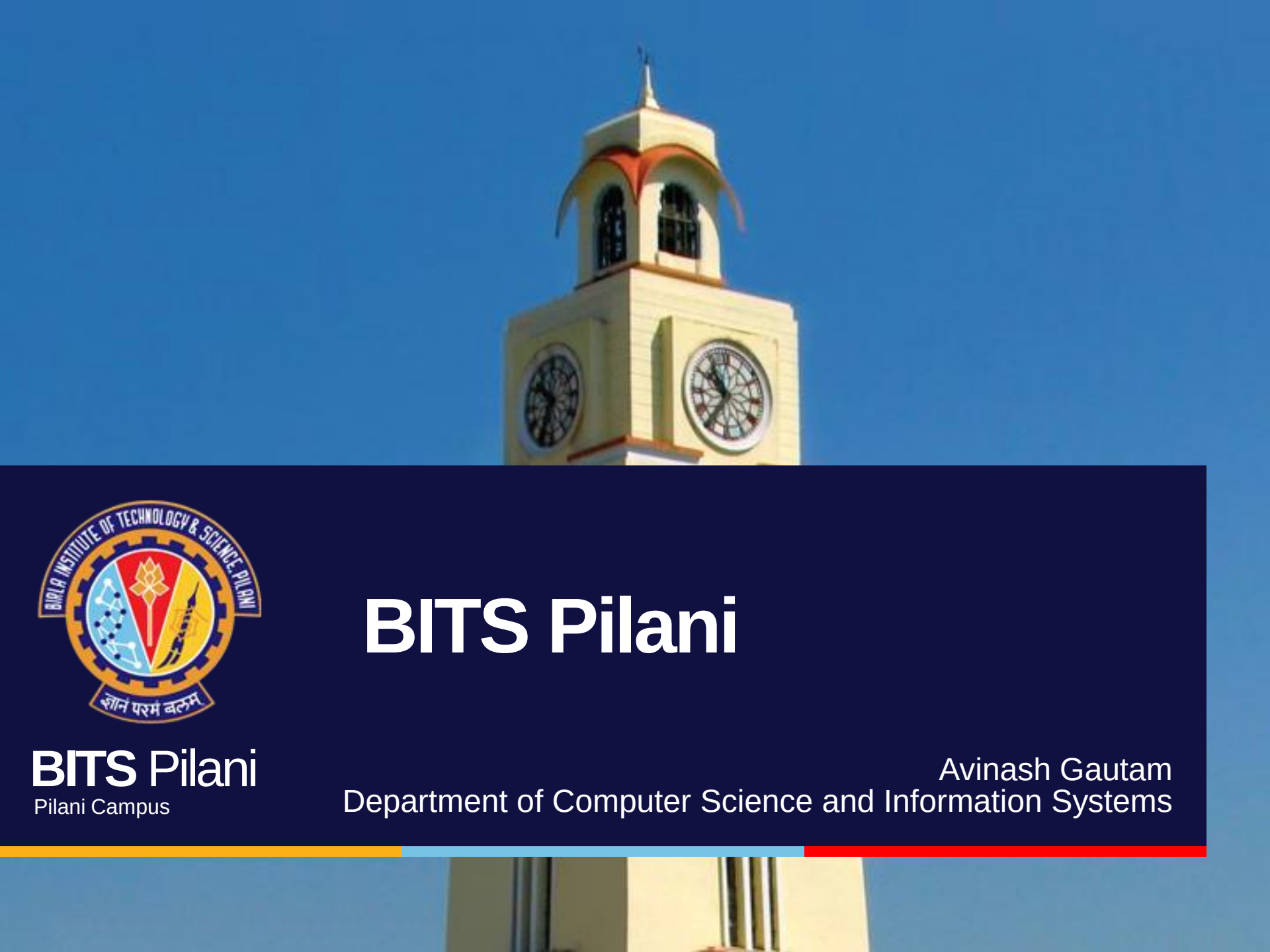


BITS Pilani

BITS Pilani
Pilani Campus

Avinash Gautam
Department of Computer Science and Information Systems

Design Class Diagrams

Chapters 16

Applying UML and Patterns

Craig Larman

Design Class Diagrams (DCDs)

- During analysis, emphasize domain concepts
- During design, shift to software artifacts
- UML has no explicit notation for DCDs
- Uniform UML notation supports smoother development from analysis to design

Domain model vs. Design Class Diagram – differences?

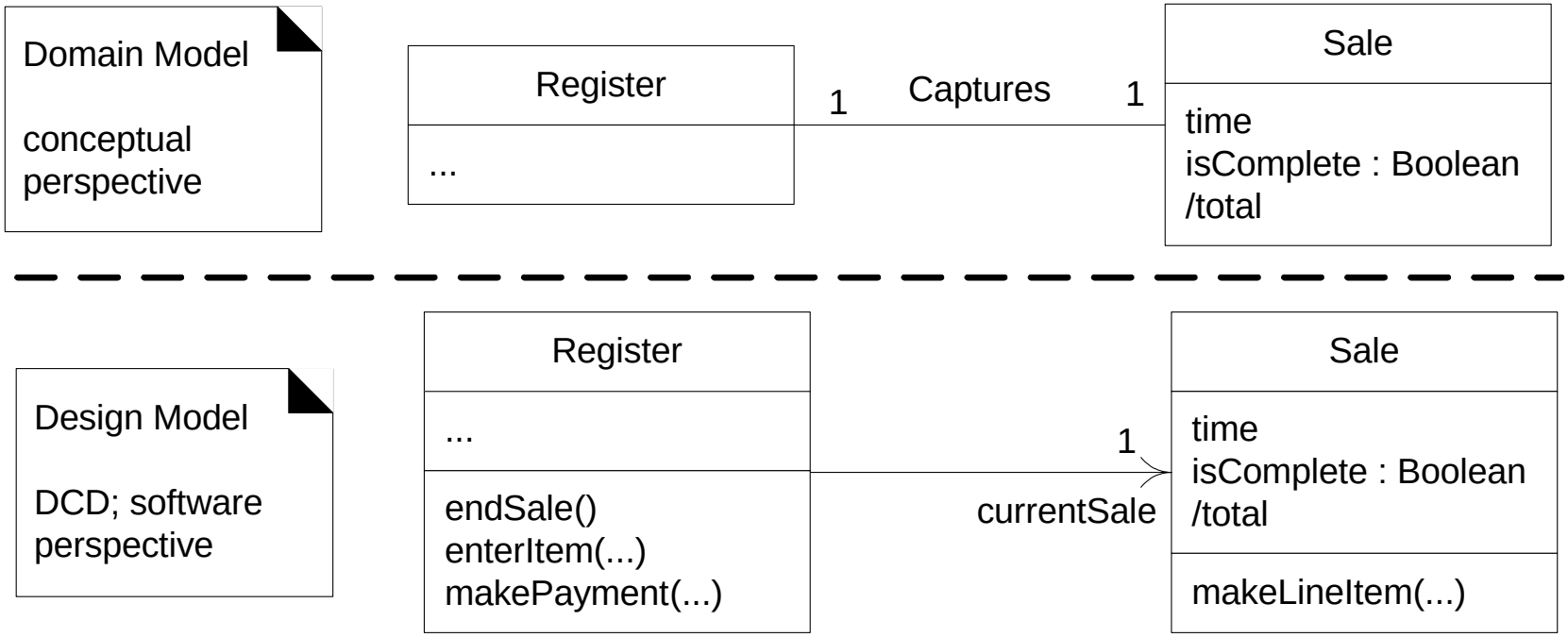


Figure 16.2

Developing a Domain Class Diagram: the NextGen POS DCD

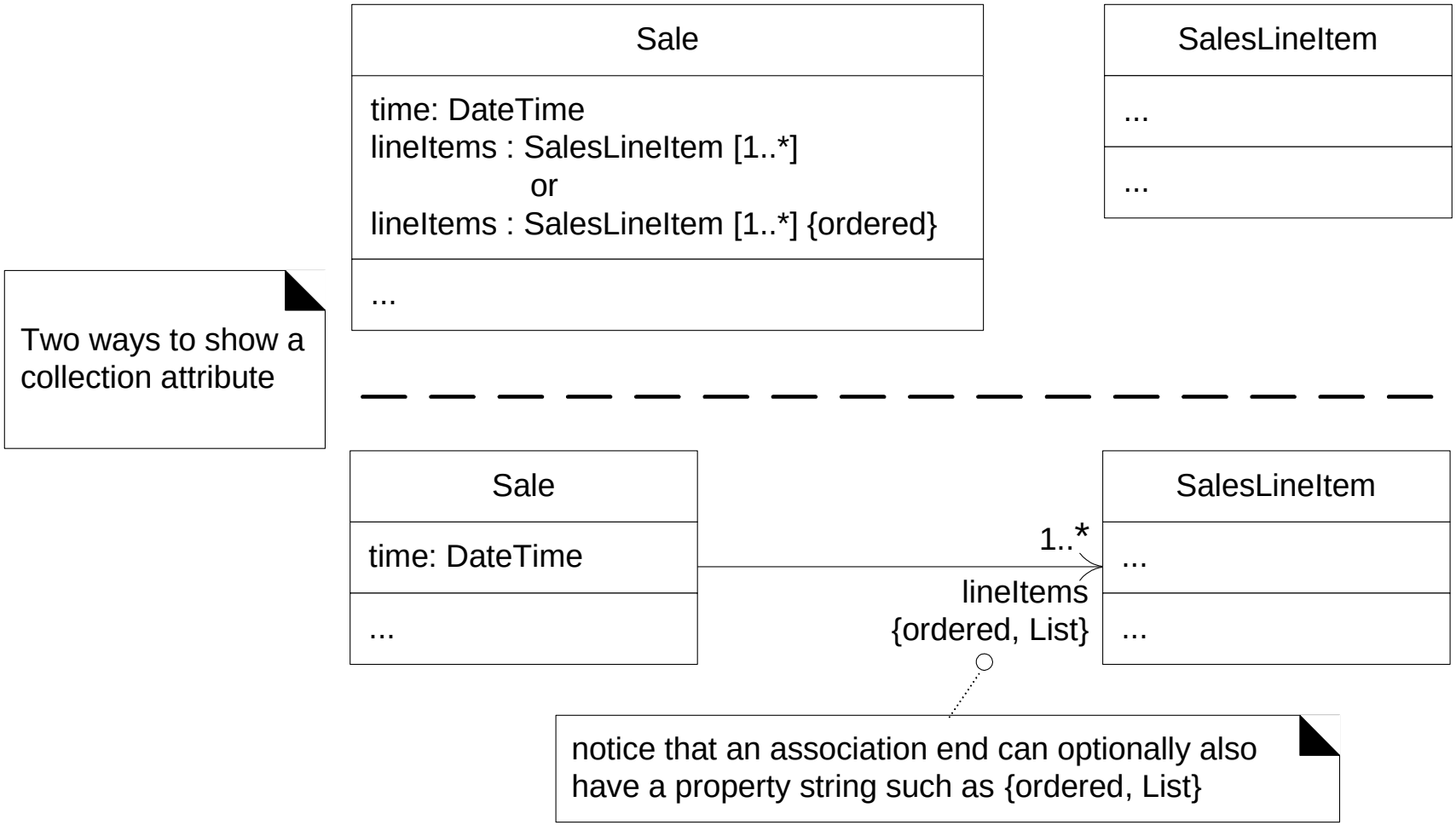
1) Identify **software classes**:

Register	Sale
ProductCatalog	ProductSpecification
Store	SalesLineItem
Payment	

2) Begin drawing a **class diagram**

3) Include the **attributes from** the domain model

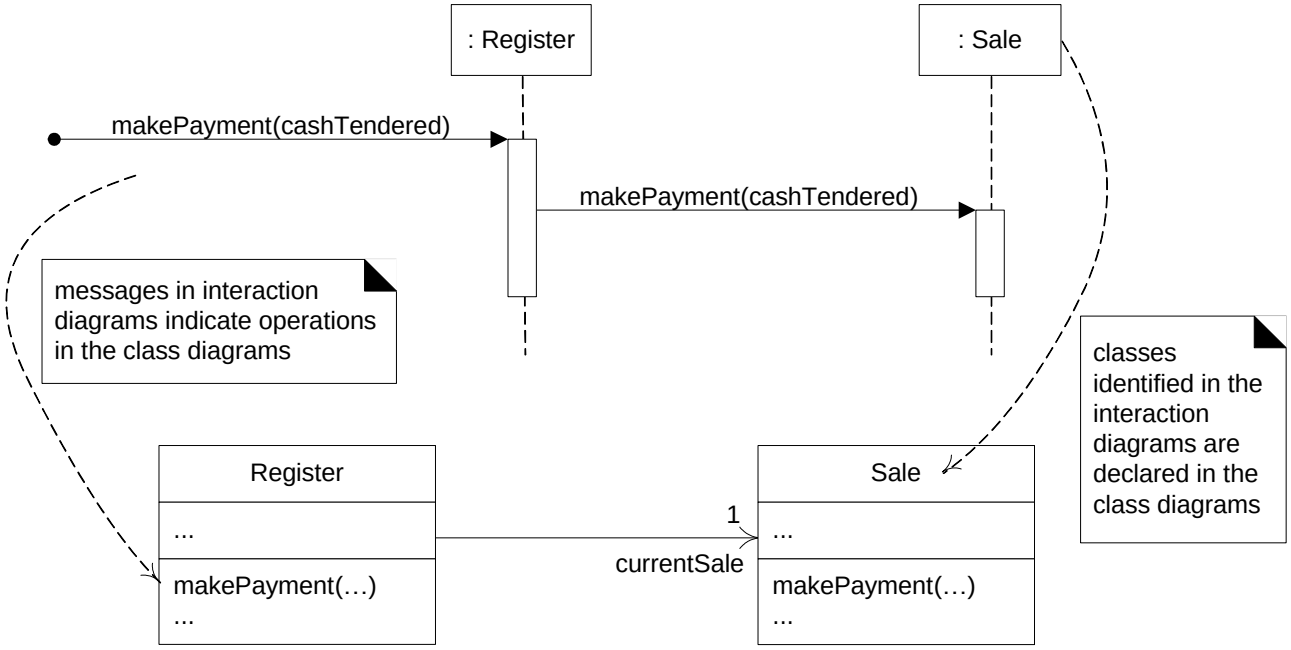
How to show attribute collections?



