OODP– Session 2

Session times

PT group 1 – Monday	18:00-21:00	room: Malet 403
PT group 2 – Thursday	18:00-21:00	room: Malet 403
FT - Tuesday	13:30-17:00	room: Malet 407

Email: oded@dcs.bbk.ac.uk

Web Page: http://www.dcs.bbk.ac.uk/~oded

Visiting Hours: Tuesday 17:00 to 19:00

Last Time

- In the beginning there was CHAOS
- Waterfall
- Agile
  - Iterative, Incremental, Interactive
- Unified Process (UP)
  - Inception
    - Supplementary Specifications, Vision, Glossary
    - Use Case Model

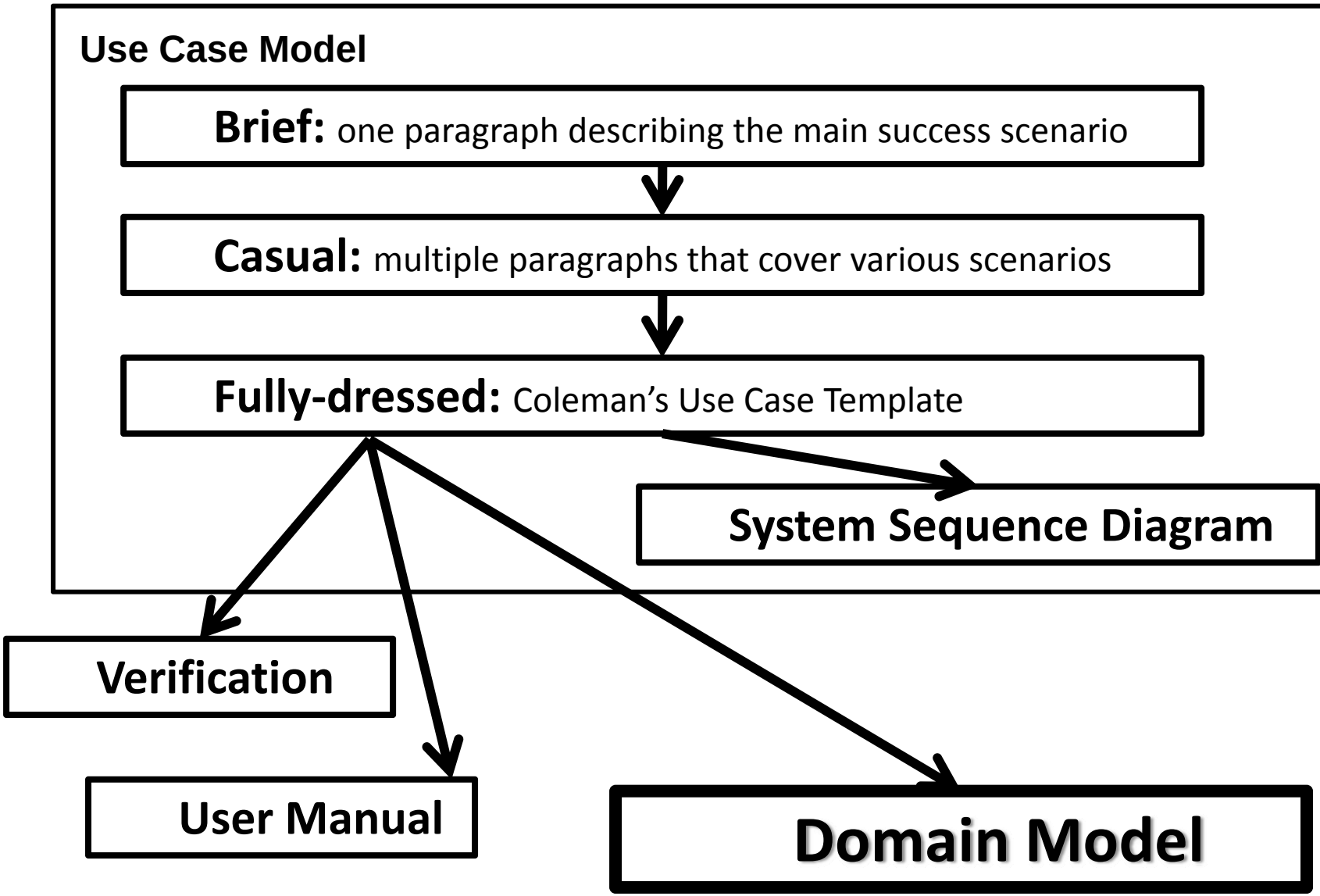
Remarks on Inception

- Most use cases written in brief format; 10-20% of cases written in fully dressed format
- Most influential and risky quality requirements identified
- First version of the Vision and SS documents

Remarks on Use Case Model

- It is the interface point with the costumers so keep it simple
- Helps uncover “structure” which might be hidden by a feature list
- Not just a starting point for Analysis and Design, but also for verification

Use case model and the next stage



Use cases scenarios

- An *actor* is something with behavior, such as a person, computer system, or organization; e.g. a cashier.
- A *scenario* – also called *use case instance* - is a specific sequence of actions and interactions between actors and the system under discussion. It is one path through the use case.
- A *use case* is a collection of related success and failure scenarios that describe actors using a system to support a goal.

Process Sale

Main Success Scenario: A customer arrives at a checkout with items to purchase …

Alternate Scenarios:

If the system rejects customer's credit card…

If the customer has less cash than the required payment…

Guidelines for writing use cases

- Try to answer the question: *How can using the system provide observable value to the users or fulfill their goals?*
- Don't think of system requirements in terms of a check list of features or functions.
- Use cases emphasize the functional requirements (other requirement types are sometimes included, but have a secondary role).
- Use cases are text documents, not diagrams.
- *Black-box* use cases are highly recommended. We describe the responsibilities of a system, not the internal workings.

  Process Sale: … The system records the sale …

  NOT: The system generates an SQL INSERT stmt for the sale…

Guidelines for writing use cases (cont)

- Identify all the different users in the system
- Create a user profile for each category of user, including all the roles the users play that are relevant to the system.
- For each role, identify all the significant goals the users have that the system will support. A statement of the system's value proposition is useful in identifying significant goals.
- Create a use-case for each goal, following the use-case template. Maintain the same level of abstraction throughout the use case. Steps in higher-level use cases may be treated as goals for lower-level (i.e. mode detailed) sub-use cases.
- Structure the use cases.
- Review and validate with users.

Elaboration

Elaboration Overview

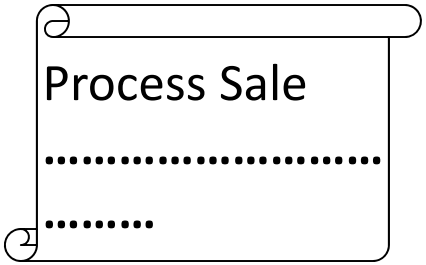
- Program and Test, core and critical parts of the project
- Achieve a stable version of the majority of the requirements
- Mitigate or retire major risks

2-3 iterations

Artifacts at the Beginning of Elaboration

Artifact	Description
Domain Model	**Visual representation of the domain model**
Design Model	**Logical Design Diagrams**
Software Architecture Document	**Summary of key architectural issues and their resolution**
Data Model	**Database schemas, mapping strategies between object and non-object representation**
Use-Case Storyboards, UI Prototypes	**User interface description, usability models...**

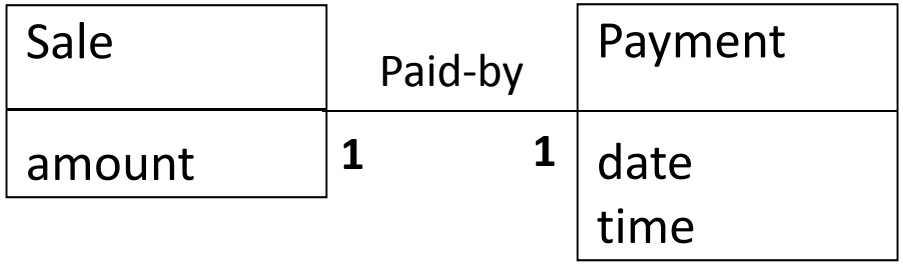
From requirements and analysis to design

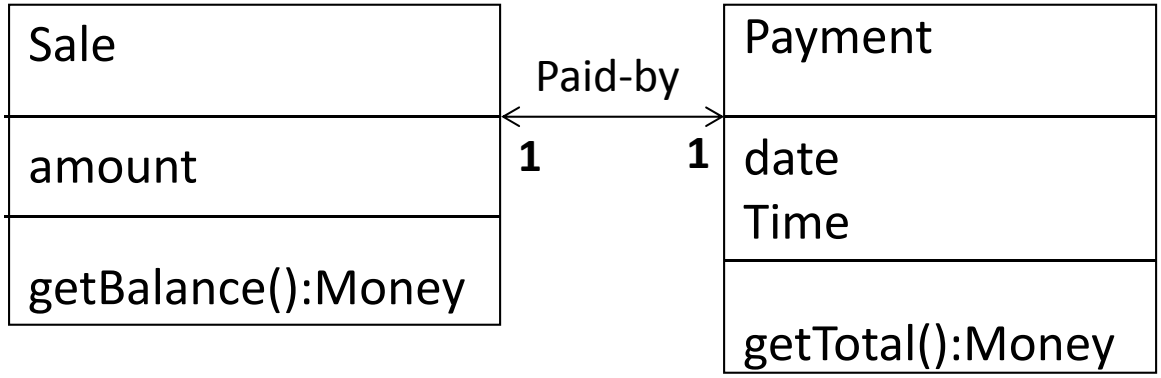


UP Use-case Model

System sequence diagram

UP Domain Model





UP Design Model

System Sequence Diagrams

What else can we learn from the use cases

- We have seen that through the use cases we learn what are the actors that interact with our system

- Now we want to study what are the interactions between the actors and the system

- We use System Sequence Diagrams to find what are the events our system needs to deal with

This gets us almost as close as possible to the “border” of the system, afterwards we do the internals.

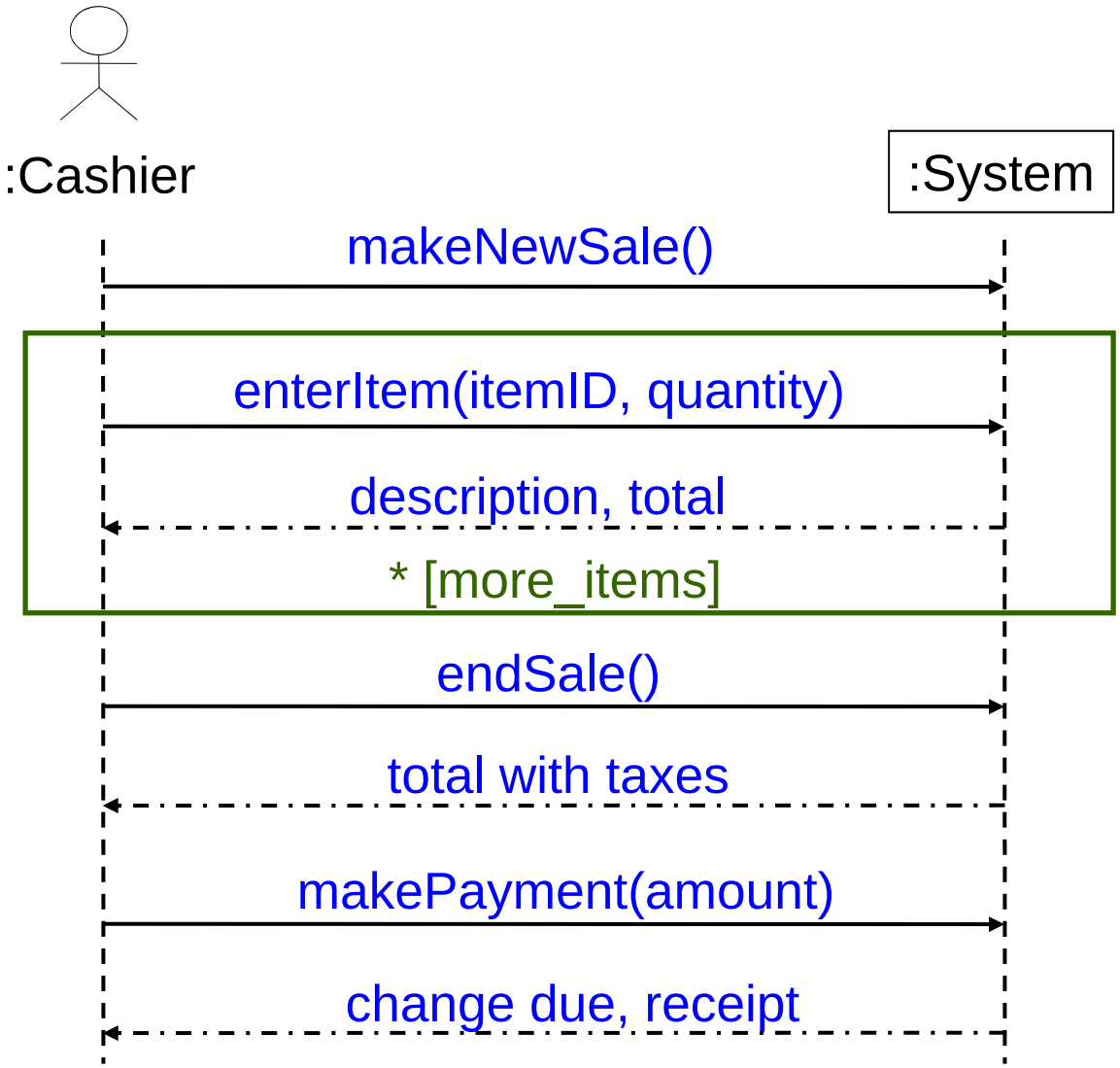
System sequence diagrams (SSDs)

- Use cases describe how actors interact with the software system
- UML notation for illustrating actor interactions.
- Given a scenario of a use case, an SSD shows:
  - the external actors that interact directly with the system,
  - the system as a black box and
  - the system events that the actors generate.
- The ordering of events should follow their order in the use case.
- SSDs are part of the use-case model.
- SSDs are visualizations of interactions implied in the use cases.

NextGen POS system

Use Case	1.Process_Sale
Descirption	A user arrives at a POS sales point and tries to purchase an item.
Actors	Cashier (Primary)
Assumptions	Cashier is identified and authenticated
Steps	1. Customer arrives at POS checkout with goods to purchase. 2. Cashier starts a new sale. 3. WHILE more items DO 1. Cashier enters item identifier. 2. System records sale line item and presents item description, price and running total. END WHILE 4. System presents total with taxes calculated. 5. Cashier tells Customer the total, and asks for payment. 6. Customer pays and System handles payment. 7. System presents receipt. 8. Customer leaves with receipt and goods (if any).

An SSD example



Domain Model

Domain Model

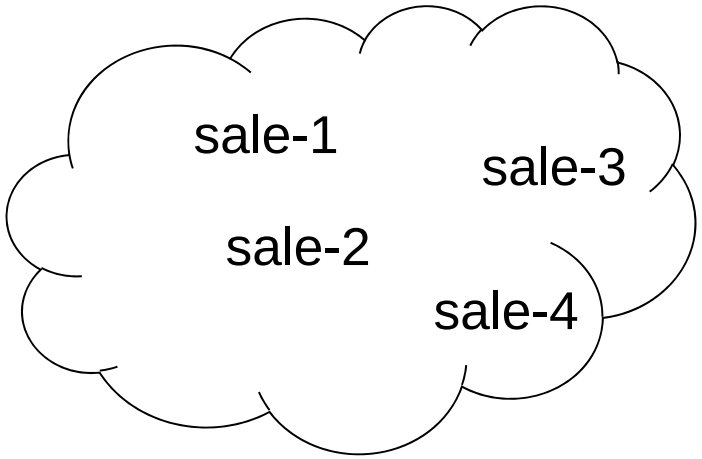
- A visual representation of conceptual classes or real-world objects in a domain of interest
- ***IS NOT*** a visualization of software components
- **IS**
  - *Class diagrams* (UML notation) show:
    - conceptual classes
    - attributes of conceptual classes
    - associations between conceptual classes

Conceptual classes

- Three ways of viewing a conceptual class:
  - Symbol – representation
  - Intension – definition
  - Extension - context
- In a domain model, we are interested in both the symbol and the intension of a conceptual class.

