Design Model: Creating Design Class Diagrams

Software Design and Analysis

CSCI 2040

Objectives

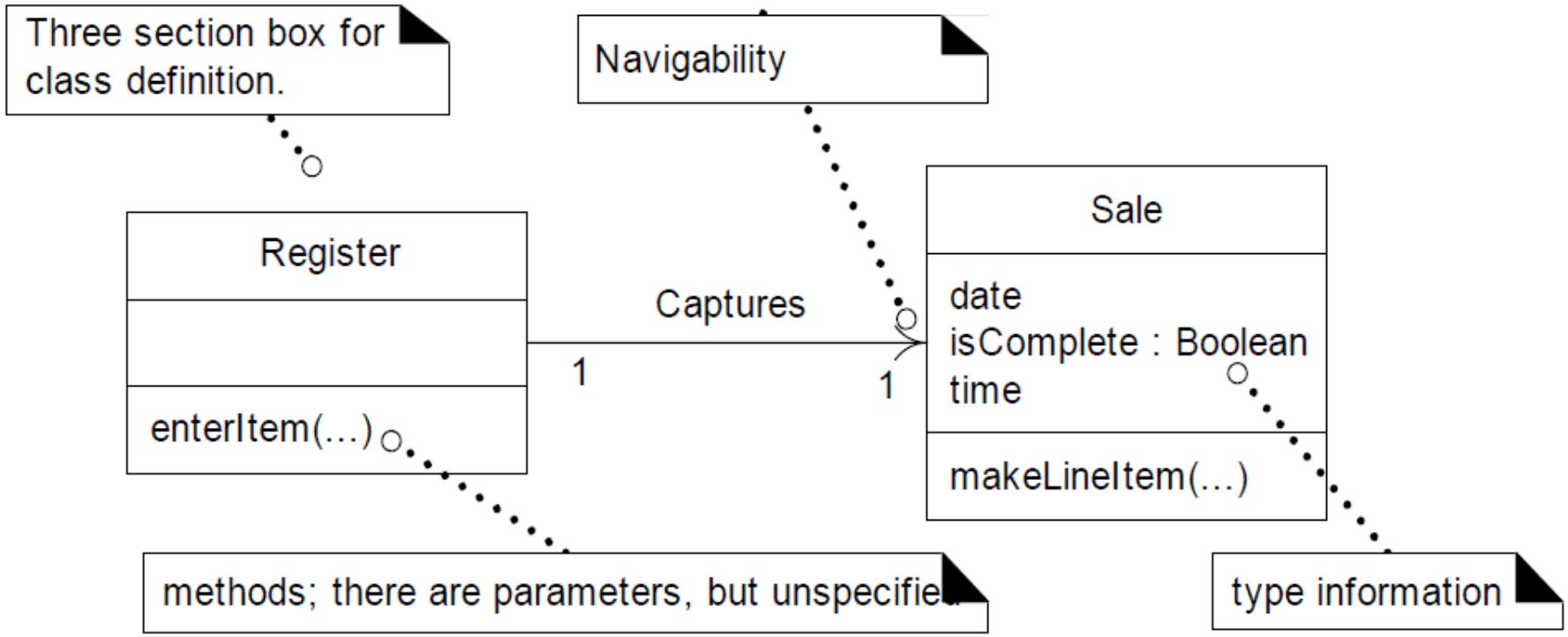
- Create design class diagrams (DCDs)
- Identify the classes, methods, and associations to show in a DCD.
- Map design artifacts to code in an object-oriented language.

Design Class Diagrams

Introduction

- The UML has notation for showing design details in class diagrams.
  - i.e., identify the specification for the software classes (and interfaces) that participate in the software solution,
  - and annotate them with design details, such as methods.
- Although this presentation of DCDs *follows* the creation of Interaction Diagrams, in practice they are usually created in parallel.

Sample Design Class Diagram

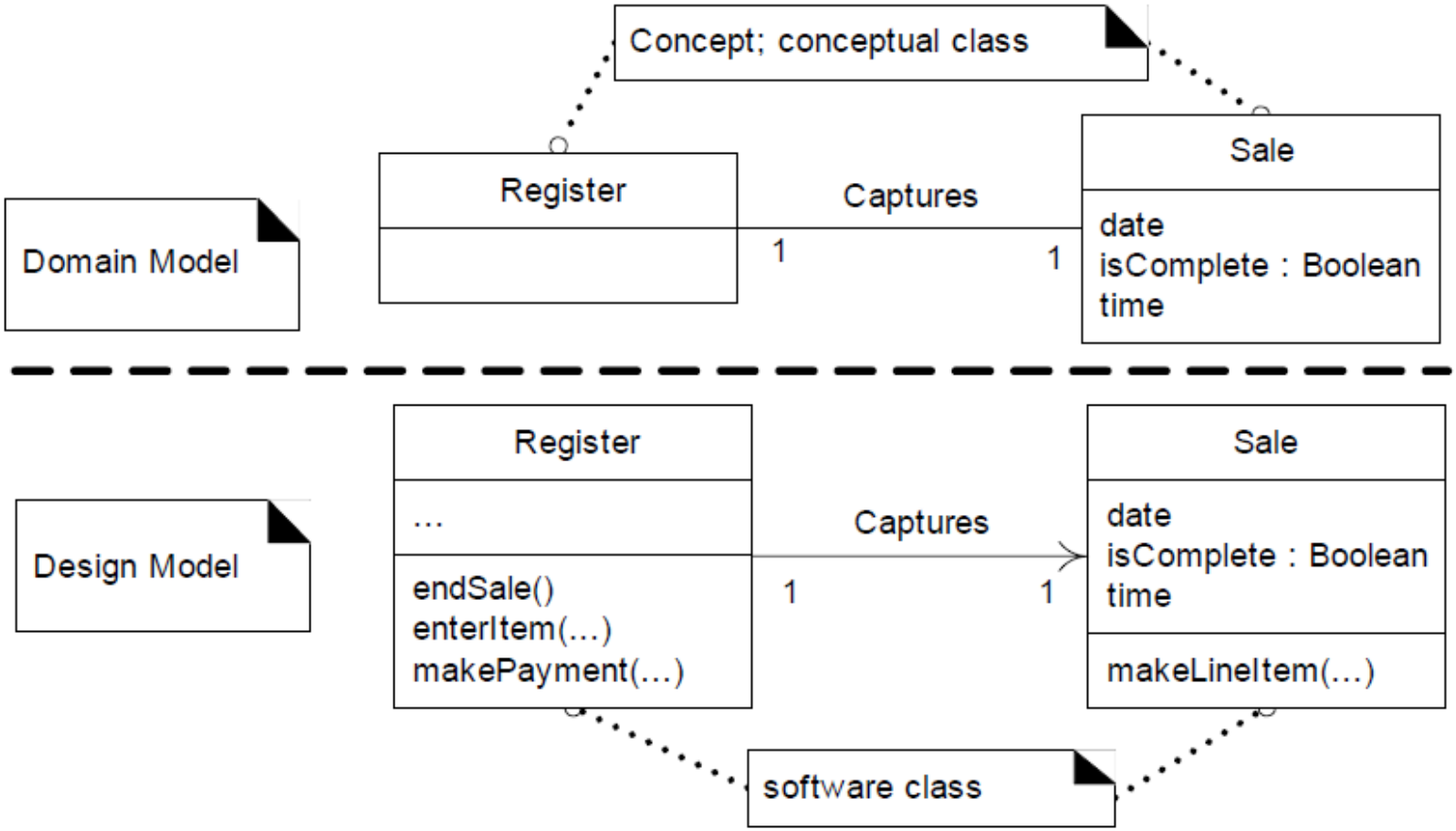


DCD

- **A design class diagram** (DCD) illustrates the specifications for software classes and interfaces (for example, Java interfaces) in an application. Typical information includes:
  - classes, associations and attributes
  - interfaces, with their operations and constants
  - methods
  - attribute type information
  - navigability
  - Dependencies, visibility