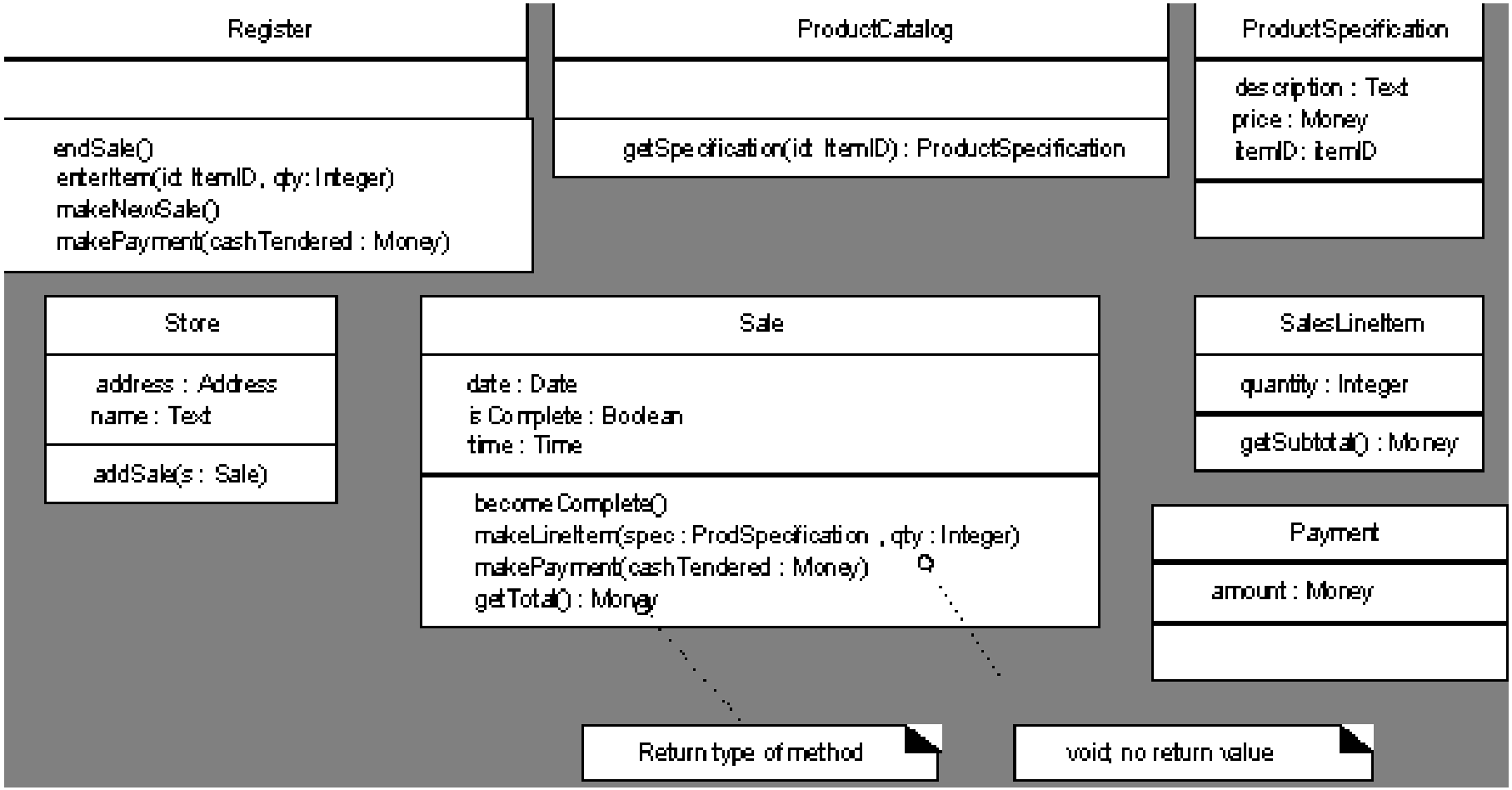
4) Add method names

—from interaction diagrams

—model class & interaction diagrams in parallel

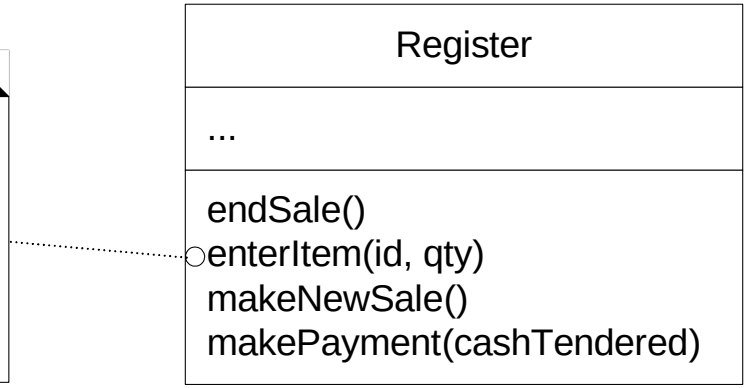


Parameters, return types optional? —readability vs. code generation



Method body pseudo-code also optional

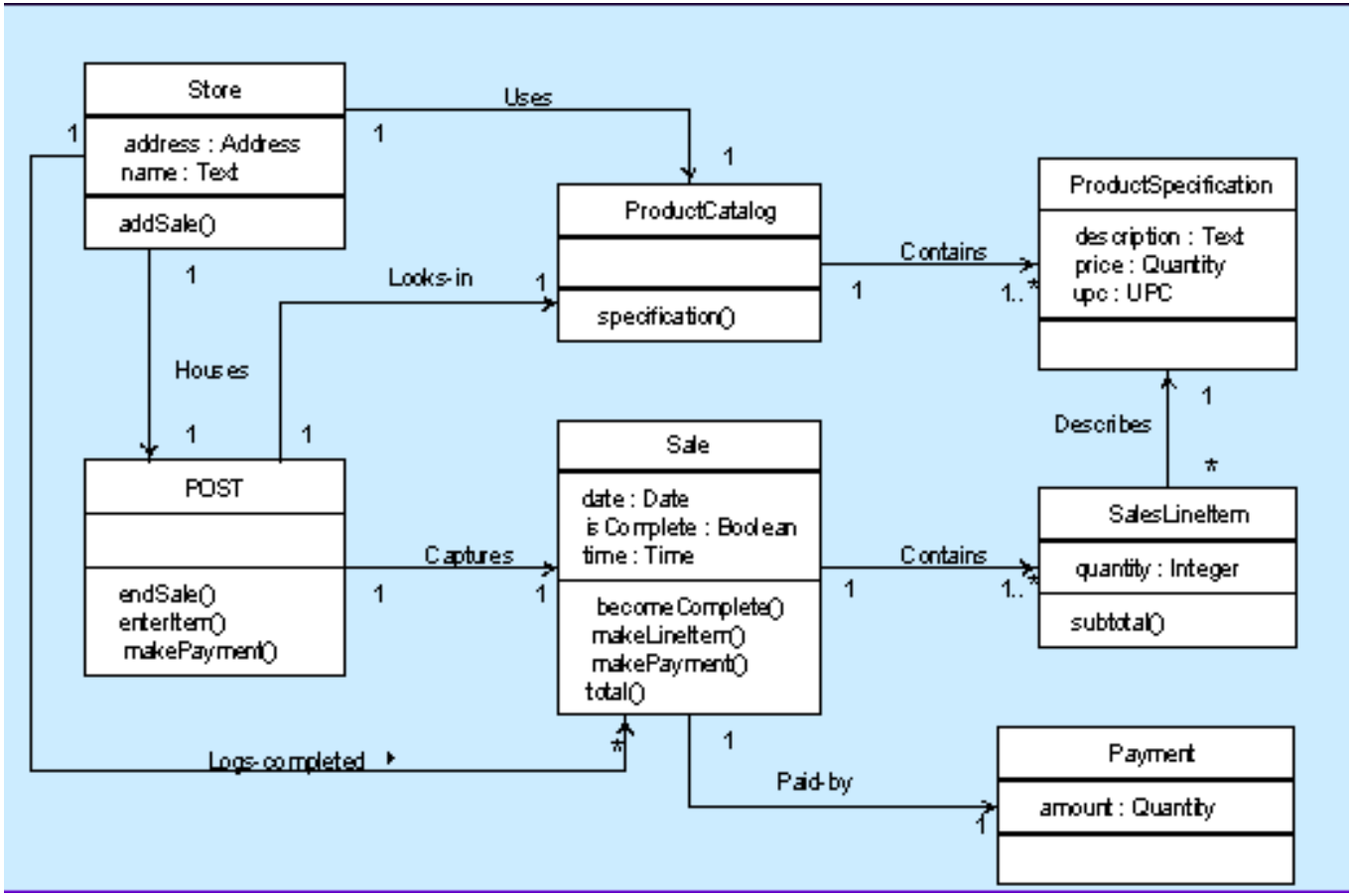
```
«method»
// pseudo-code or a specific language is OK
public void enterItem( id, qty )
{
   ProductDescription desc = catalog.getProductDescription(id);
   sale.makeLineItem(desc, qty);
}
```



5) Add associations and navigability —Navigability implies visibility of attributes

What attribute does ProductCatalog implicitly contain?

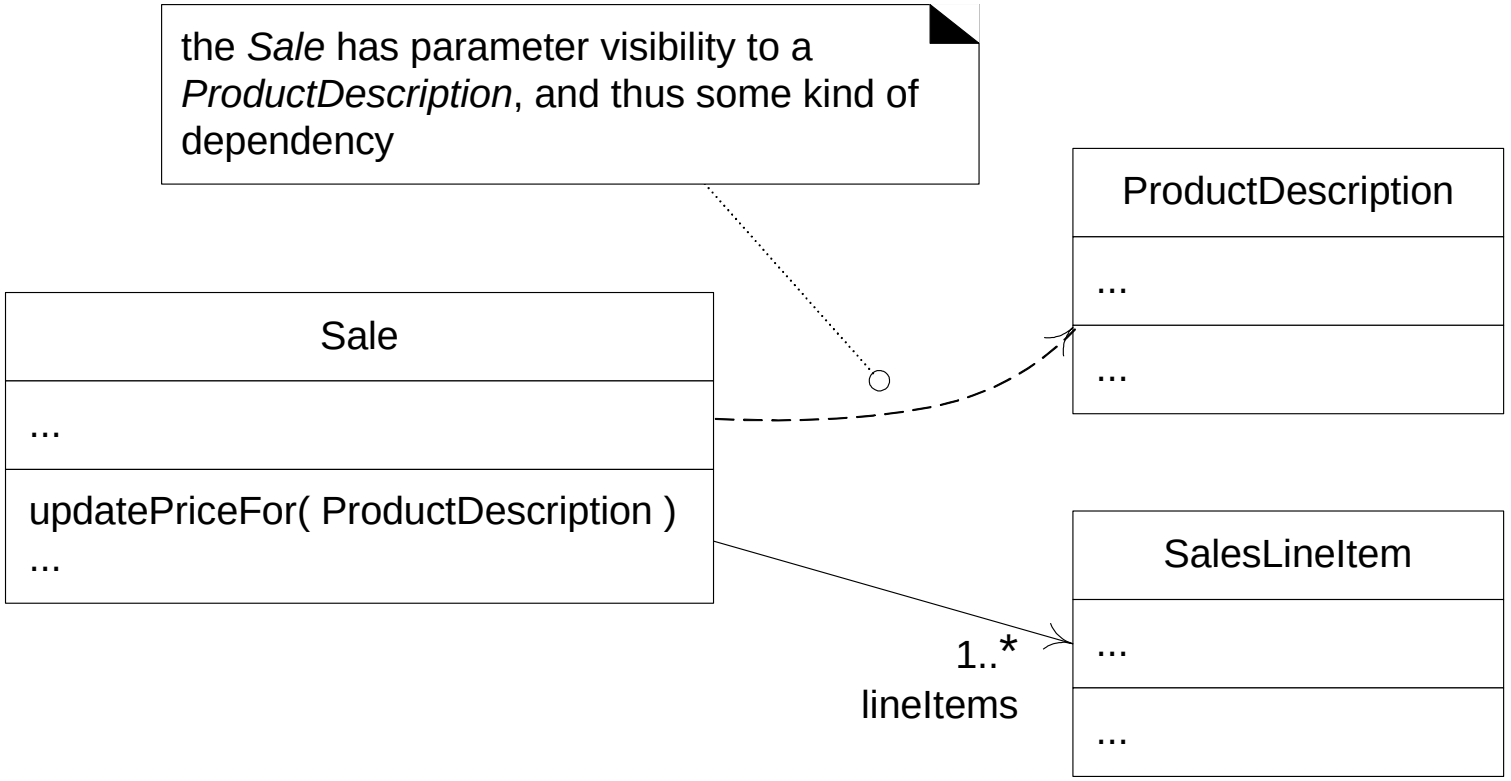
How does navigability clarify this design?



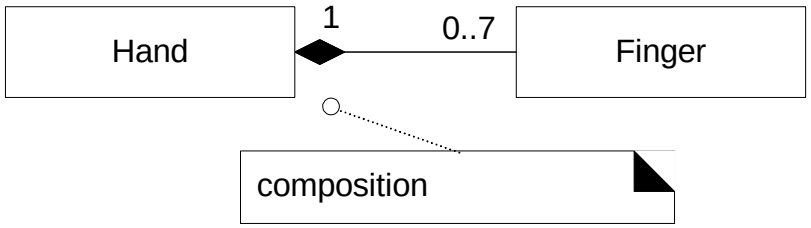
6) Adding dependency relationships

- Indicates that one element has knowledge of another element
- I.e., a change in specification of one thing may affect another thing that uses it, but not necessarily the reverse
- A dashed directed line
- Typically non-attribute visibility between classes

What does dependency add to this DCD?