Sale
date time

"A sale represents the event of a purchase transaction. It has a date and a time."



How to identify conceptual classes

1. Reuse and modify existing models
2. Use a category list
3. Identify noun phrases

Category list

Conceptual class category	Examples
Business transaction	**Sale, Payment, Reservation**
Transaction line items	**SalesLineItem**
Product or service related to a transaction or transaction line item	**Item** **Flight, Seat, Meal**
Where is the transaction recorded	**Register, Ledger**
Place of transaction	**Store, airplane, seat**

NextGen POS system

Use Case	1.Process_Sale
Descirption	A user arrives at a POS sales point and tries to purchase an item.
Actors	Cashier (Primary)
Assumptions	Cashier is identified and authenticated
Steps	1. Customer arrives at POS checkout with goods to purchase. 2. Cashier starts a new sale. 3. WHILE more items DO 1. Cashier enters item identifier. 2. System records sale line item and presents item description, price and running total. END WHILE 4. System presents total with taxes calculated. 5. Cashier tells Customer the total, and asks for payment. 6. Customer pays and System handles payment. 7. System presents receipt. 8. Customer leaves with receipt and goods (if any).

How to identify conceptual classes

Linguistic analysis: Identify noun phrases in the use cases.

Process Sale …

Main Success Scenario:

1. **Customer** arrives at a **POS checkout** with **goods** 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 **total**. …

Use a conceptual class category list.

- physical objects
- specifications, descriptions
- places
- transactions
- transaction line items
- roles of people
- containers or contained item
- event
- catalog etc.

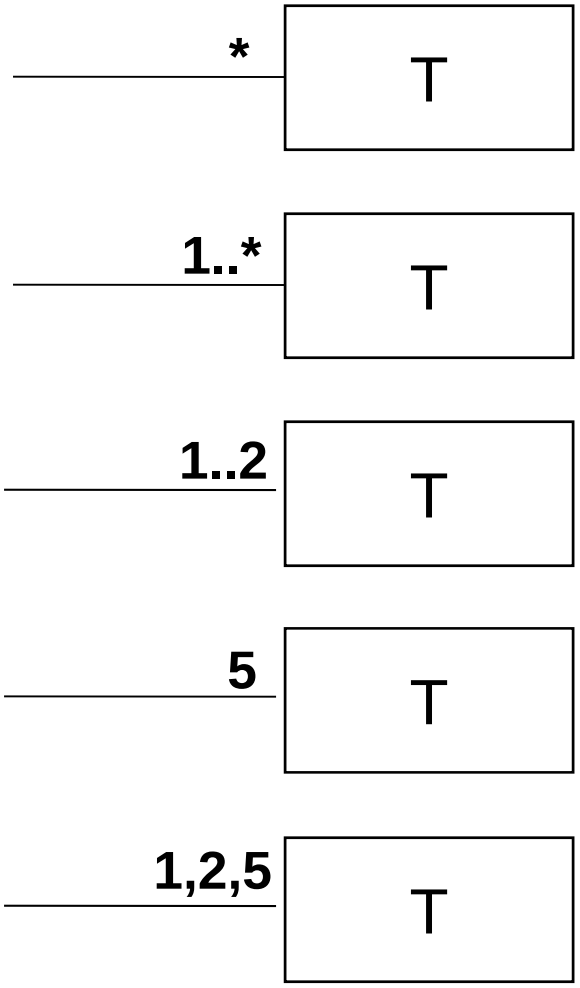
How to identify associations

- An association is a relationship between object instances that indicates some meaningful and interesting connection
  - worth *remembering*
  - derived from the Common Associations List
    - *A is a physical (or logical) part of B*
    - *A is physically (or logically) contained in*
    - *A is a description of B*
    - *A is known/captured/logged/recorded in B*
    - *A uses or manages B*
    - *A is related to a transaction B*
    - *Etc.*

Candidate associations

- Relationships that need to be remembered:
  - Register **Records** Sale (why?)
  - Sale **Paid-by** Payment (why?)
  - ProductCatalog **Records** ProductSpecification (why?)
- Relationships derived from the Common Associations List
  - SalesLineItem **Is-contained-in** Sale
  - Store **Contains** Item
  - ProductSpecification **Describes** Item
  - Cashier **Is-member-of** Store
  - Cashier **Uses/Manages** Register
  - Customer/Cashier Makes/Receives (Is-related-to) Payment
  - Etc.

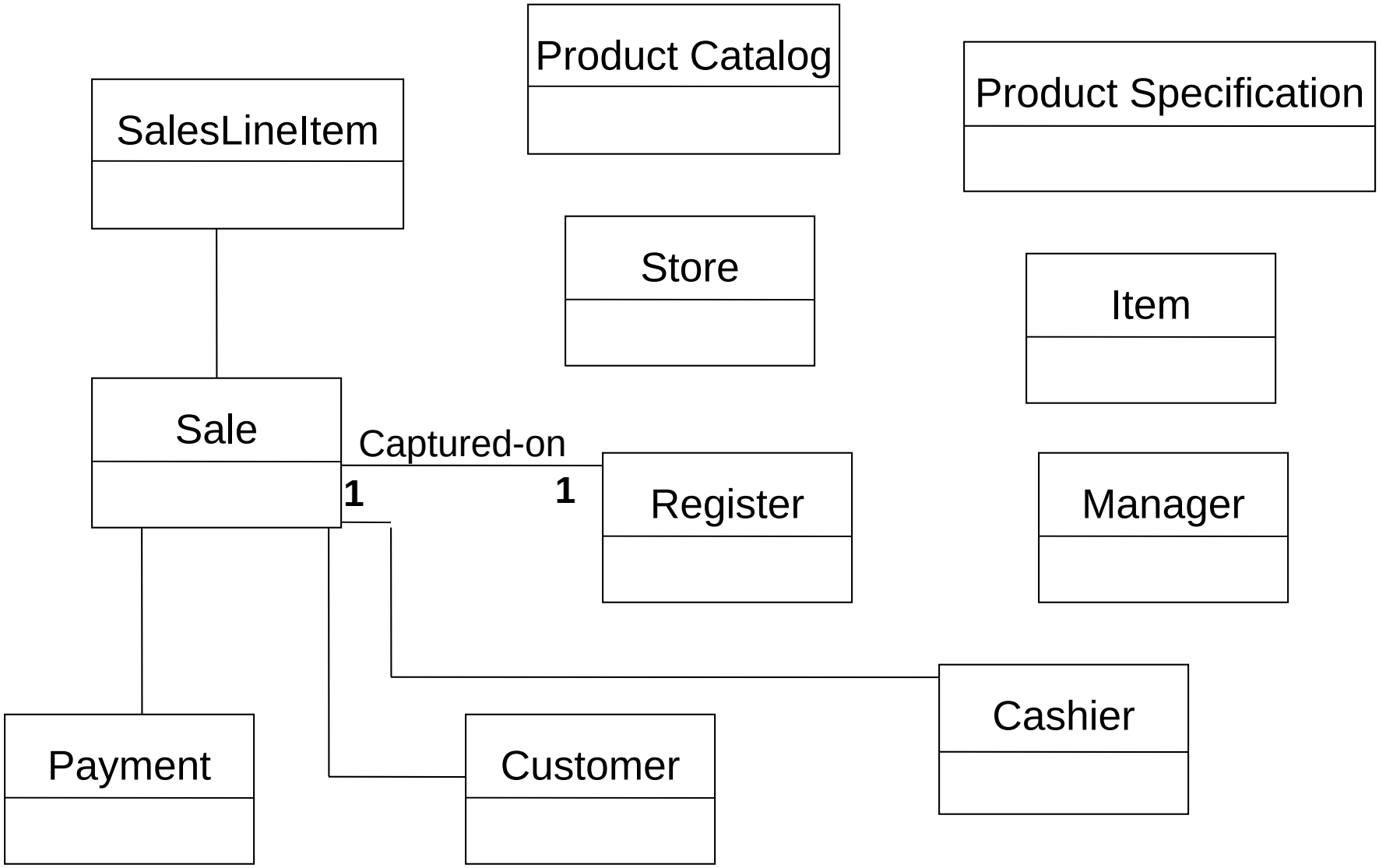
Multiplicity of associations



Attention!

The multiplicity value of a relationship indicates how many object instances can be validly associated with another, **at a particular moment**, rather than over a span of time.

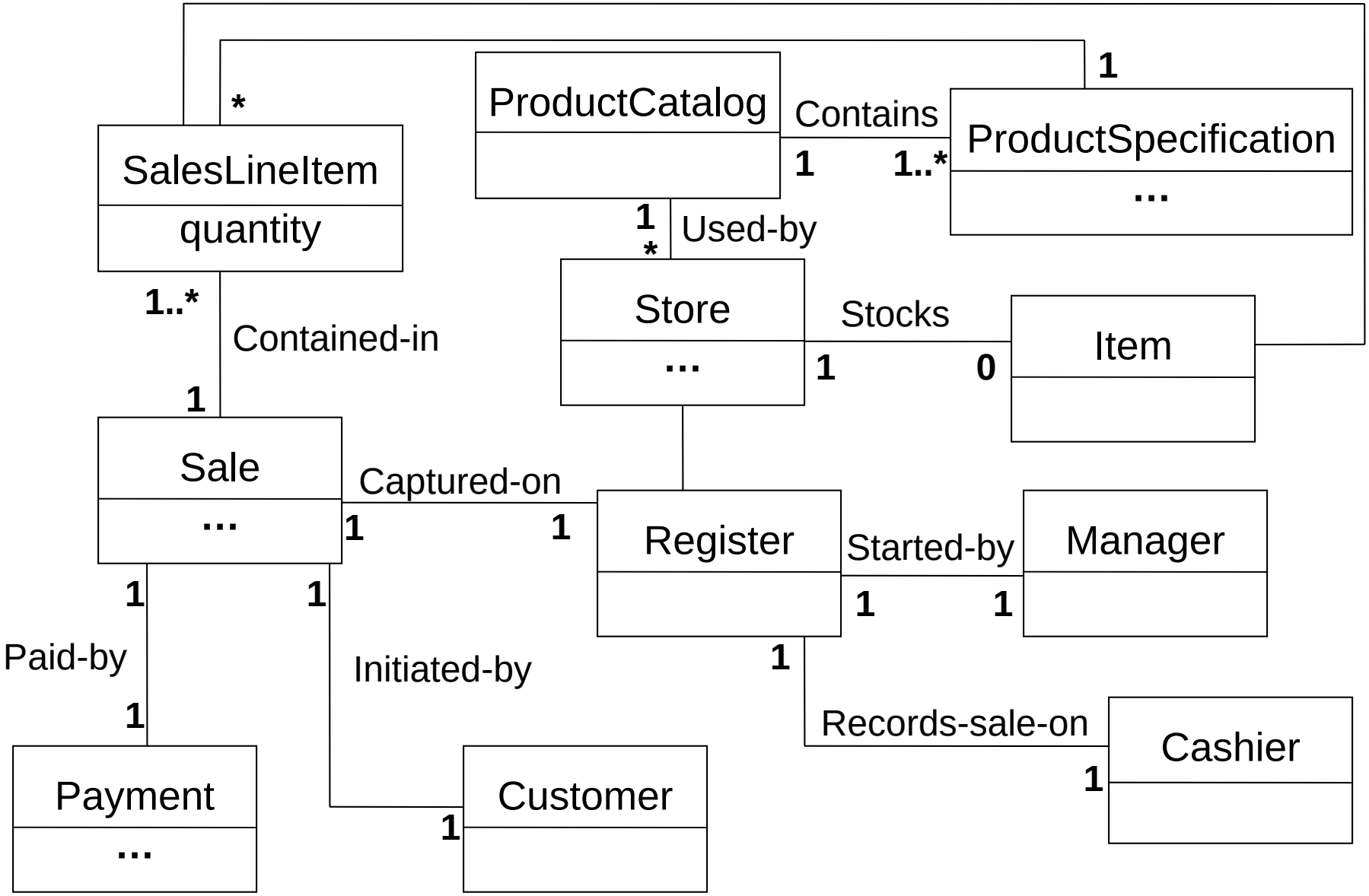
Adding associations (names/multipl.)



How to identify attributes

- Attributes in a domain model should preferably be **simple attributes** or **data types**
- Common types: boolean, date, number, string, text, time
- Other types: address, color, geometrics, phone number, national insurance number, universal product code, postal codes
- Do not relate conceptual classes with an attribute!
- Represent a data type as a non-primitive class if:
  - *It is composed of separate sections*
  - *Operations are associated with it (e.g. parsing)*
  - *It has other attributes*
  - *It is an abstraction of one or more types*

Adding attributes



Pitfalls

Initial attempt

Sale
store

Shouldn't "*store*" be a separate class

Item
serial no description price prodId

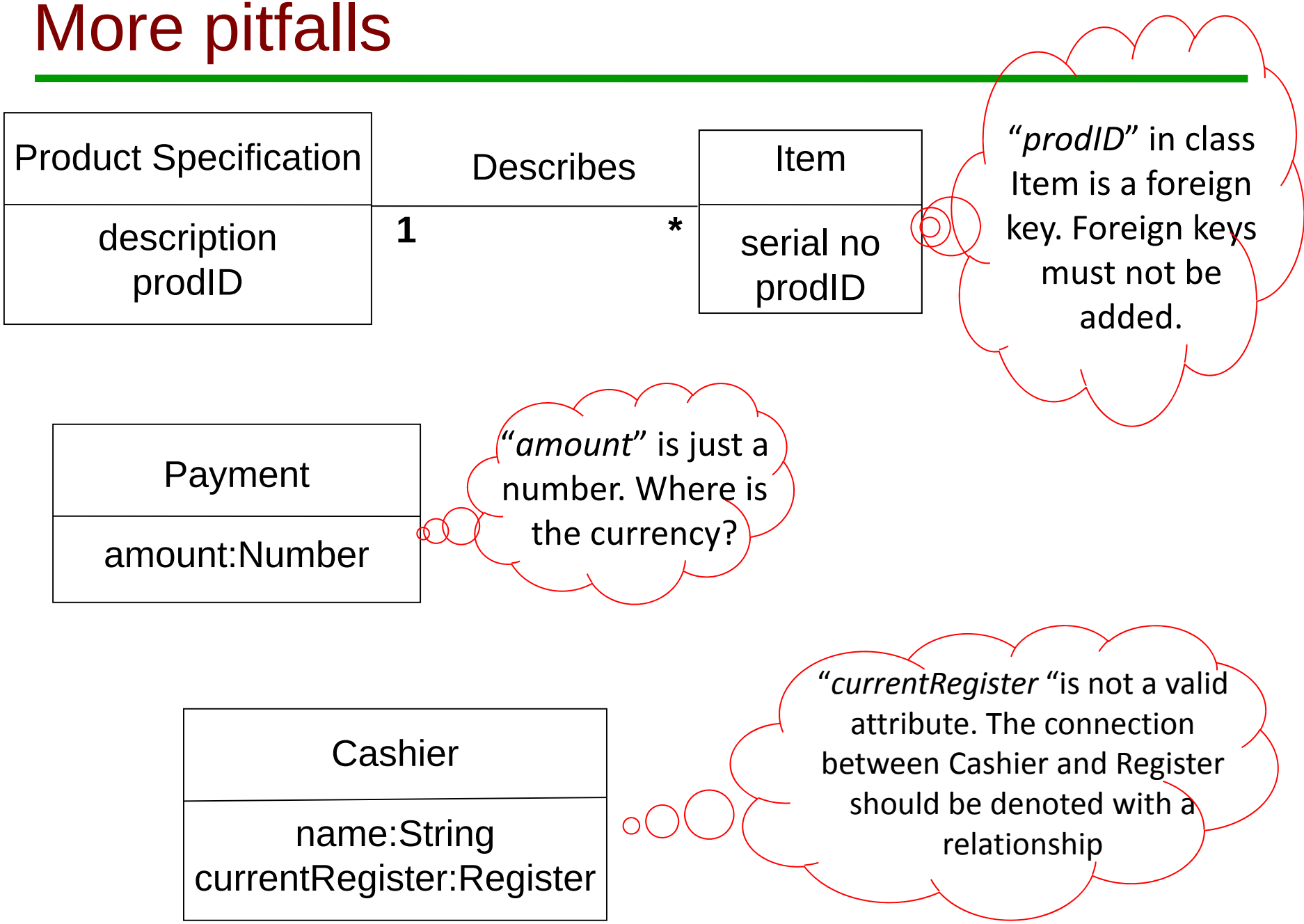
A specification or description class is needed