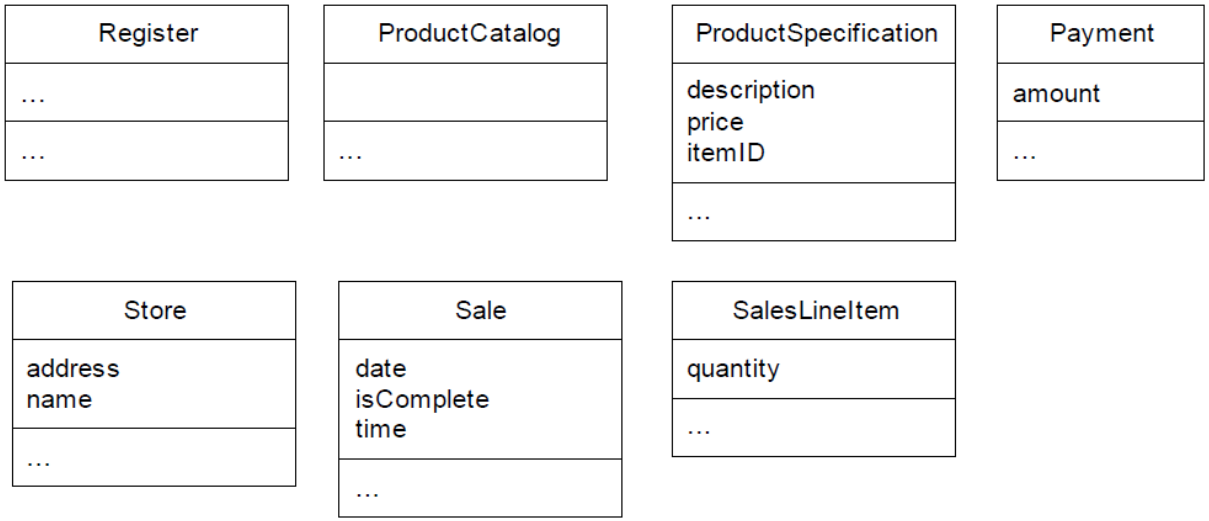
Domain Model vs Design Model Classes

- Domain Model does not represent a software definition, design Model does!

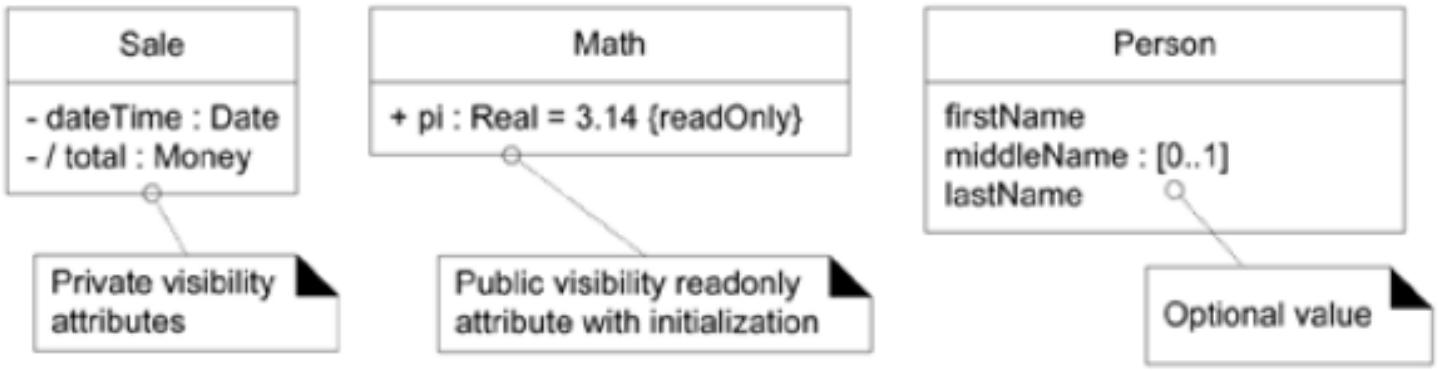


Software Classes in Application

- The first step in the creation of DCDs is to identify those classes that participate in the software solution.
  - These can be found by scanning all the Interaction Diagrams (created based on Domain Model) and listing the classes mentioned.

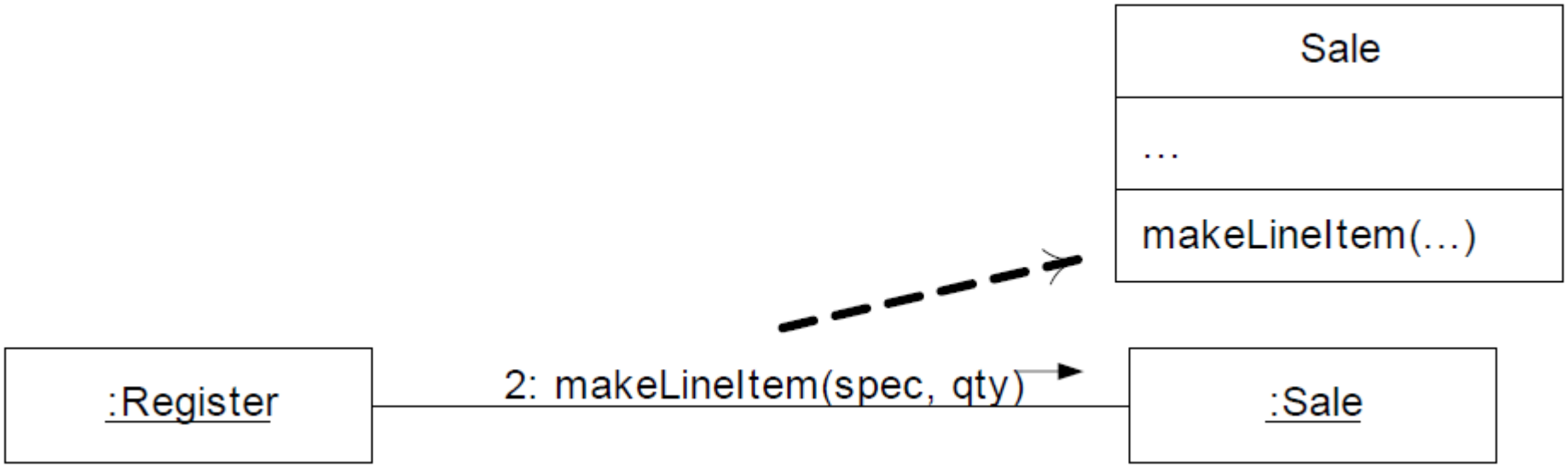


Visibility

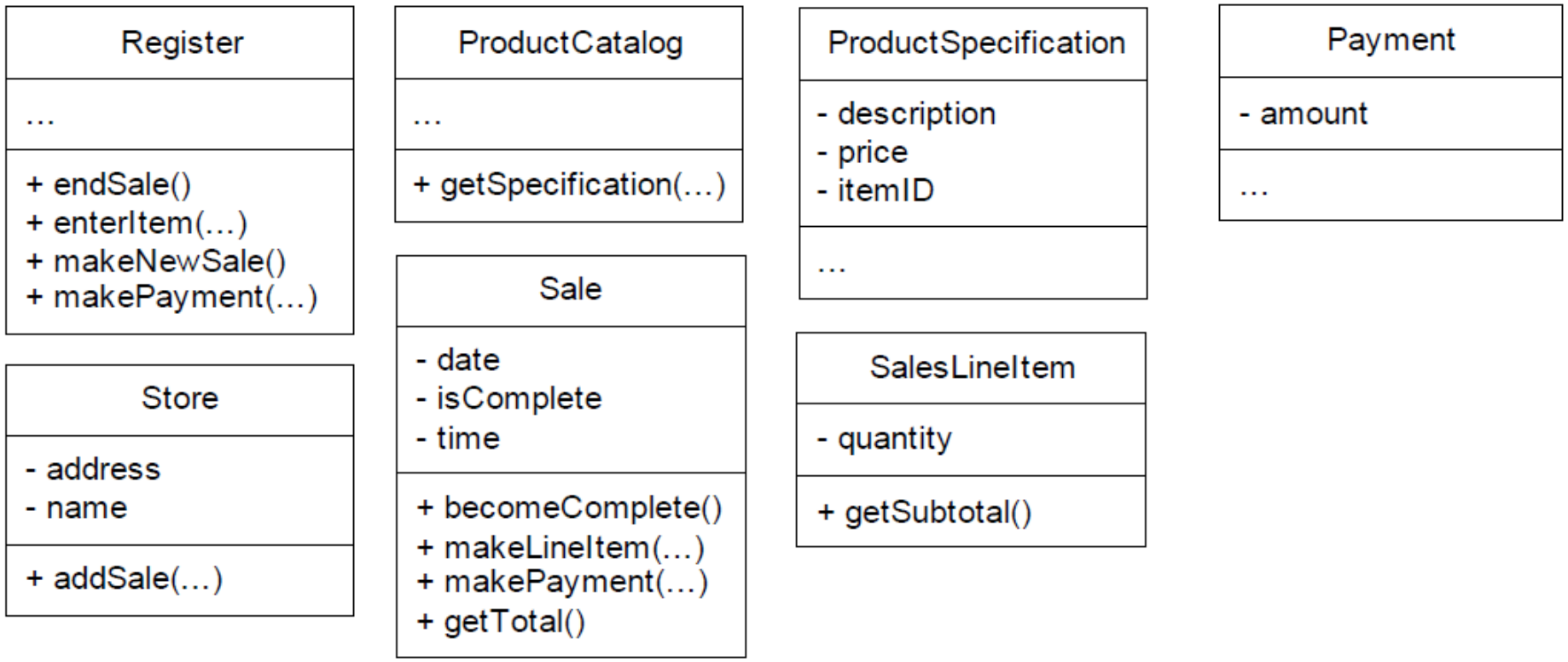

Sale
- dateTime : Date
- / total : Money
Private visibility attributes
Math
+ pi : Real = 3.14 {readOnly}
Public visibility readonly attribute with initialization
Person
firstName
middleName : [0..1]
lastName
Optional value

Methods Names from Interaction Diagrams

- The methods of each class can be identified by analyzing the interaction diagrams.

