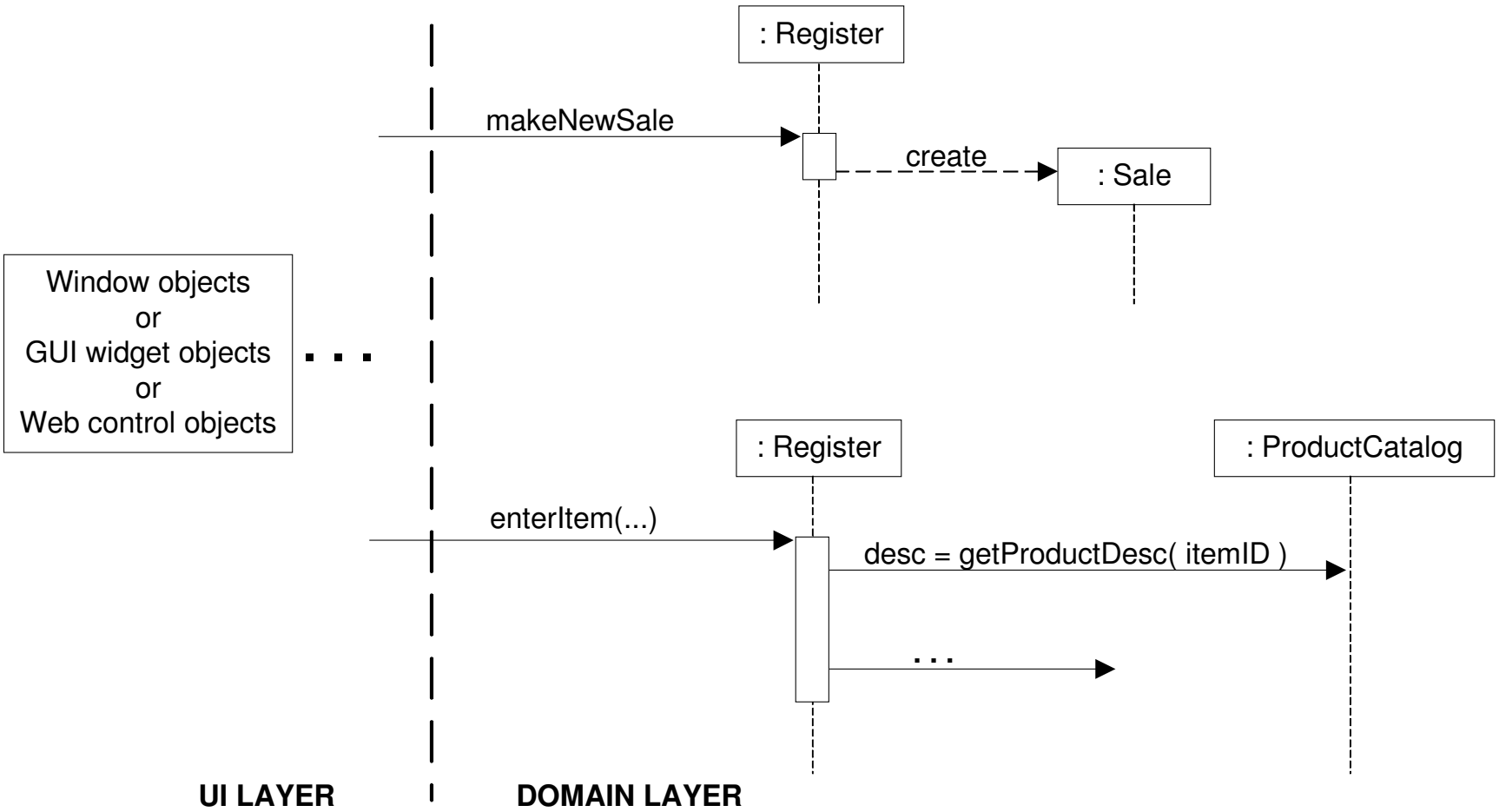
This is the starting point of the design for this use case!

Use Case Realizations



Larman, Figure 18.2

Use Case Realizations



Larman, Figure 18.3

Use Case Realizations

We can certainly start with the use cases themselves, but it's probably easier to use contracts if they exist. For example,

Contract CO1: makeNewSale

Operation: makeNewSale ()

Cross References: Use Cases: Process Sale.

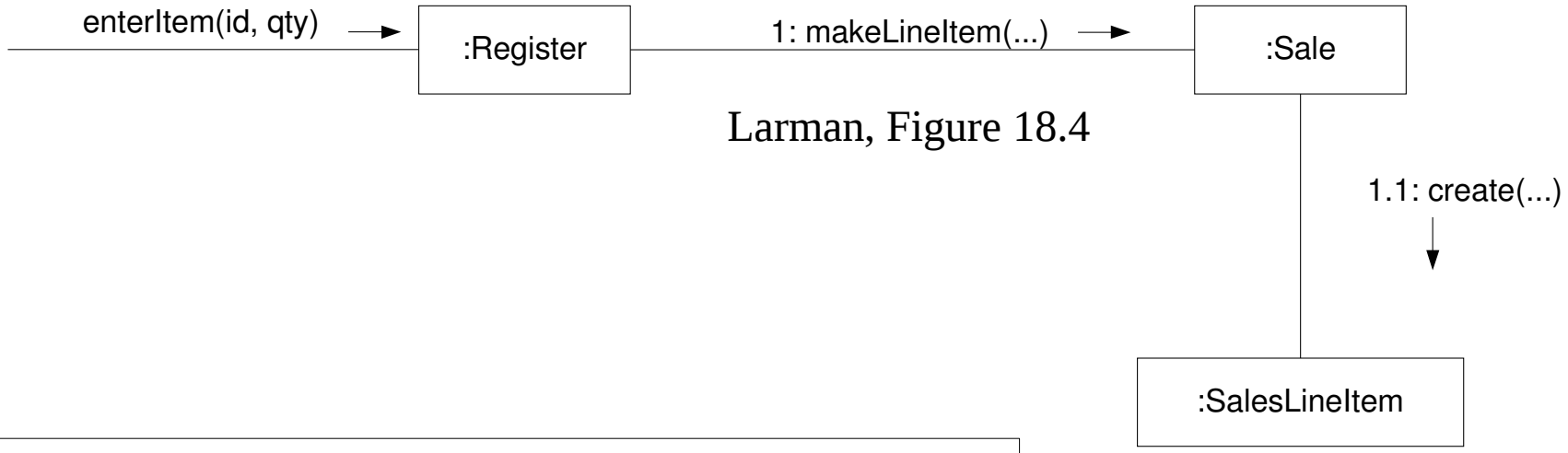
Pre-conditions: none.

Post-conditions:

- A Sale instance *s* was created. (instance creation)
- s was associated with the Register (association formed)
- Attributes of s were initialized

Along with the use case text, the postcondition state changes give us the message interactions that will be needed to satisfy the requirements.

Use Case Realizations



Larman, Figure 18.4

Contract CO2: enterItem

Operation: enterItem(itemID: ItemID, quantity: integer)

Cross References: Use Cases: Process Sale.

Pre-conditions: There is a sale underway..

Post-conditions:

- A SalesLineItem instance *sli* was created. (instance creation)
- [...]

Object Design: makeNewSale

Recall the contract for makeNewSale. To design this operation, step 1 is to choose a Controller. Some possibilities:

Contract CO1: makeNewSale

Operation: makeNewSale ()

Cross References: Use Cases: Process Sale.

Pre-conditions: none.