Improved object model

Sale	Store

Item	Product Specification
serial no	description price prodId

More pitfalls



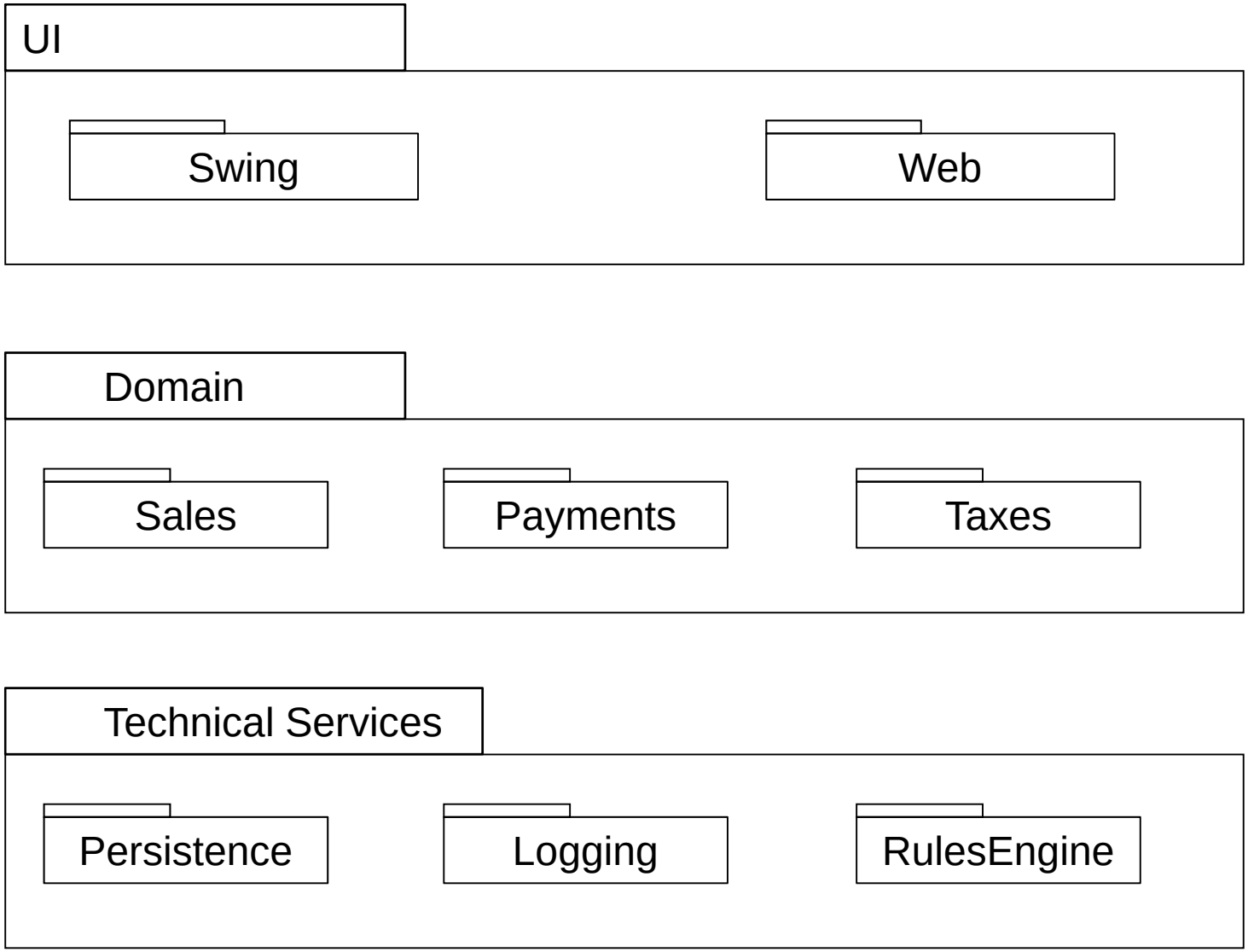
Logical Architecture

Logical Architecture

- Large scale organization of software classes into
  - Layers that are a coarse grain grouping of
    - Classes
    - Packages
    - Subsystems

Logical because, there is no decision of how these elements are deployed

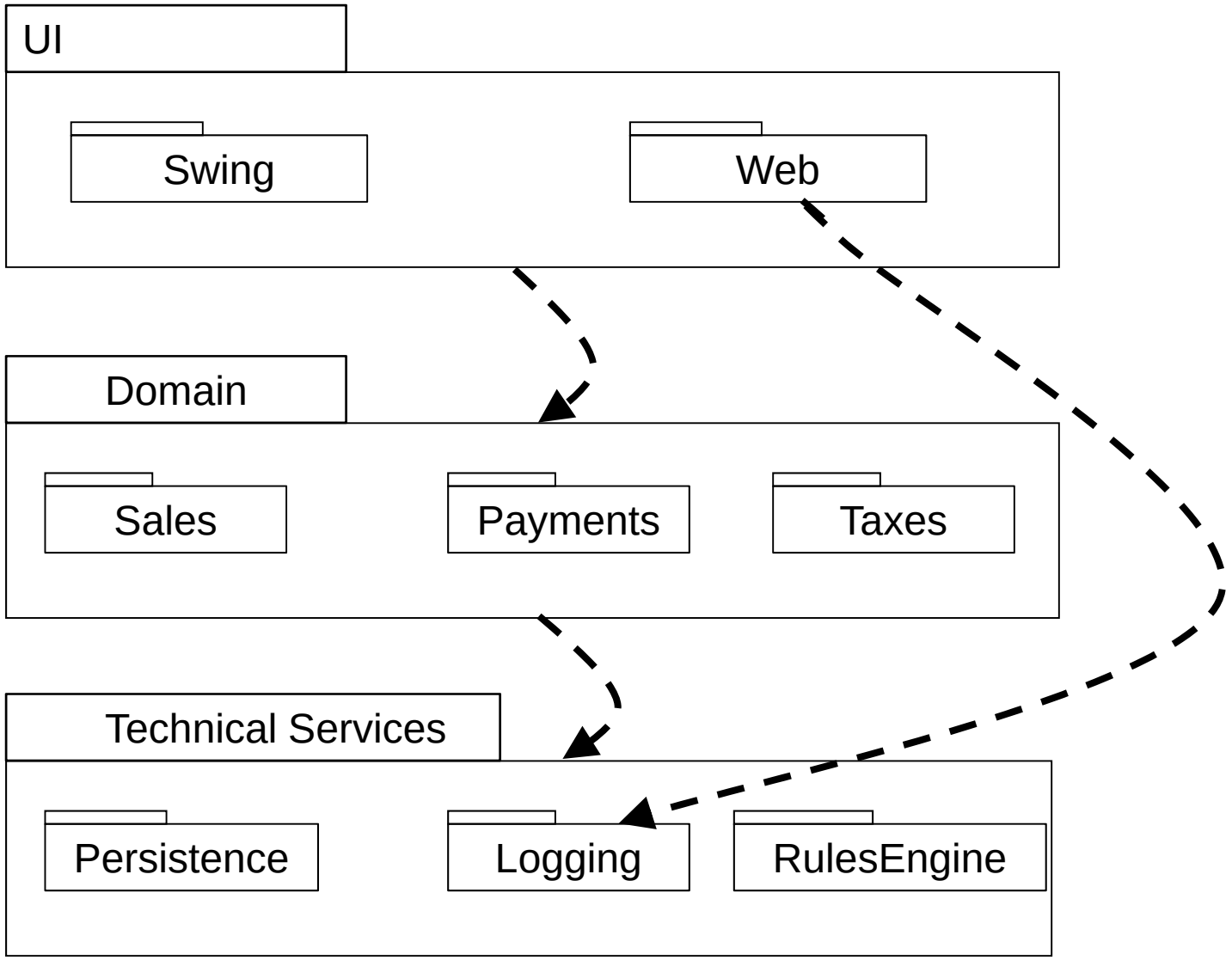
Example Layers



Layers?

- UI – user interface
- Application logic and Domain Objects
  - Software object representing domain concepts that fulfil application requirements
- Technical Services
  - General purpose objects and subsystems providing supporting technical services.

Layer Interaction



Strict,
Relaxed

Design Model

Domain Model, Design Model what's the difference?

- Both use the object-oriented paradigm
- Domain Model describes the problem at hand (does the right thing), whereas the Design Model designs a good solution for the problem ("does the thing right")
- Domain Model provides a static view of conceptual classes, whereas the Design Model provides both a static and a dynamic view of software objects and classes.

Interaction Diagrams

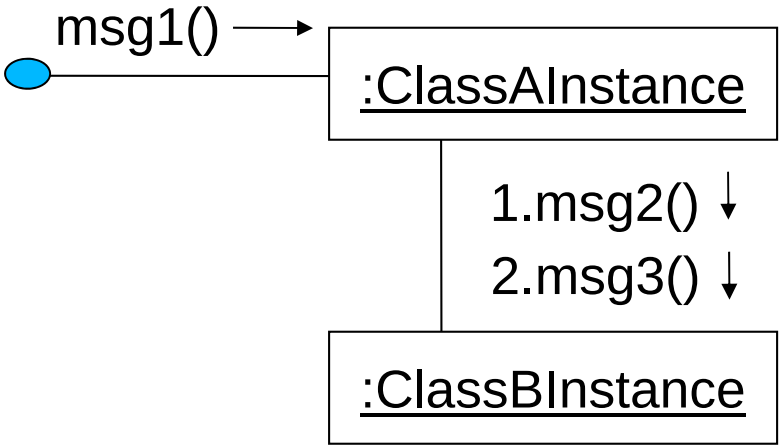
Interaction diagrams

- Interaction diagrams are the heart of the Design Model
- They illustrate how objects interact via messages and collaborate to fulfill the requirements
- They provide a dynamic view to the system
- Many design decisions are taken in the process of generating interaction diagrams

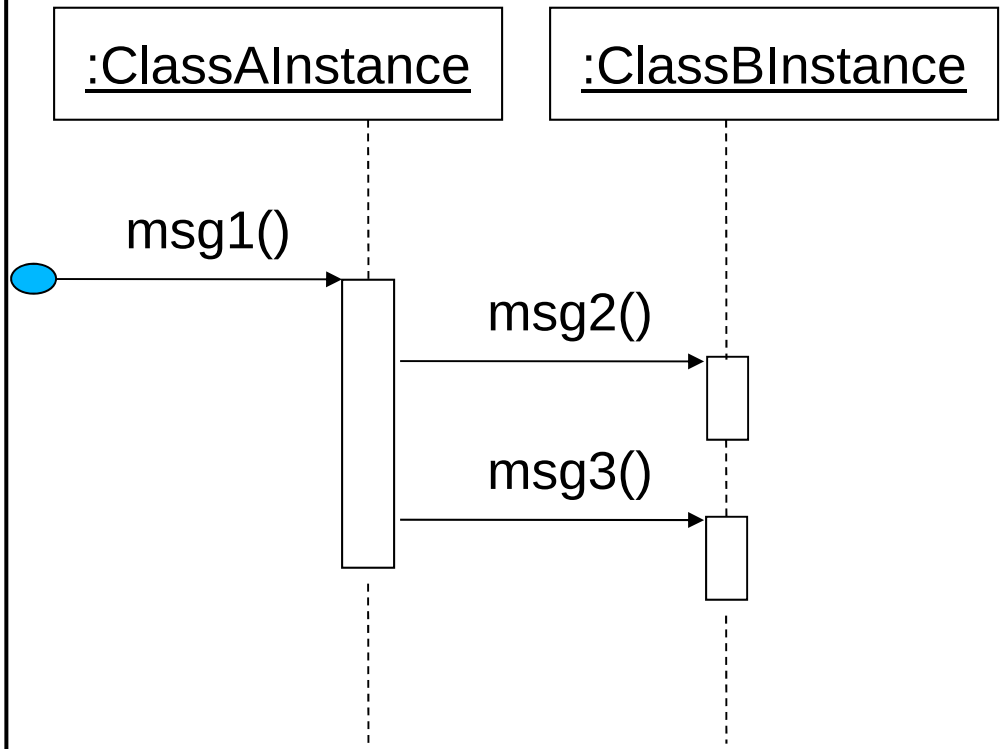
  => their creation requires strong design skills
- But, first let’s look at the notation!