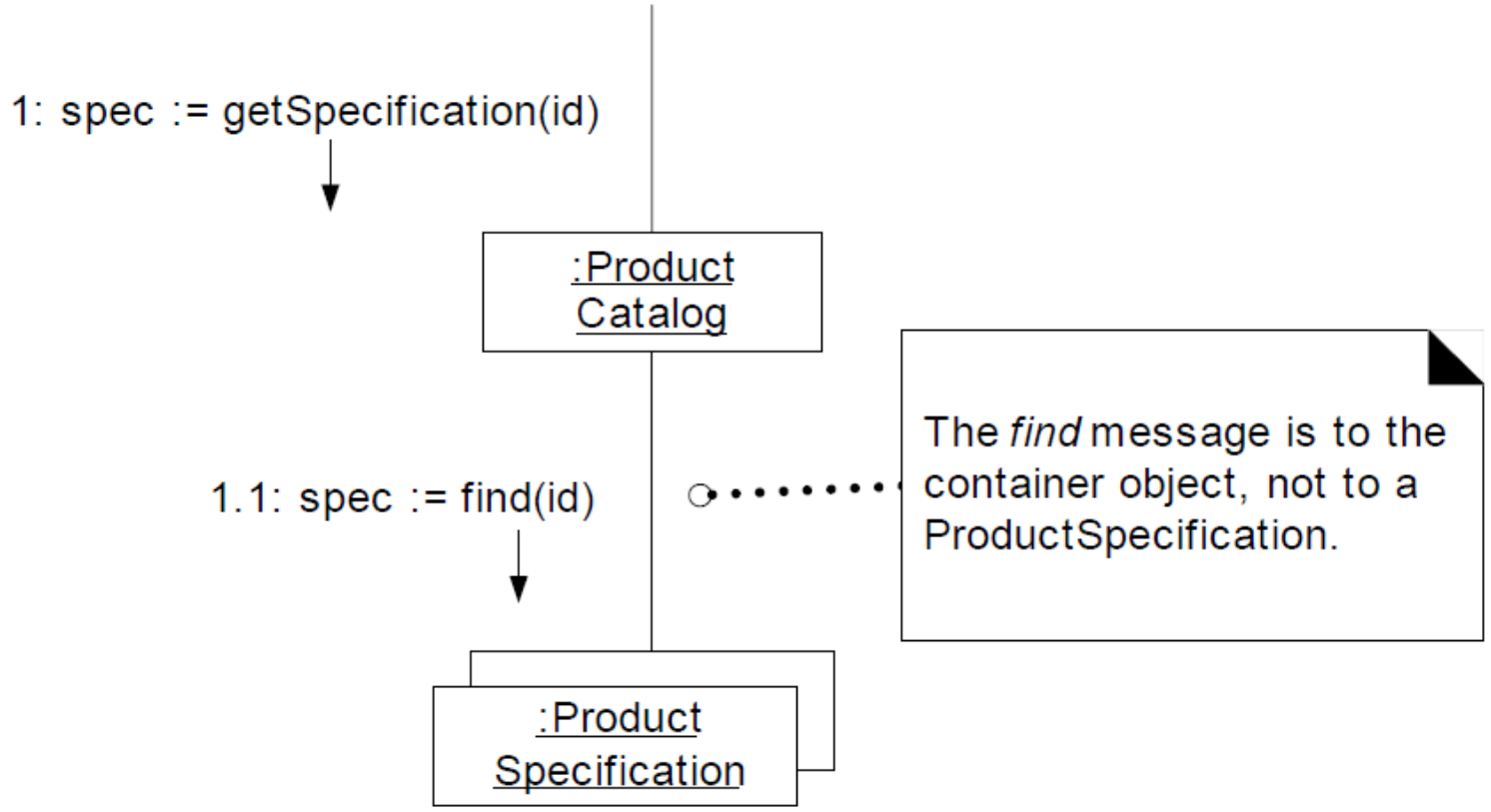


Methods in Application



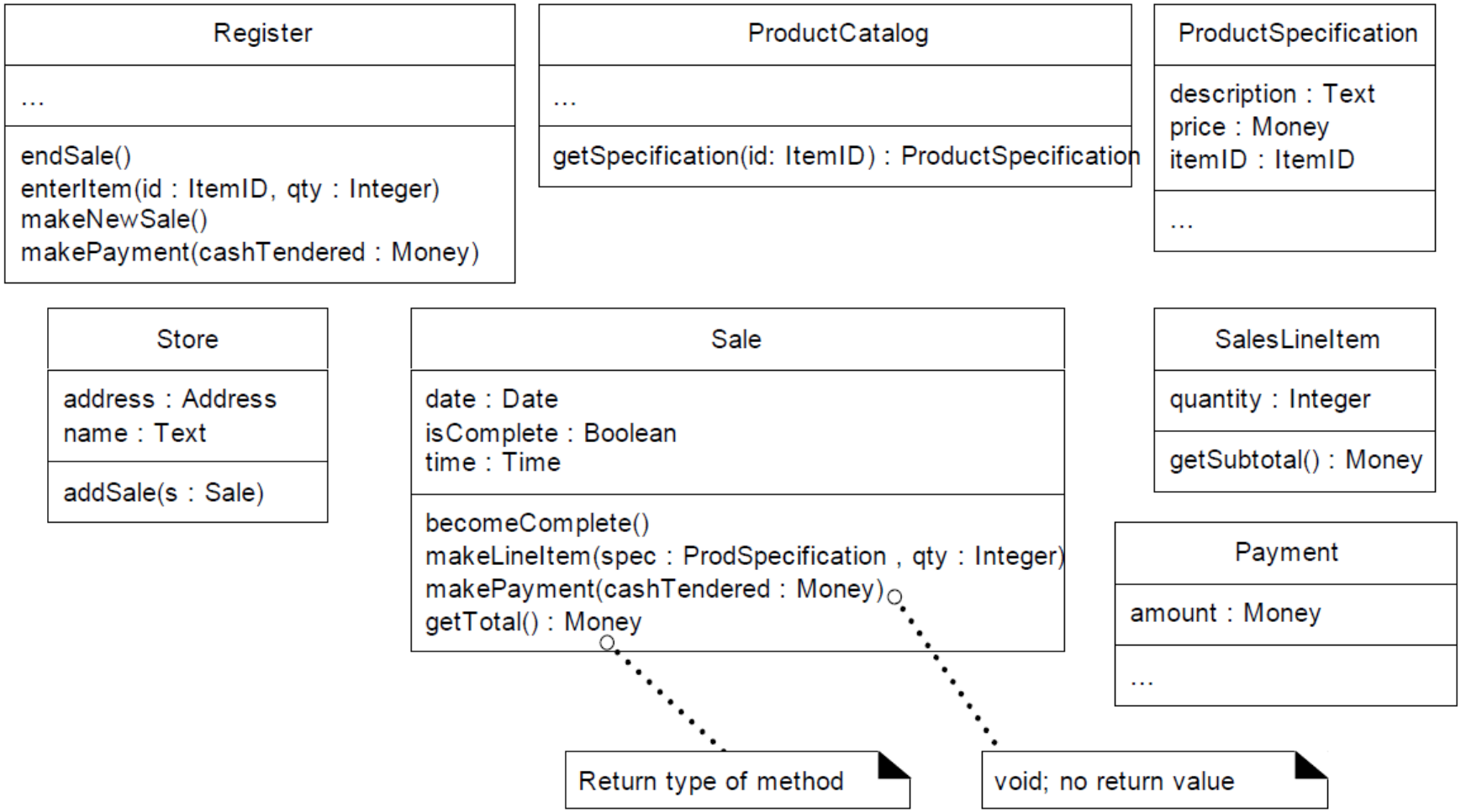
Message to Multiobject

- The *find* method is not part of the *Product Specification* class

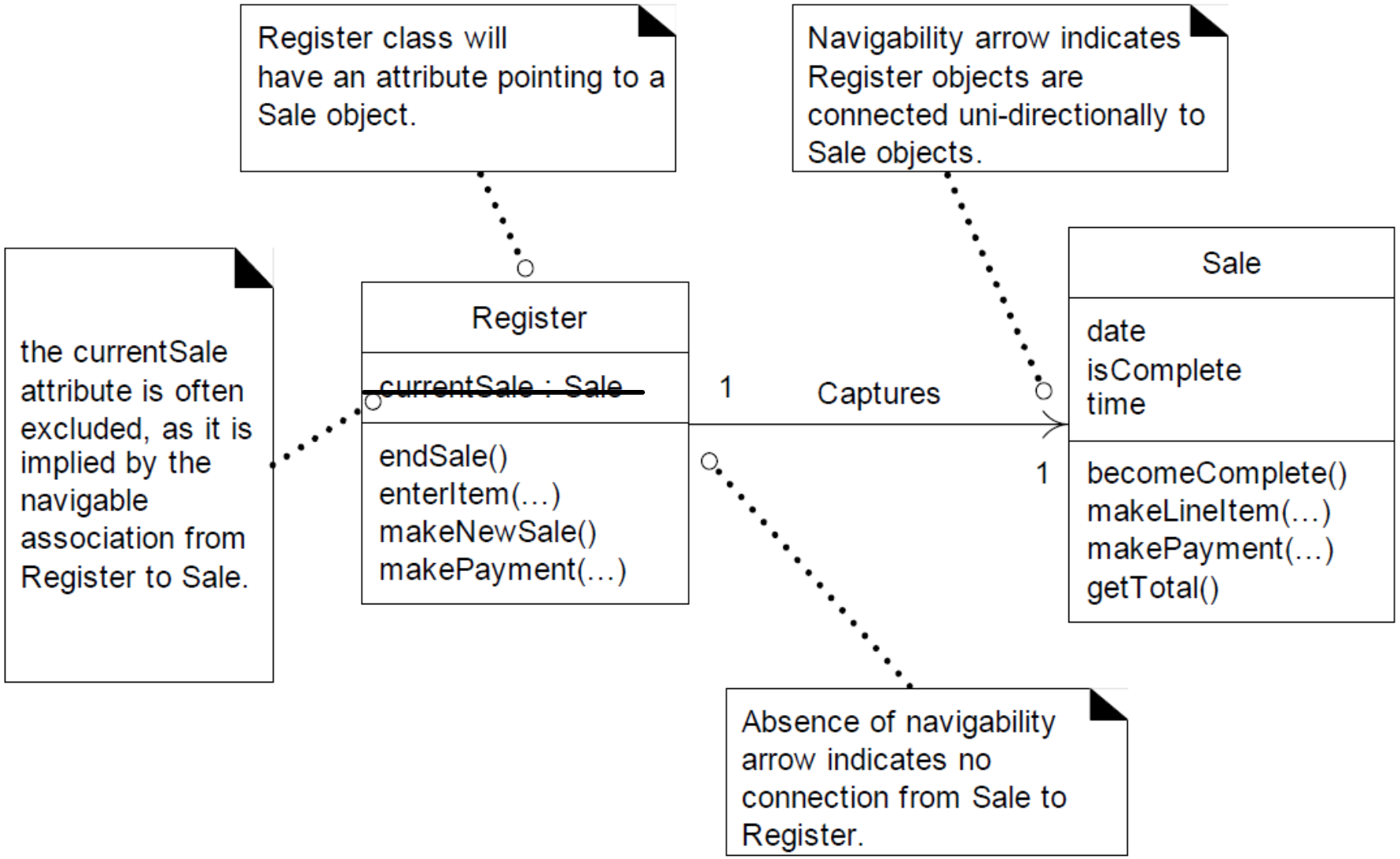


Adding Type Information

- The types of the attributes, method parameters, and method return values