Post-conditions:

- A Sale instance *s* was created. (instance creation)
- s was associated with the Register (association formed)
- Attributes of s were initialized

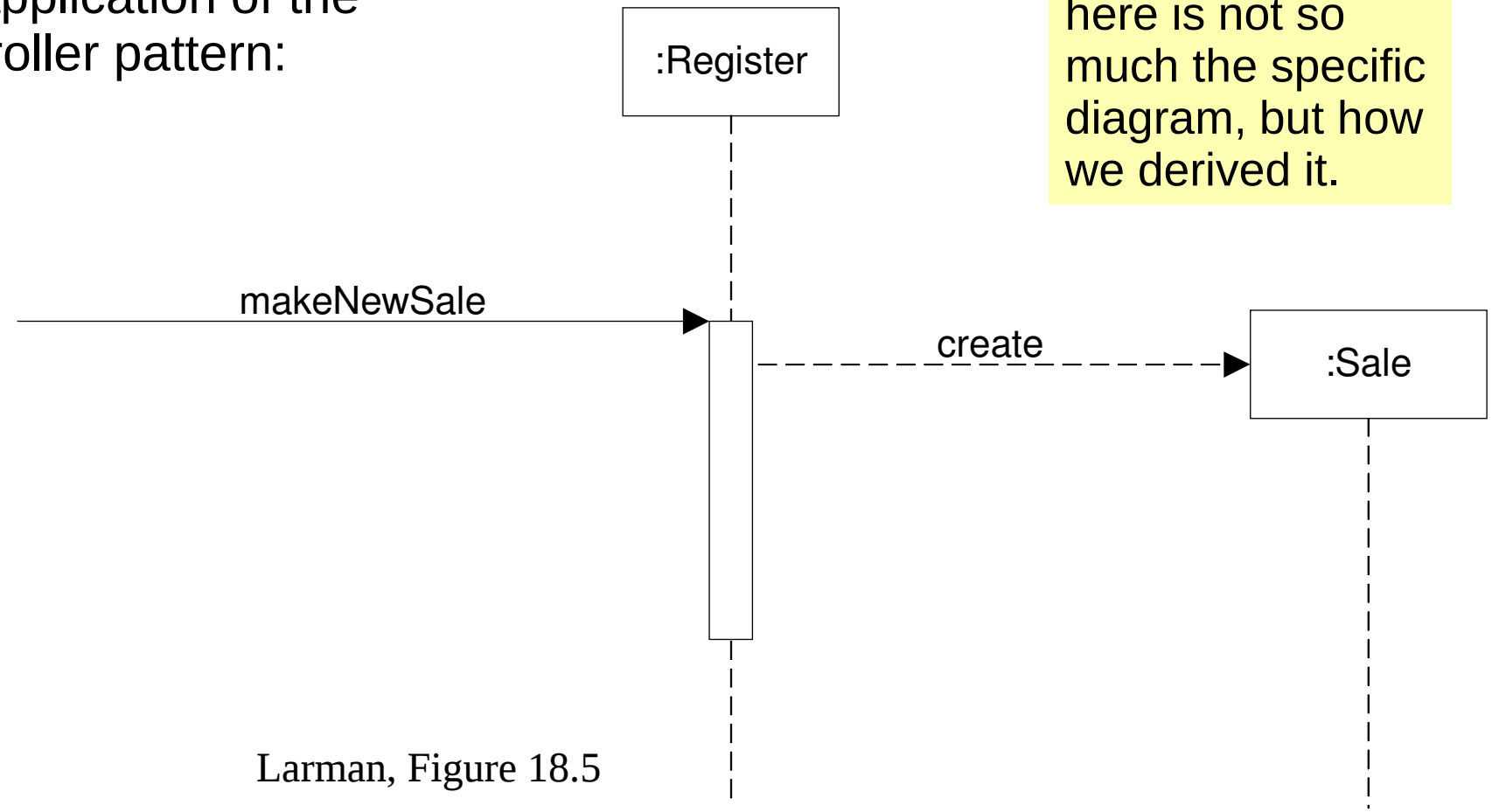
- Store
- Register
- POSSystem

- ProcessSaleHandler
- ProcessSaleSession

In this case, Register will do well enough since there aren't many system operations.

Object Design: makeNewSale

Our design starts out with this application of the Controller pattern:



Larman, Figure 18.5

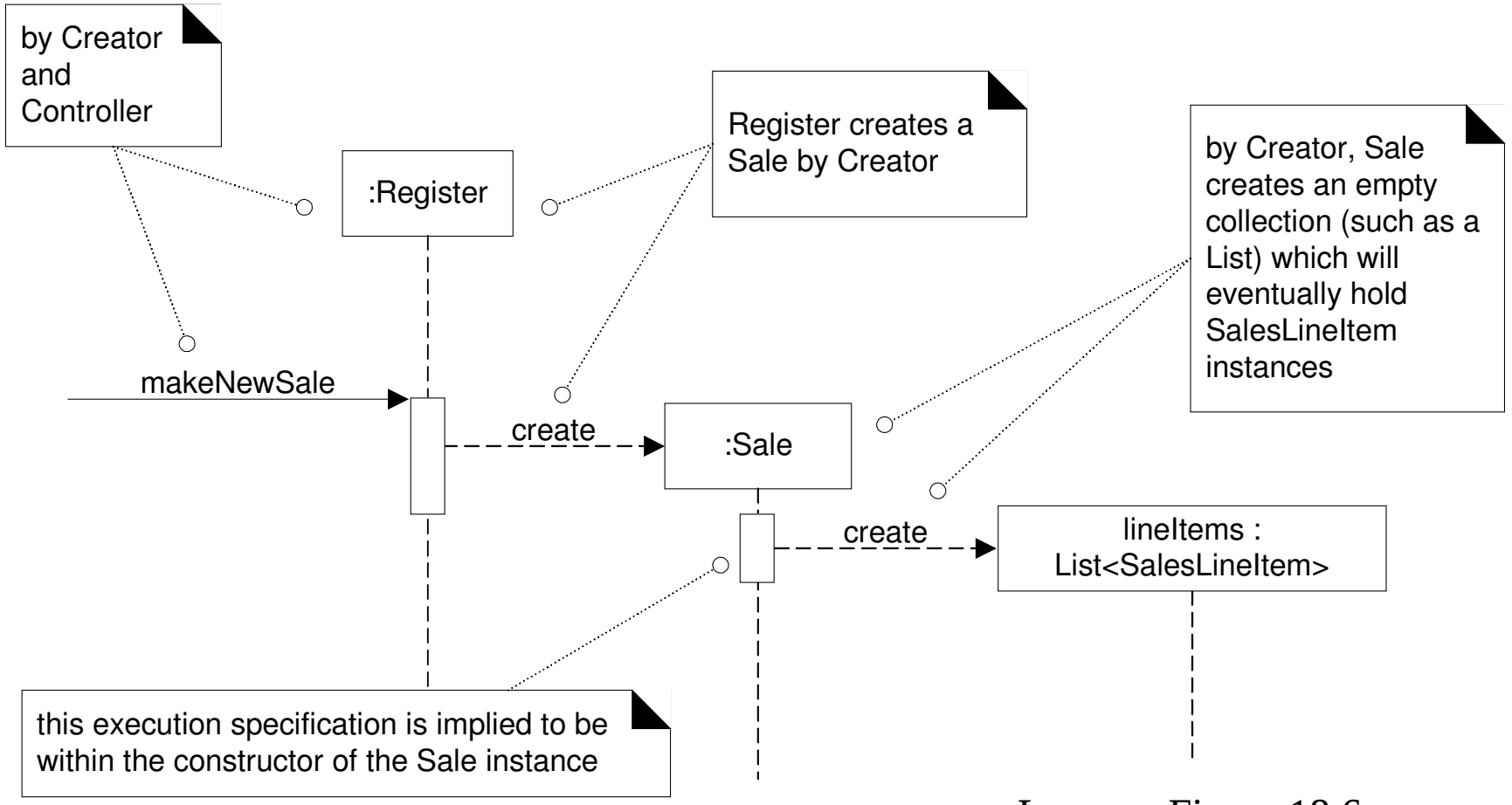
Note that the important thing here is not so much the specific diagram, but how we derived it.

Object Design: makeNewSale

- Now that we have a Controller, the next step is to consider creation of the Sale object.
- The Creator pattern suggests that Register is the obvious candidate — which shouldn't be surprising, especially if you stop to think what the word 'register' actually means. :-)
- When a Sale is created, it will need an empty collection in which to store SalesLineItems. As Creator suggests, Sale itself is the obvious place to create this.

Object Design: makeNewSale

Here's the full picture — but once again, the analysis is more important than the result:



Larman, Figure 18.6

Object Design: enterItem

Now let's look at the full set of postconditions for enterItem:

Contract CO2: enterItem

Operation: enterItem(itemID: ItemID, quantity: integer)

Cross References: Use Cases: Process Sale.

Pre-conditions: There is a sale underway..