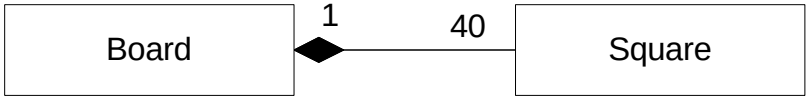


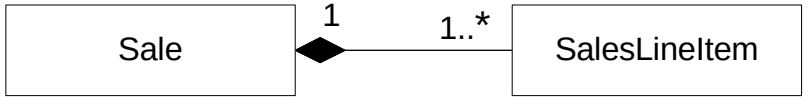
Composition (whole-part) relations



composition means
-a part instance (*Square*) can only be part of one composite (*Board*) at a time

-the composite has sole responsibility for management of its parts, especially creation and deletion





Association classes

—model association with attributes & operations

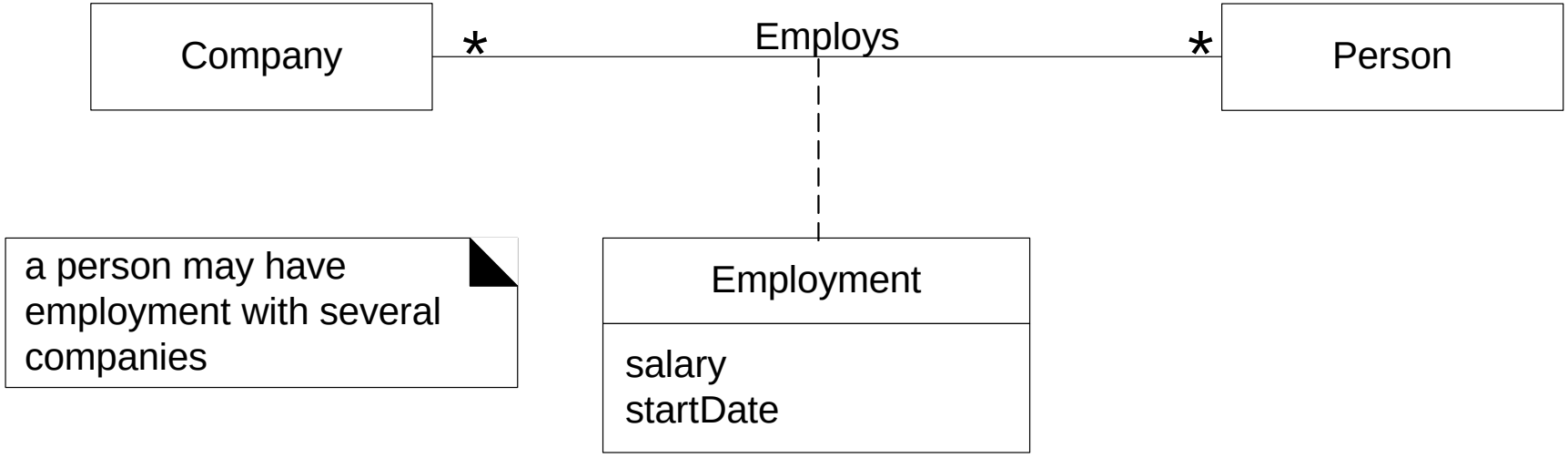
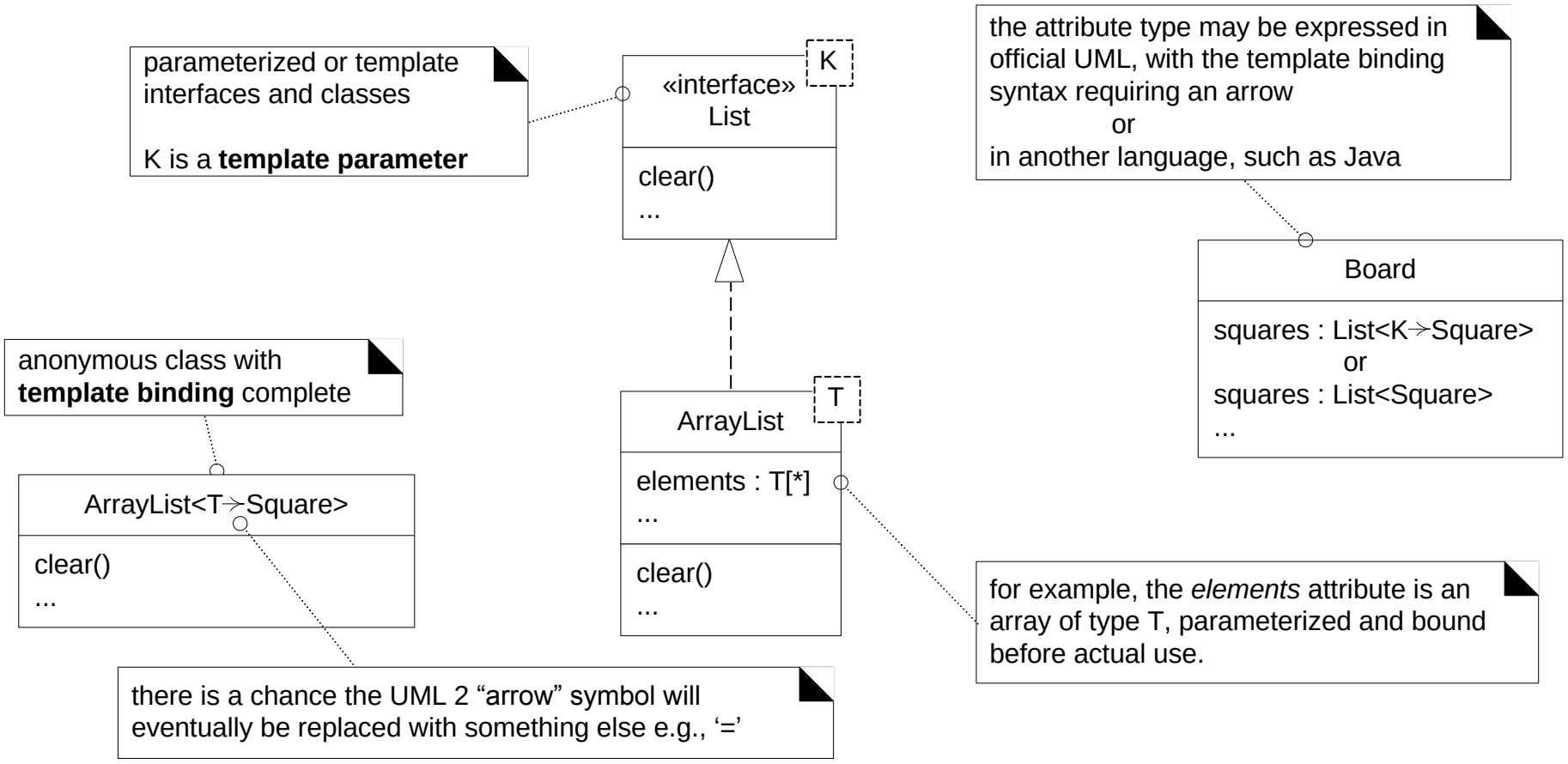


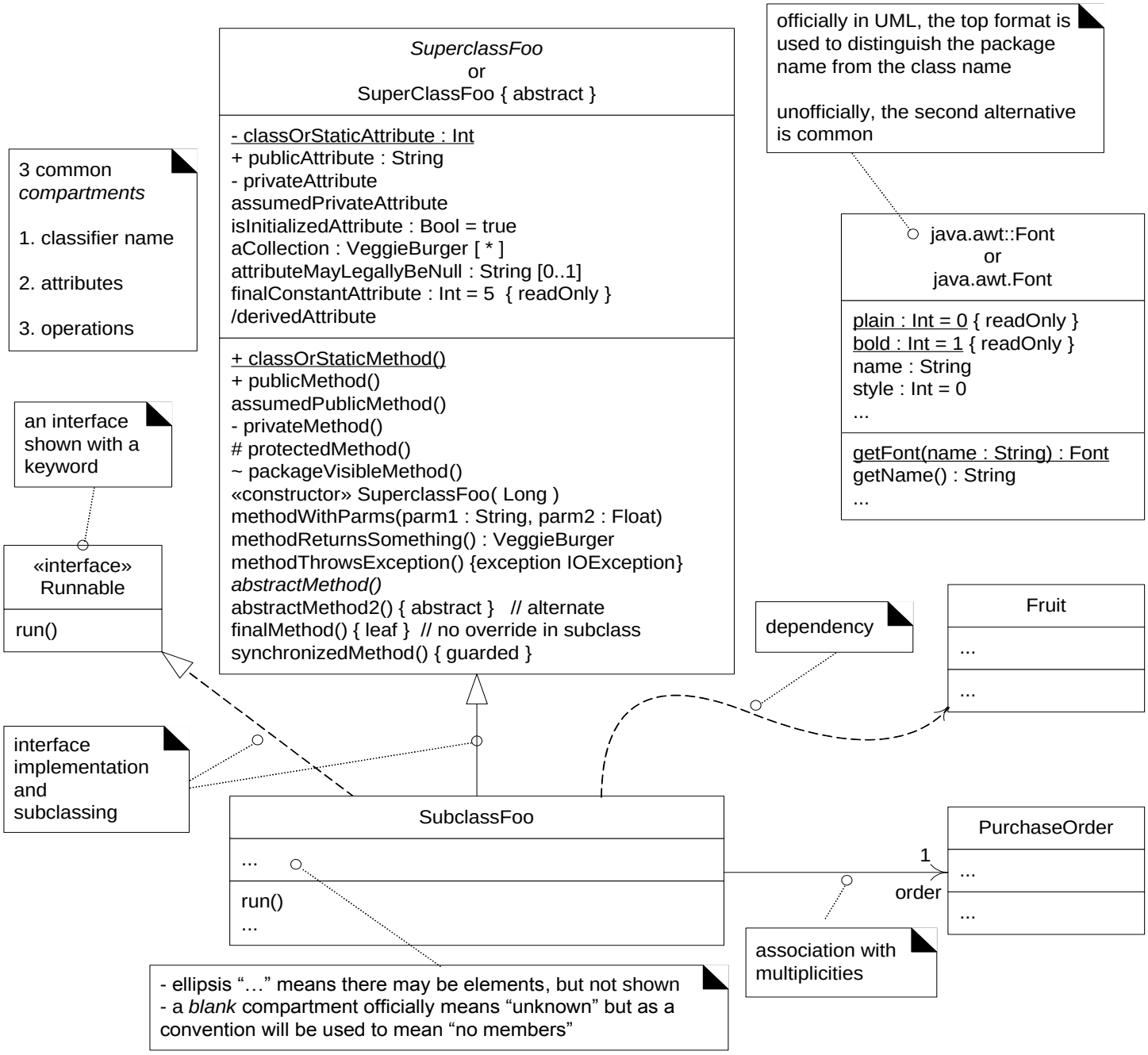
Figure 16.16

Interfaces and Template Classes

—Interface is a predefined «stereotype»

—Templates take parameters in corner




SuperclassFoo
or
SuperClassFoo { abstract }
- classOrStaticAttribute : Int
+ publicAttribute : String
- privateAttribute
assumedPrivateAttribute
isInitializedAttribute : Bool = true
aCollection : VeggieBurger [*]
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }
/derivedAttribute
+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
- privateMethod()
protectedMethod()
~ packageVisibleMethod()
«constructor» SuperclassFoo(Long)
methodWithParms(parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() {exception IOException}
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }
3 common compartments
1. classifier name
2. attributes
3. operations
officially in UML, the top format is used to distinguish the package name from the class name
unofficially, the second alternative is common
java.awt::Font
or
java.awt.Font
plain : Int = 0 { readOnly }
bold : Int = 1 { readOnly }
name : String
style : Int = 0
...
getFont(name : String) : Font
getName() : String
...
an interface shown with a keyword
«interface»
Runnable
run()
dependency
Fruit
...
...
interface implementation and subclassing
SubclassFoo
...
run()
...
PurchaseOrder
...
...
1
order
association with multiplicities
- ellipsis "…" means there may be elements, but not shown
- a blank compartment officially means "unknown" but as a convention will be used to mean "no members"

REVIEW

innovate achieve lead

Use-Case Diagrams

Use cases

- First developed by Ivar Jacobson
  - Now part of the UML (though not necessarily object-oriented)
  - Emphasizes user's point of view
  - Explains everything in the user's language
- A "use *case*" is a set of cases or scenarios for using a system, tied together by a common user goal
  - Essentially descriptive answers to questions that start with "What does the system do if …"
  - E.g., "What does the auto-teller do if a customer has just deposited a check within 24 hours and there's not enough in the account without the check to provide the desired withdrawal?"
  - Use case describes what the auto-teller does in that situation
- Use case model = the set of all use cases
- Why are use cases good for brainstorming requirements?

Brief Use Case format

Brief format narrates a story or scenario of use in prose form, e.g.:

Rent Videos. A Customer arrives with videos to rent. The Clerk enters their ID, and each video ID. The System outputs information on each. The Clerk requests the rental report. The System outputs it, which is given to the Customer with their videos.

Fully dressed Use Case (from Fowler & Scott, *UML Distilled*)

Use Case: Buy a Product (Describe user's goal in user's language)

Actors: Customer, System (Why is it a good idea to define actors?)

1. Customer browsers through catalog and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information (address; next-day or 3-day delivery)
4. System presents full pricing information, including shipping
5. Customer fills in credit card information
6. System authorizes purchase
7. System confirms sale immediately
8. System sends confirming email to customer

(Did we get the main scenario right?)

Alternative: *Authorization Failure* (At what step might this happen?)

6a. At step 6, system fails to authorize credit purchase
Allow customer to re-enter credit card information and re-try

Alternative: *Regular customer* (At what step might this happen?)