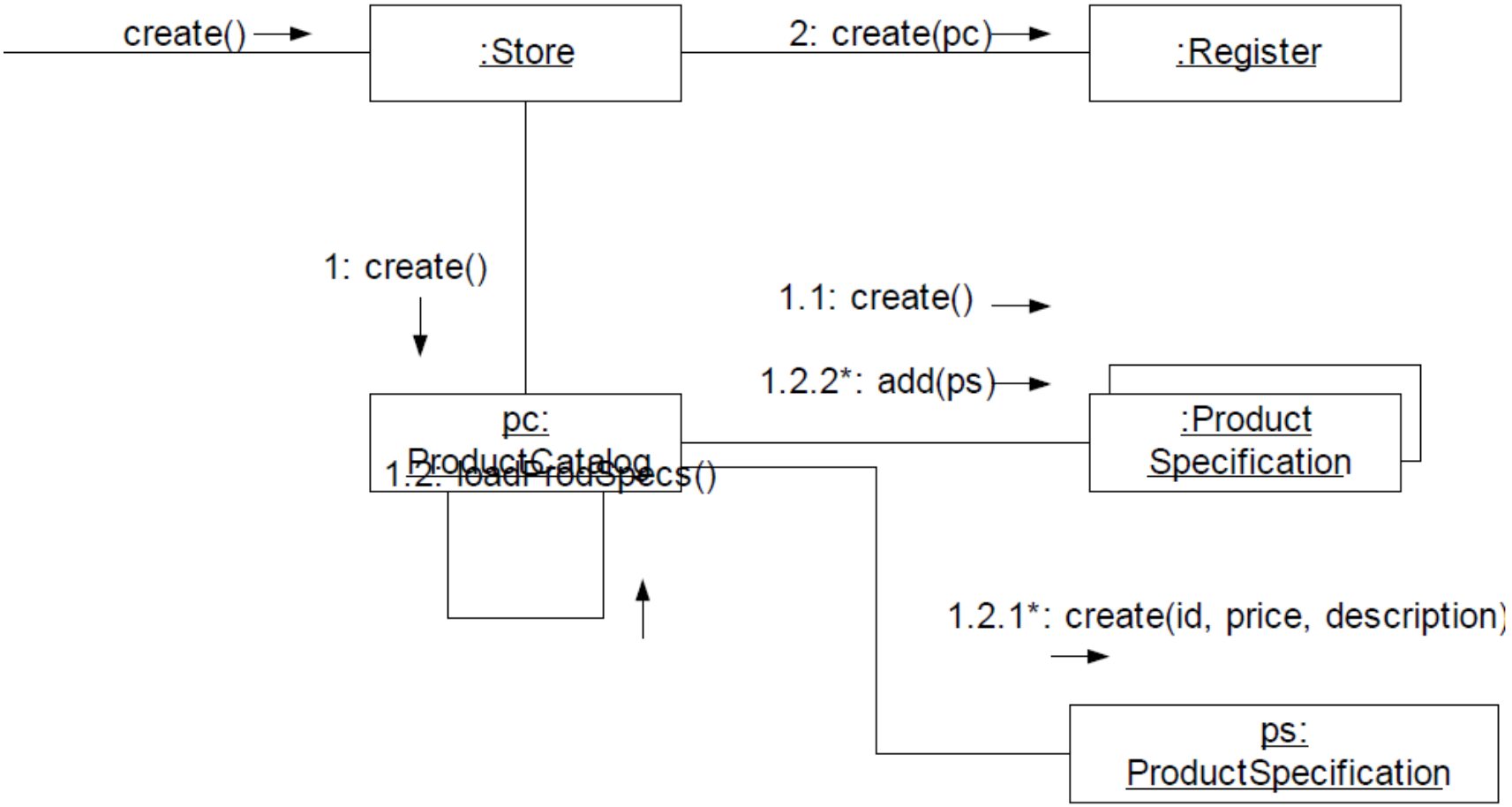


Showing Navigability and Attribute Visibility

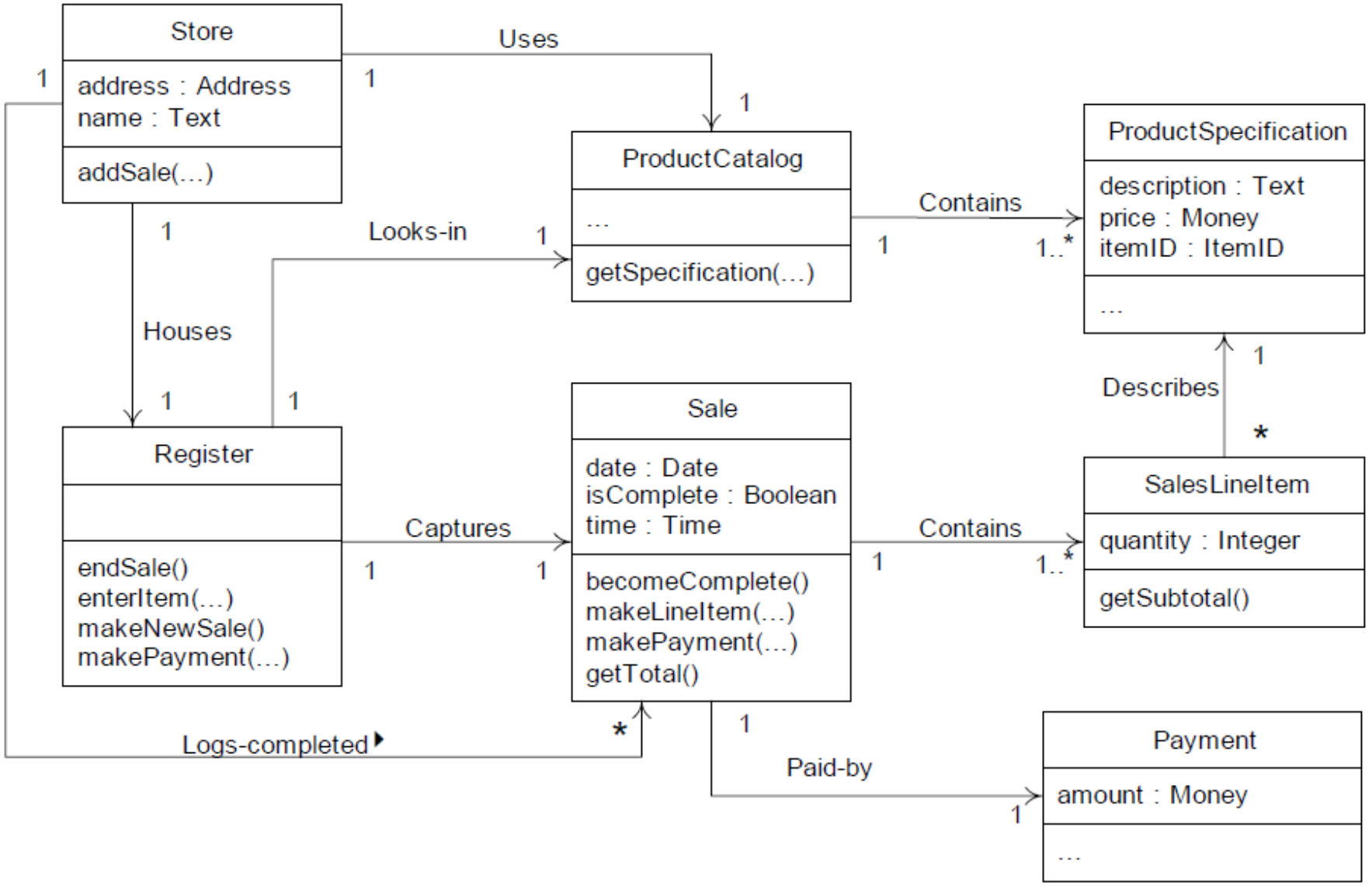


Navigability

- Navigability is identified from interaction diagrams

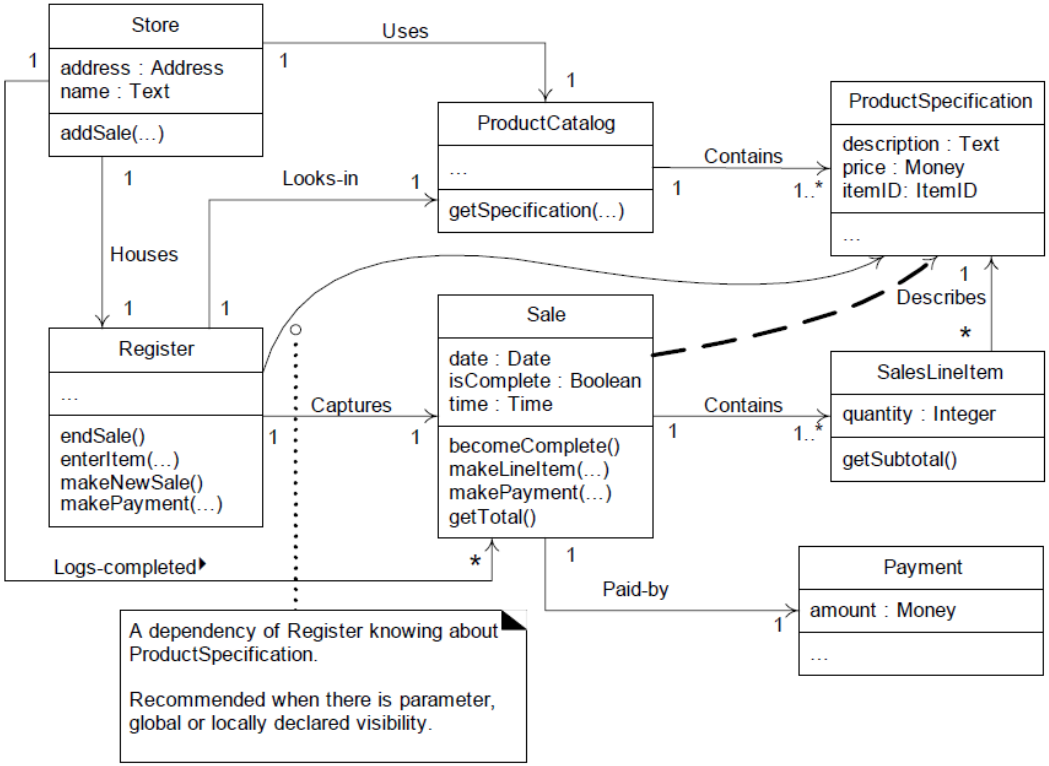


Associations with Navigability Adornments

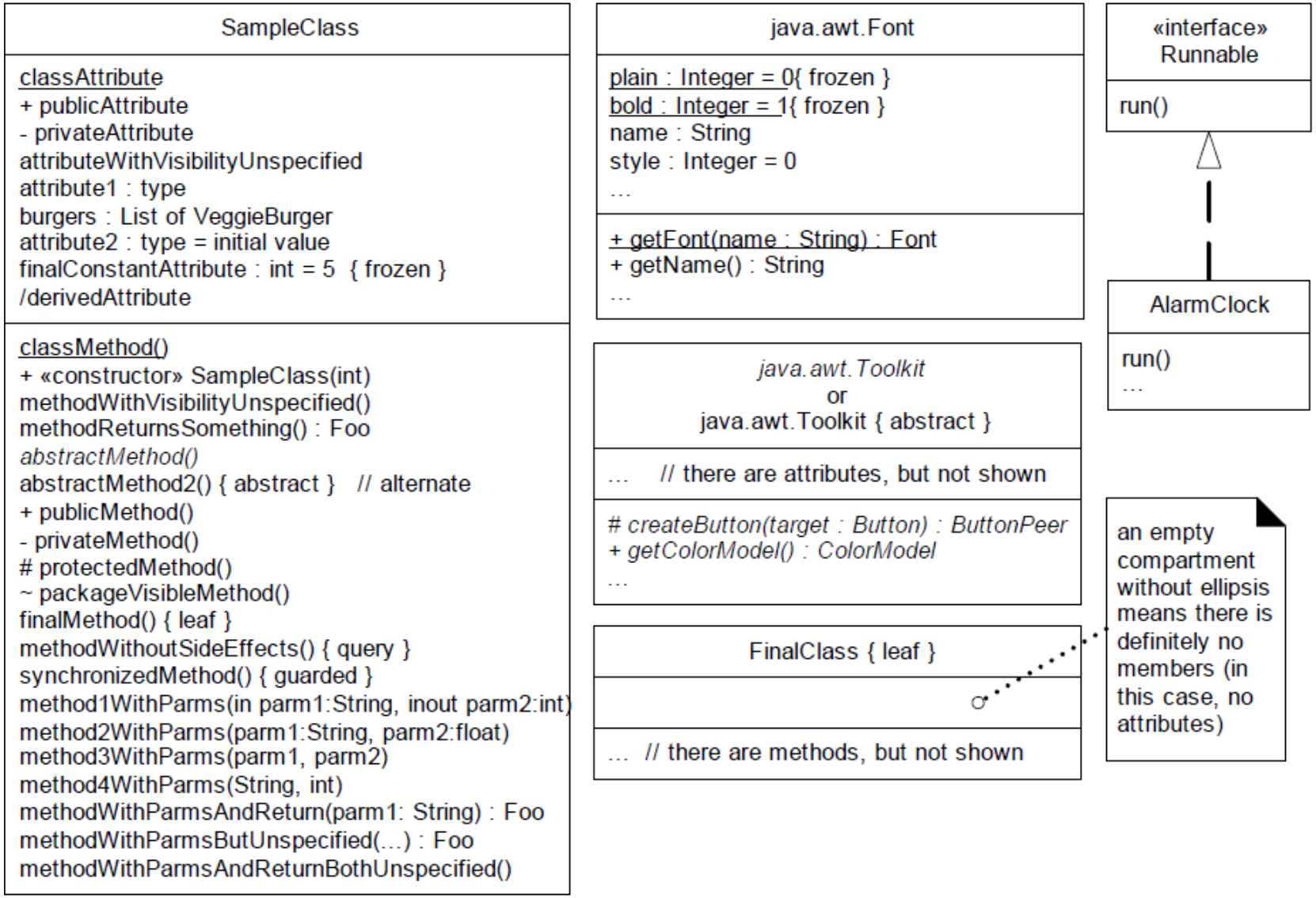