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)
- *sli* was associated with a ProductDescription, based on itemID match (association formed)

What are the design decisions to be made here?

Object Design: enterItem

Design questions:

- Which Controller class should we use?
  - By the same logic as for makeNewSale, the Controller should be Register.
- Should we display Item Description and Price?
  - The use case says we should, but non-GUI objects such as Register and Sale shouldn't normally be involved in output. We'll return to this requirement later; for now, we'll just ensure that we have the information we'd need in order to be able to display these values.
- How to create a new SalesLineItem?
  - The postconditions require that a SalesLineItem be created. The domain model states that a Sale contains SalesLineItems, which suggests that a software Sale object could do likewise. The Creator pattern tells us that it's reasonable for Sale to create the SalesLineItem.

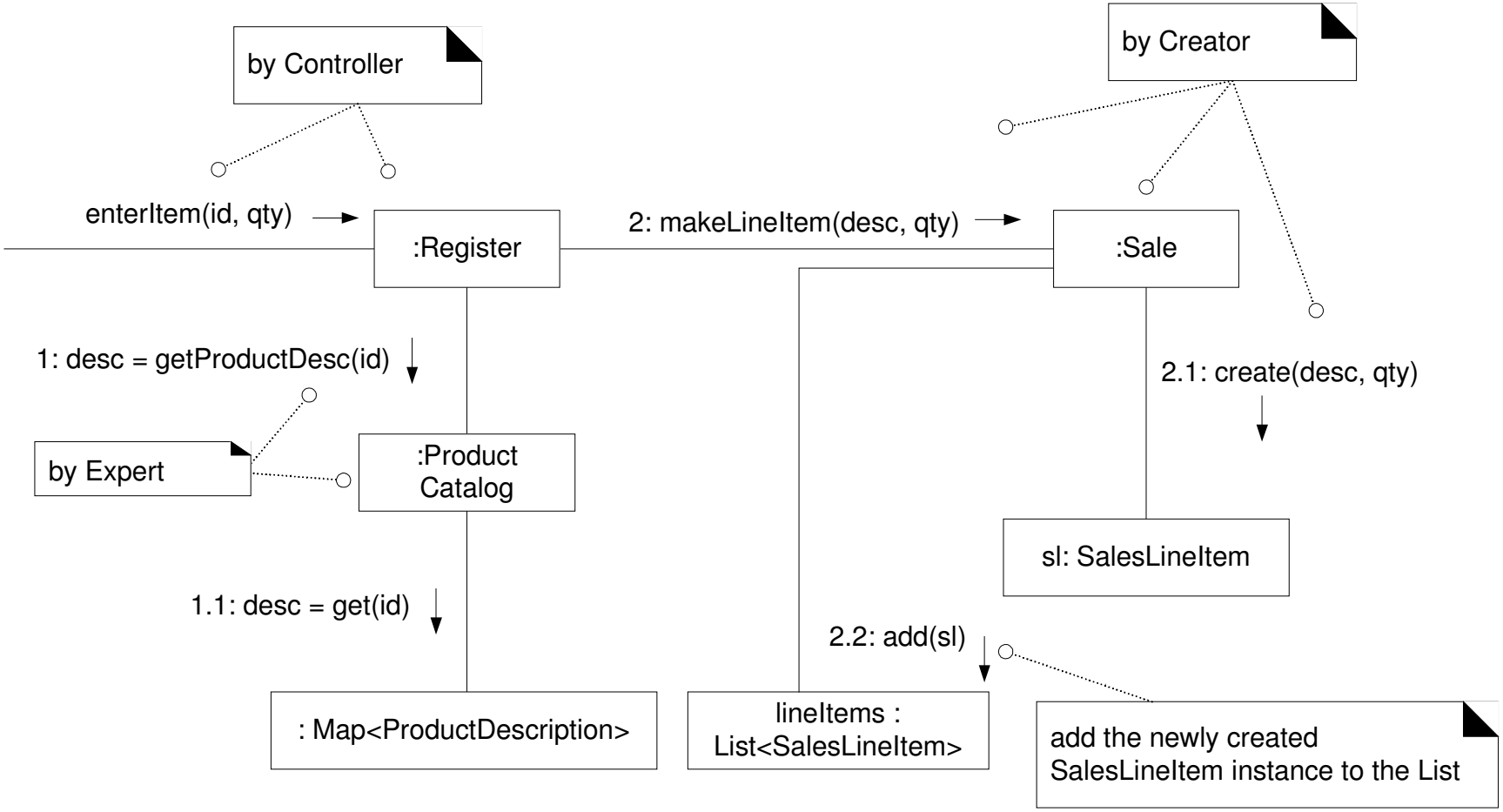
Object Design: enterItem

Design questions, continued:

- How to find a ProductDescription?
  - A SalesLineItem needs a ProductDescription to match the incoming itemID. In other words, we must look up the itemID to find the description. Who should be responsible for this lookup? This is a job for Information Expert, which suggests that ProductCatalog is the class which knows about product descriptions — so let's design a ProductCatalog class which matches this domain concept, and which contains a getProductDescription method.
- Who should instigate the ProductDescription lookup?
  - Given that ProductCatalog will do the lookup, who should send it the message asking it to do so? It's reasonable to assume that both a Register and a ProductCatalog instance were created at startup (this assumption should be recorded!), so we can safely have the Register assume this responsibility. This implies the concept of *visibility*, which we'll come back to shortly.

Object Design: enterItem

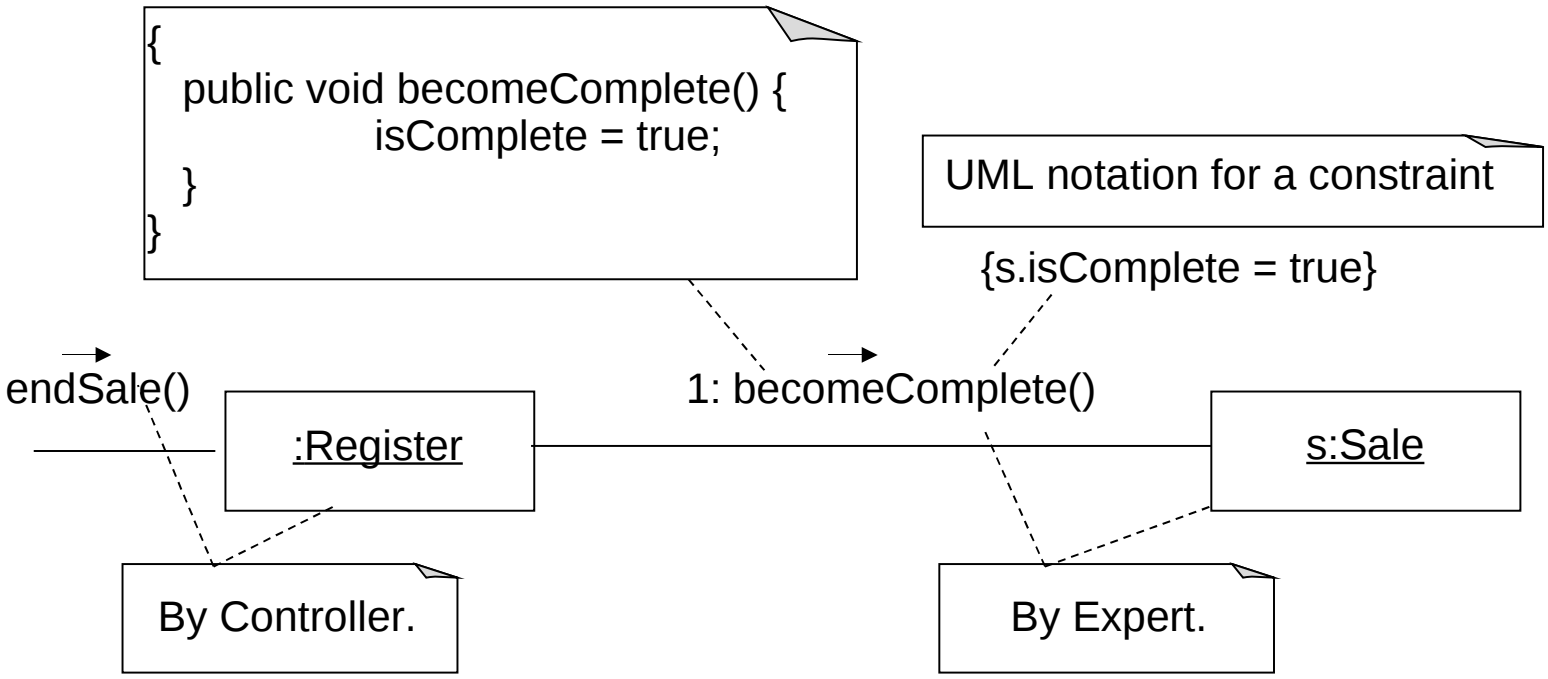
Putting everything together, we get this picture:



Larman, Figure 18.7

Object Design: endSale

- **Contract CO3: endSale**
- **…**
- **Post-conditions:**
  - Sale.isComplete became true (attribute modification)

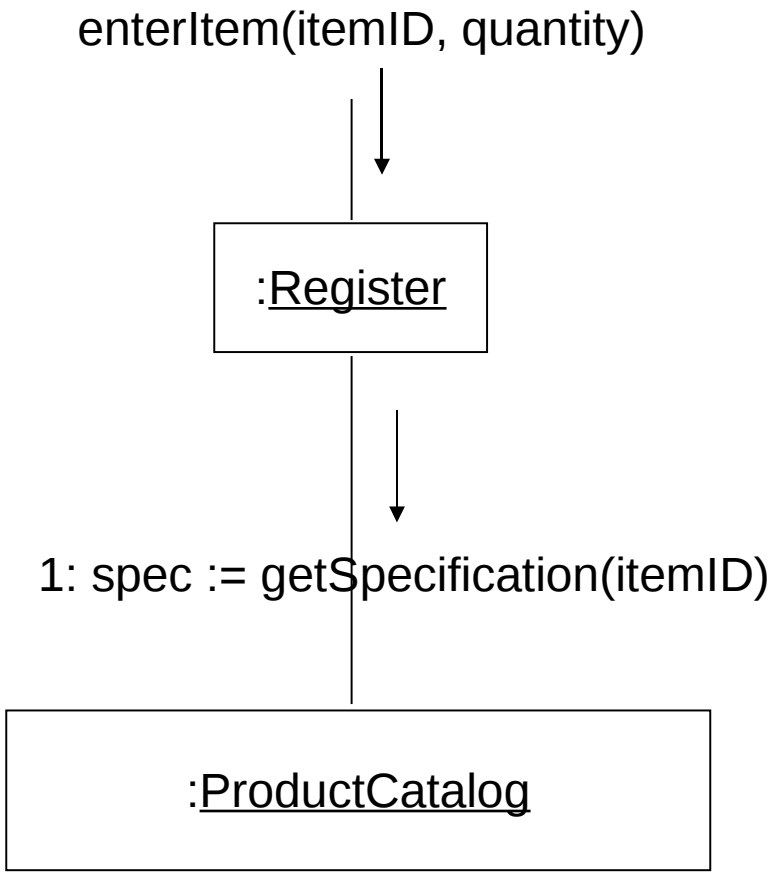


Design Model: Determining Visibility

Introduction

- Visibility: the ability of an object to “see” or have reference to another object.
- For a sender object to send a message to a receiver object, the receiver must be visible to the sender – the sender must have some kind of reference (or pointer) to the receiver object.

Visibility Between Objects

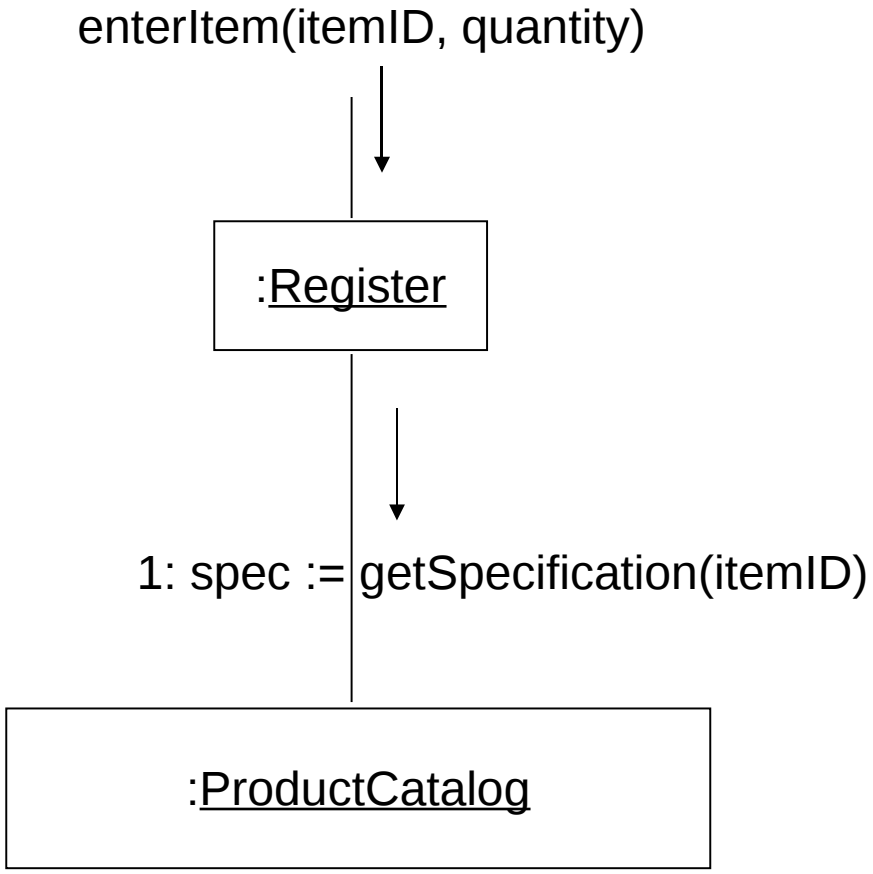


- The getSpecification message sent from a Register to a ProductCatalog implies that the ProductCatalog instance is visible to the Register instance.

Visibility

- How do we determine whether one resource (such as an instance) is within the scope of another?
- Visibility can be achieved from object A to object B in four common ways:
  - Attribute visibility: B is an attribute of A.
  - Parameter visibility: B is a parameter of a method of A.
  - Local visibility: B is a (non-parameter) local object in a method of A.
  - Global visibility: B is in some way globally visible.

