

CSSE2003

Software Engineering Studio

Semester 2, 2009

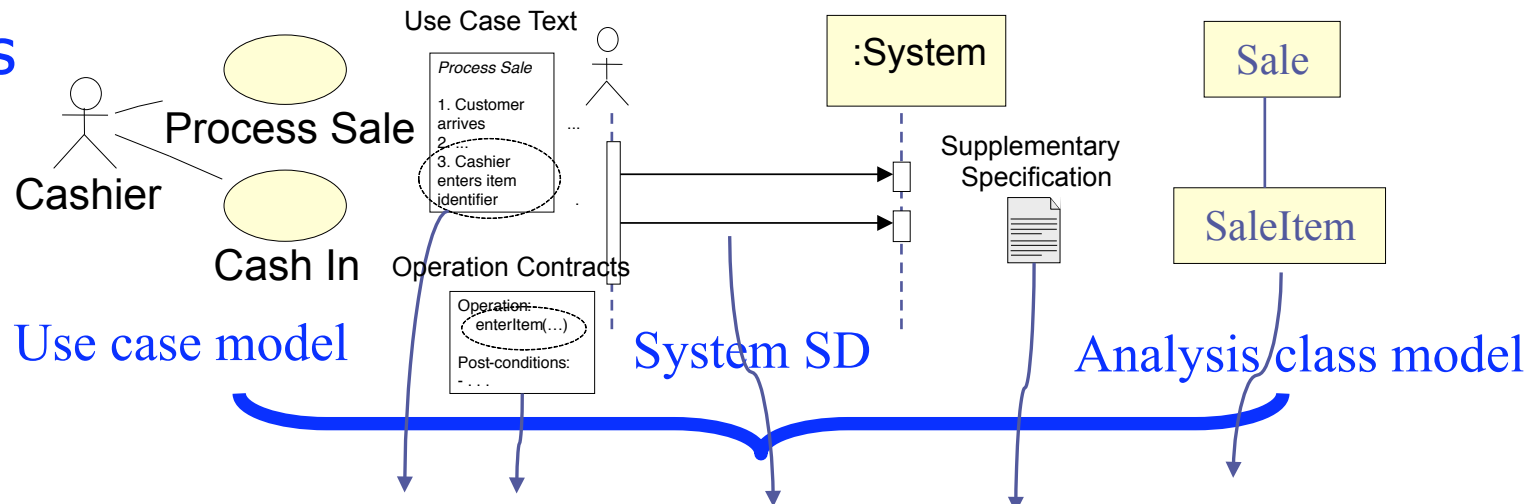
11: Design Guidelines

Lecture Summary

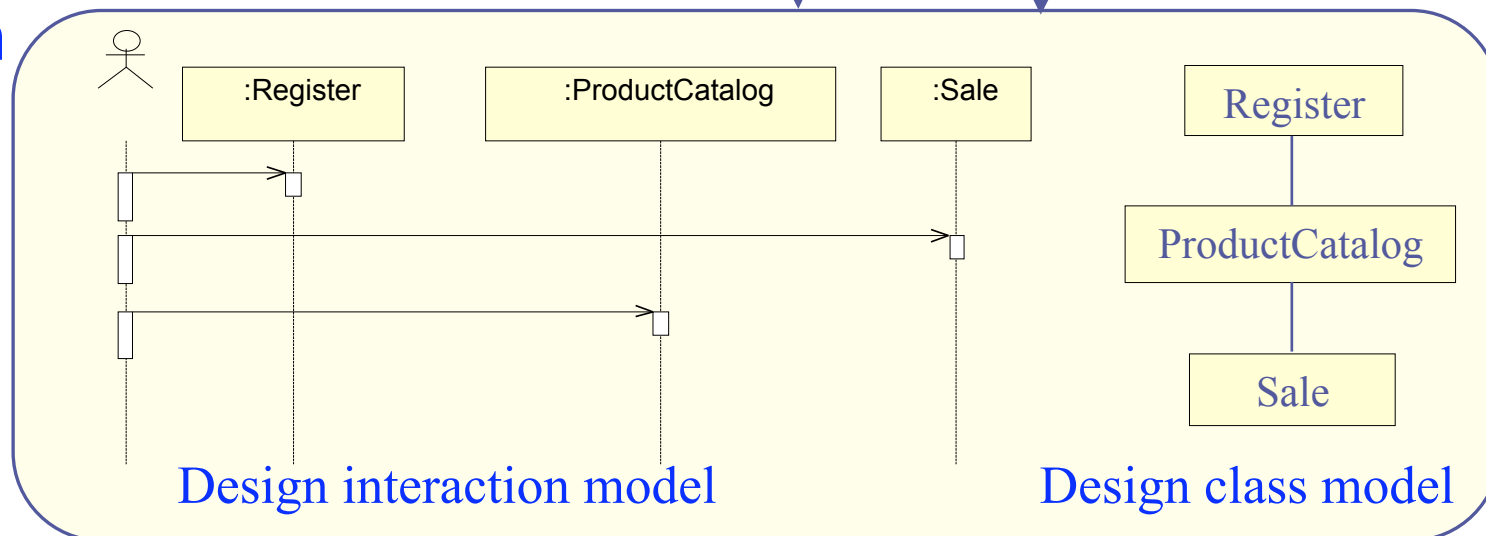
- ◆ Assigning responsibilities to objects
- ◆ GRASP patterns:
 - Expert
 - Creator
 - Controller
 - Low Coupling
 - High Cohesion
- ◆ Designing for visibility
- ◆ Creating design class diagrams

Object-Oriented Analysis and Design?

Analysis



Design



UML: What's Important?



Just a diagramming notation standard.
Not a method, process, or design guide.

Harmful is knowing how to
read and draw UML diagrams,
but not being an expert in
design and patterns.



Important is object and architectural design skills, not UML
diagrams, drawing, or CASE tools.
A critical skill is designing or thinking in objects. This *can* be
practiced based on explainable principles.