Adding Dependency Relationship Indicating Non-Attribute Visibility

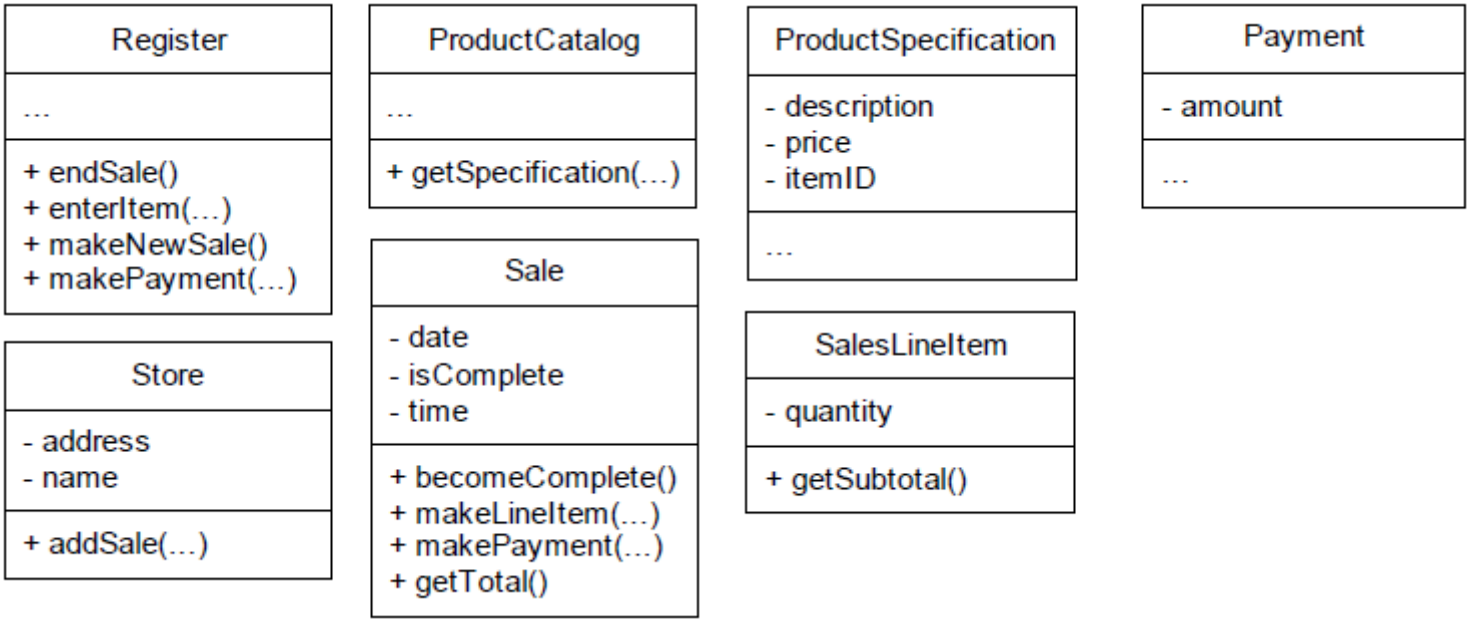
- Declared visibility with dashed line, indicates non-attribute visibility.
  - Sale does not keep reference to ProductSpecification.
  - Sale receives a ProductSpecification as a parameter in the makeLineItem message (through Register)



Notation for Member Details