Visibility

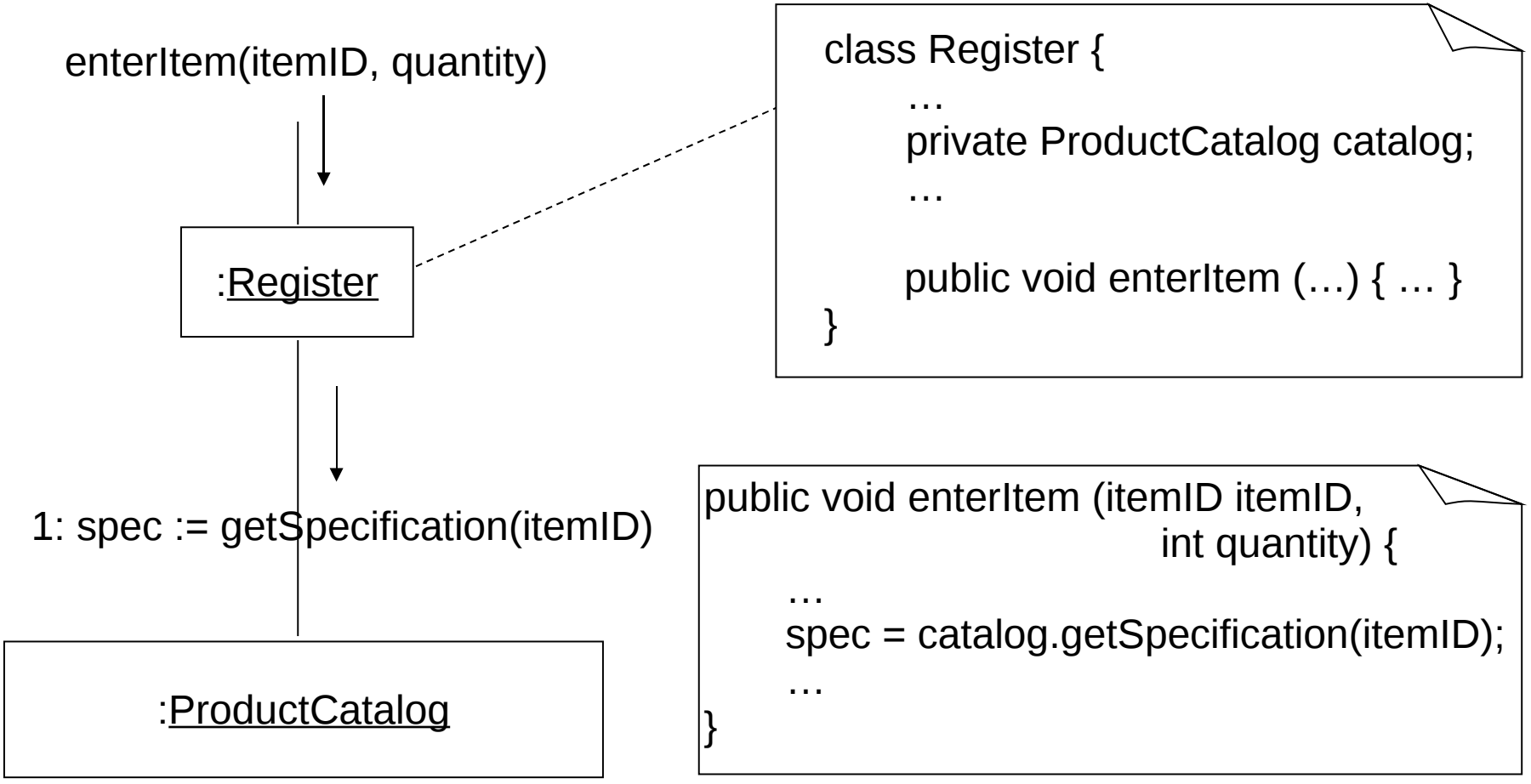


- The ProductCatalog must be visible to the Register.
- A typical visibility solution is that a reference to the ProductCatalog instance is maintained as an attribute of the Register.

Attribute Visibility

- Attribute visibility from A to B exists when B is an attribute of A.
- This is a relatively permanent visibility, because it persists as long as A and B exist.
- In the enterItem collaboration diagram, Register needs to send the getSpecification message to a ProductCatalog. Thus, visibility from Register to ProductCatalog is required.

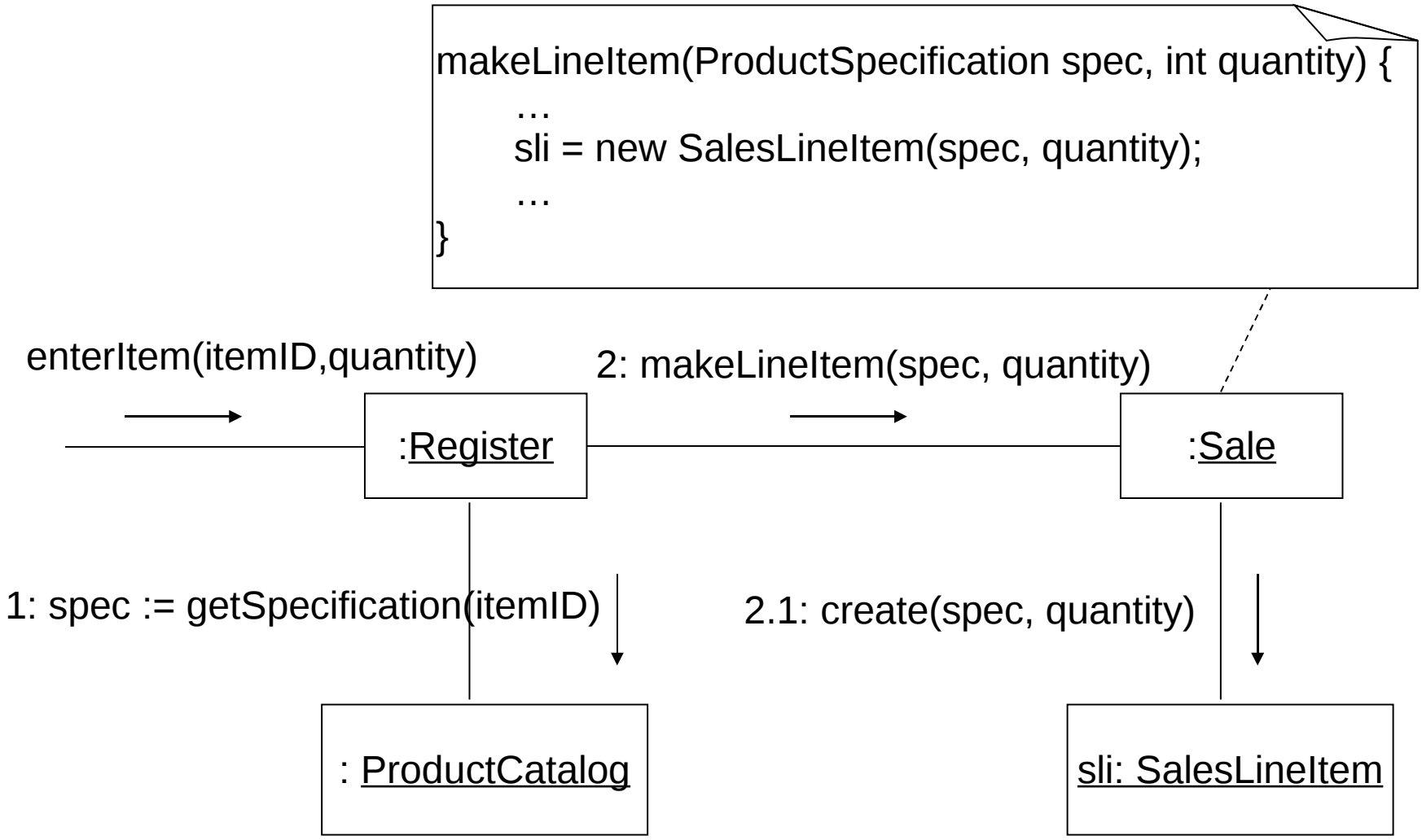
Attribute Visibility



Parameter Visibility

- Parameter visibility from A to B exists when B is passed as a parameter to a method of A.
- This is a relatively temporary visibility, because it persists only within the scope of the method.
- When the makeLineItem message is sent to a Sale instance, a ProductSpecification instance is passed as a parameter.

Parameter Visibility



Parameter Visibility

```
// constructor
SalesLineItem(ProductSpecification spec,
                    int quantity) {
      …
      // parameter to attribute visibility
      productSpec = spec;
      …
}
```

- When Sale creates a new SalesLineItem, it passes a ProductSpecification to its constructor.
- We can assign ProductSpecification to an attribute in the constructor, thus transforming parameter visibility to attribute visibility.

Local Visibility

- Locally declared visibility from A to B exists when B is declared as a local object within a method of A.
- This is a relatively temporary visibility because it persists only within the scope of the method. It can be achieved as follows:
  1. Create a new local instance and assign it to a local variable.
  2. Assign the return object from a method invocation to a local variable.

```
enterItem(itemID, quantity) {
        …
        ProductSpecification spec = catalog.getSpecification(itemID);
        ...
}
```

Global Visibility

- Global visibility from A to B exists when B is global to A.
- This is a relatively permanent visibility because it persists as long as A and B exist.
- One way to achieve this is to assign an instance to a global variable (possible in C++ but not in Java).