Responsibility-Driven Design (RDD)

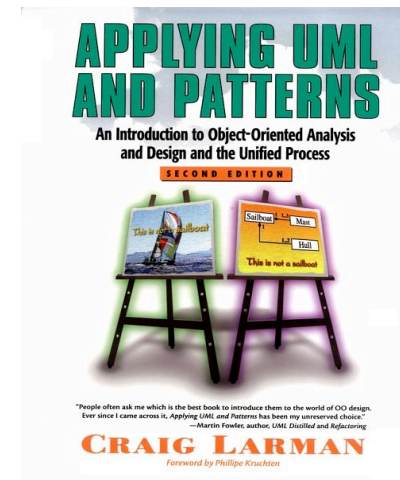
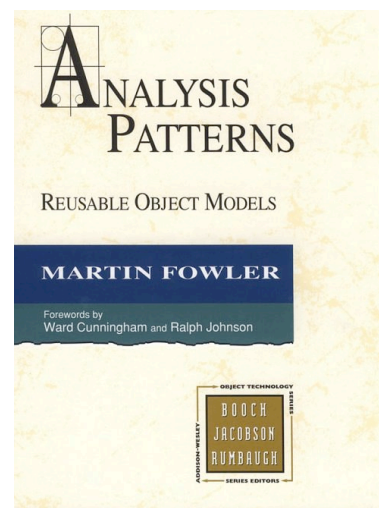
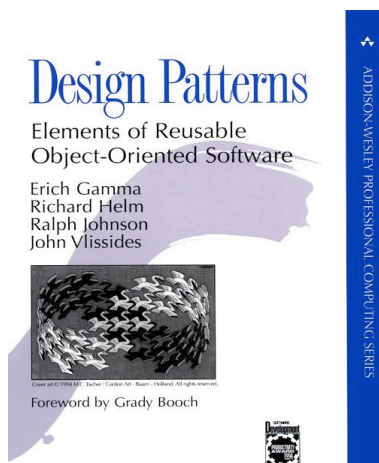
- ◆ Detailed object design is usually done from the point of view of the *metaphor* of:
 - Objects have responsibilities
 - Objects collaborate
- ◆ In RDD, we do object design such that we ask questions such as:
 - What are the responsibilities of this object?
 - Who does it collaborate with?

DEFINITION: Responsibilities

- ◆ Responsibilities are an abstraction.
 - The responsibility for persistence (e.g. database)
 - ◆ Large-grained responsibility
 - The responsibility for the sales tax calculation
 - ◆ More fine-grained responsibility
- ◆ They are implemented with methods in objects.
 - 1 method in 1 object
 - 5 methods in 1 object
 - 50 methods across 10 objects

DEFINITION: Patterns

- ◆ Patterns are *named* problem-solution pairs to common problems, typically showing a popular, robust solution.
 - “Façade” “Information Expert” ...
- ◆ They provide a *vocabulary* of design.



GRASP Patterns

- ◆ **General Responsibility Assignment Software Patterns.**
- ◆ Guiding principles to help us assign responsibilities.
- ◆ Very fundamental, basic principles of object design.
- ◆ 5 core GRASP patterns:
 - Expert
 - Creator
 - Controller
 - Low Coupling
 - High Cohesion



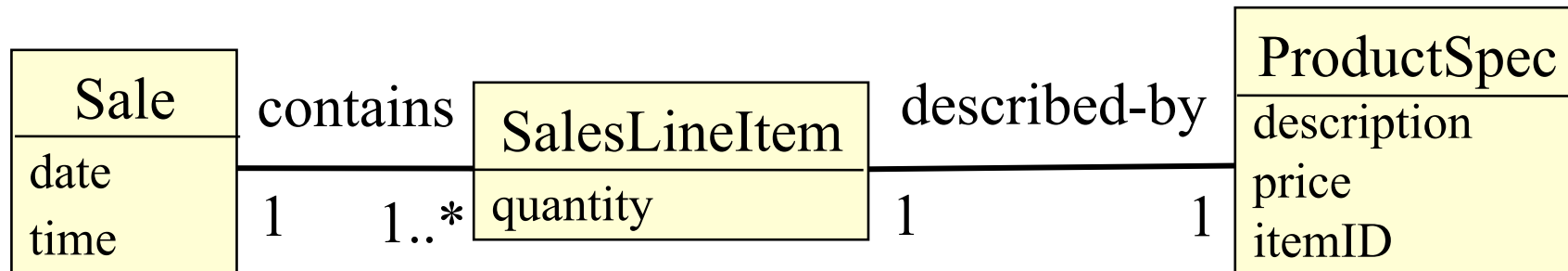
Memorizing and applying these is the most important.

Types of Responsibilities

- ◆ **Doing** responsibilities of an object:
 - doing something itself
 - initiating action in other objects
 - controlling and coordinating activities in other objects
- ◆ **Knowing** responsibilities of an object:
 - knowing about private encapsulated data
 - knowing about related objects
 - knowing about things it can derive or calculate

PATTERN: Information Expert

- ◆ What is most basic, general principle of responsibility assignment?
- ◆ Assign a responsibility to the object that has the information necessary to fulfill it.
- ◆ *What should record the grand total of a sale?*



- ◆ To calculate a total, all SalesLineItems must be known and the price of each item must be known.

A Design Solution - Expert

- ◆ The responsibility goes to Sale because Sale records all SalesLineItems: method getTotal()
- ◆ Introduces a responsibility on SalesLineItem because SalesLineItem records the quantity of each item: method getSubtotal()
- ◆ Introduces a responsibility on ProductSpec because ProductSpec records the price of each item: method getPrice()

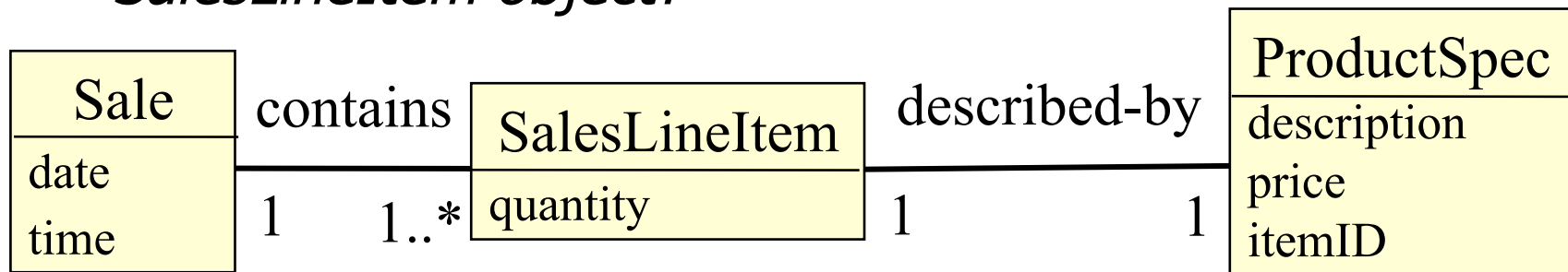
Sale
date
time
getTotal()