Interaction diagrams

Communication diagram



Sequence diagram



Common interaction diagram notation

- Classes and instances

class example:

instance examples:

- Message expression syntax

return:= message ({parameter:parameterType}) : returnType

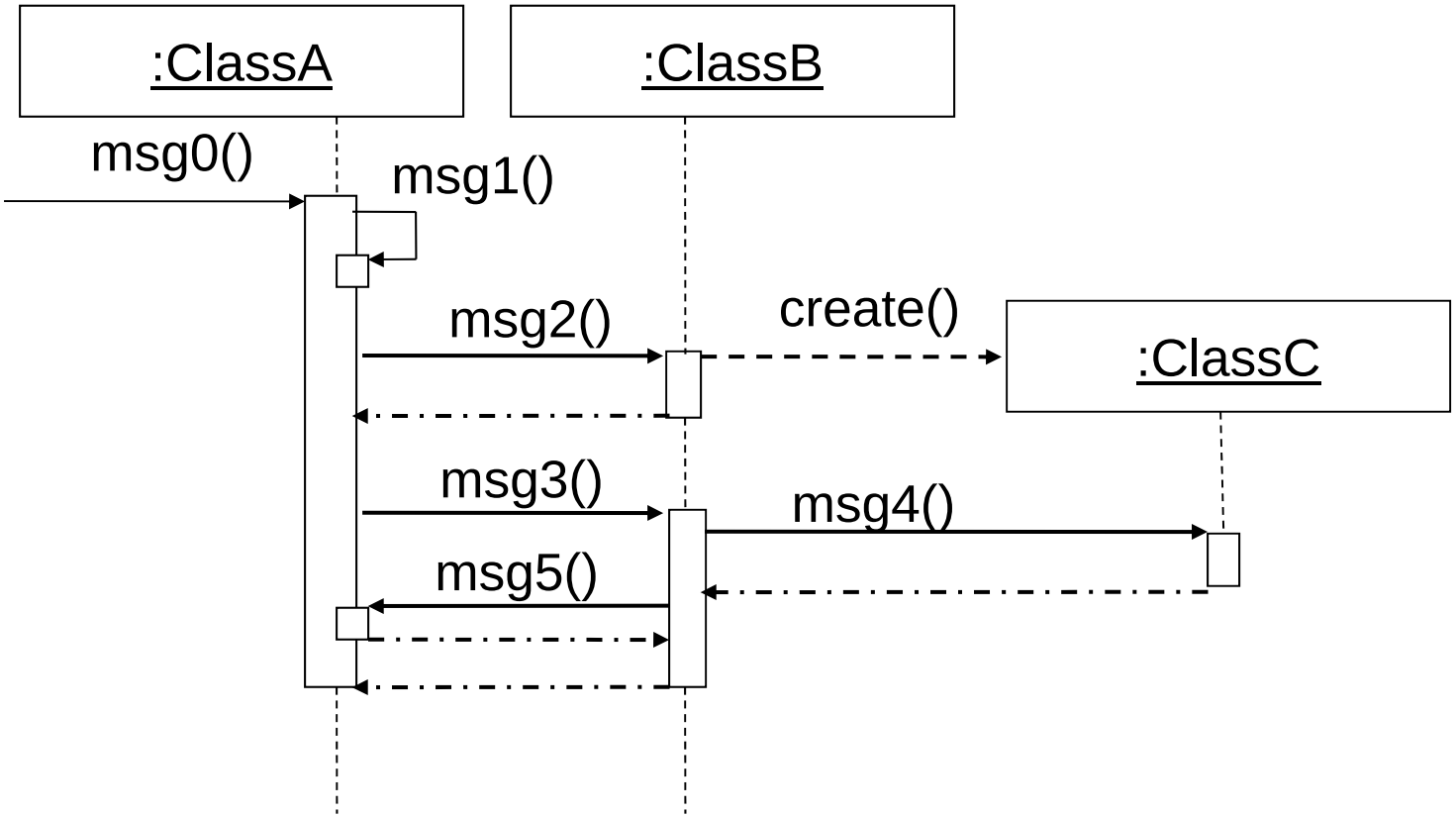
message example:

spec := getProductSpec (id:ItemID) : ProductSpec

(type information can be omitted)

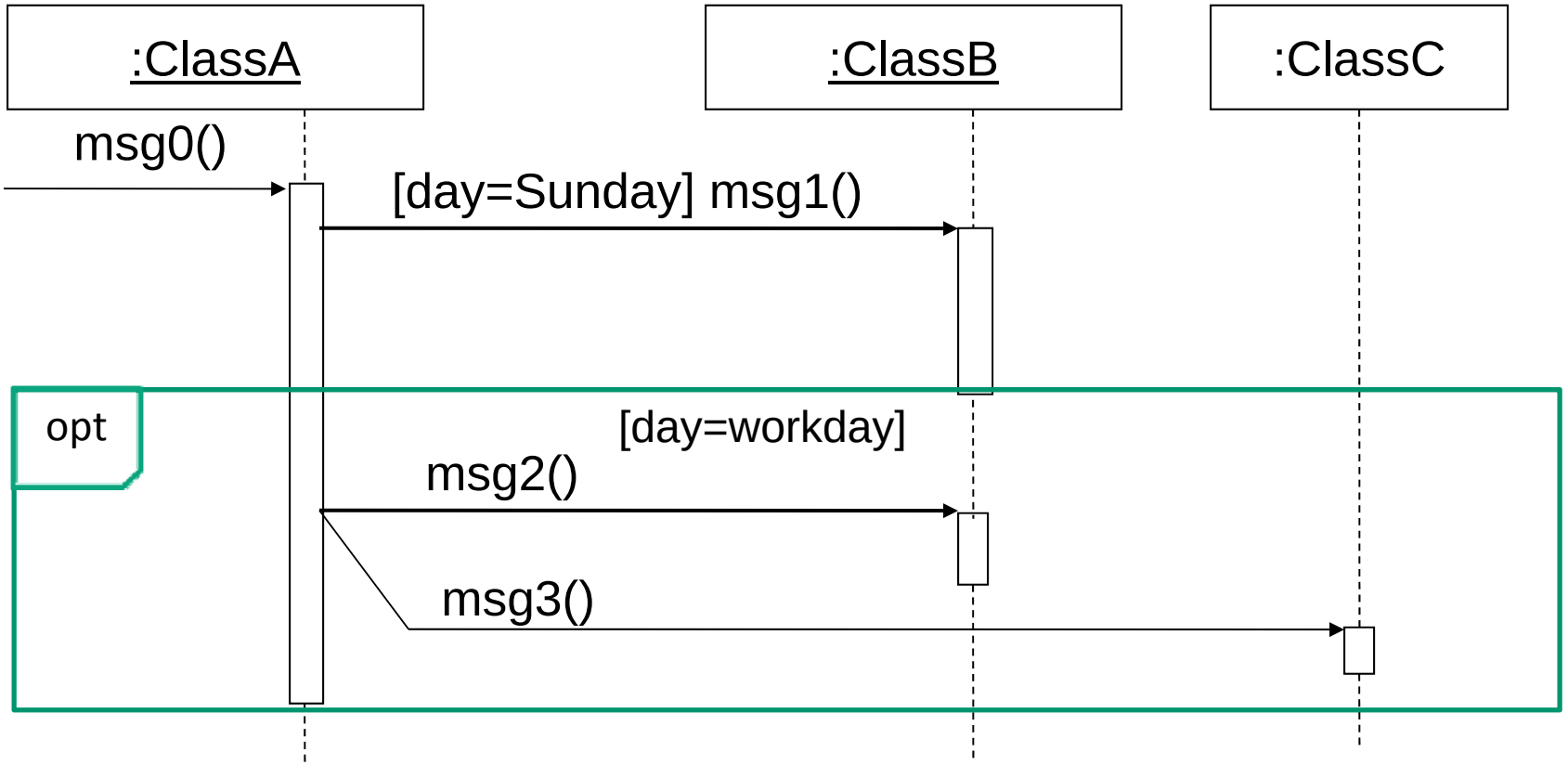
Sequence diagram notation

- Sequence is inferred by order of arrowed lines (top-bottom)
- Activation boxes show message nesting
- Returns are illustrated with dotted lines



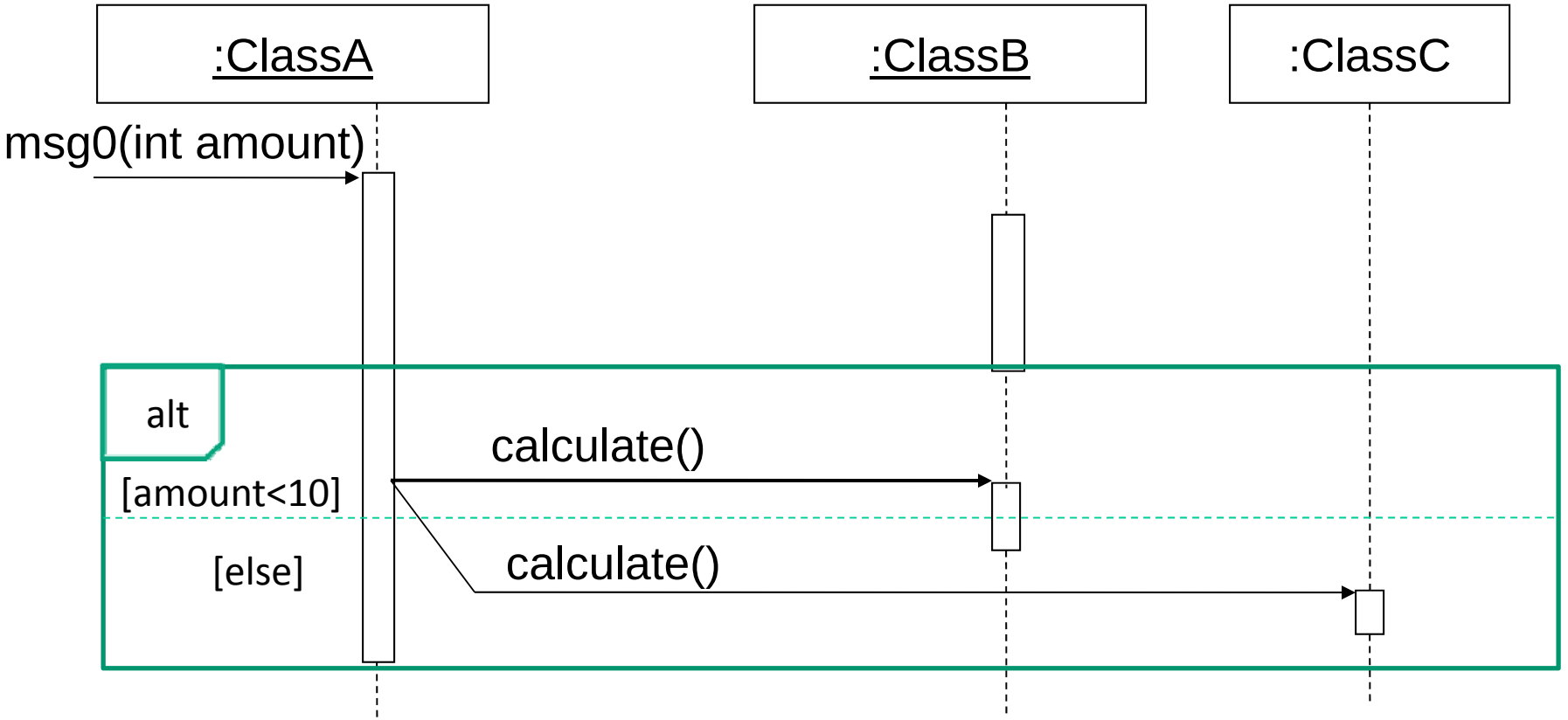
Sequence diagram notation (cont.)

- Conditional messages
- Messages to class objects



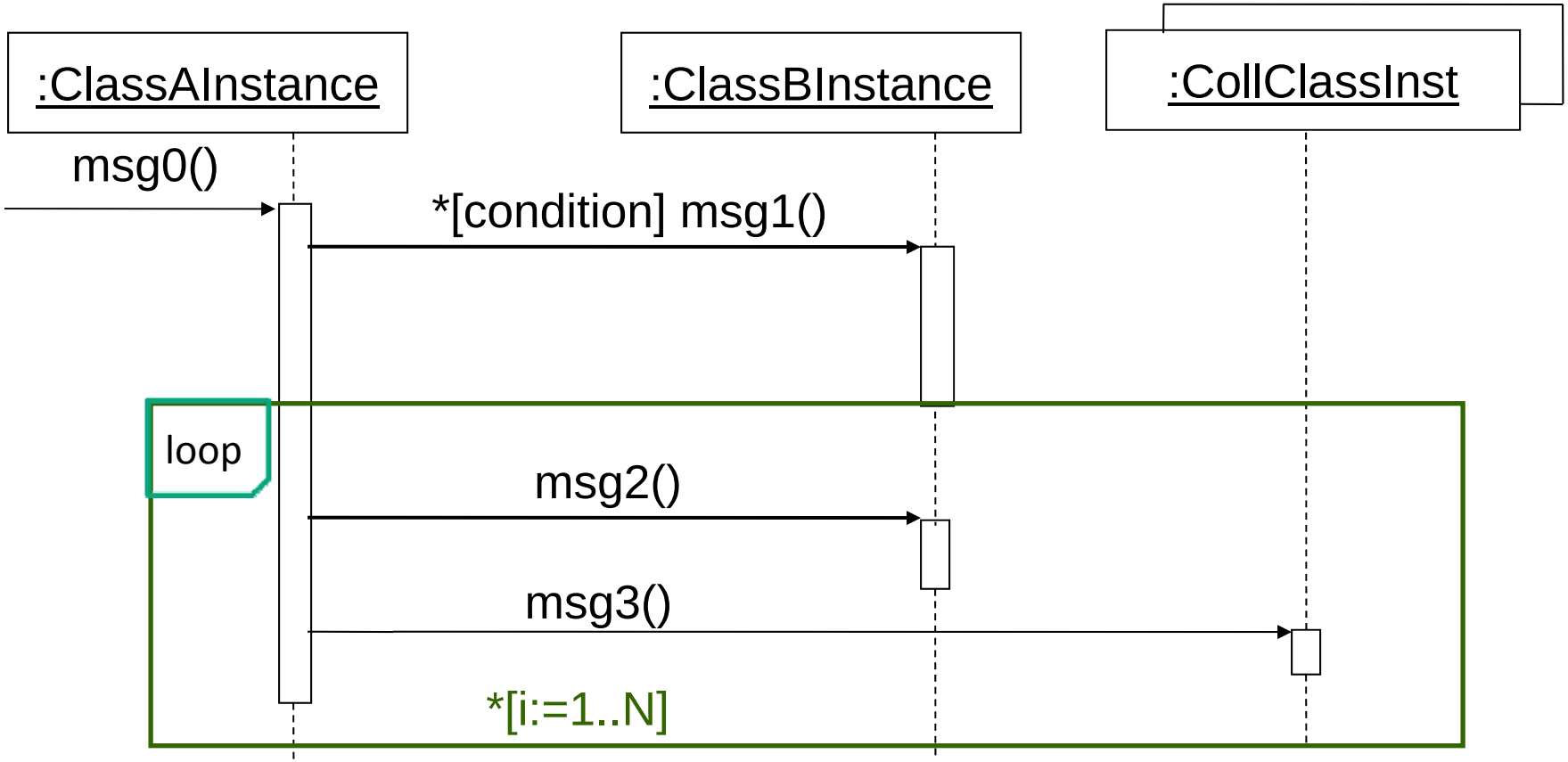
Sequence diagram notation (cont.)