Members Details in POS Class Diagram

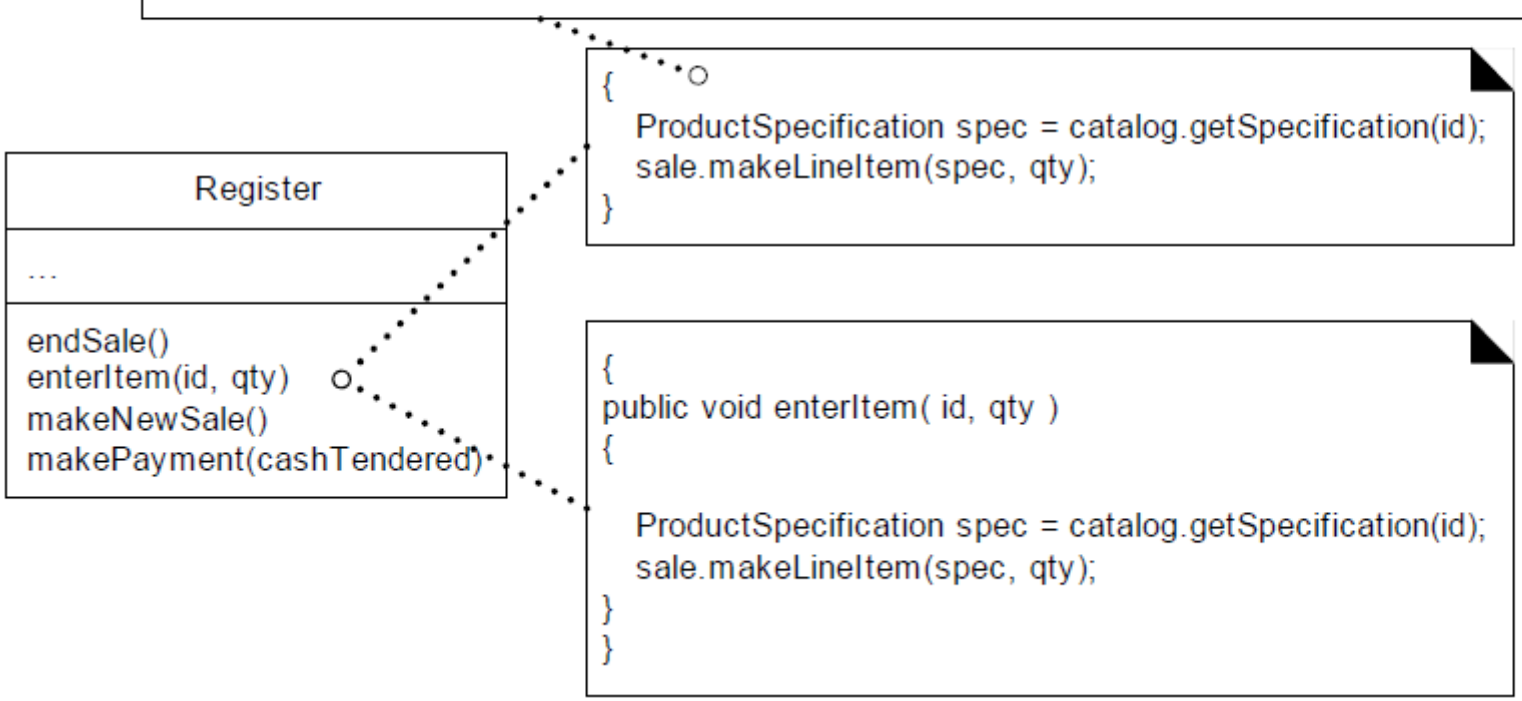


Method Body Notation

UML notation
A method body implementation may be shown in a UML note box. It should be placed within braces, wh
signifies it is semantic influence (it is more than just a comment).

The synax may be pseudo-code, or any language.