SalesLineItem
quantity
getSubtotal()

ProductSpec
description
price
itemID
getPrice()

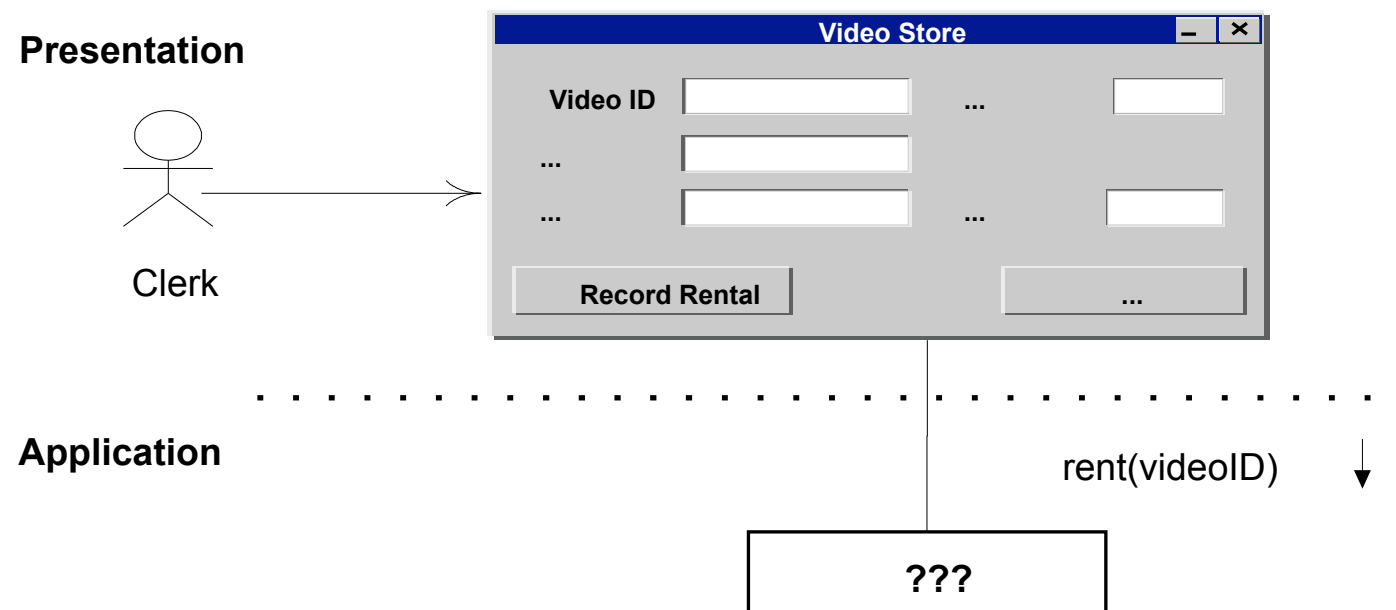
PATTERN: Creator

- ◆ What object creates an X?
 - Ignores special-case patterns such as *Factory*.
- ◆ Choose an object C, such that:
 - C contains or aggregates X
 - C closely uses X
 - C has the initializing data for X
- ◆ *What should be responsible for creating a SalesLineItem object?*



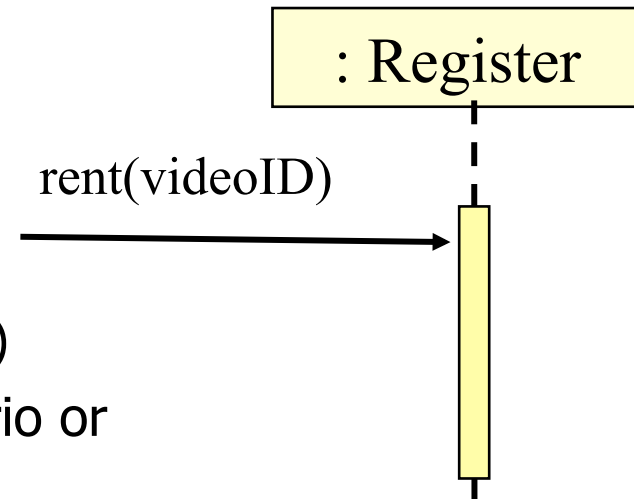
PATTERN: Controller

- ◆ A system operation is generated by an actor, e.g. clicking a mouse button.
- ◆ *What object in the domain (or application coordination layer) receives requests for work from the UI layer?*



A Design Solution - Controller

- ◆ Choose a class whose name suggests:
 - The overall "system," device, or subsystem (A kind of Façade class)
 - Or, represents the use case scenario or session
- ◆ Responsibility for a system operation should not be assigned to a GUI class. Potential reuse is improved by making GUI classes a Facade that delegates to a controller object.
- ◆ A controller object should, in turn, delegate to and coordinate other objects (for cohesion).