3a. System displays current shipping information, pricing information, and last four digits of credit card information

3b. Customer may accept or override these defaults
Return to primary scenario at step 6

Scenario, use case and goal

- A *scenario* is a specific sequence of actions and interactions in a use case.
  - One path through the use case
  - E.g., The scenario of buying a product

- A *use case* is a collection of success and failure scenarios describing an actor using a system to support a goal.

Heuristics for writing use case text

- Avoid implementation specific language in use cases, such as IF-THEN-ELSE or GUI elements or specific people or depts
  - Which is better: “The clerk pushes the OK button.” or: “The clerk signifies the transaction is done.”?
  - The latter defers a UI consideration until design.
- Write use cases with the user’s vocabulary, the way a users would describe performing the task
- Use cases never initiate actions; actors do.
  - Actors can be people, computer systems or any external entity that initiate an action.
- Use case interaction produces something of value to an actor
- Create use cases & requirements incrementally and iteratively
  - Start with an outline or high-level description
  - Work from a vision and scope statement
  - Then broaden and deepen, then narrow and prune

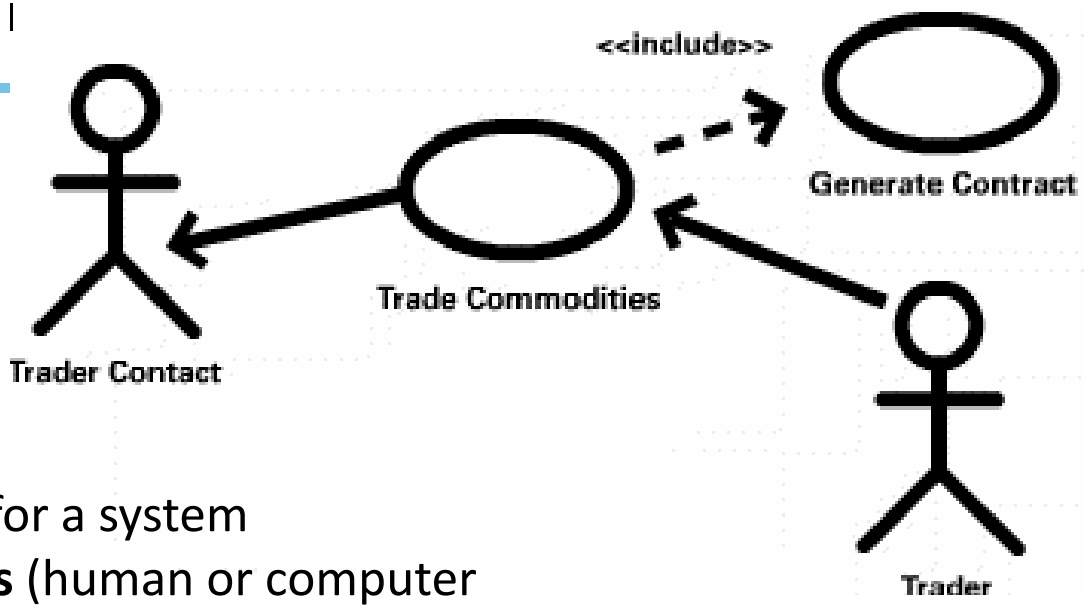
More use case pointers

- Add pre-conditions and post-conditions in each use case:
  - What is the state of affairs before and after use case occurs?
  - Why?
- Some analysts distinguish between business and system use cases:
  - System use cases focus on interaction between actors within a software system
  - Business use cases focuses on how a business interacts with actual customers or events
  - Fowler prefers to focus on business use cases first, then come up with system use cases to satisfy them
  - Note iterative approach in developing use cases, too

Text and Diagrams

- Use case *text* provides the detailed description of a particular use case
- Use case *diagram* provides an overview of interactions between actors and use cases

Use case diagram



- Bird's eye view of use cases for a system
- Stick figures represent **actors** (human or computer in **roles**)
- Ellipses are **use cases** (behavior or functionality seen by users)
- What can user do with the system?
  - E.g., Trader interacts with Trader Contract via a Trade Commodities transaction
- **<<include>>** relationship inserts a chunk of behavior (another use case)
- <<extend>> adds to a more general use case

Advantages of use cases

- What do you think?
- Systematic and intuitive way to capture functional requirements?
- Facilitates communication between user and system analyst:
  - *Text* descriptions explain functional behavior in user's language
  - *Diagrams* can show relationship between use case behaviors
  - When should we bother with diagrams?
- Use cases can drive the whole development process:
  - Analysis understand what user wants with use cases
  - Design and implementation realizes them
  - How can use case help with early design of UI prototype?
  - How can use cases set up test plans?
  - How can use cases help with writing a user manual?

Use cases assignment

- Develop a set of use cases
  and a use case diagram for a simple ATM
  (simple means you can just consider two kinds of accounts, savings and checking, and two transactions, deposits and withdrawals)
- Due anytime Sunday, September 7

  on Blackboard

Requirements Assignments

By Monday, September 8, email me a tentative project title, customer and level of commitment to the project, and other team members their roles.

By Monday, September 15, email me a link to a web site containing the above (perhaps revised), plus:

1. Write a high-level requirements specification
2. Write uses cases describing crucial system behavior
3. Preliminary estimates (person-hours) for project, including rest of analysis design, implementation & testing
4. Assess any risks associated with the project

OO domain modeling with UML class diagrams and CRC cards

What is a Domain Model?

- Illustrates meaningful conceptual classes in problem domain
- Represents real-world concepts, not software components
- Software-oriented class diagrams will be developed later, during design

A Domain Model is Conceptual, not a Software Artifact

Conceptual Class:

Sale
amt item

vs.

Software Artifacts:

SalesDatabase

Sale
Double amt; Item item;
void print()

What's the difference?

Identify conceptual classes from noun phrases

- Vision and Scope, Glossary and Use Cases are good for this type of linguistic analysis

However:

- Words may be ambiguous or synonymous
- Noun phrases may also be attributes or parameters rather than classes:
  - If it stores state information or it has multiple behaviors, then it's a class
  - If it's just a number or a string, then it's probably an attribute

From NPs to classes or attributes

Consider the following problem description, analyzed for Subjects, Verbs, Objects:

The ATM verifies whether the customer's card number and PIN are correct.
S**C** V O**R** O **A** **A**
If it is, then the customer can check the account balance, deposit cash, and withdraw cash.
S **R** V O **A** V **A** V **A**
Checking the balance simply displays the account balance.
M **A** V O **A**
Depositing asks the customer to enter the amount, then updates the account balance.
M V O**R** V **A** V O **A**
Withdraw cash asks the customer for the amount to withdraw; if the account has enough cash,
M **A** V O**R** **A** V O V **A**
the account balance is updated. The ATM prints the customer's account balance on a receipt.
O **A** V S V **C** O **A** O

Analyze each **subject** and **object** as follows:

- Does it represent a person performing an action? Then it's an actor, '**R**'.
- Is it also a verb (such as 'deposit')? Then it may be a method, '**M**'.
- Is it a simple value, such as 'color' (string) or 'money' (number)? Then it is probably an attribute, '**A**'.
- Which NPs are unmarked? Make it '**C**' for class.

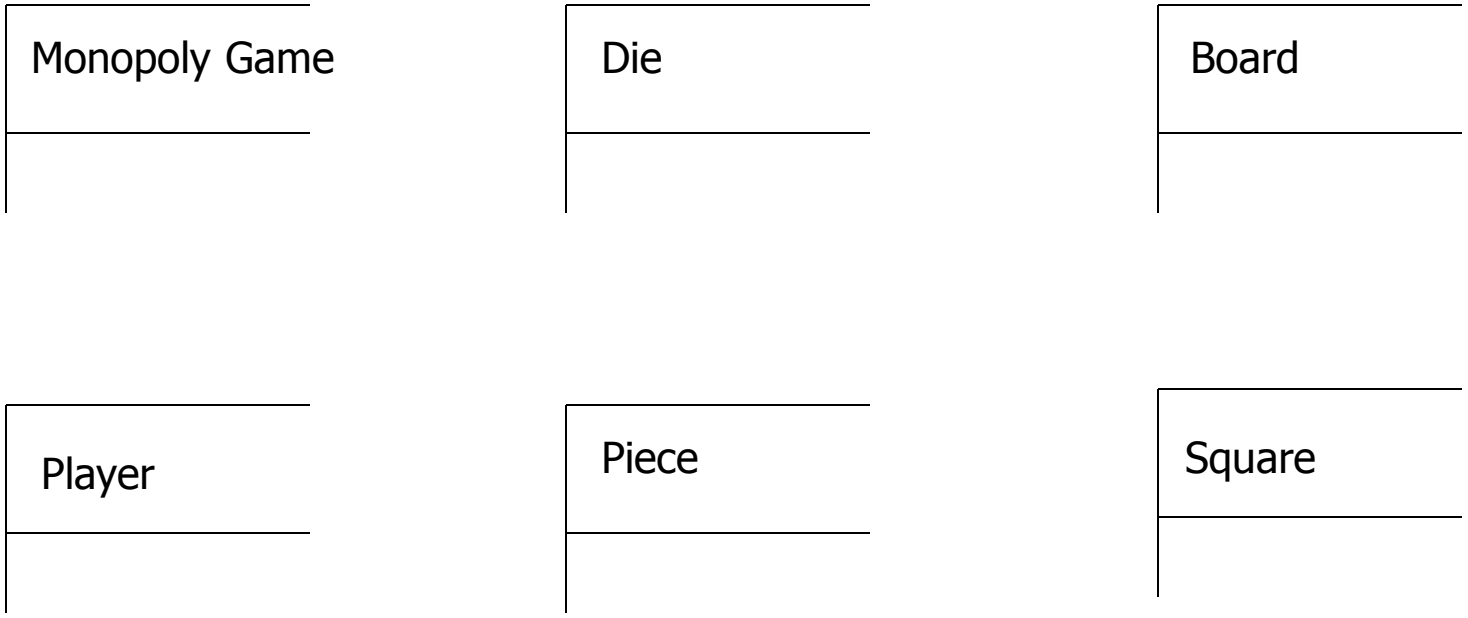
Verbs can also be classes, for example:

- Deposit is a class if it retains state information

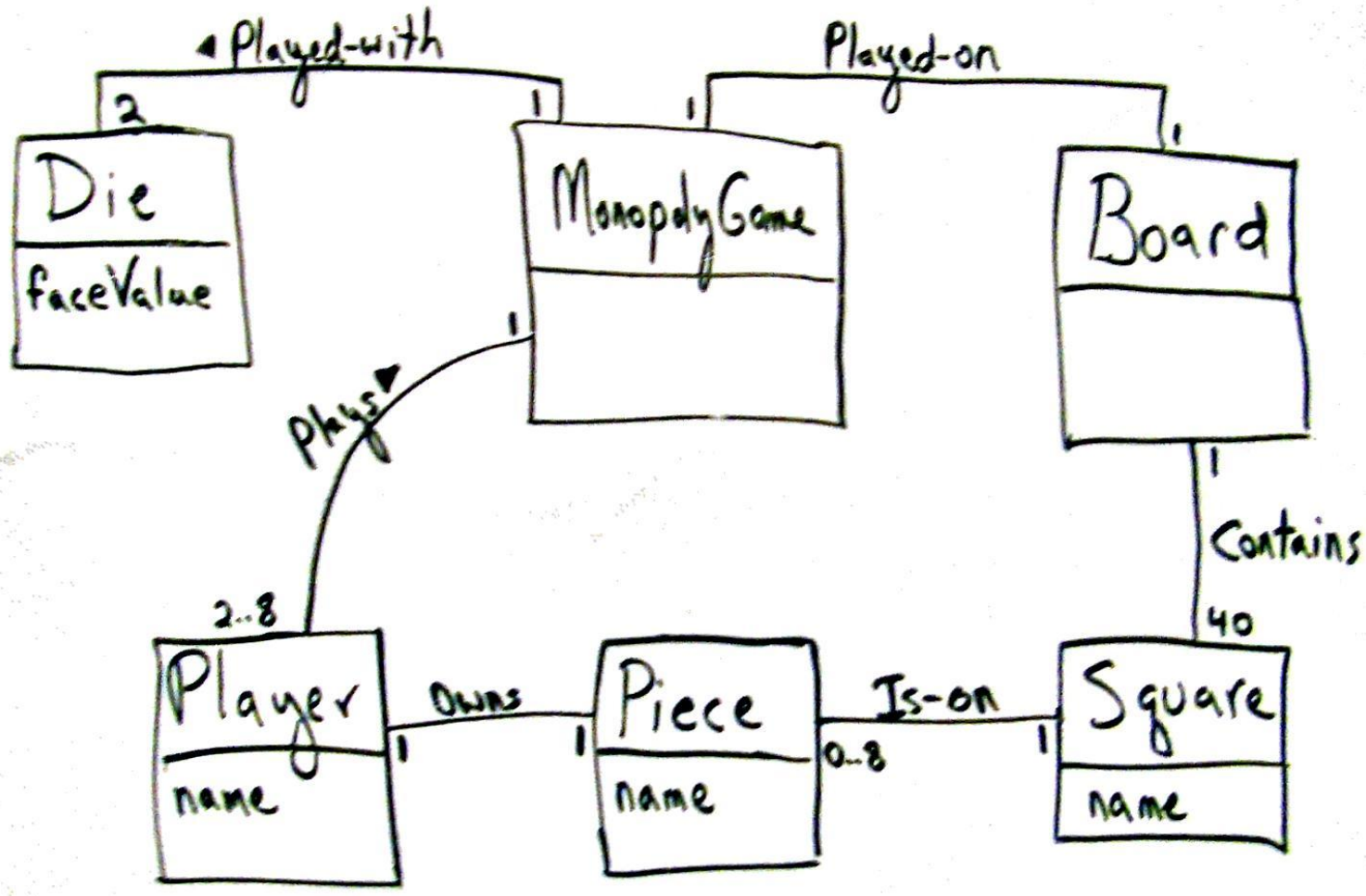
Steps to create a Domain Model

- Identify candidate conceptual classes
- Draw them in a UML domain model
- Add associations necessary to record the relationships that must be retained
- Add attributes necessary for information to be preserved
- Use existing names for things, the vocabulary of the domain

Monopoly Game domain model (first identify concepts as classes)



Monopoly Game domain model
Larman, Figure 9.28



Discovering the Domain Model with CRC cards

- Ordinary index cards
  - Each card represents a **class** of objects.
- Each card has three components
  - Name, Responsibilities, Collaborators

Class Name	
Responsibilities	Collaborators
. . .	. . .