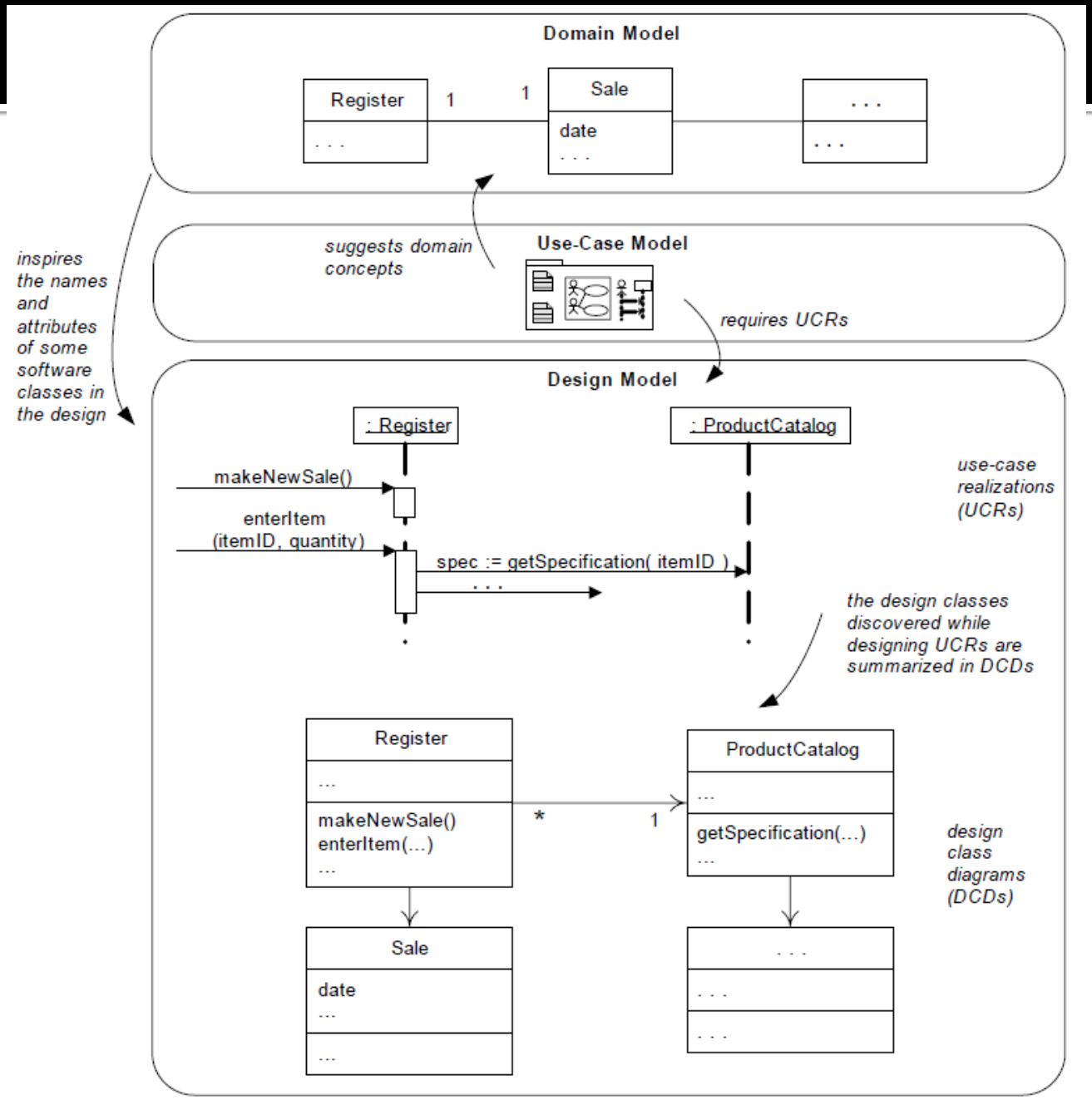
It is common to exclude the method signature (public void ...), but it is legal to include it.



DCDs within UP

Discipline	Artifact Iteration->	Incep. Il	Elab. El. .En	Const. C1..Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model (SSDs)	s	r		
	Vision	s	r		
	Supplementary Specifications	s	r		
	Glossary	s	r		
Design	*Design Model*		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