Signs of a Poor Controller

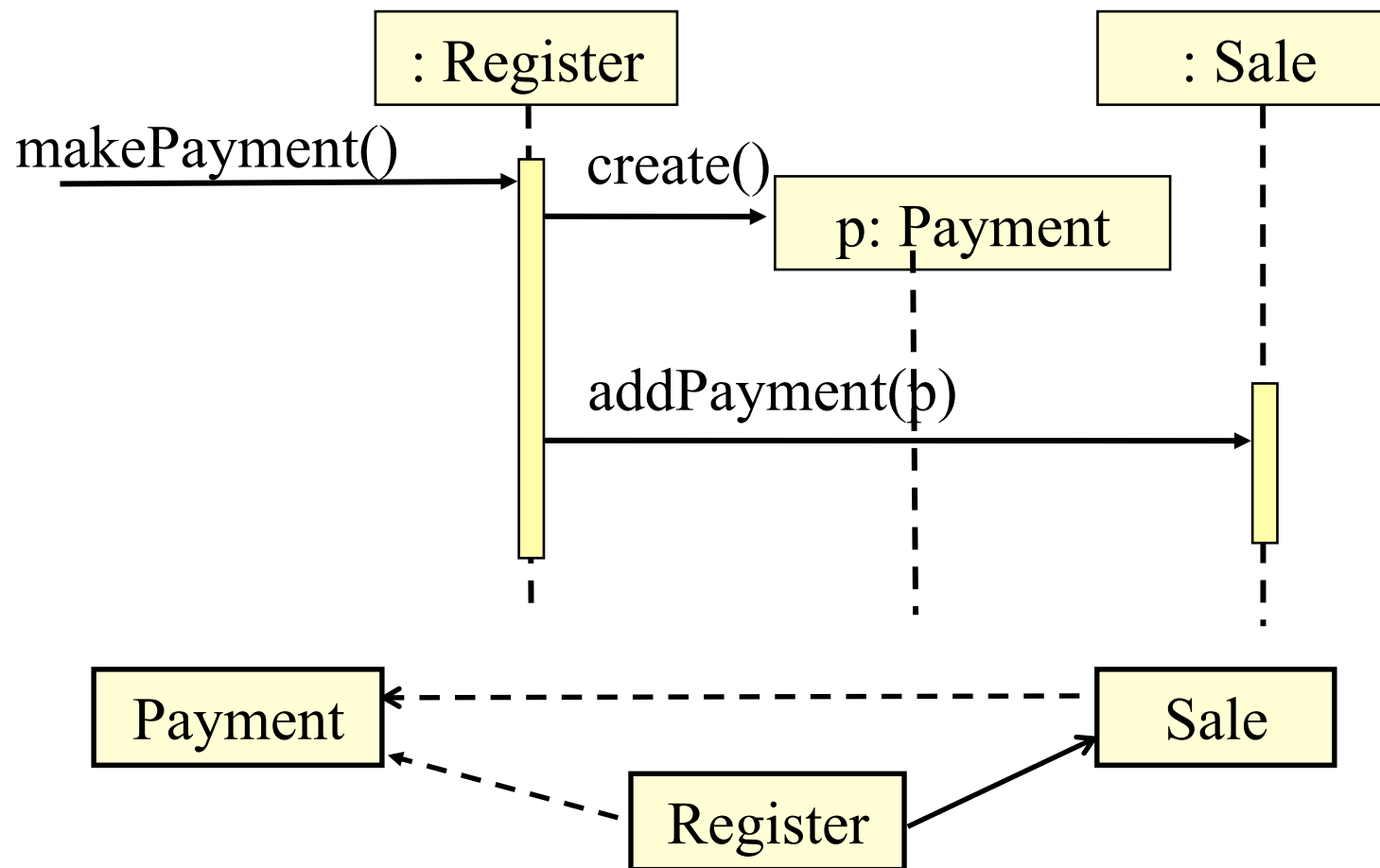
- ◆ A single controller object that handles all system operations (when there are many).
- ◆ A controller object that does not delegate to other objects (low cohesion).
- ◆ A controller object that records information that should be distributed to other objects, or that duplicates such information (avoid duplicating volatile information).

PATTERN: Low Coupling

- ◆ Coupling is a measure of how dependent one class is upon other classes.
- ◆ Coupling is necessary to support object interaction but loose-coupling supports design modularity.
- ◆ Assign responsibilities so that coupling remains low.
- ◆ A highly coupled class makes it harder to:
 - understand the class in isolation
 - reuse the class since it depends on many others
 - isolate the effect of change during system maintenance
- ◆ *What should create a Payment object and associate it with the Sale?*