Figure 15.1: Class-Responsibility-Collaborators (CRC) card

Responsibilities

- Key idea: objects have responsibilities
  - As if they were simple agents (or actors in scenarios)
- Anthropomorphism of class responsibilities gets away from thinking about classes as just data holders
  - "Object think" focuses on their active behaviors
- Each object is responsible for specific actions
  - Client can expect predictable behaviors
- Responsibility also implies independence:
  - To trust an object to behave as expected is to rely upon its autonomy and modularity
  - Harder to trust objects easily caught up in dependencies caused by global variables and side effects.

Class names

- **Class Name** creates the vocabulary of our analysis
  - Use nouns as class names, think of them as simple agents
  - Verbs can also be made into nouns, if they are maintain state
  - E.g., “reads card” suggests **CardReader**, managing bank cards
- Use pronounceable names:
  - If you cannot read aloud, it is not a good name
- Use capitalization to initialize Class names and demarcate multi-word names
  - E.g., CardReader rather than CARDREADER or card_reader
  - Why do most OO developers prefer this convention?
- Avoid obscure, ambiguous abbreviations
  - E.g., is TermProcess something that terminates or something that runs on a terminal?
- Try *not* to use digits within a name, such as CardReader2
  - Better for instances than classes of objects

Responsibilities section

- Describes a class's **behaviors**
  - Describe *what* is to be done, not *how*!
- Use short verb phrases
  - E.g.: "reads card" or "look up words"
- How do constraints of index cards guide class analysis?
  - A good measure of appropriate complexity
  - If you cannot fit enough tasks on a card, maybe you need to divide tasks between classes, on different cards?

Collaborators

- Lists important **suppliers** and possibly clients of a class
- Why are classes that supply services more important here?
  - Suppliers are necessary for the description of responsibilities
- As you write down responsibilities for a class, add any suppliers needed for them
  - For example “read dictionary” obviously implies that a “dictionary” as a collaborator
- Developing CRC cards is first a process of discovering classes and their responsibilities
  - People naturally perceive the world as categories of objects
  - In object-oriented analysis, one discovers new categories relevant to a problem domain

Class diagrams in UML

- Classes are boxes, lines are associations
  - Add decorations to lines to refine them
  - But before we study the decorations….

A heuristic for class diagram design

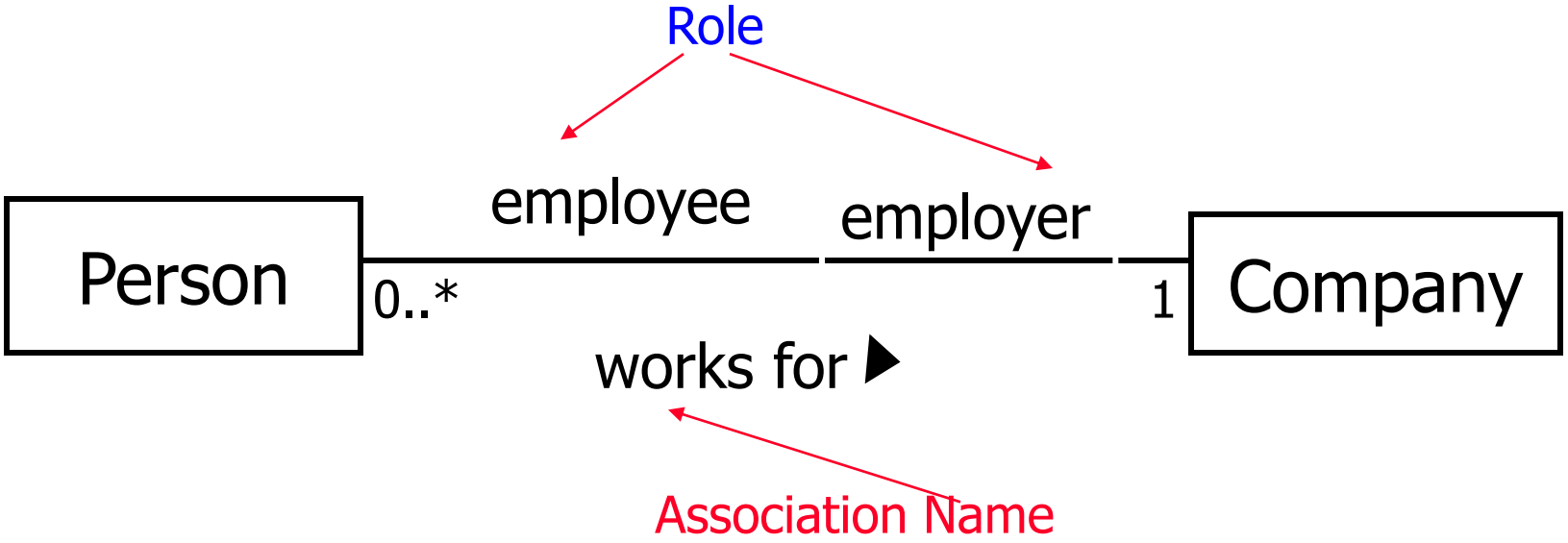
- *Don't put any decorations on the associations at first*
- Semantics of relations tends to be *vague*
  - “Is a" relationship can mean SUBTYPE ("a square is a polygon")
  - or INSTANCE-OF ("George is a square")
  - or IDENTICAL-TO ("The morning star is the evening star")
  - or PROPERTY-OF ("A circle is a round object")
  - or ROLE-OF ("George is a President")
  - or MADE-OF ("My house is a brick one")
  - or simply EXISTS (“To be or not to be…").
  - In many languages, there is no verb "is" at all, or it’s rarely used.
- Let the meaning of relations *emerge* from what they relate
  - Vagueness is natural: start off vague, get more specific gradually
- UML supports this heuristic by starting with simple undirected lines (associations)
- Later, add detail to your relationship structures

Two basic relationships of O-O

- OOA typically distinguishes two relations: **is-a** and **has-a**
- That's what O-O programming languages implement:
  - Smalltalk and Java: instance variables and inheritance
  - C++: data members and class derivation
- Meyer calls these **inheritance** and **client/supplier**
- Booch: **generalization/specialization** and **association** or **aggregation** or **composition**
- Coad and Yourdon: **gen/spec** and **whole/part**
- UML calls these **generalization** and **association** or **aggregation** or **composition**

Associations

- A link between two classes (“has a”)
  - Typically modeled as a member reference
  - Notation from Extended Entity Relation (EER) models

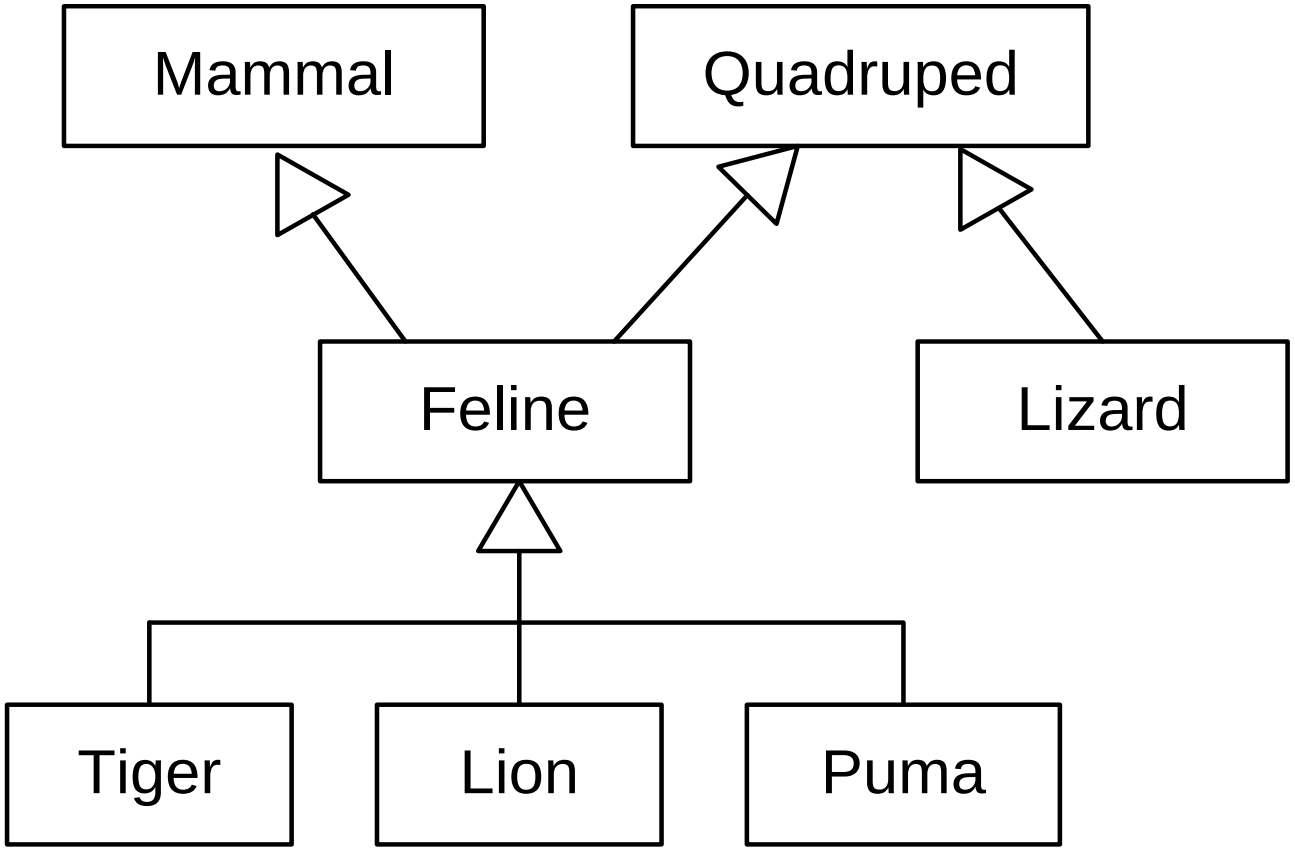


- Role names and multiplicity at association ends
- Direction arrow to aid reading of association name

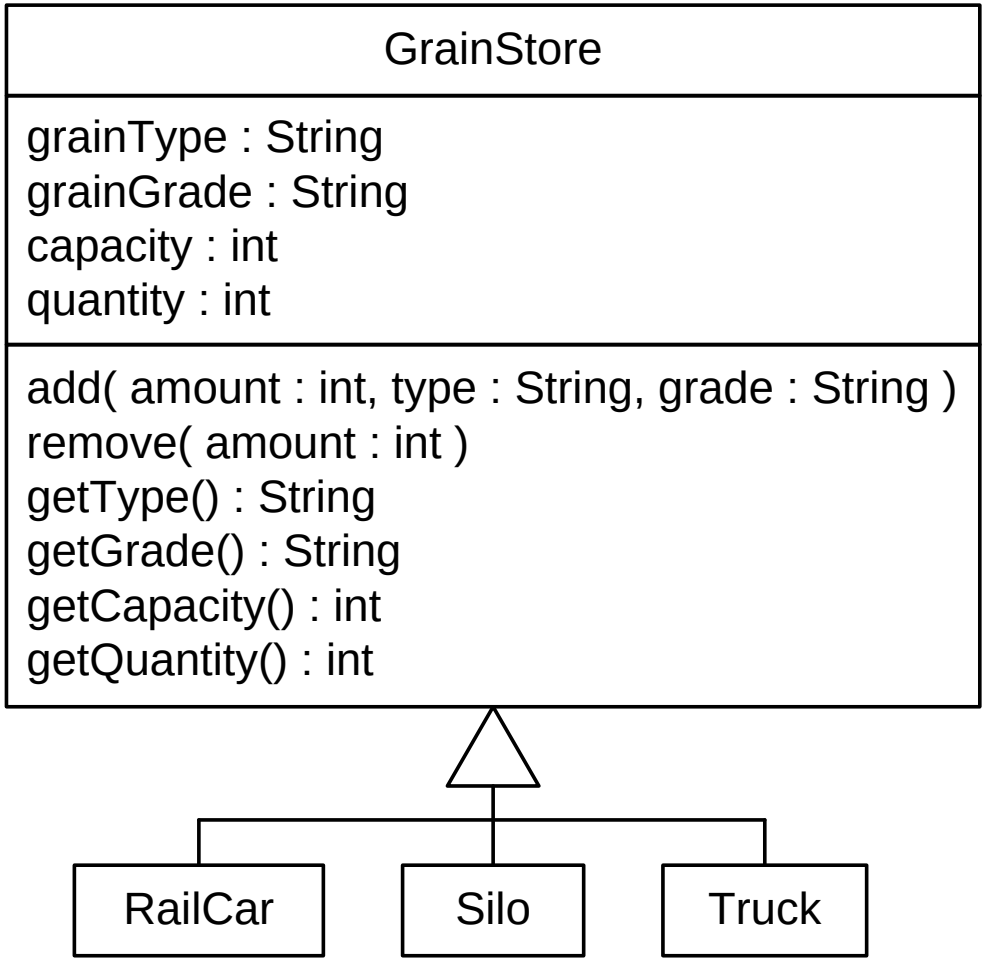
Generalization / Specialization

- Classes may be subclasses of other classes
  - "a kind of" or "is a"
  - Cars are a kind of vehicle (Or a car is a vehicle)
  - Chairs are a kind of furniture
  - People are a kind of mammal
- Generalization captures commonality:
  - What do cars have in common with vehicles?
  - Same properties and behaviors
- UML: generalization/specialization relationship
  - Drawn as a line with an arrow head on one end
- Inheritance in OO programming languages

Generalization Example



Advantages of Inheritance?