Design Model: Creating Design Class Diagrams

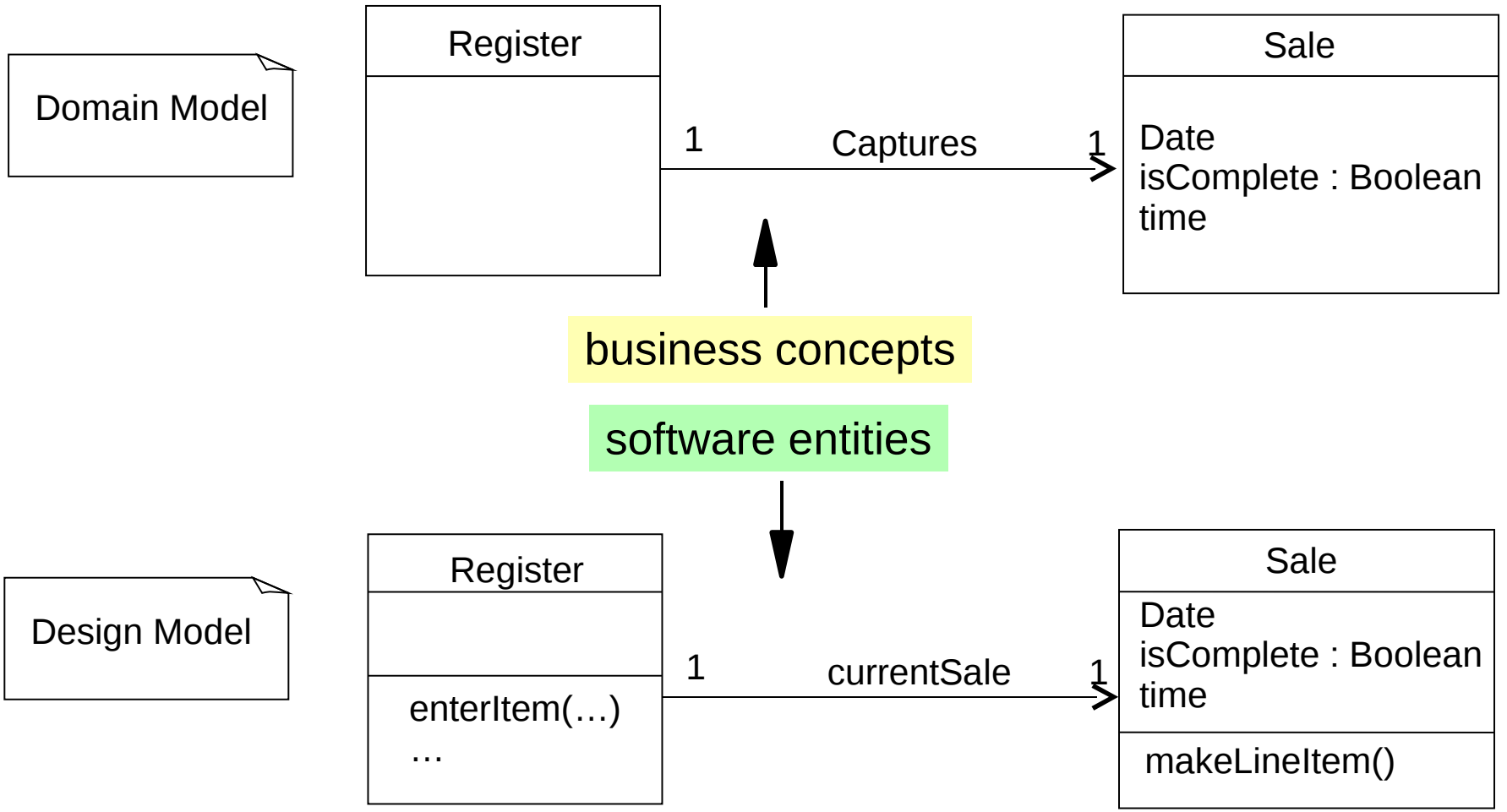
When to create DCDs

- Once the interaction diagrams have been completed it is possible to identify the specification for the software classes and interfaces.
- A class diagram differs from a Domain Model by showing software entities rather than real-world concepts. In a sense, the Domain Model and DCD are two different views of the same thing, but only in a sense.
- The UML has notation to define design details in static structure, or class diagrams.

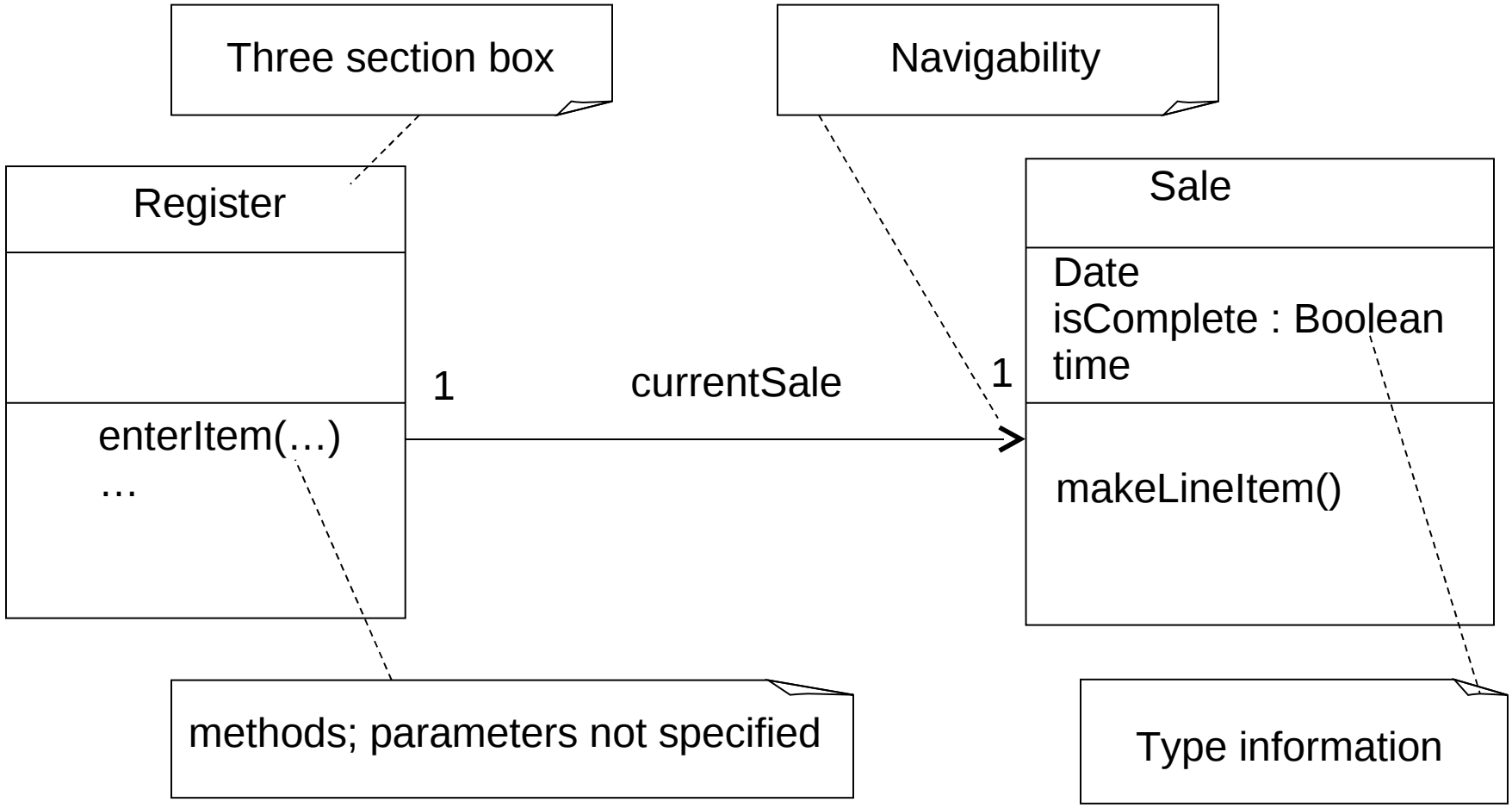
DCD and UP Terminology

- Typical information in a DCD includes:
  - classes, associations and attributes
  - interfaces (with operations and constants)
  - methods
  - attribute type information
  - navigability
  - dependencies
- The DCD depends upon the Domain Model and interaction diagrams.
- The UP defines a Design Model which includes interaction and class diagrams.