- Conditional messages
- Mutually exclusive conditional messages
- Messages to class objects



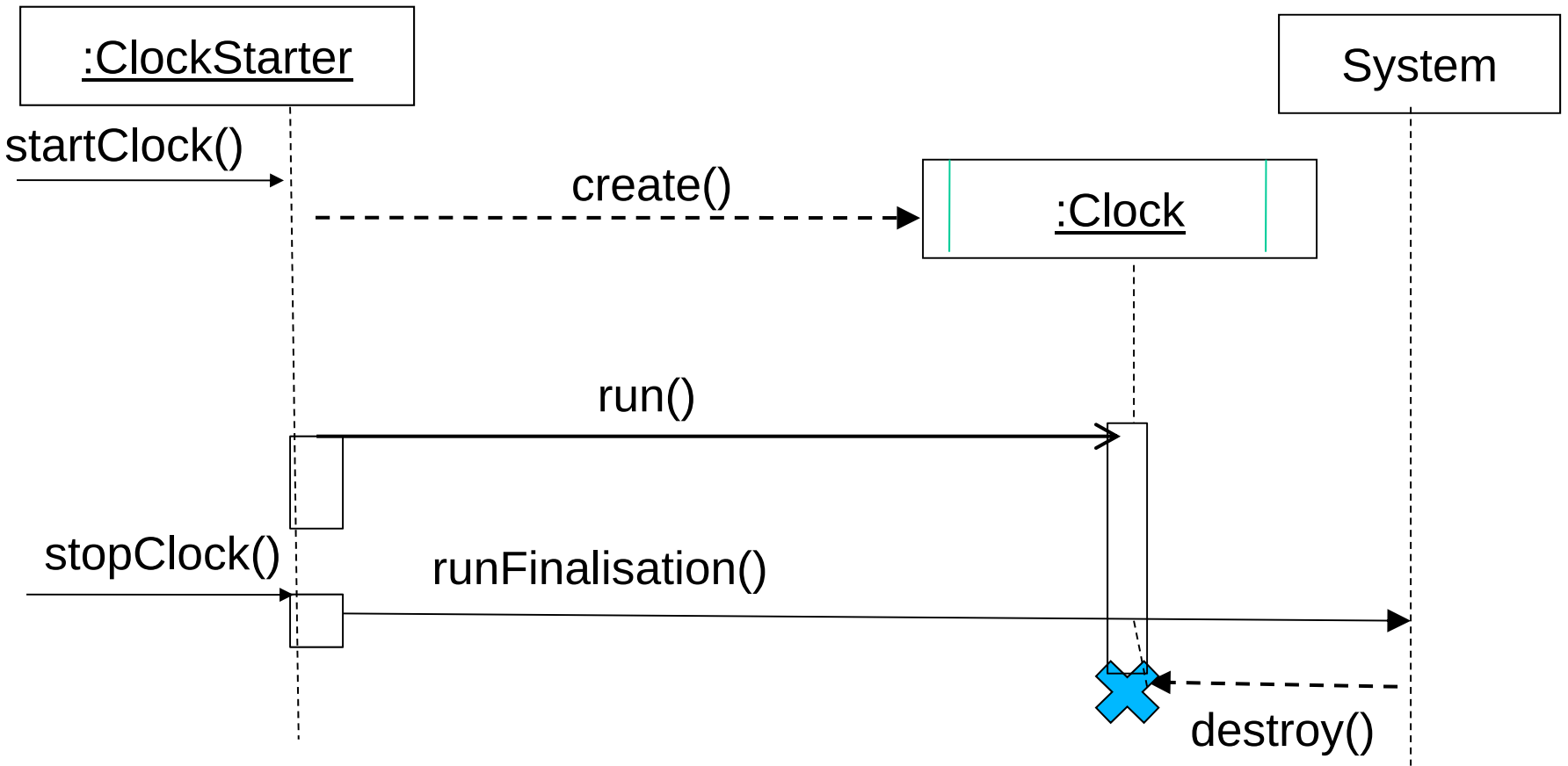
Sequence diagram notation (cont.)

- Iteration of a message
- Iteration of multiple messages
- Iteration over the items of a Collection object



Asynchronous call

```
public class ClockStarter{
public void StartClock()
{
        //Clock implements the runnable interface
        Thread t=new Thread(new Clock());
        t.start(); //asynchronous call to the run method on clock
}}
```



Example: Cellular phone

- Consider the software that controls a very simple cellular telephone.
- Such a phone has buttons for dialling digits, and a “send” button for initiating a call.
- It has “dialler” hardware and software that gathers the digits to be dialled and emits the appropriate tones.
- It has a cellular radio that deals with the connection to the cellular network.
- It has a microphone, a speaker, and a display.

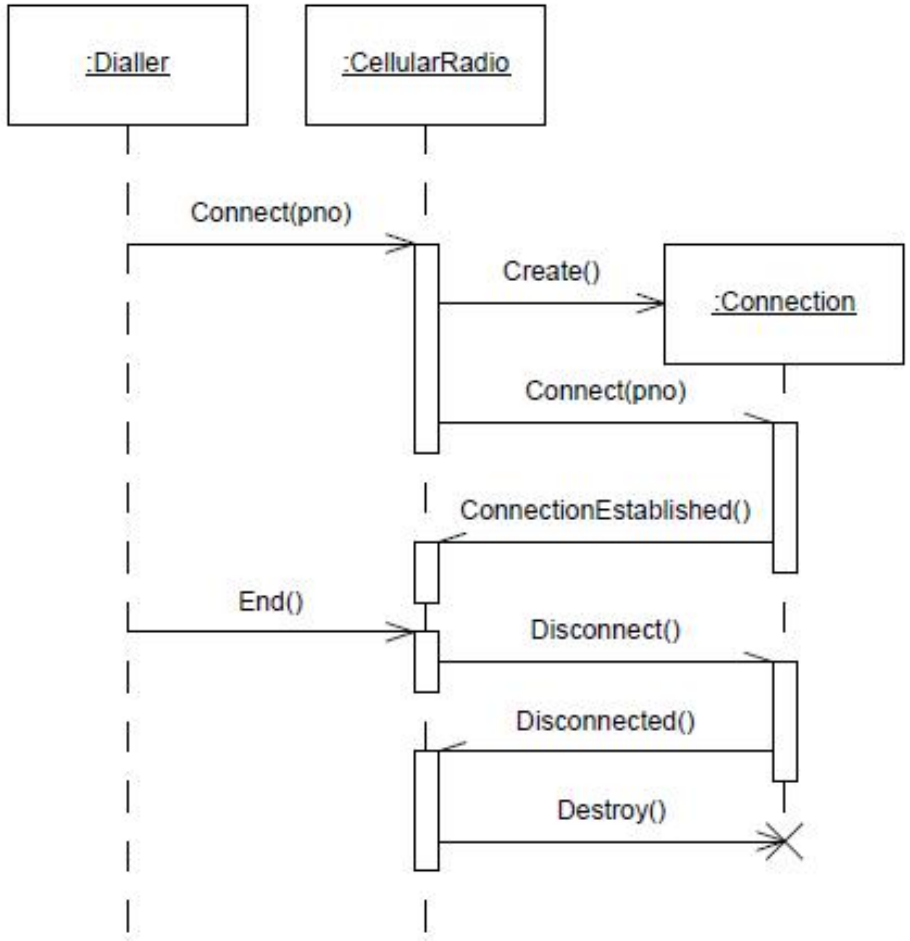
Cellular phone Use Case

Use case: Make Phone Call

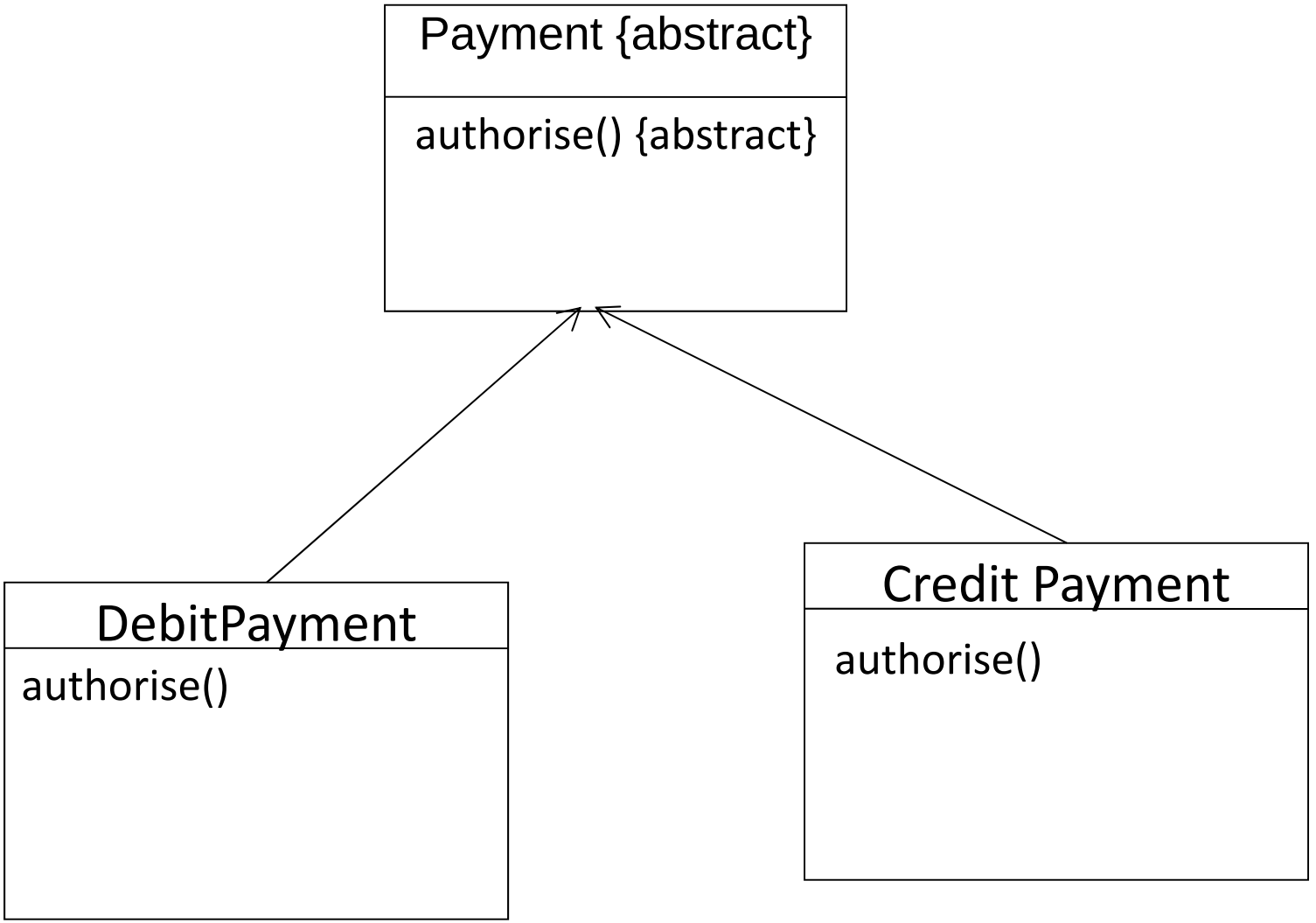
- 1. User presses the digit buttons to enter the phone number.
- 2. For each digit, the display is updated to add the digit to the phone number.
- 3. For each digit, the dialler generates the corresponding tone and emits it from the speaker.
- 4. User presses “Send”
- 5. The “in use” indicator is illuminated on the display
- 6. The cellular radio establishes a connection to the network.
- 7. The accumulated digits are sent to the network.
- 8. The connection is made to the called party.

Cellular Phone: Connection

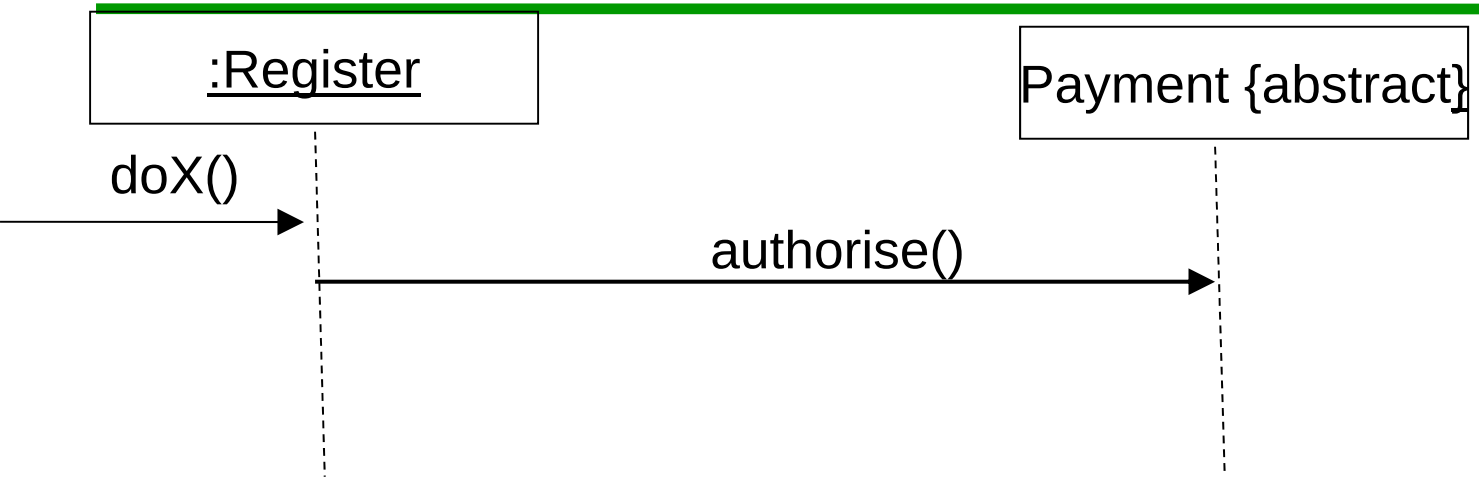
An asynchronous message is a message that returns immediately after spawning a new thread in the receiving object. The Connect message, for instance, returns immediately to the CellularRadio object. Yet you can see by the activation rectangle on the connection lifeline that the Connect method continues to execute. The Connect method is executing in a separate thread. This demonstrates the power that sequence diagrams have for showing concurrent multi-threaded interactions.