Five activities of OOA

1) Class-&-object: describe problem domain in terms of classes of objects
2) Structure: describe relationships between classes
3) Subject: organize classes into clusters or packages
4) Attributes: describe data held by objects
5) Services: describe behaviors that objects can perform

System Sequence Diagrams

Based on Craig Larman, Chapter 10

Dynamic behaviors

- Class diagrams represent ***static*** relationships.
- What about modeling ***dynamic*** behavior?
- **Interaction** diagrams model how groups of object collaborate to perform some behavior
  - Typically captures the behavior of a single use case

Use Case: Order Entry
1) An Order Entry window sends a “prepare” message to an Order
2) The Order sends “prepare” to each Order Line on the Order
3) Each Order Line checks the given Stock Item
4) Remove appropriate quantity of Stock Item from stock
5) Create a deliver item

Alternative: Insufficient Stock
3a) if Stock Item falls below reorder level
then Stock Item requests reorder

Sequence diagrams

- Vertical line is called an object's **lifeline**
  - Represents an object's life during interaction
- Object deletion denoted by X, ending a lifeline
  - Horizontal arrow is a message between two objects
- Order of messages sequences top to bottom
- Messages labeled with message name
  - Optionally arguments and control information
- Control information may express conditions:
  - such as [hasStock], or iteration
- Returns (dashed lines) are optional
  - Use them to add clarity

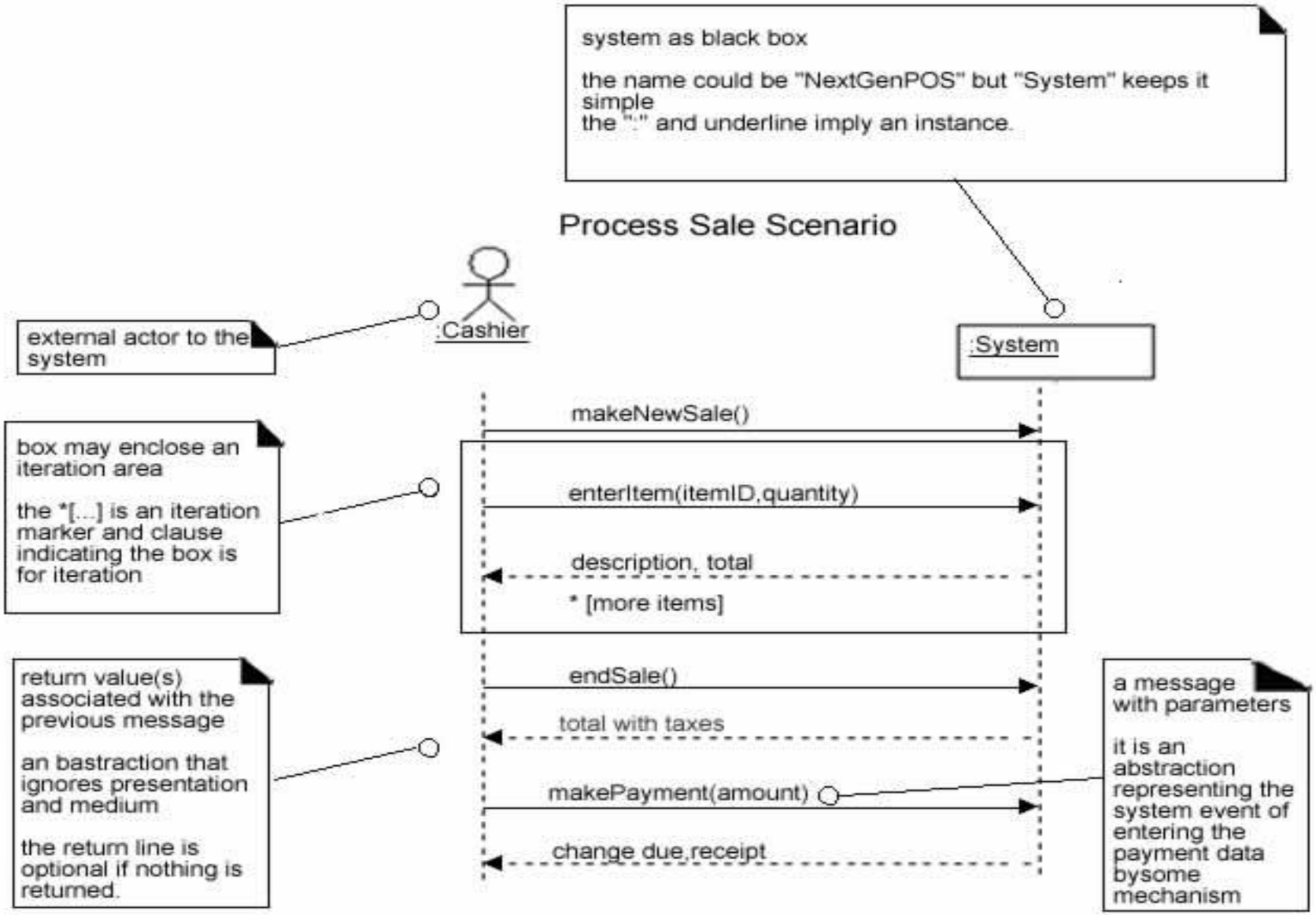
System Sequence Diagram (SSD)

For a use case scenario, an SSD shows:

- The System (as a black box)
- The external actors that interact with System
- The System events that the actors generate
- SSD shows operations of the System in response to events, in temporal order
- Develop SSDs for the main success scenario of a selected use case, then frequent and salient alternative scenarios

SSD for Process Sale scenario

(Larman, page 175)



From Use Case to Sequence System Diagram

How to construct an SSD from a use case:

1. Draw System as black box on right side
2. For each actor that directly operates on the System, draw a stick figure and a lifeline.
3. For each System events that each actor generates in use case, draw a message.
4. Optionally, include use case text to left of diagram.

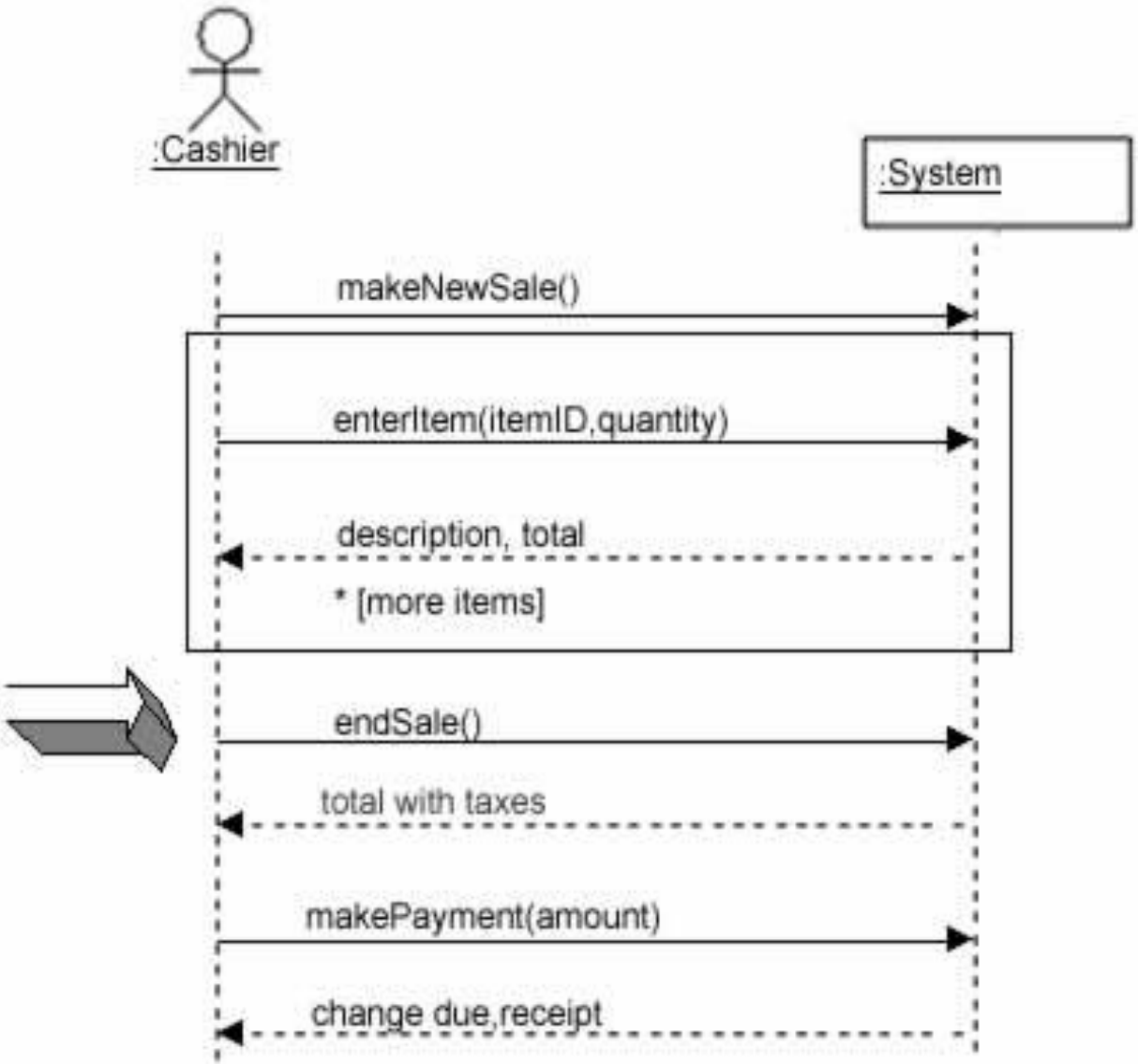
Example: use cases to SSD

Simple Cash-only Process Sale scenario:

1.customer arrives at aPOS check out with goods and/or services to purchase.
2.Cashier starts a new sale.
3.Cashier enters a new item identifier.
4.System records new sale line item and presents item description,price and running total.

Cashier repeats steps 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.
....



Identifying the right Actor

- In the process Sale example, does the customer interact directly with the POS system?
- Who does?
- Cashier interacts with the system directly
- Cashier is the generator of the system events
- Why is this an important observation?

Naming System events & operations

- System events and associated system operations should be expressed at the level of intent
- Rather than physical input medium or UI widget
- Start operation names with verb (from use case)
- Which is better, scanBarCode or enterItem?

SSDs within the Unified Process

Create System Sequence Diagrams during Elaboration in order to:

- Identify System events and major operations
- Write System operation contracts (Contracts describe detailed system behavior)
- Support better estimates
- Remember, there is a season for everything:
  it is not necessary to create SSDs for all scenarios of all use cases, at least not at the same time

Interaction Diagrams

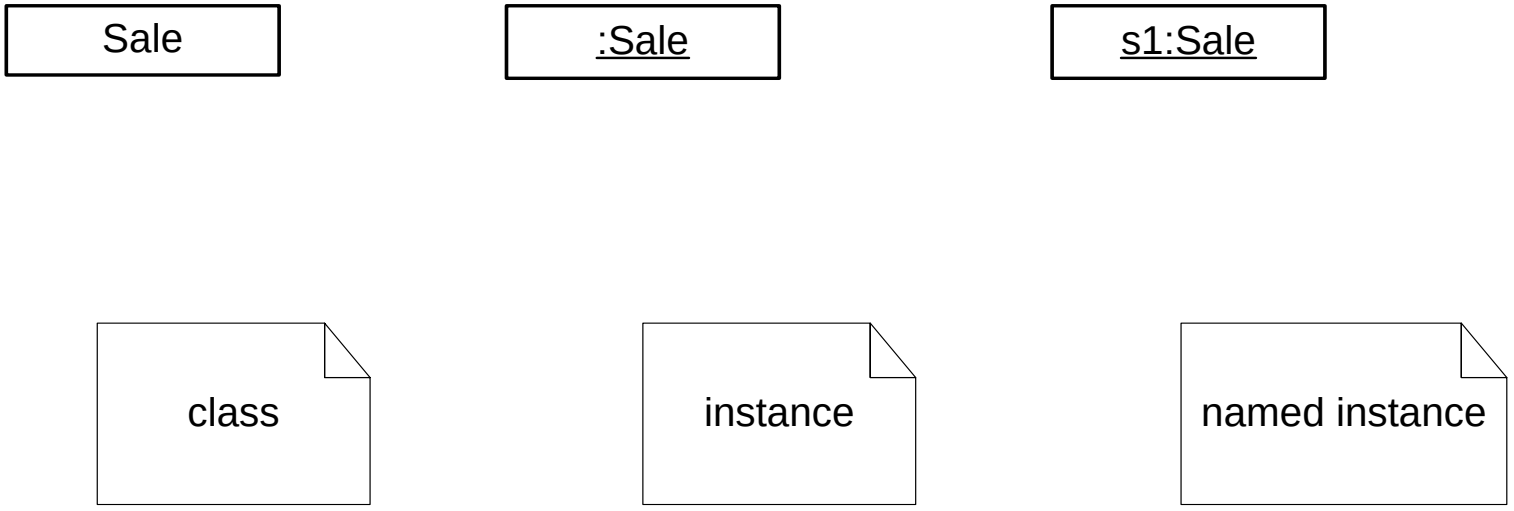
Designing with interaction and class diagrams

- Beginners often emphasize Class diagrams
- Interaction diagrams deserve more attention
- Some tools can help:
  - Convert between sequence and communication diagrams automatically
  - Reflect changes in class and interaction diagrams in parallel

Two kinds of UML Interaction Diagrams

- **Sequence** Diagrams: show object interactions arranged in time sequence, *vertically*
- **Communication** Diagrams: show object interactions arranged as a flow of objects and their links to each other, *numerically*
- Semantically equivalent, structurally different
  - Sequence diagram emphasize time ordering
  - Communication diagrams make object linkages explicit