Domain Model vs. Design Model Classes

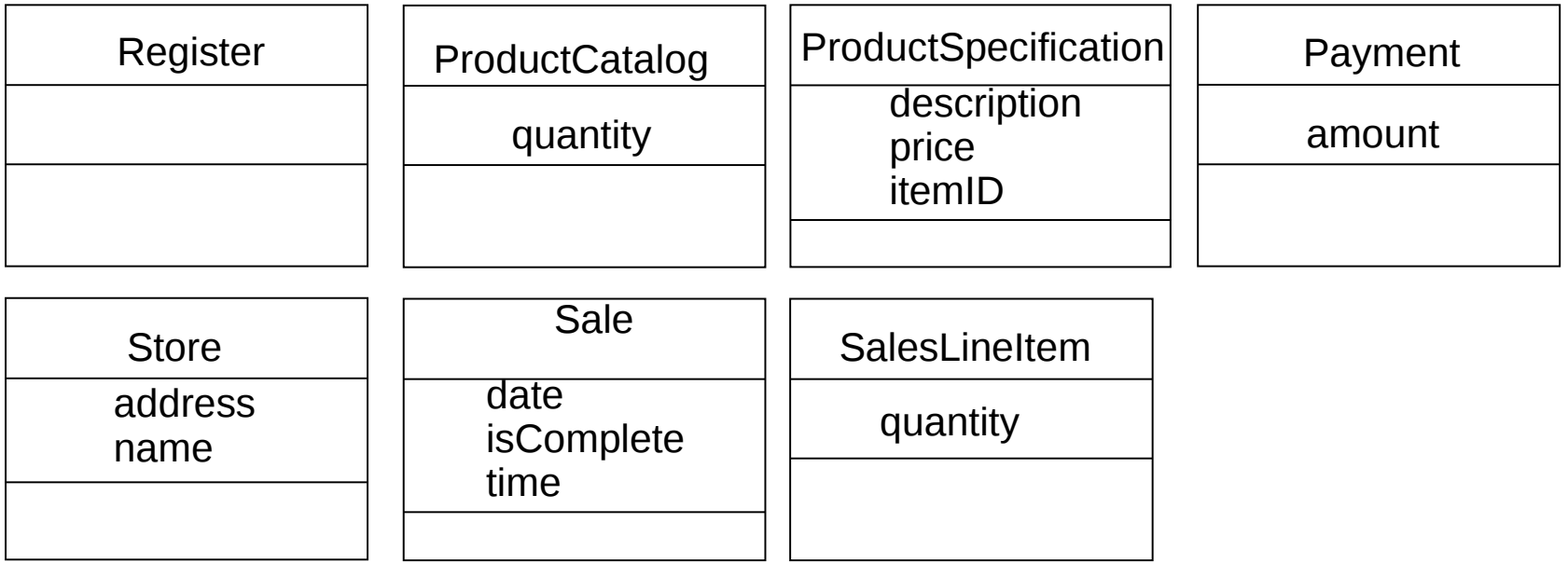


An Example DCD



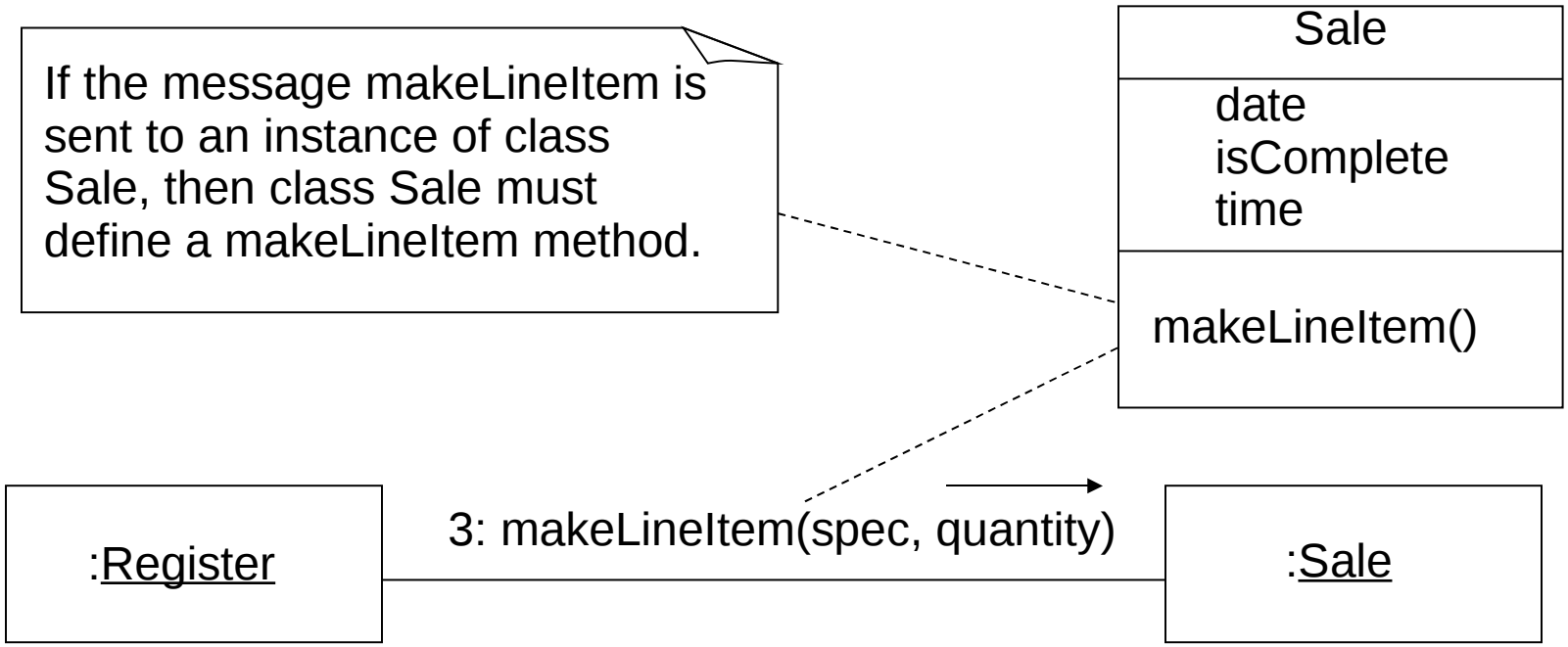
Creating a NextGen POS DCD

- Identify all the classes participating in the software solution. Do this by analyzing the interaction diagrams. Draw them in a class diagram.
- Duplicate the attributes from the associated concepts in the Domain Model.