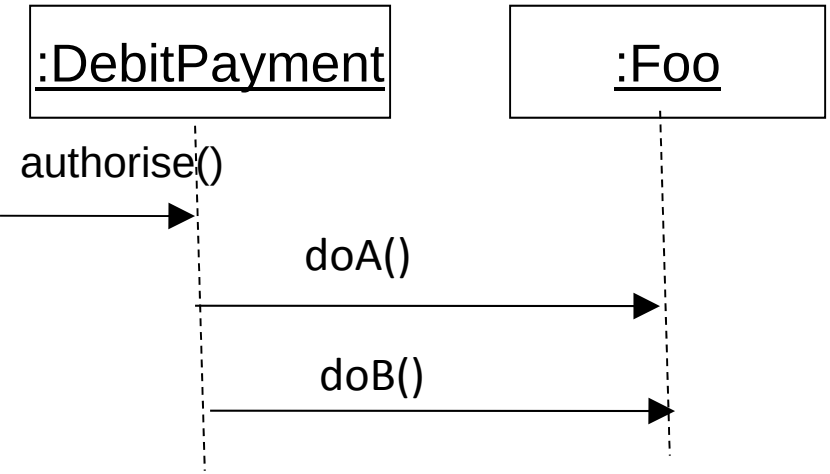


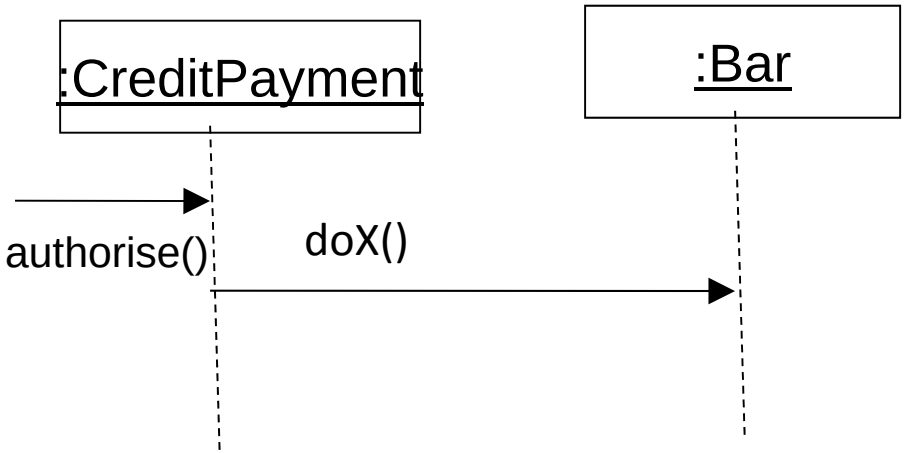
Polymorphism



Inheritance

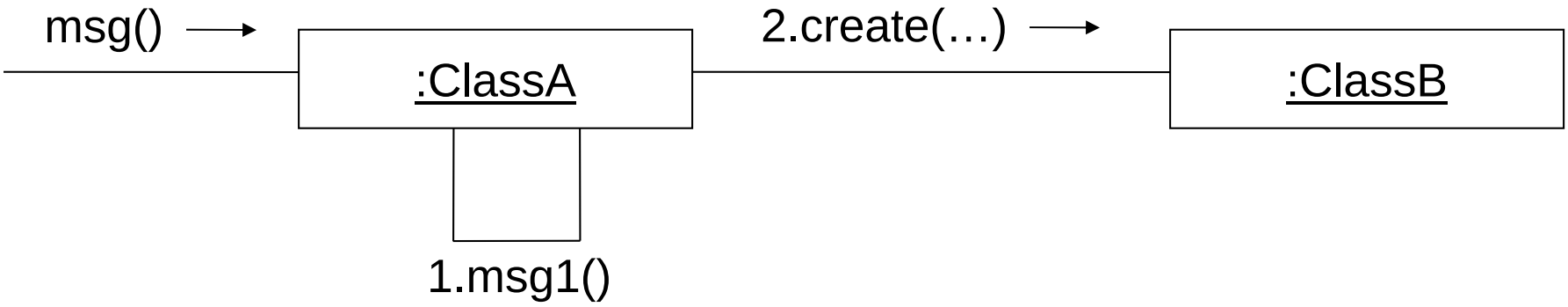






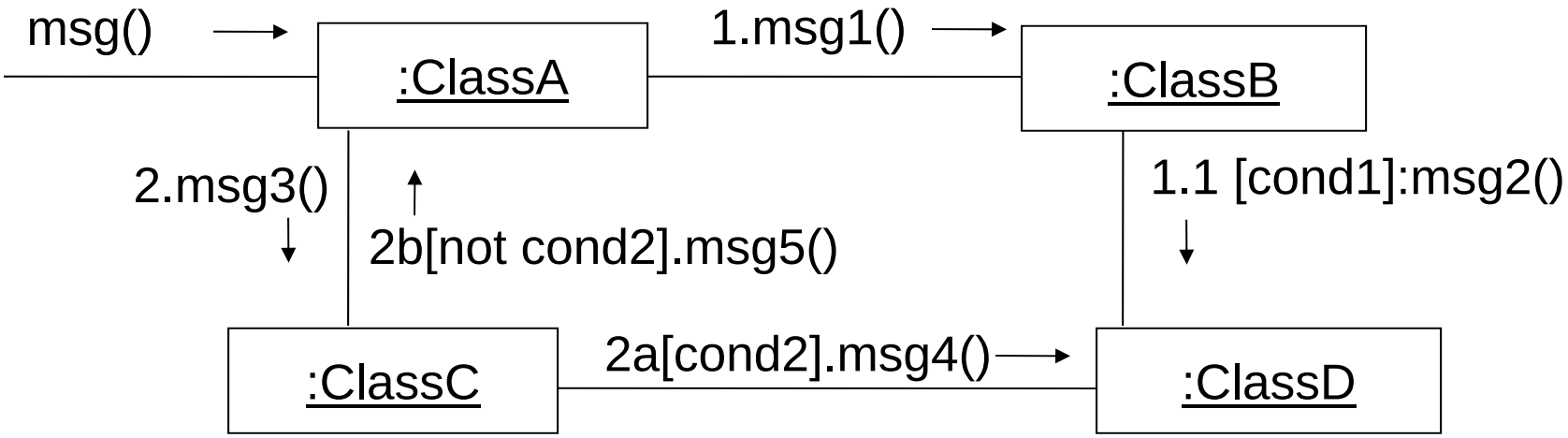
Communication diagram notation

- Message number sequencing
- Message direction
- Creation messages
- Self messages



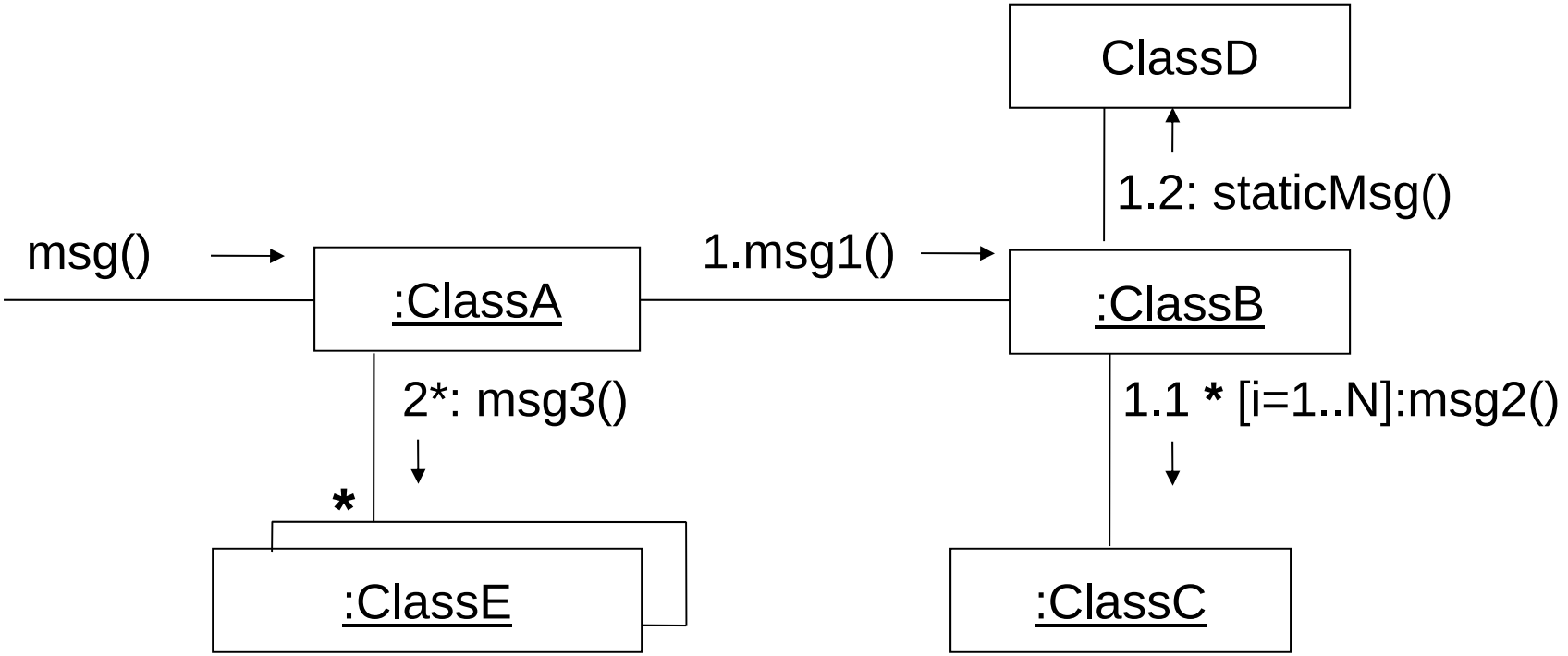
Communication diagram notation (cont.)

- Nesting of messages
- Conditional messages
- Mutually exclusive conditional paths

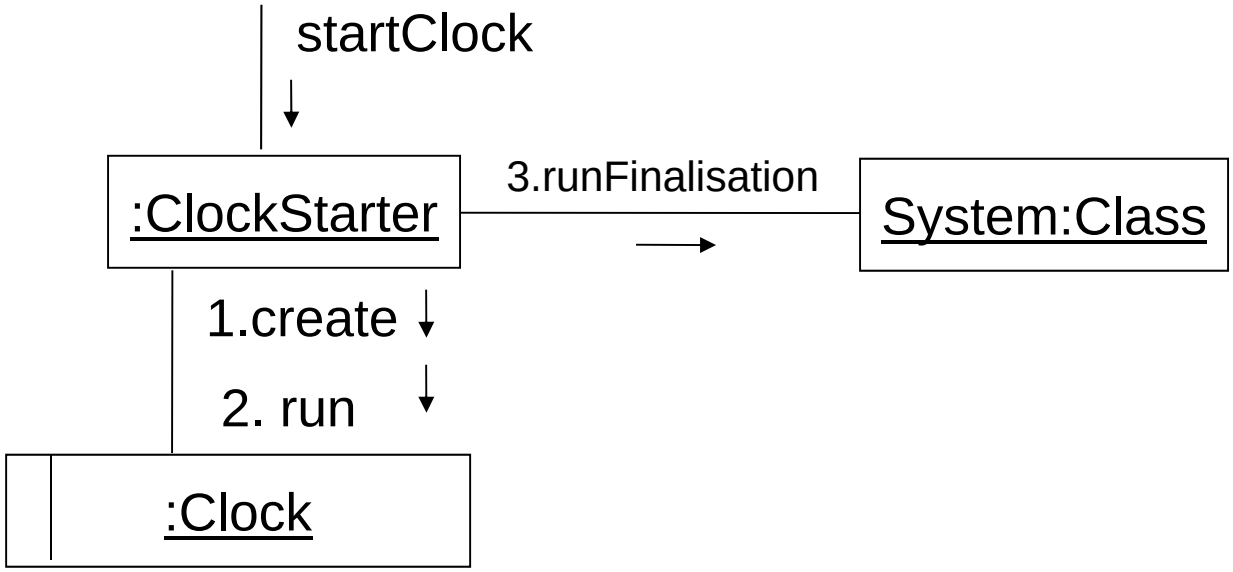


Communication diagram notation (cont.)

- Iteration or looping
- Iteration over a collection
- Messages to a class, rather than an instance



Asynchronous invocation



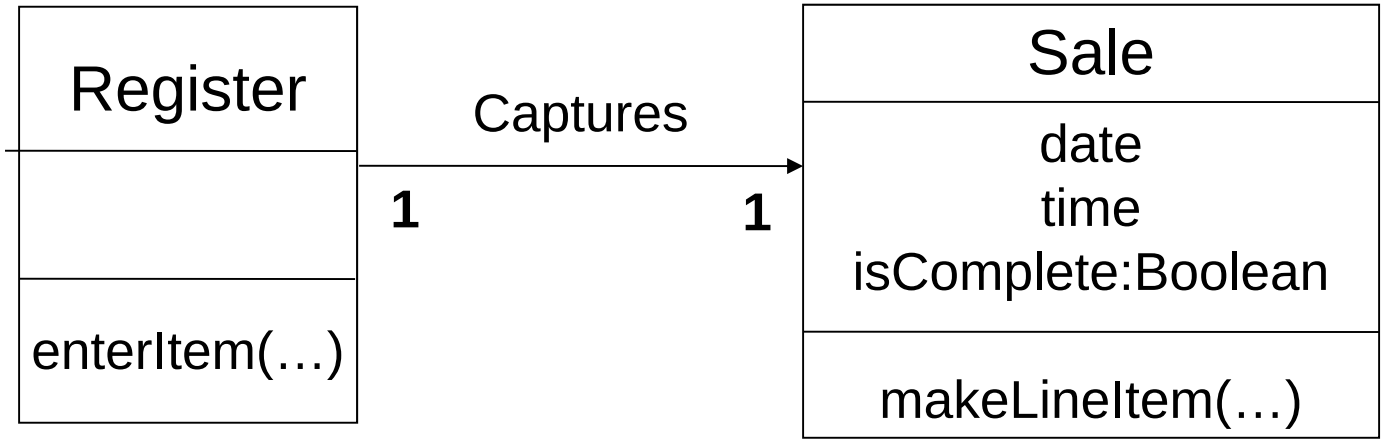
Sequence Diagram Vs. Communication Diagram

Diagram Type	Strength	Weakness
Sequence	1. Forces a clear exposition of order of messages 2. Rich notations	1. Space consuming 2. Rigid format constraints
Communication	1. Space Economical 2. Flexibility to add new objects	1. Less notational options 2. May get messy

Design Class Diagrams

Design Class Diagrams (DCDs)

- DCDs, Design Class Diagrams, provide a static view of the Design Model
- DCDs are created in parallel with Interaction diagrams
- Example DCD:



Design Class Diagrams (DCDs)

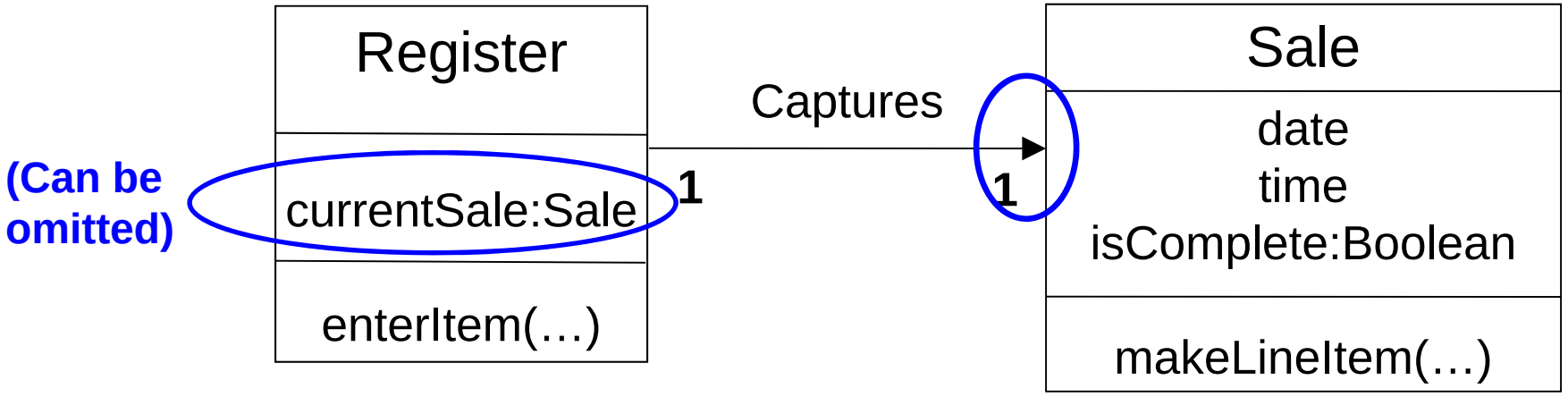
- DCDs illustrate:
  - Classes and interfaces
  - Associations (and navigability)
  - Attributes and often their types
  - Methods, their parameters, and often parameter and method result types
- How are DCDs different from class diagrams of the Domain Model?
- How are DCDs different from interaction diagrams?

How to generate DCDs

- Identify software classes by looking at the Interaction diagrams or the Domain Model class diagrams. Some of the classes in the Domain Model may be omitted.
- Add attributes based on the attributes of the classes in the Domain Model (or the attributes of objects in the Interaction diagrams). In the Design Model, additional attributes may be added (that are not defined in the Domain Model). Types of attributes may also be added.
- Add methods by analyzing the Interaction diagrams. If a message is sent to an instance of a class A, then class A in a DCD must define a method with the same name.
- Add associations by drawing inspiration from the Domain Model and considering *visibility* between objects

Associations in DCDs

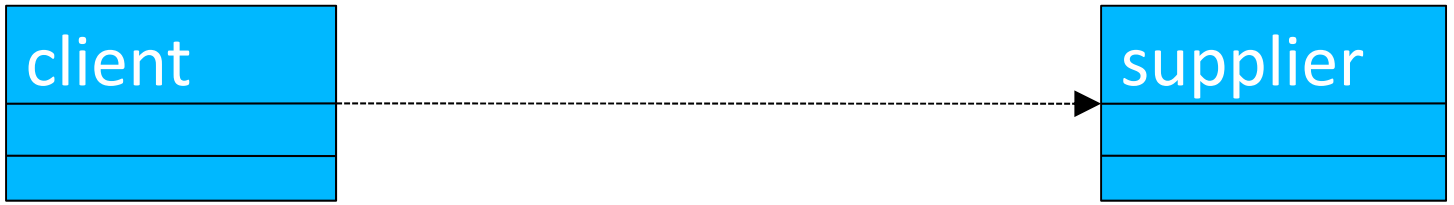
- Associations in DCDs are formed based on attribute visibility:
- Navigability arrows indicate attribute visibility from the source to the destination class. Attributes pointing to visible objects are usually omitted.