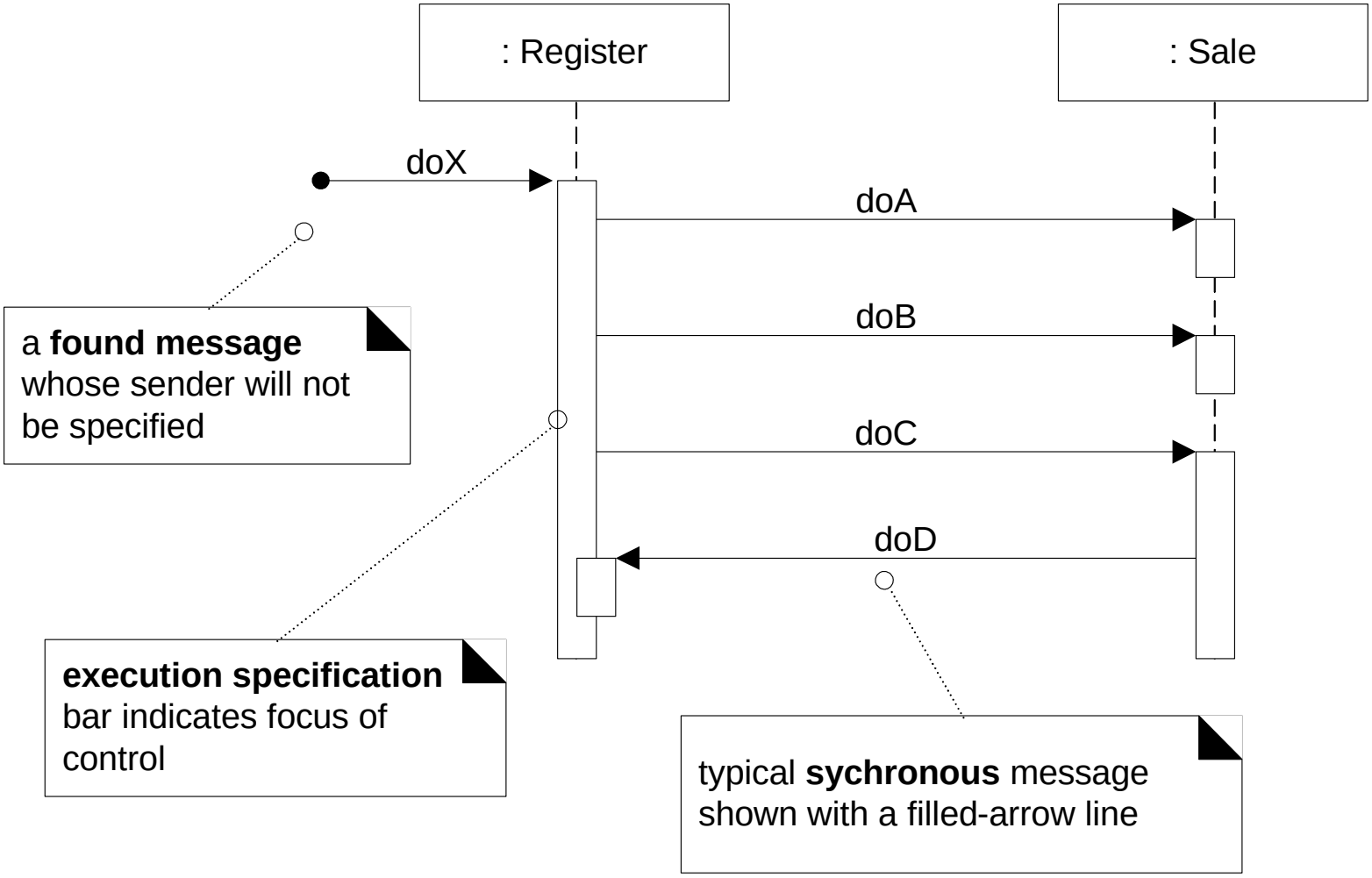
Interaction Diagram notation



Which would you expect to find most often in Interaction diagrams?

What do you think of "<u>:Sale</u>" instead of "aSale"?

Sequence diagram notation



What does vertical placement communicate?

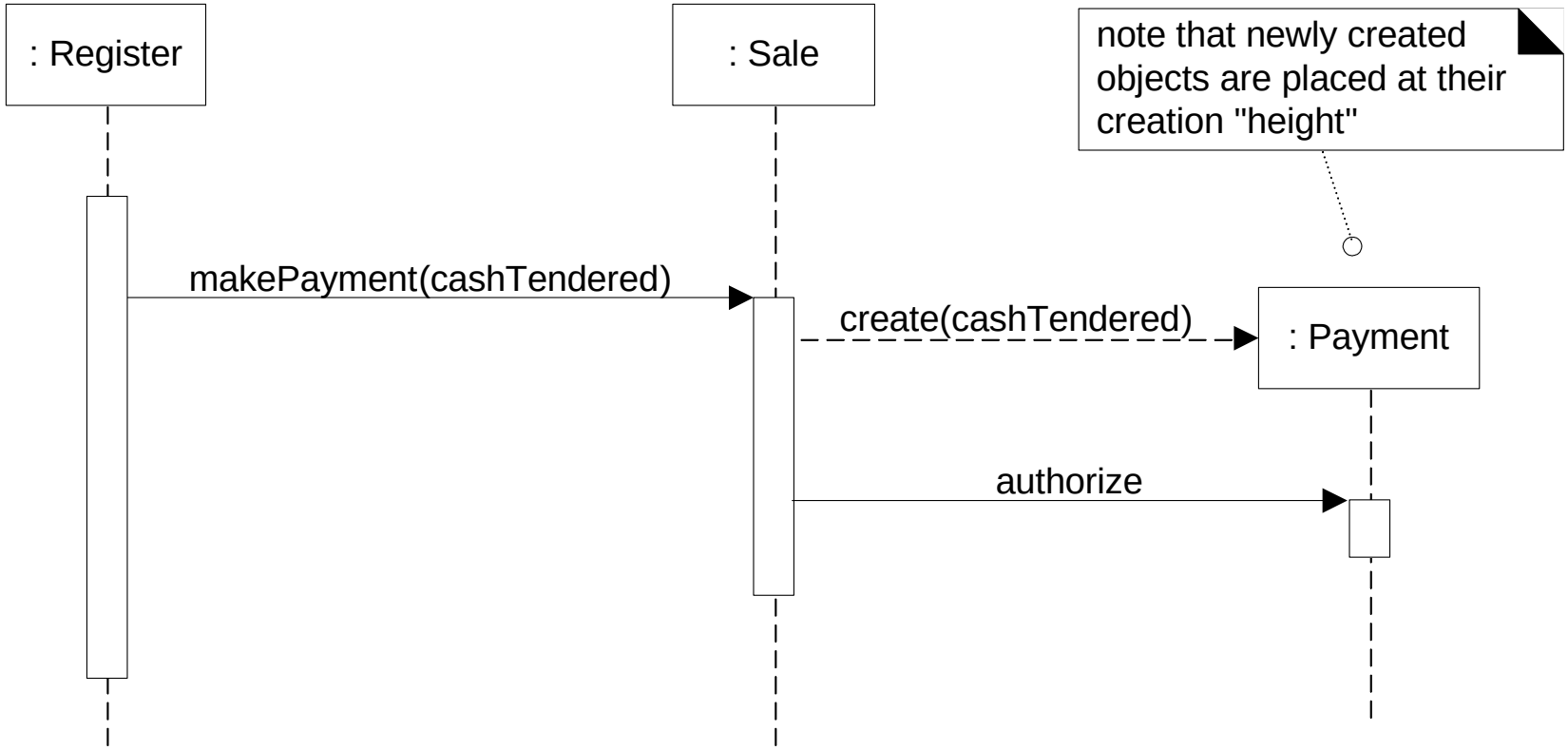
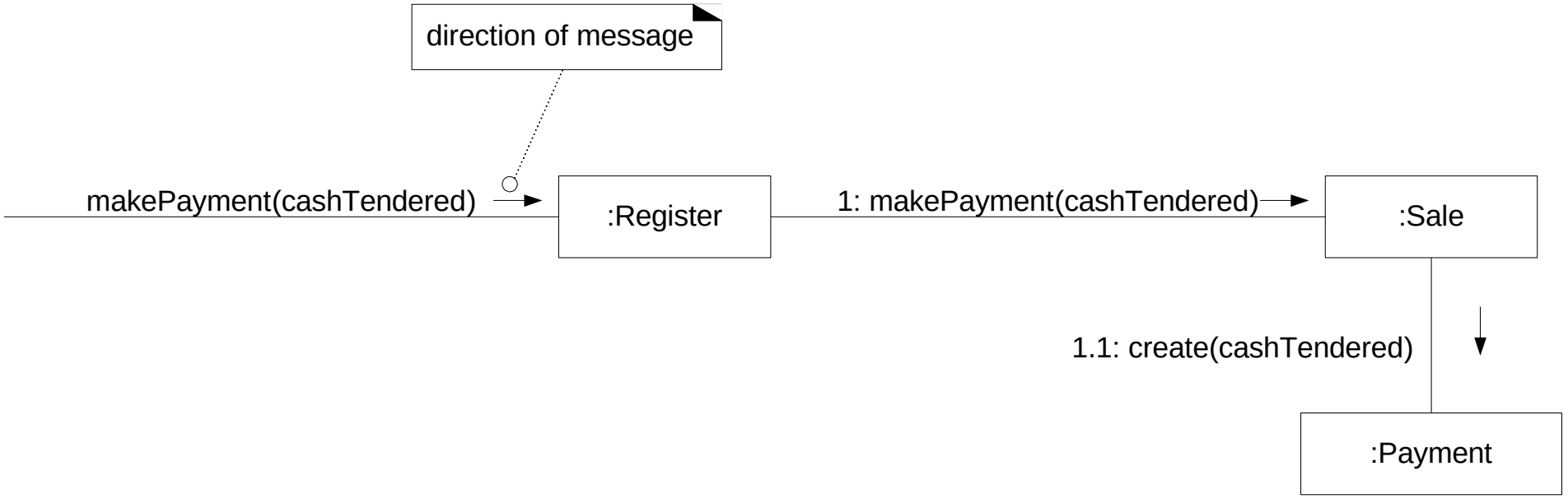


Figure 15.10

Communication Diagram: makePayment



What do the numbers communicate?

What does a create message communicate?

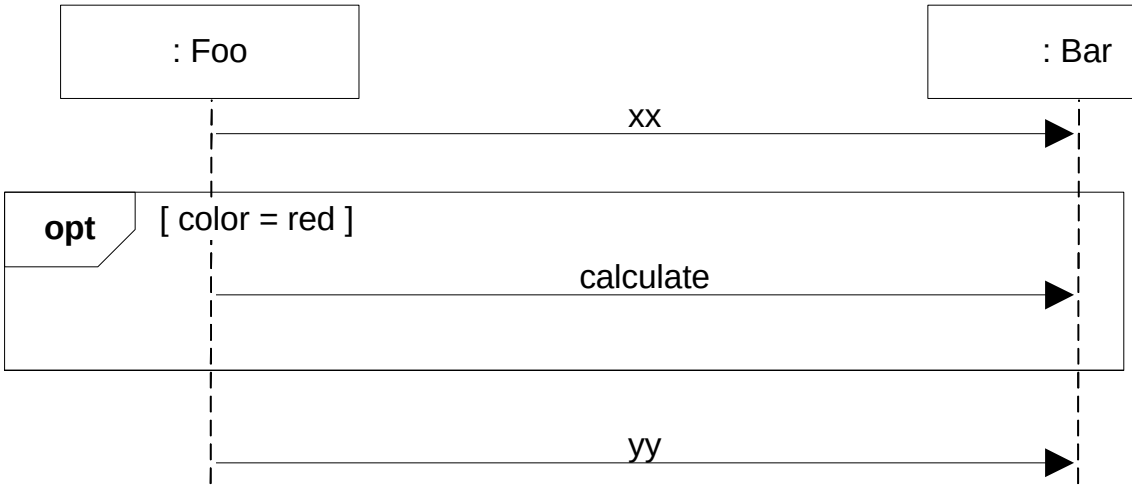
Communication (aka Collaboration) diagrams

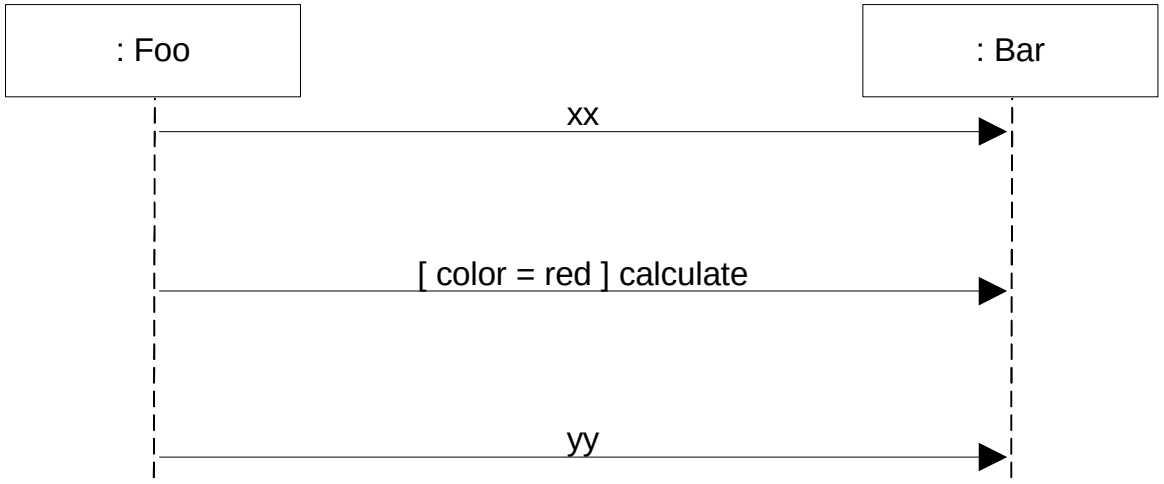
- Objects are rectangular icons
  - e.g., Order Entry Window, Order, etc.
- Messages are arrows between icons
  - e.g., prepare()
- Numbers on messages indicate sequence
  - Also spatial layout helps show flow
- *Which do you prefer: sequence or communication?*
- Fowler doesn't use *communication* diagrams
  - Show flow clearly, but awkward modeling alternatives
- UML notation for control logic has changed in UML 2 but Fowler isn't impressed