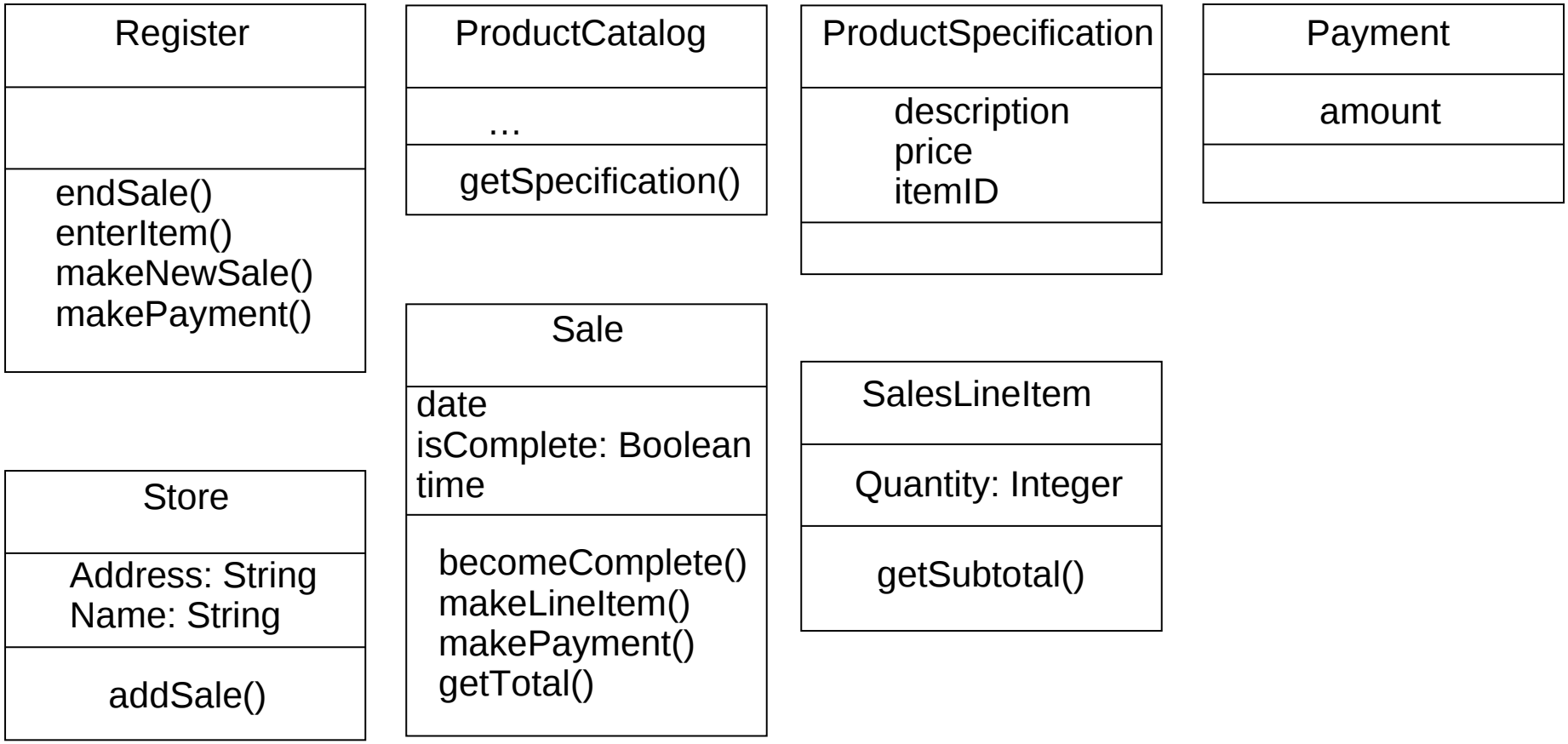
Creating a NextGen POS DCD

- Add method names by analyzing the interaction diagrams.
  - The methods for each class can be identified by analyzing the interaction diagrams.

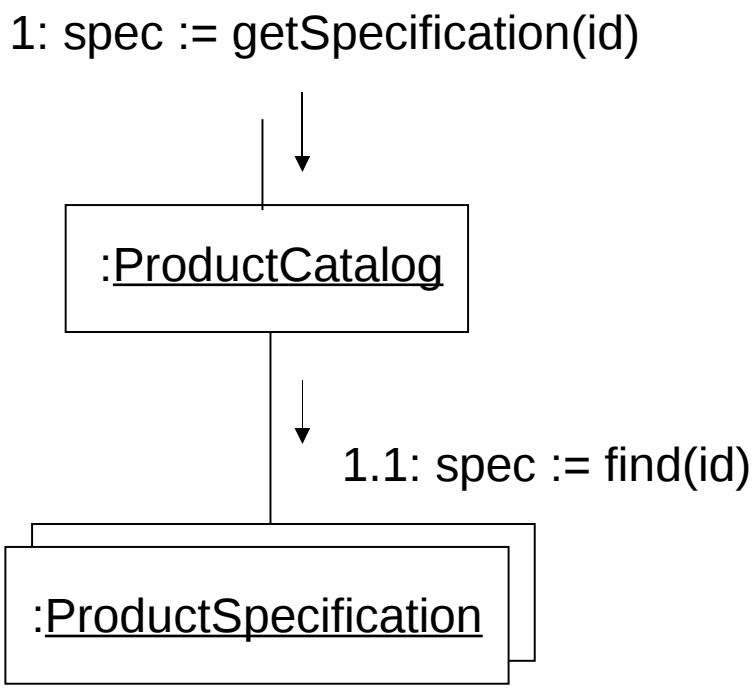


Creating a NextGen POS DCD

- Add type information to the attributes and methods.



Method Names - Multiobjects

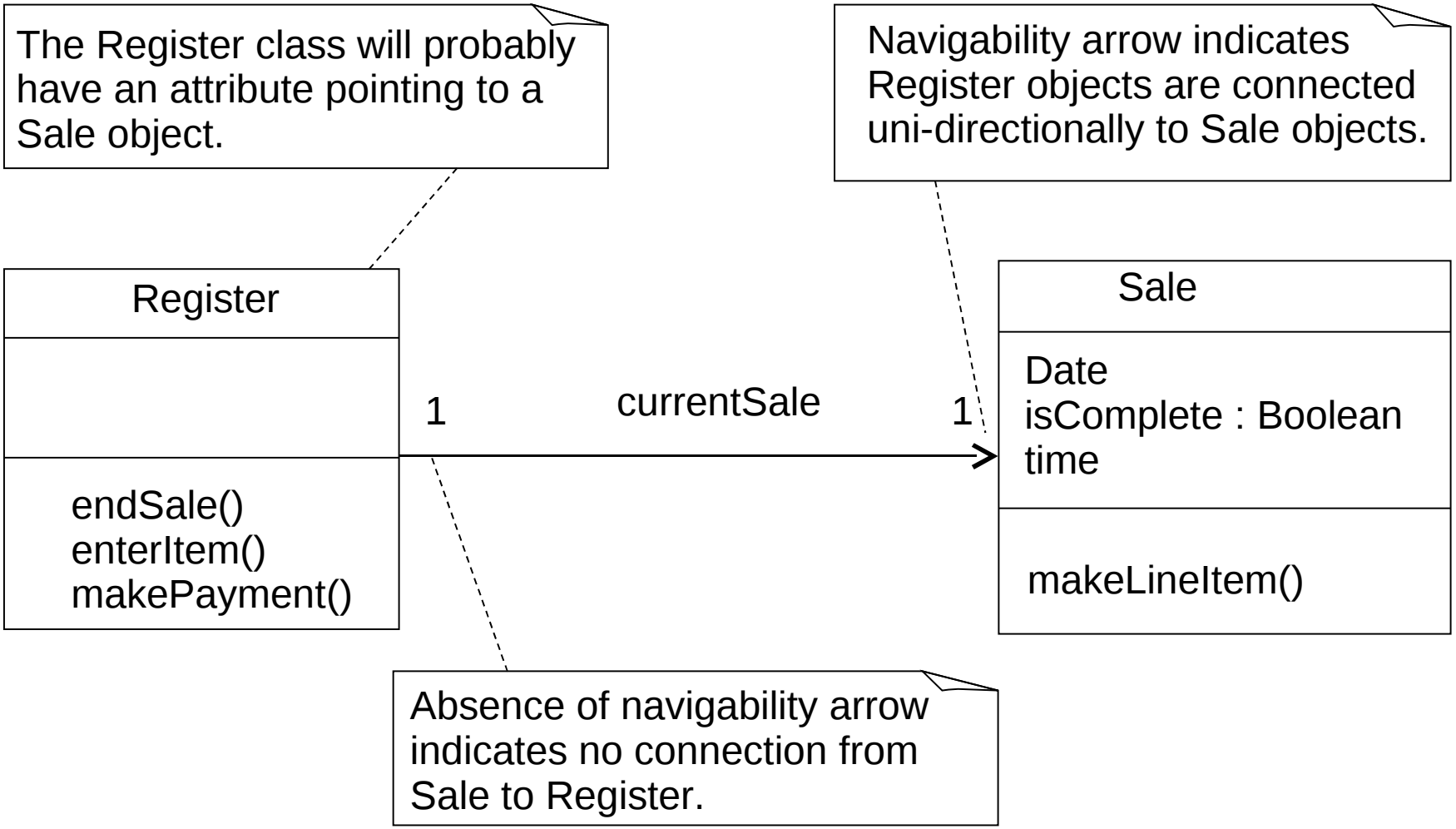


- The find message to the multiobject should be interpreted as a message to the container/ collection object.
- The find method is not part of the ProductSpecification class.

Associations, Navigability, and Dependency Relationships

- Add the associations necessary to support the required attribute visibility.
  - Each end of an association is called a role.
- Navigability is a property of the role, implying visibility of the source to the target class.
  - Attribute visibility is implied.
  - Add navigability arrows to the associations to indicate the direction of attribute visibility where applicable.
  - Common situations suggesting a need to define an association with navigability from A to B:
    - A sends a message to B.
    - A creates an instance of B.
    - A needs to maintain a connection to B
- Add dependency relationship lines to indicate non-attribute visibility.

Creating a NextGen POS DCD



Adding Navigability and Dependency Relationships