Parameter visibility: makeLineItem(ProductSpecification spec, int qty)

Dependency

- In UML modelling, a dependency relationship is a relationship in which changes to one model element (the supplier) impact another model element (the client). You can use dependency relationships in class diagrams, component diagrams, deployment diagrams, and use case diagrams.



Adding dependency relationships

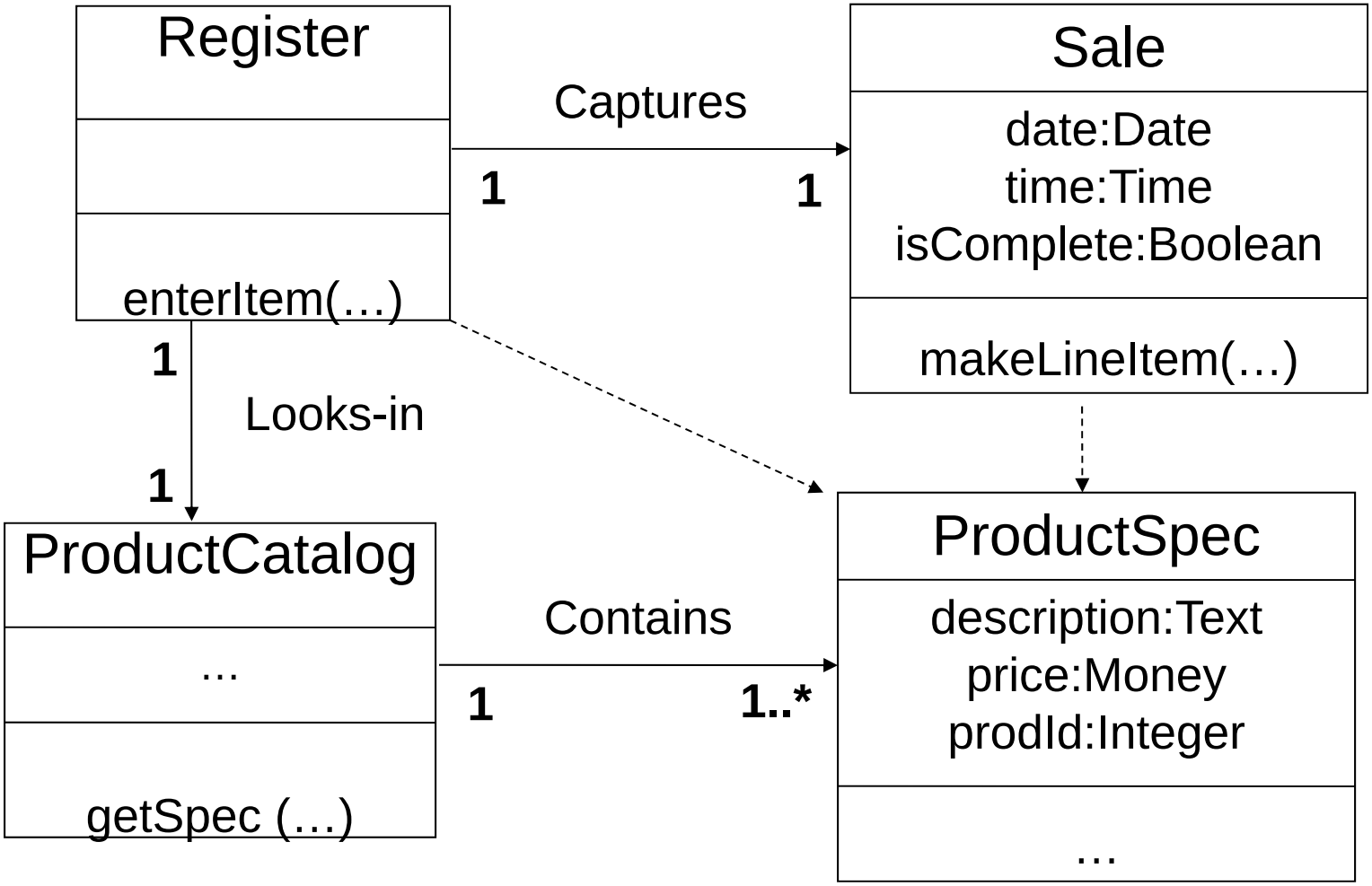
- Dependency relationship indicates that one element (of any kind, including classes, use cases, and so on) has knowledge of another element.
- A dependency is a using relationship that states a change in specification of one thing may affect another thing that uses it, but not necessarily the reverse.

Continue…

- The dependency relationship is useful to depict non-attribute visibility between classes.
  - Parameters
  - Global or local visibility
  - A dashed arrow line 
  - A dashed directed line

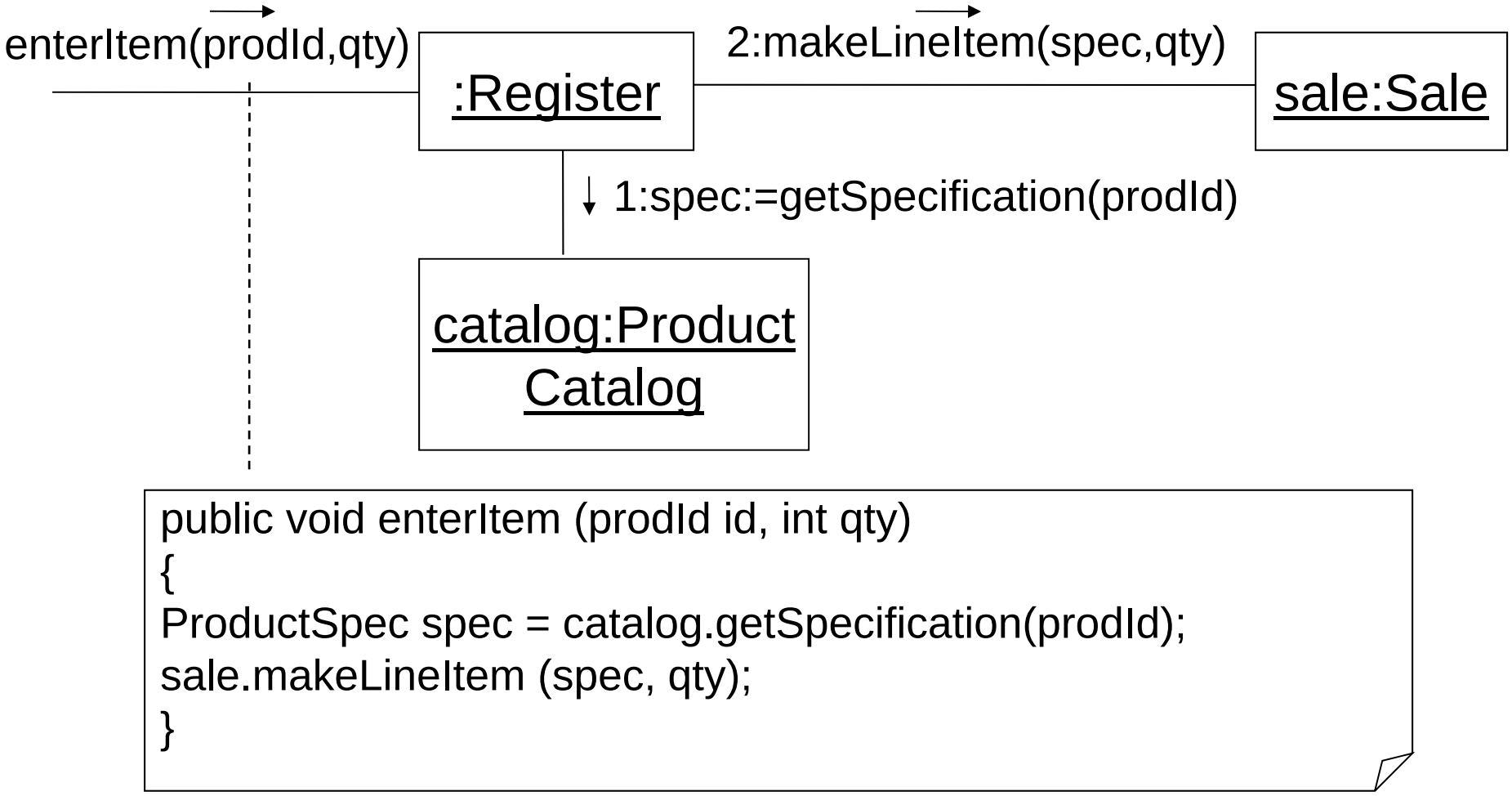
Dependency relationships in DCDs

- Dependency relationships indicate non-attribute visibility and are illustrated with dashed arrow lines.

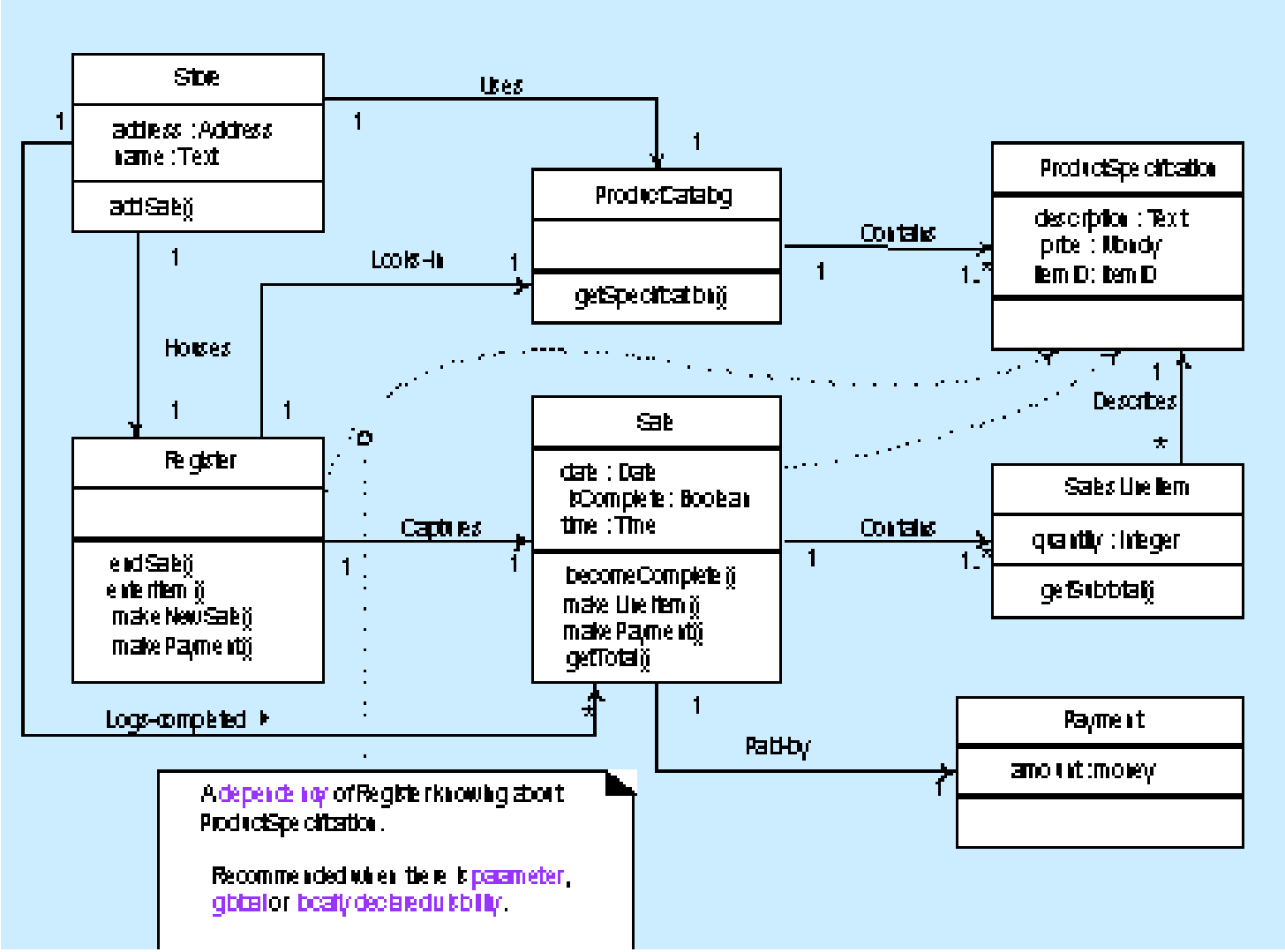


Creating methods from interaction diagrams

- An interaction diagram shows messages sent in response to a method invocation.