Control logic in Interaction Diagrams

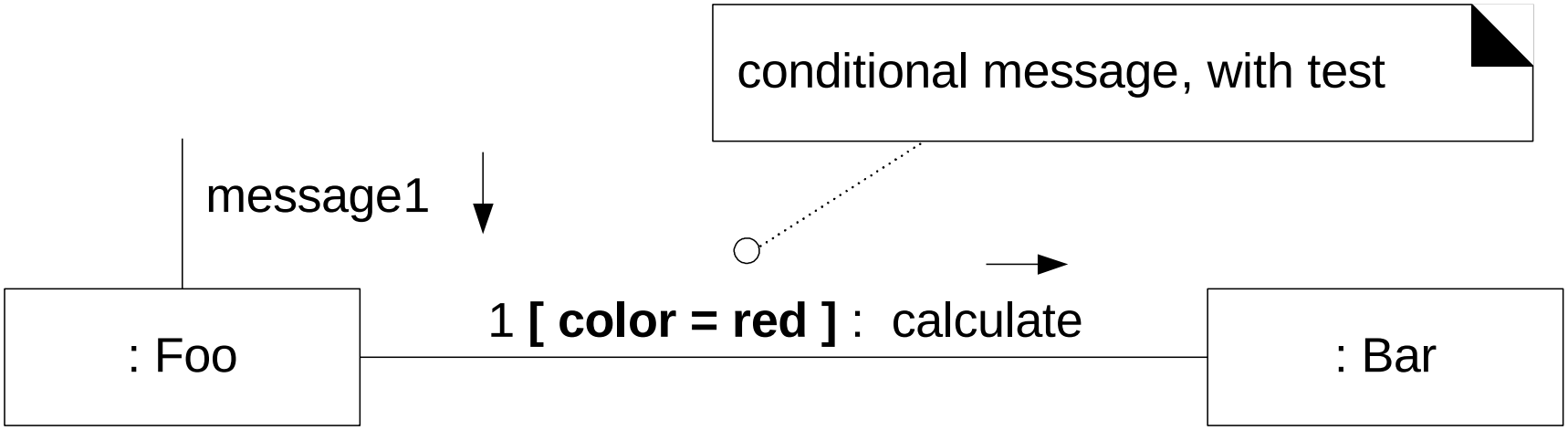
- Conditional Message
  - *[variable = value]* : message()
  - Message is sent only if clause evaluates to *true*
- Iteration (Looping)
  - ** [i := 1..N]*: message()
  - "*" is required; [...] clause is optional
- Communication diagrams add Seq. Numbers before conditional messages or loops

Logic in sequence diagrams: which notation do you prefer?

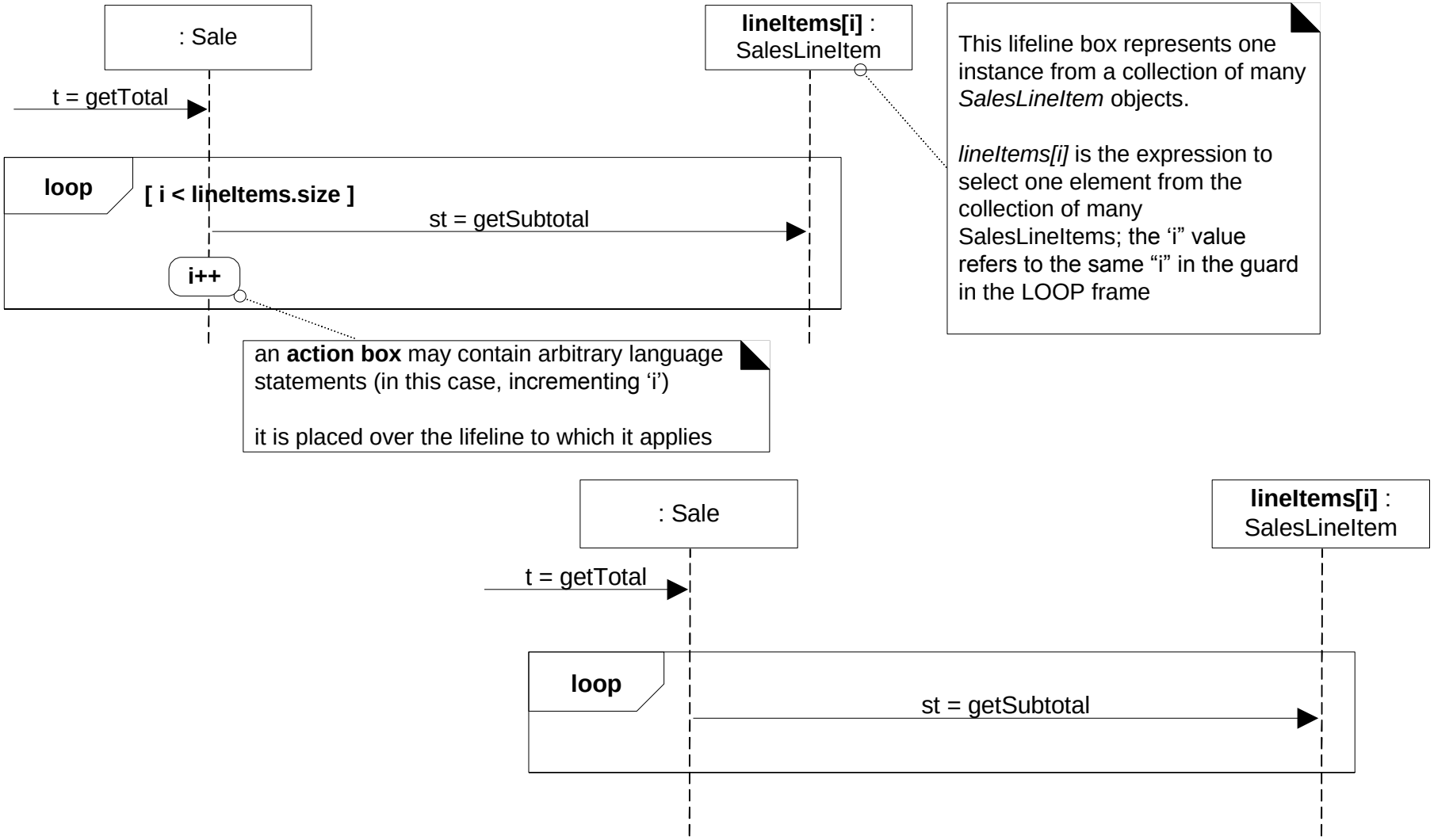




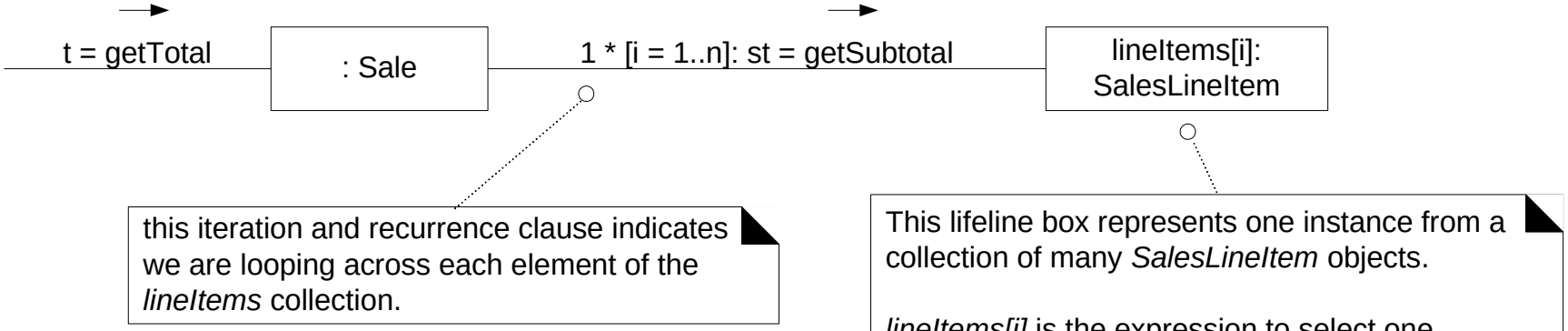
Logic in communication diagrams

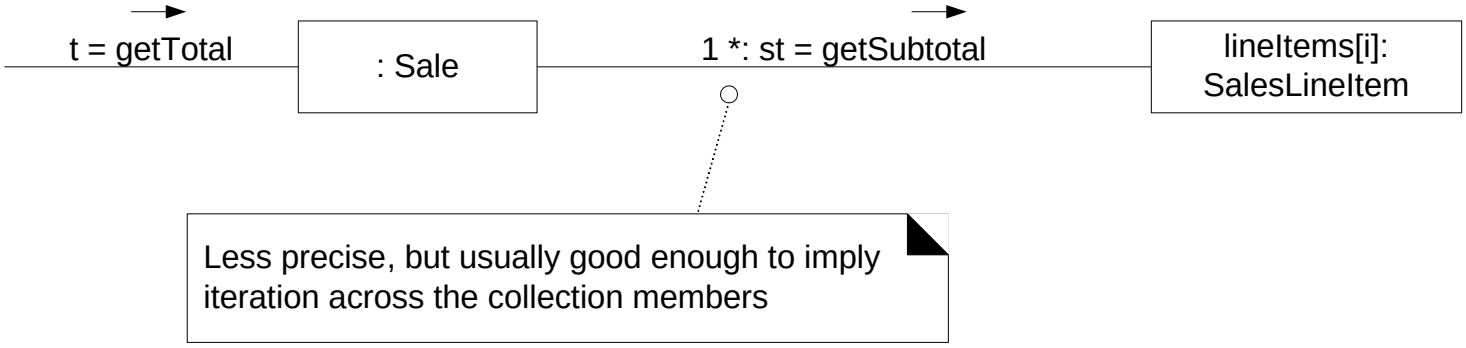


Loops in sequence diagrams: which notation do you prefer?

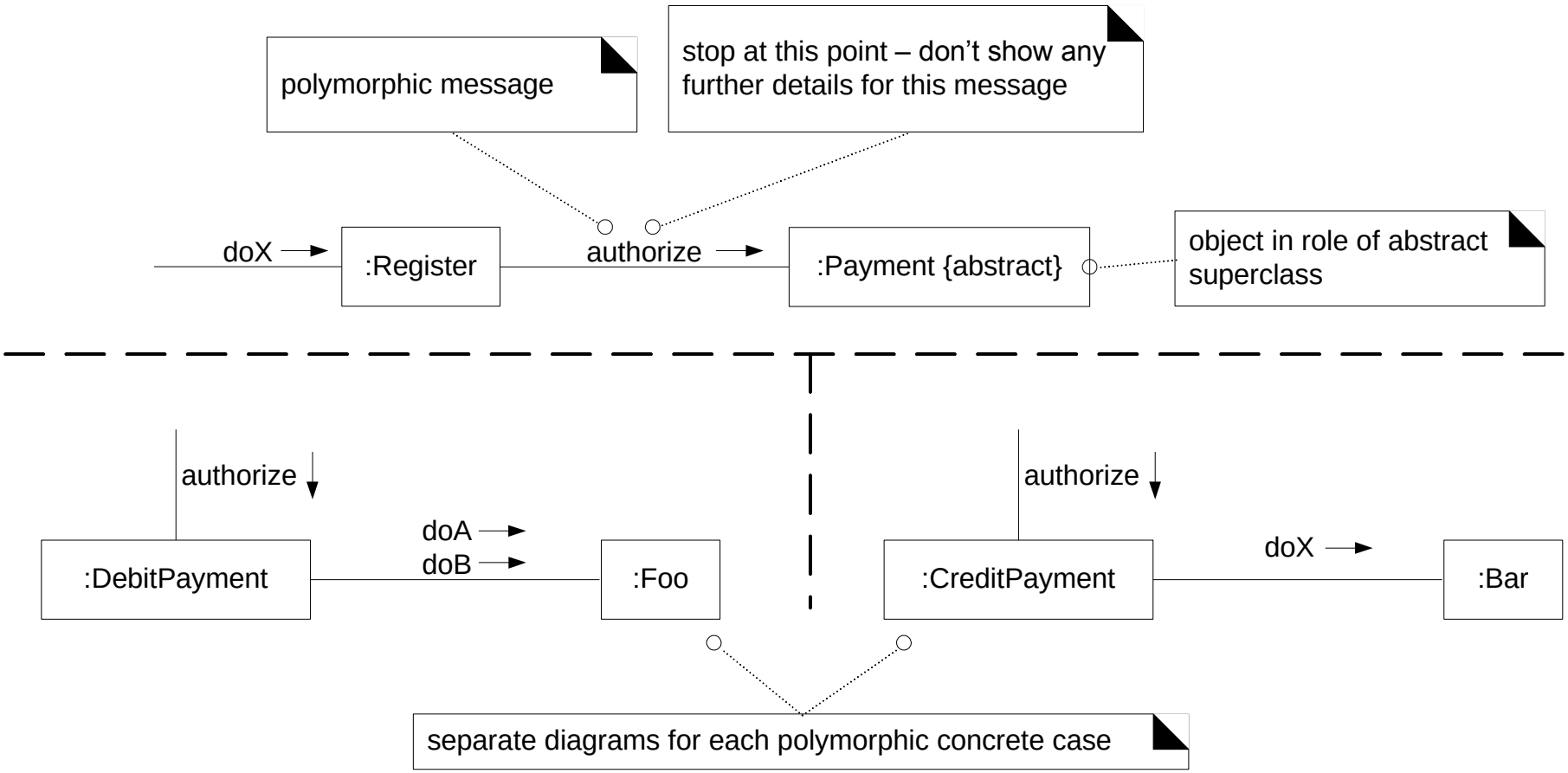


Iteration in communication diagrams





Polymorphism: How is it shown in interaction diagrams?



Thank You