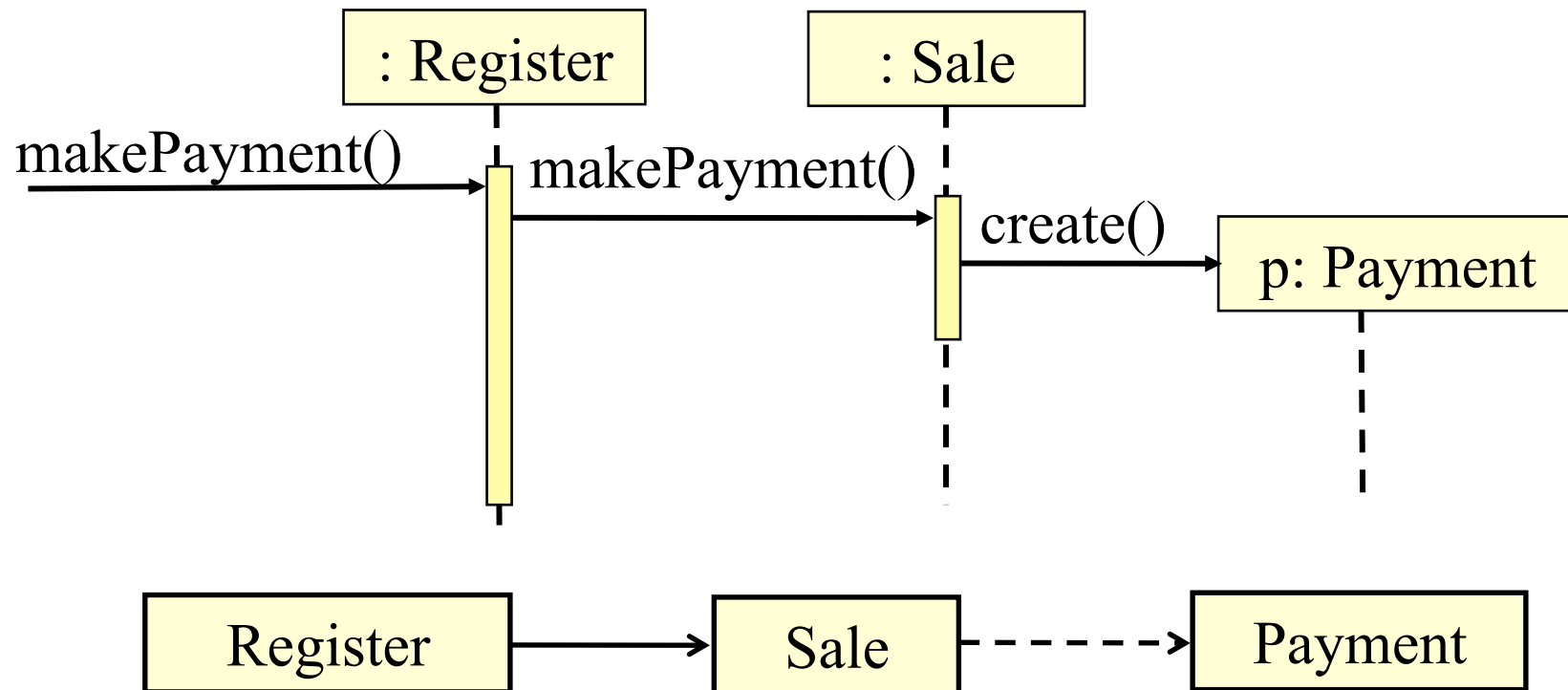
One Possible Design Solution

- ◆ Assign responsibility to the Register



Another Design Solution – Low Coupling

- ◆ Using the Low Coupling principle, delegate responsibility to Sale



PATTERN: High Cohesion

- ◆ Cohesion is a measure of how related the responsibilities of a class are.
- ◆ Typically a class with high cohesion has a relatively small number of methods with highly related function and does not do 'too much' work.
- ◆ A highly cohesive class has the same advantages as a loosely coupled class. It is easier to understand, to modify, and to reuse.
- ◆ Delegation is one technique to increase cohesion. E.g. the Register class delegates the payment creation responsibility to the Sale class.
- ◆ Split uncohesive classes into families of classes that share the work.

Coupling and Cohesion

- ◆ Bad cohesion usually creates bad coupling, and vice versa.
- ◆ What counts as good coupling is a qualitative and relative judgment, not a numeric one.
- ◆ Design principles can not be considered in isolation. Typically they are antagonistic.
- ◆ A subclass is strongly coupled to a superclass, which suggests caution in using inheritance.

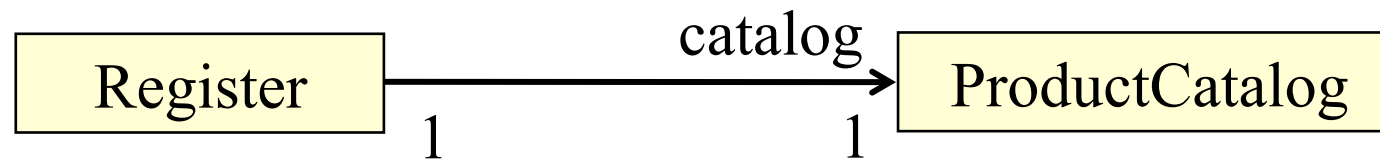


Designing for Visibility

- ◆ To send a message to B, A must be visible to B. It doesn't happen by "magic."
- ◆ Kinds of visibility:
 - Attribute: b is referenced as an attribute of a .
 - Parameter: b is referenced as a parameter of a method in a .
 - Local: b is referenced as a local variable of a method in a .
 - Global: b is made globally visible in some way.

Attribute Visibility

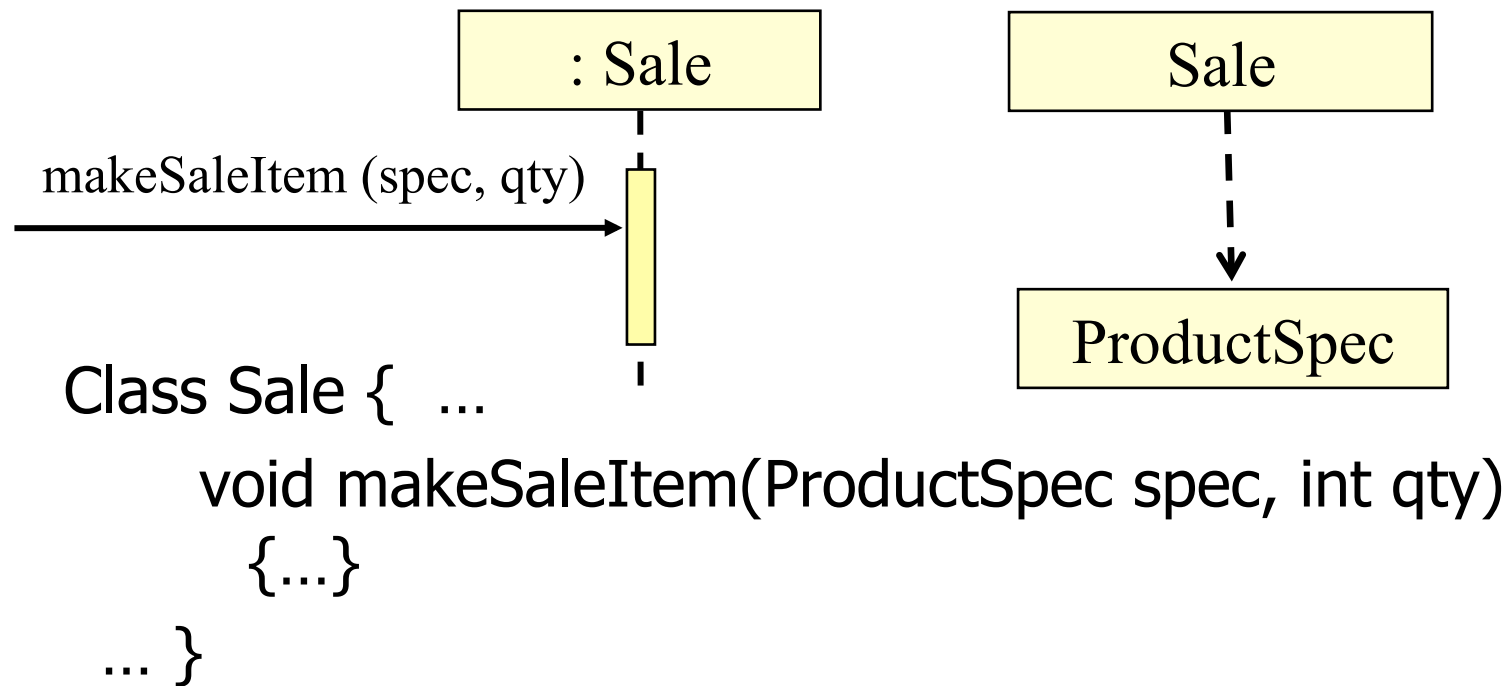
- ◆ Permanent relationship that persists as long as a and b
- ◆ Modelled as an association in a class diagram



```
Class Register {
    ...
    ProductCatalog catalog;
    ...
}
```