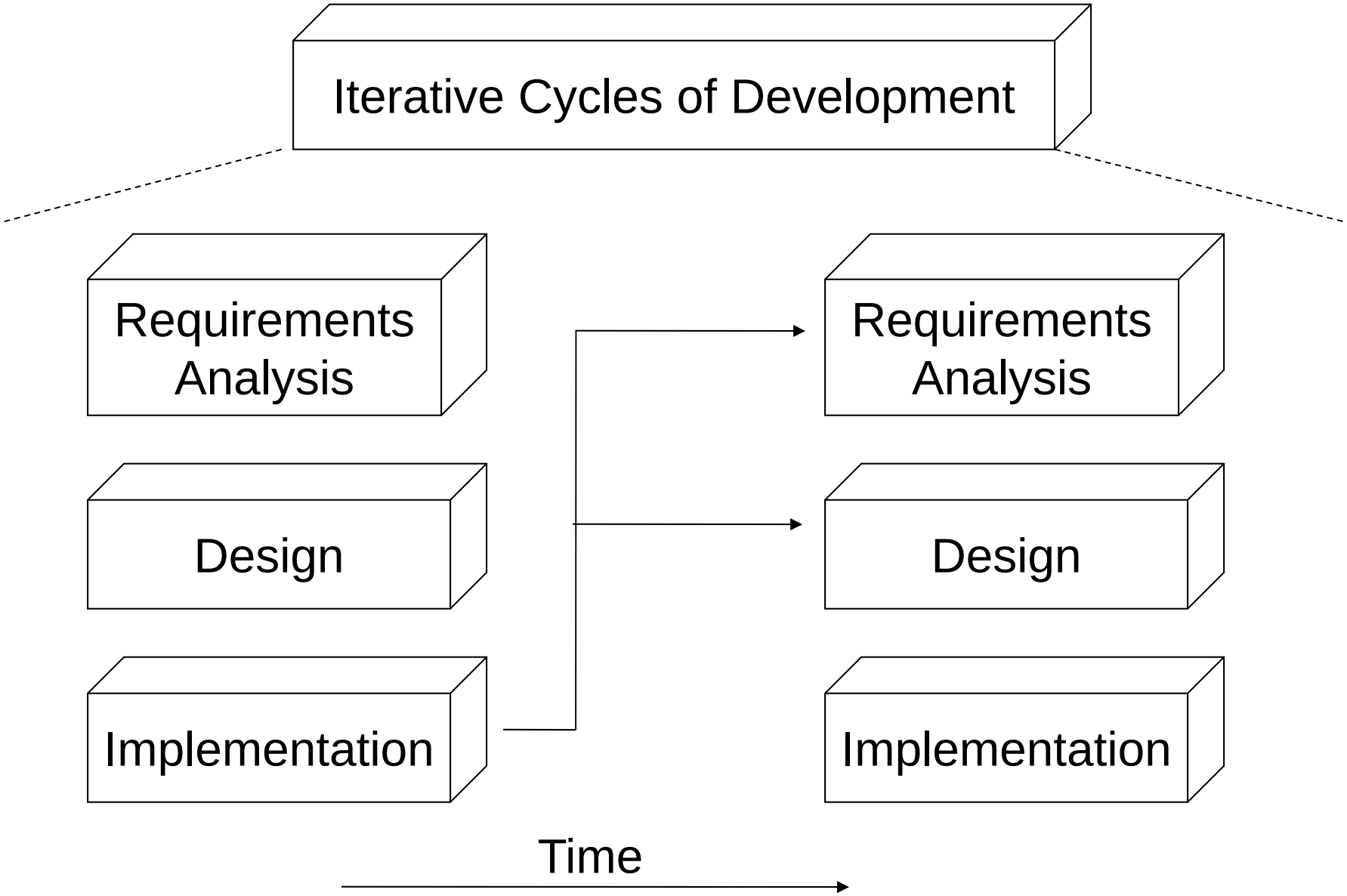


Dependency relationships non-attribute visibility



Implementation Model

Code changes and the iterative model

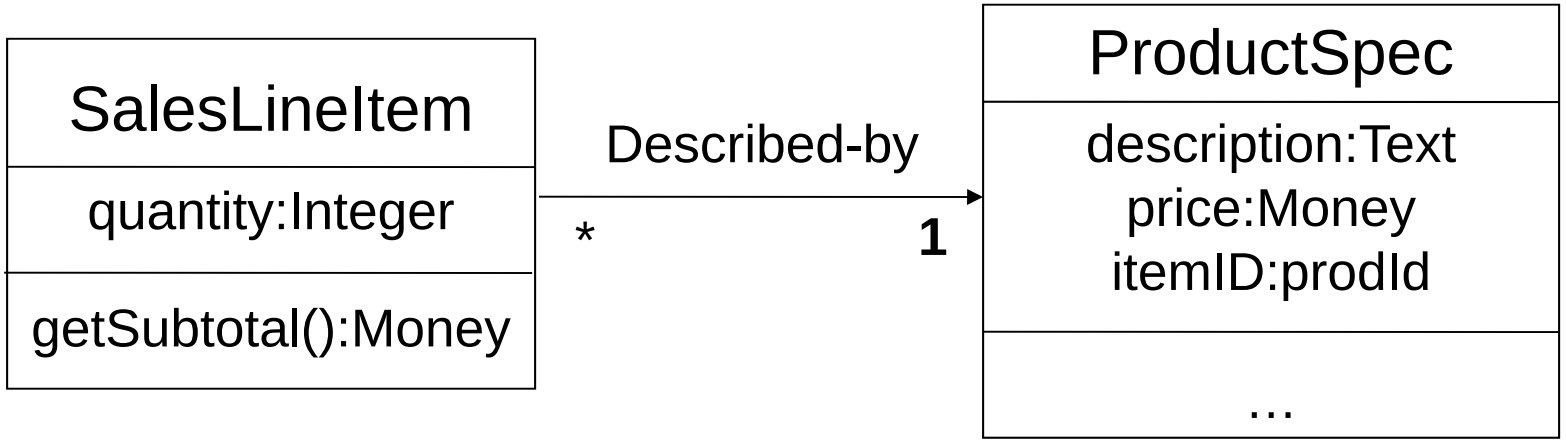


Implementation model

- Interaction diagrams and DCDs: input to code generation
- The Implementation Model includes:
  - source code for classes
  - database schema
  - XML/HTML pages
  - etc.
- Writing code in an OO programming language (like Java) is not considered part of OOA/D
- Most of the creative work took place during OOA/D. Transition to implementation is mostly mechanical
- Expect deviations from the design during programming

Creating class definitions from DCDs

- Straightforward mapping of attribute definitions and method signatures. Don't forget to add constructors.




```
public class SalesLineItem
{
private int quantity;

public SalesLineItem(ProductSpec spec, int qty) { … }
public Money getSubtotal() { … }
}
```

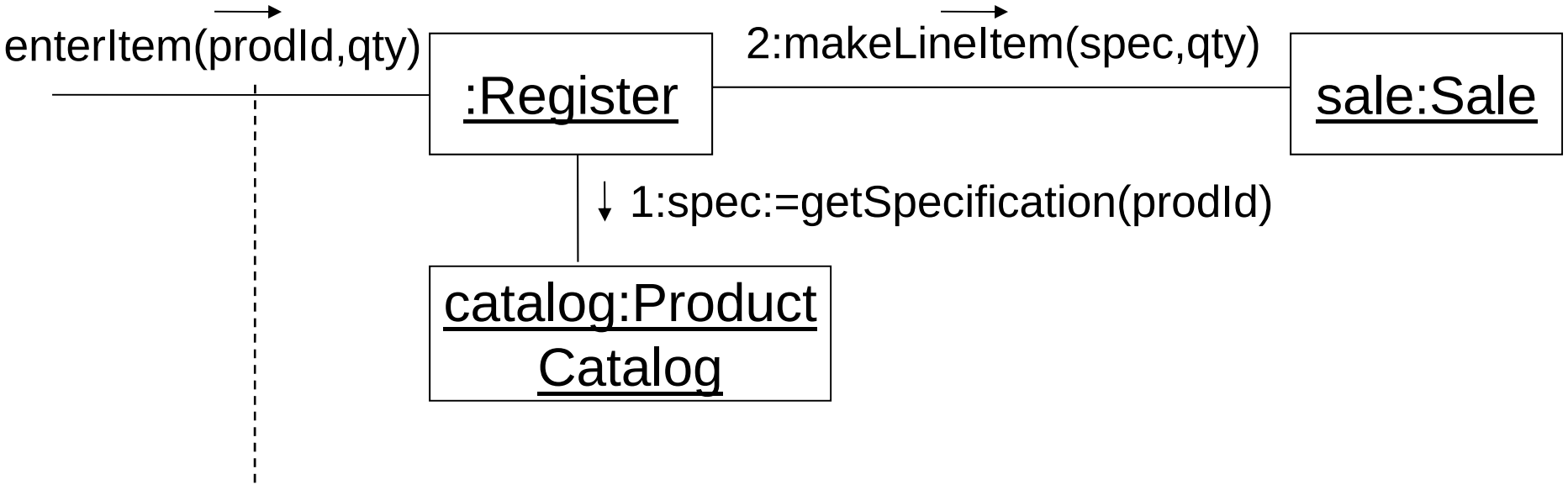
Reference attributes

- A reference attribute is an attribute that refers to another object, not to a primitive type such as String, Number and so on.


```
public class SalesLineItem
{
private int quantity;
private ProductSpec productSpec;
public SalesLineItem(ProductSpec spec, int qty) { … }
public Money getSubtotal() { … }
}
```

Creating methods from interaction diagrams

- An interaction diagram shows messages sent in response to a method invocation.




```
public void enterItem (ItemID prodId, int qty)
{
ProductSpec spec = catalog.getSpecification(prodId);
sale.makeLineItem (spec, qty);
}
```

Now what?

Main design consideration

- A critical task in the Design Model is to **assign responsibilities to objects.**
- The use cases describe different tasks that must be performed in their context. Who is responsible for performing these tasks? Are they going to be performed by one element (object) or delegated to (shared amongst) many elements?
- Responsibilities are obligations of an object in terms of its behaviour. There are two kinds:
  - knowing
  - doing

Object responsibilities

- **Doing** responsibilities of an object include:

  creating an object, doing a calculation, initiating action in other objects, coordinating activities in other objects

- **Knowing** responsibilities of an object include:

  knowing about private data, about related objects, about things it can derive and calculate

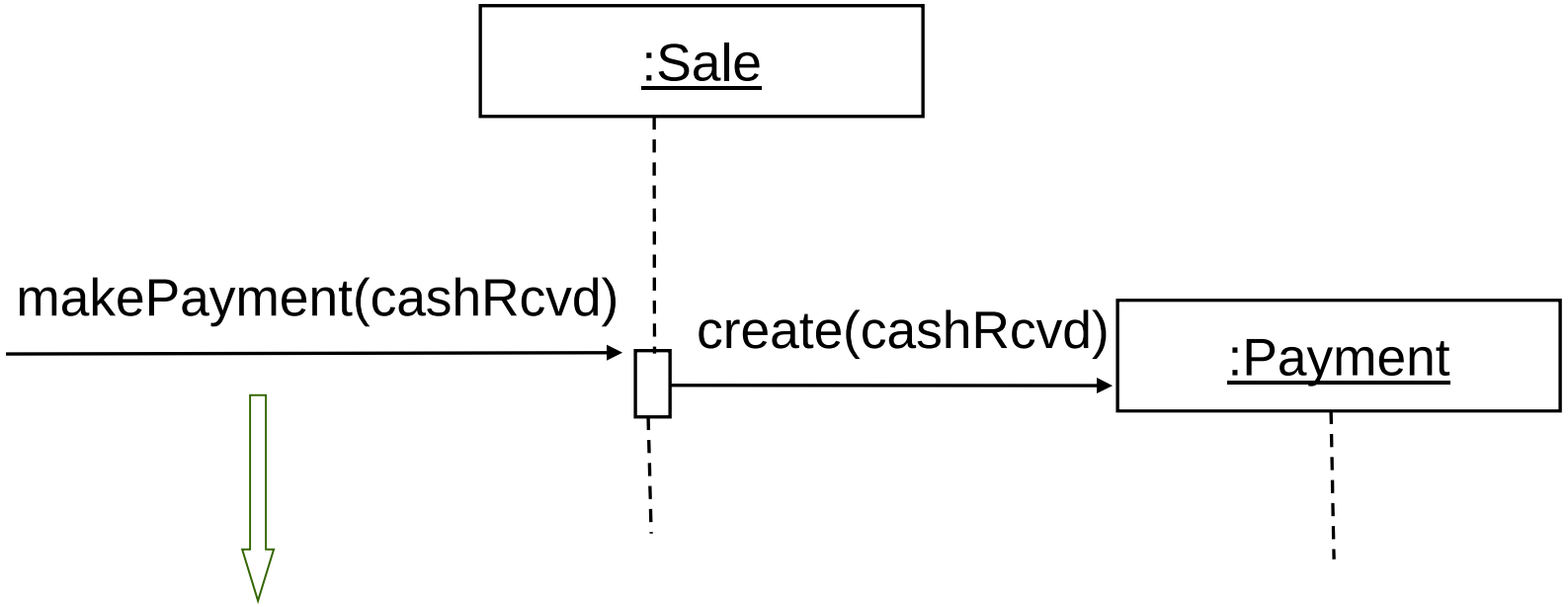
- Examples:
  - "A Sale is responsible for creating SalesLineItem"
  - "A Sale is responsible for knowing its total"

Responsibilities and methods

- In general, responsibilities are implemented using methods.
- Methods either act alone or collaborate with other object methods.
- Depending on its granularity, a responsibility may correspond to a single method in a class, or it may involve many methods across many classes packaged in a subsystem.

Responsibilities and interaction diagrams

- How do we assign responsibilities (implemented as methods) to objects? Interaction diagrams show choices in assigning responsibilities.



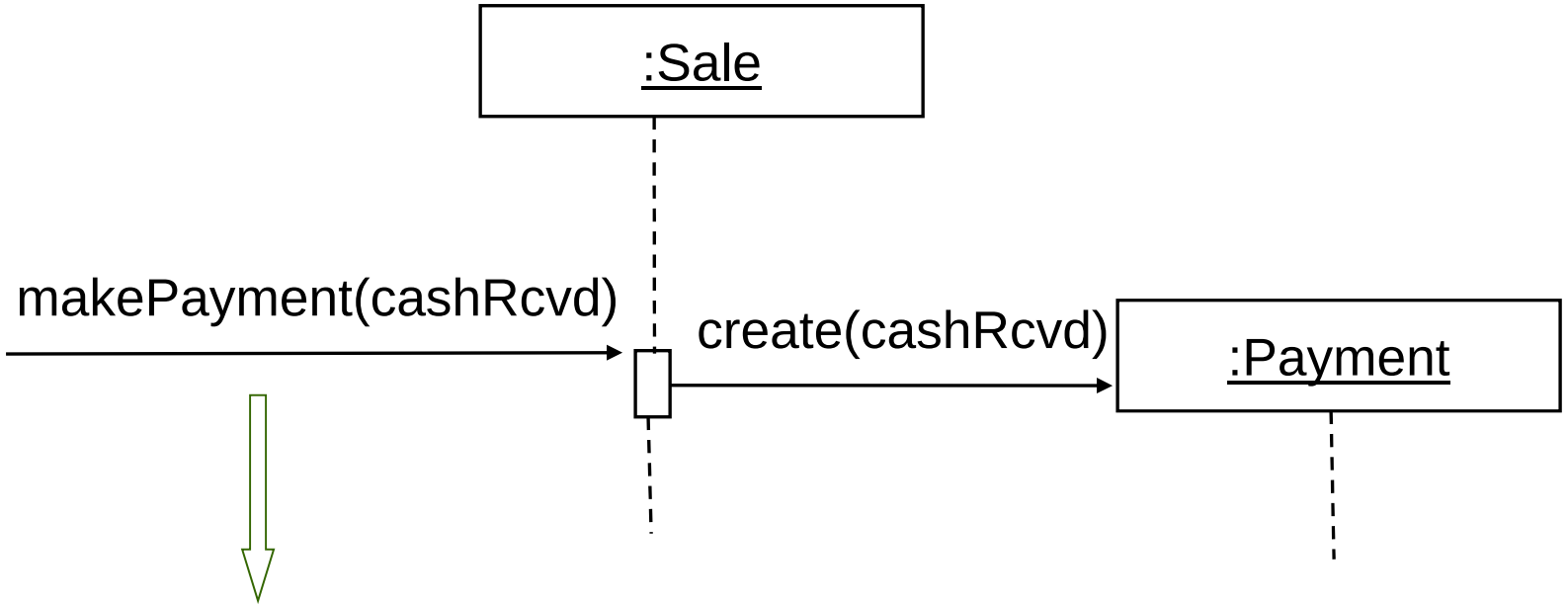
It implies Sale objects have a responsibility to create Payments. This responsibility is invoked with a makePayment message and handled with a corresponding makePayment method.

Responsibilities and methods

- In general, responsibilities are implemented using methods.
- Methods either act alone or collaborate with other object methods.
- Depending on its granularity, a responsibility may correspond to a single method in a class, or it may involve many methods across many classes packaged in a subsystem.
- Object collaboration often results in dependencies.

Responsibilities and interaction diagrams

- How do we assign responsibilities (implemented as methods) to objects? Interaction diagrams show choices in assigning responsibilities.



It implies Sale objects have a responsibility to create Payments. This responsibility is invoked with a makePayment message and handled with a corresponding makePayment method.

Patterns

- Patterns guide object-oriented developers in the creation of software.
- They codify existing tried-and-true knowledge and principles; the more widely they are used, the better.
- They are expressed in a standard format:

  Pattern Name: …

  Solution: …

  Problem It Solves: …

Pattern Example: The Information Expert

- **Pattern Name:** Information Expert

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

  Problem It Solves: What is a general principle of assigning responsibility to objects?

  Consequences: These are the results and trade-offs of applying the pattern. Critical for evaluating design decisions in terms of reusability , flexibility, portability.

Why are patterns useful?

- Patterns are very simple to use.
- They are applicable in many contexts; in fact, the notion of a new pattern could be considered an oxymoron.
- If one learns how to apply them properly, s/he will generate designs that are easy to understand, maintain and reuse.
- By referring to pattern names, designers can communicate easily and explain the reasons behind their design choices.
- Patterns provide a formal framework for generating good designs => avoid hand-waving.