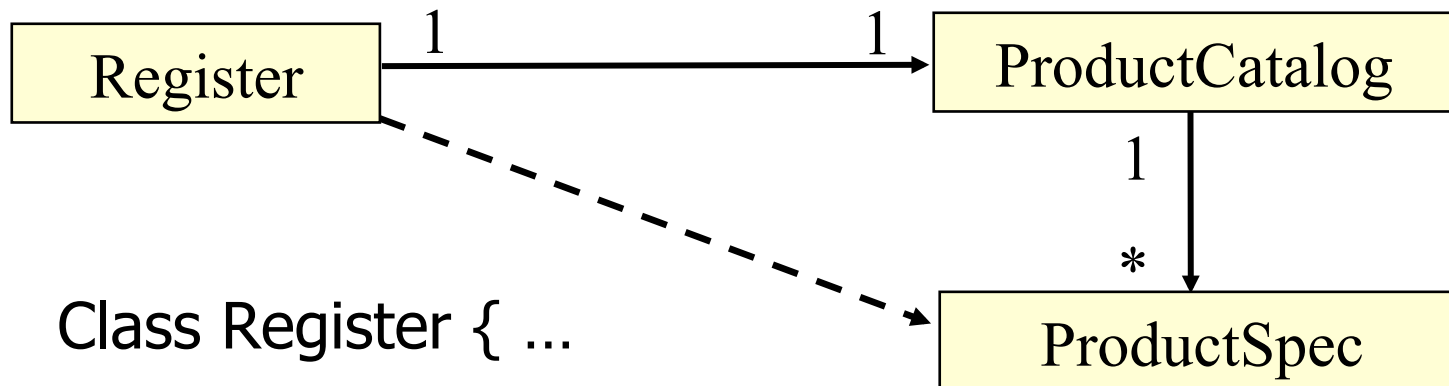
Parameter Visibility

- ◆ Temporary relationship that persists as long as the method activation
- ◆ Modelled as a dependency in a class diagram



Local Visibility

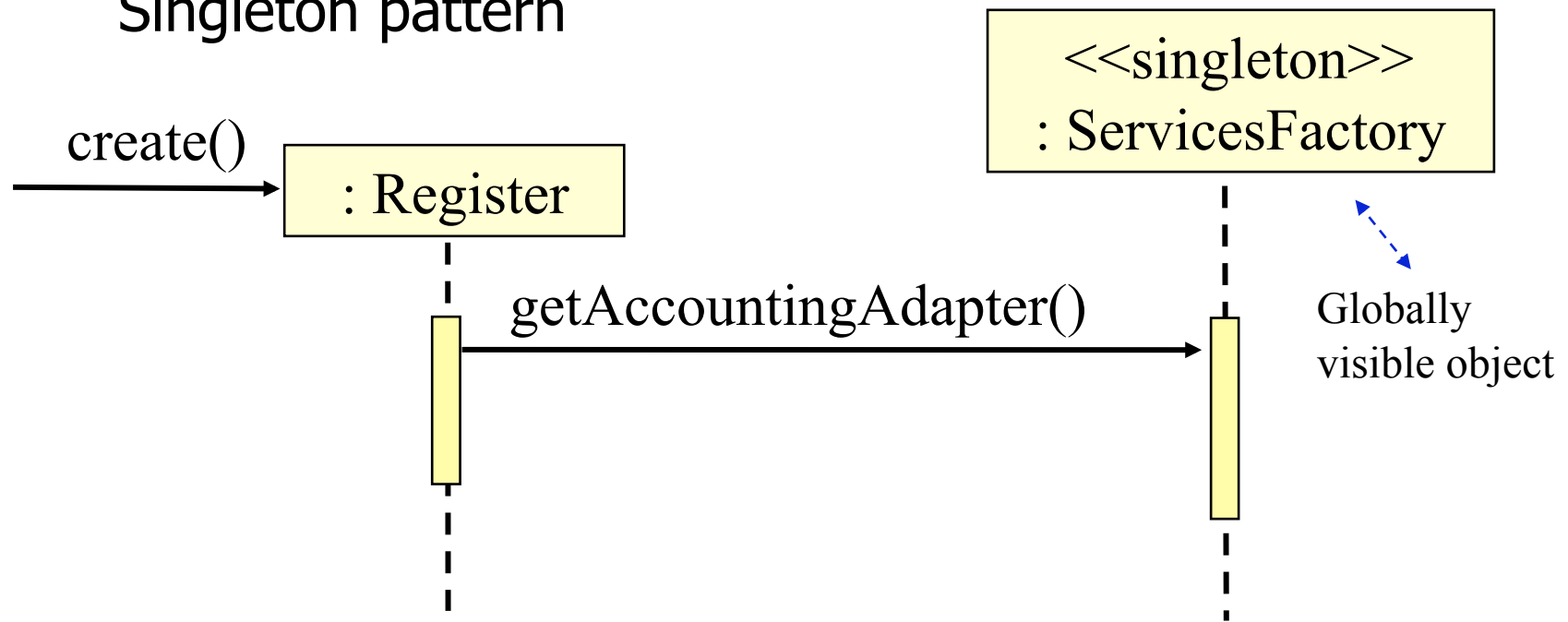
- ◆ Temporary relationship that persists as long as the method activation
- ◆ Modelled as a dependency in a class diagram



```
Class Register { ...
    ProductCatalog catalog; ...
    void enterItem(ItemID id, int qty) {...
        ProductSpec spec = catalog.getSpec(id) ...}
    ... }
```

Global Visibility

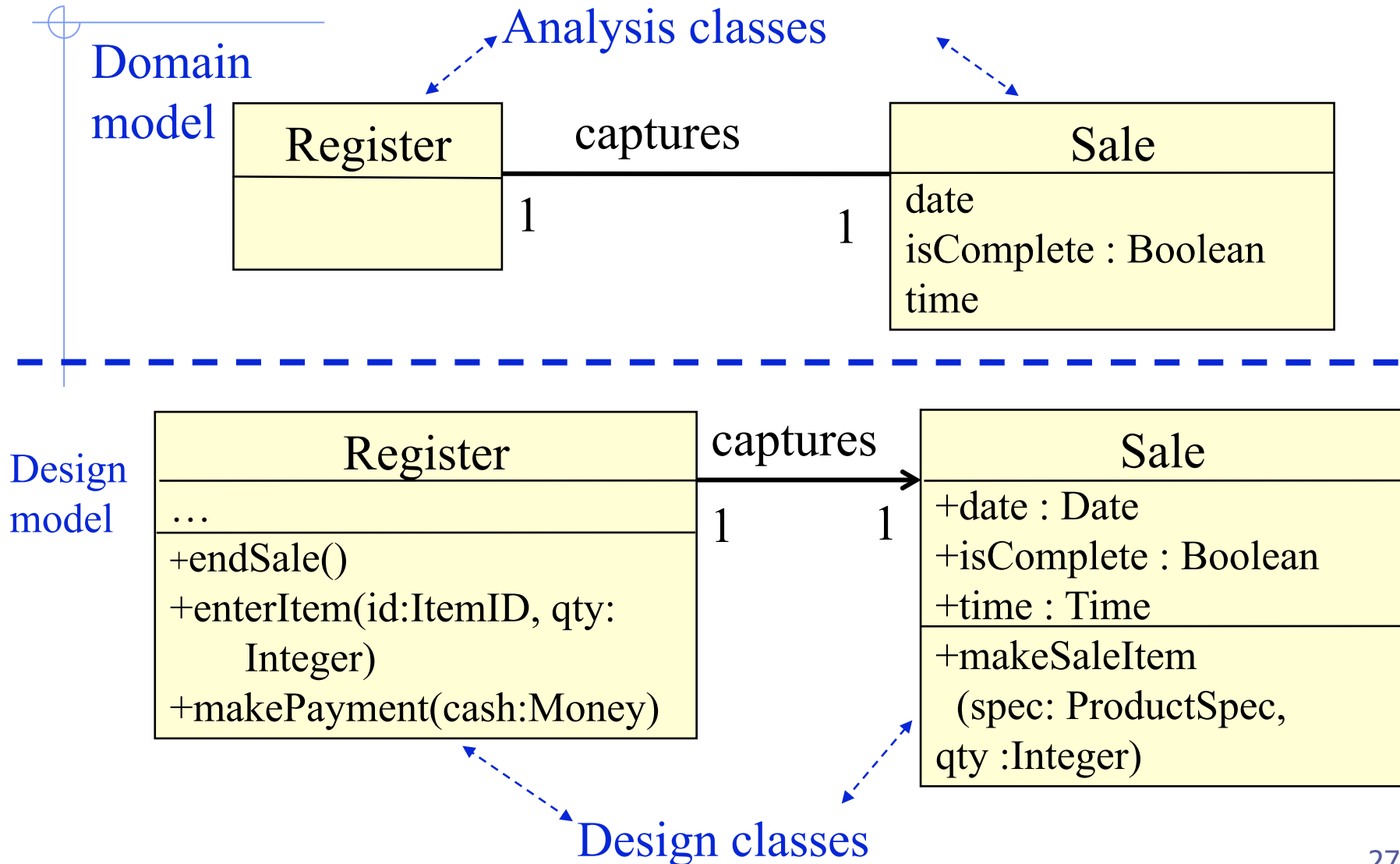
- ◆ Relatively permanent relationship
- ◆ Can be modelled as an association in a class diagram
- ◆ Preferred method for global visibility uses the Singleton pattern



Design Class Diagrams

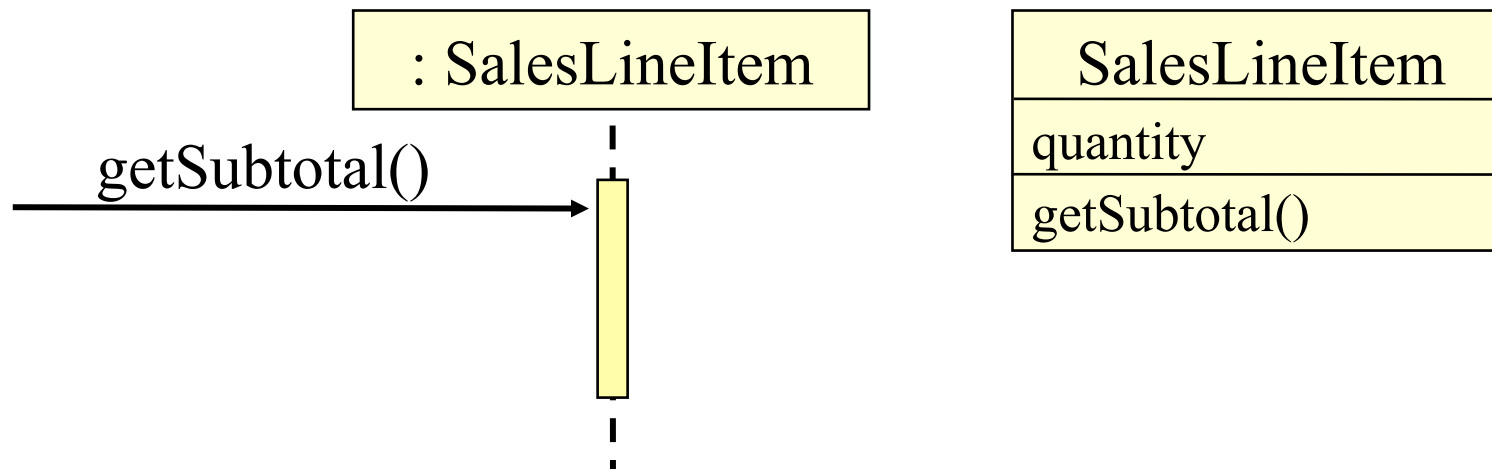
- ◆ A design class diagram defines software entities rather than domain concepts.
- ◆ Design class diagrams are developed from analysis class diagrams with further detail gathered from sequence diagrams, such as
 - additional design classes not in the analysis model
 - methods in sequence diagrams that a class must support
 - associations (and navigability) and dependency relationships necessary to support interactions in sequence diagrams

Analysis vs Design Class Models



Add Methods to Design Classes

- ◆ To find all the methods that a class must support, identify all messages sent to instances of the class in all sequence diagrams.
- ◆ If message `getSubtotal()` is sent to an object of class `SaleItem` in a sequence diagram, then the class `SaleItem` must define that method.

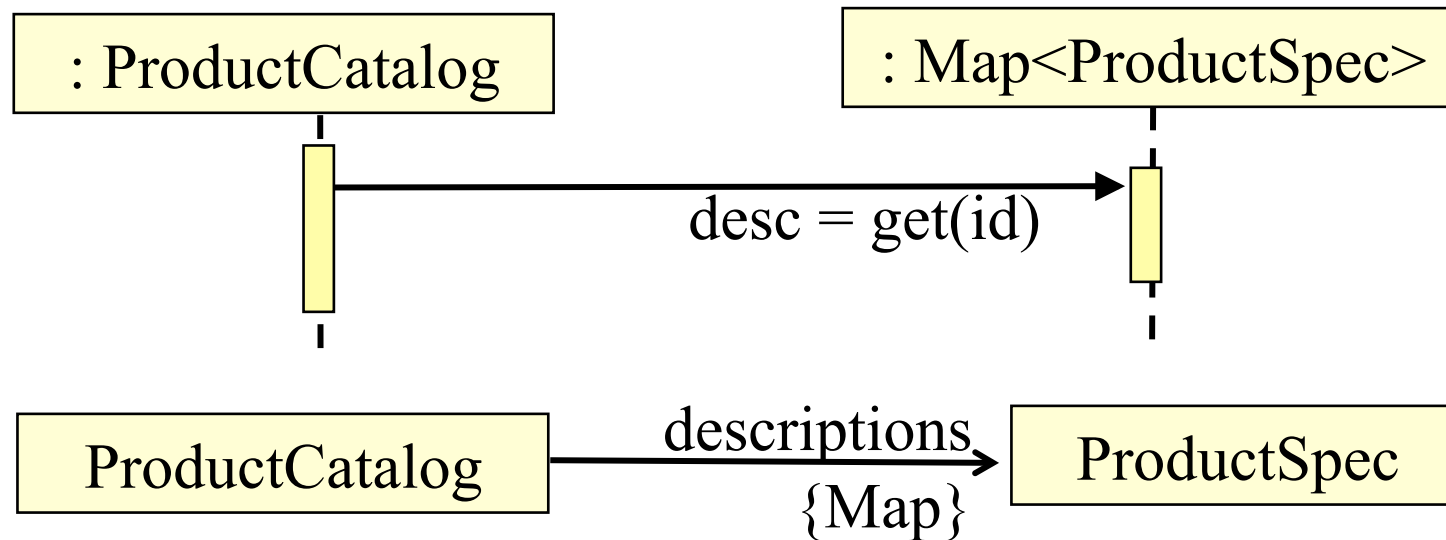


Methods

- ◆ In UML, `create()` denotes instantiation and initialisation of a new object. These are normally omitted from design class diagrams.
- ◆ Accessors that get and set attribute values are also normally omitted. This does not mean they are not important: 'routine' does not mean 'unimportant'.
- ◆ A message to a multi-object is interpreted as a message to the container object (e.g. List or Map), not to the objects in the collection. Container classes are not normally shown on class diagrams since they are normally library classes.

Method Example

- ◆ Message *get* is targeted on the container object (Map), so it is not correct to make *get* a method in the *ProductSpec* class.



Add Type Information

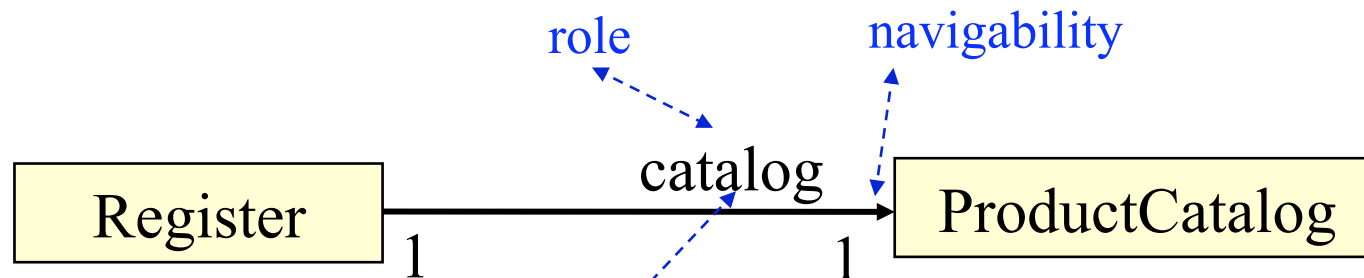
- ◆ Specifying the type of attributes and methods (including return type) is optional.
- ◆ What is the purpose of the UML? Low-level detail is clutter, but is required for automatic processing.
- ◆ Example:

UML: - x : Integer = 4 {frozen}

Java: private static final int x = 4;

Add Associations -1

- ◆ Each end of an association is *a role*.
- ◆ Each end may have a navigability arrow.
- ◆ Arrow indicates that an object of the source class can interact with an object of the target class.
- ◆ Navigability implies attribute visibility.



```
Class Register {
    ProductCatalog catalog;
    ... }
```

Add Associations -2

- ◆ In an analysis class diagram, associations are added to enhance understanding.
- ◆ In a design class diagram, associations are added to establish visibility (driven by sequence diagrams).
- ◆ In a design class diagram, an association should carry a navigability arrow (or two).
- ◆ In an implementation, an association is realised by defining a source class attribute that references an object of the target class.

Dependency Relationships

- ◆ A dashed line (with arrow) indicates a general dependency of one class on another.
- ◆ Used to show a visibility relationship that is not of the attribute type.
- ◆ To avoid clutter, notate such a relationship only when it is salient.

Follow-up Reading

- ◆ Larman: Chapters 17, 18, 19