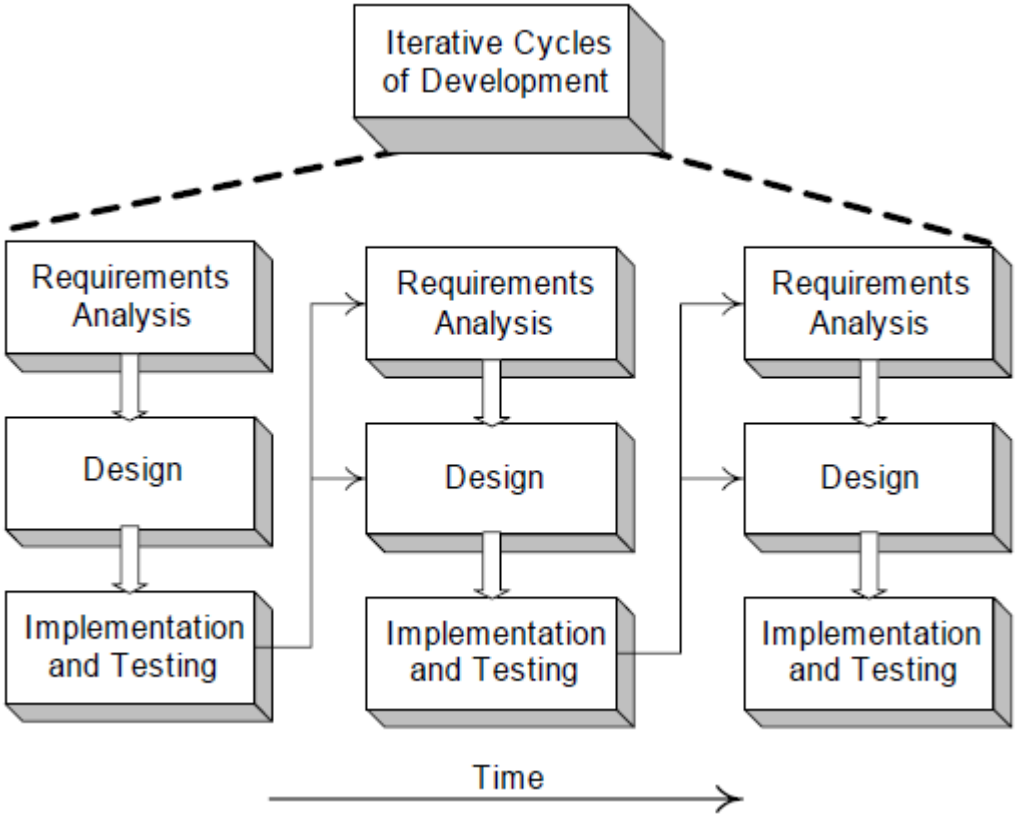
UP Artifacts



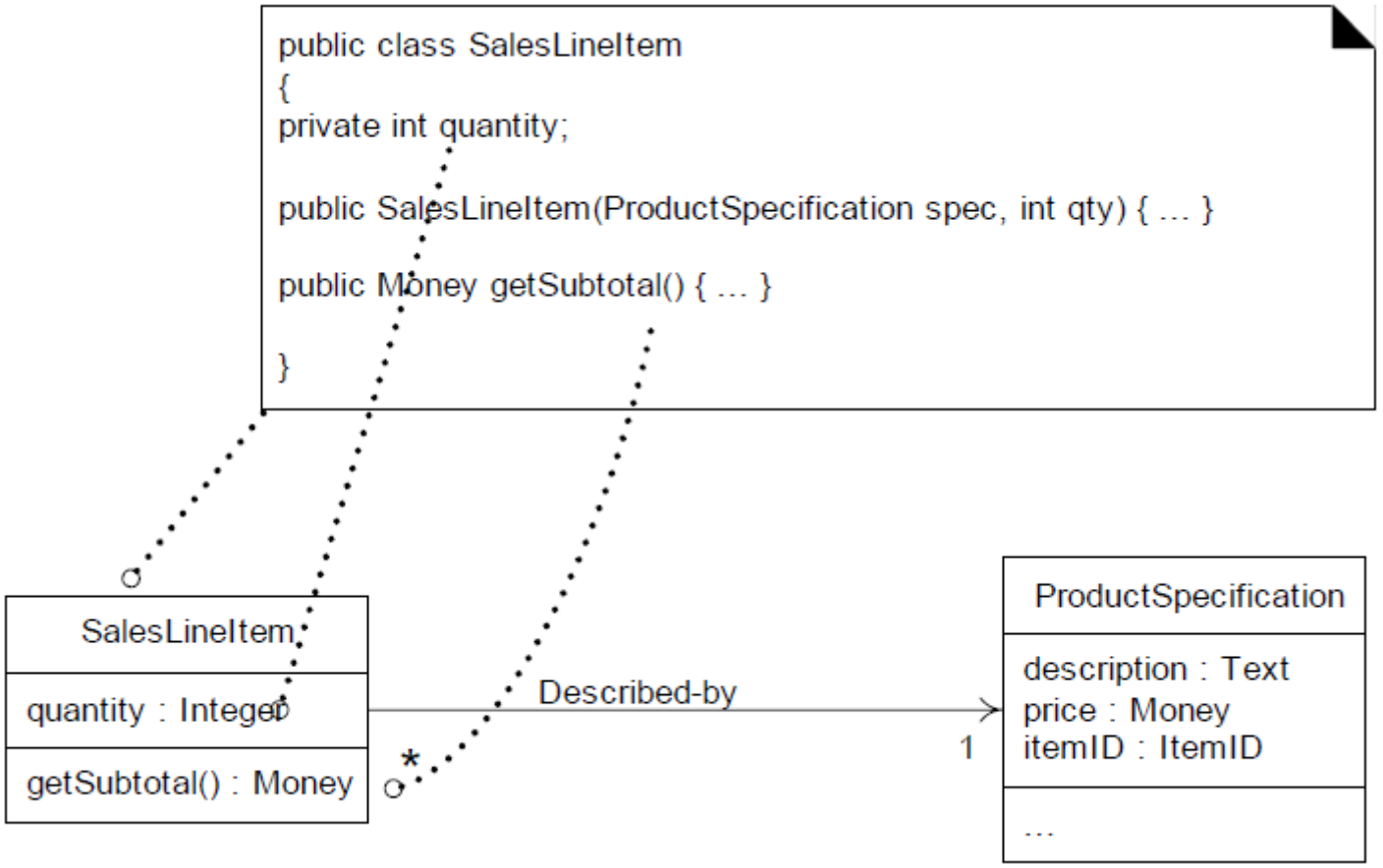
Mapping Designs to Code

Code Changes and Iterative Process

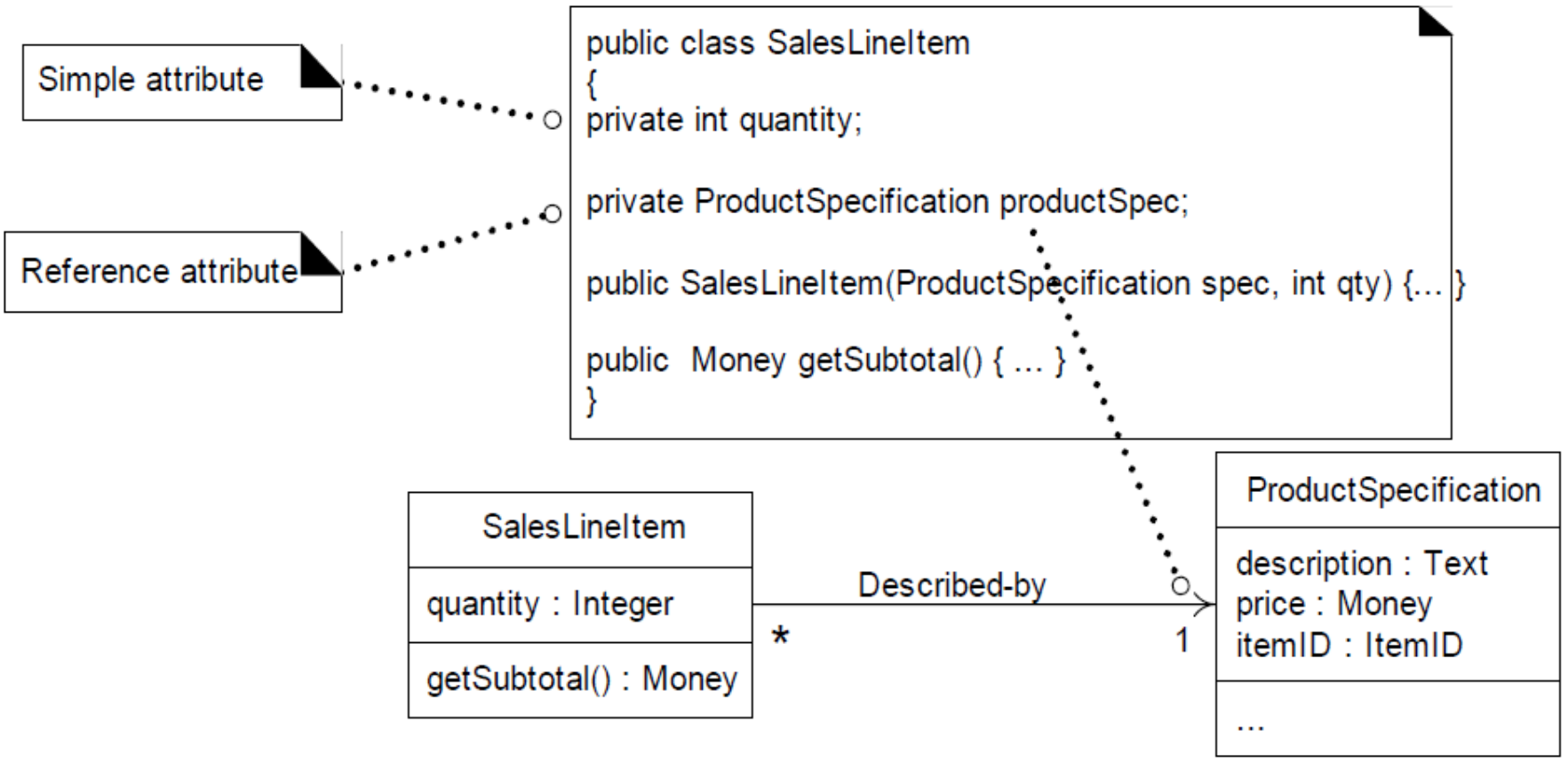
- Implementation in an iteration influences later design.



Defining Class with Methods and Attributes

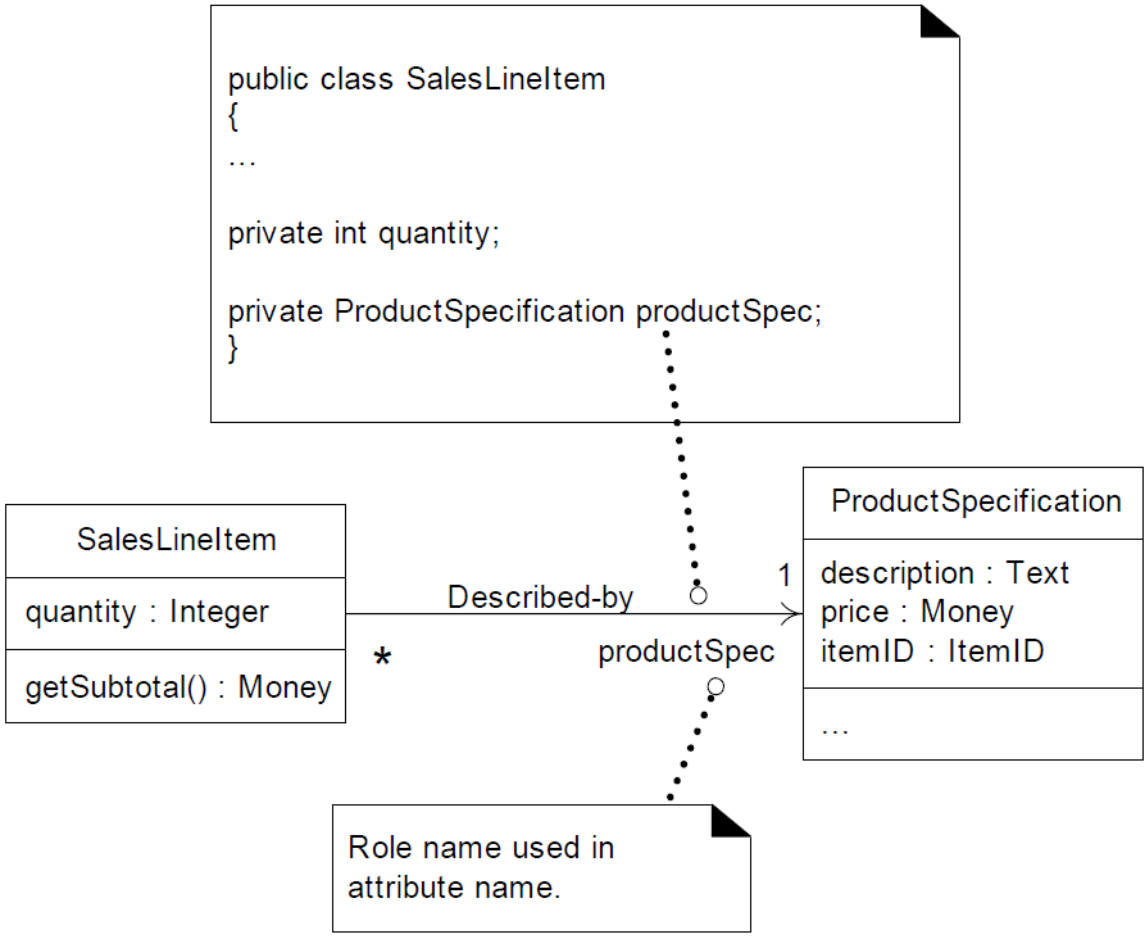


Adding Reference Attributes

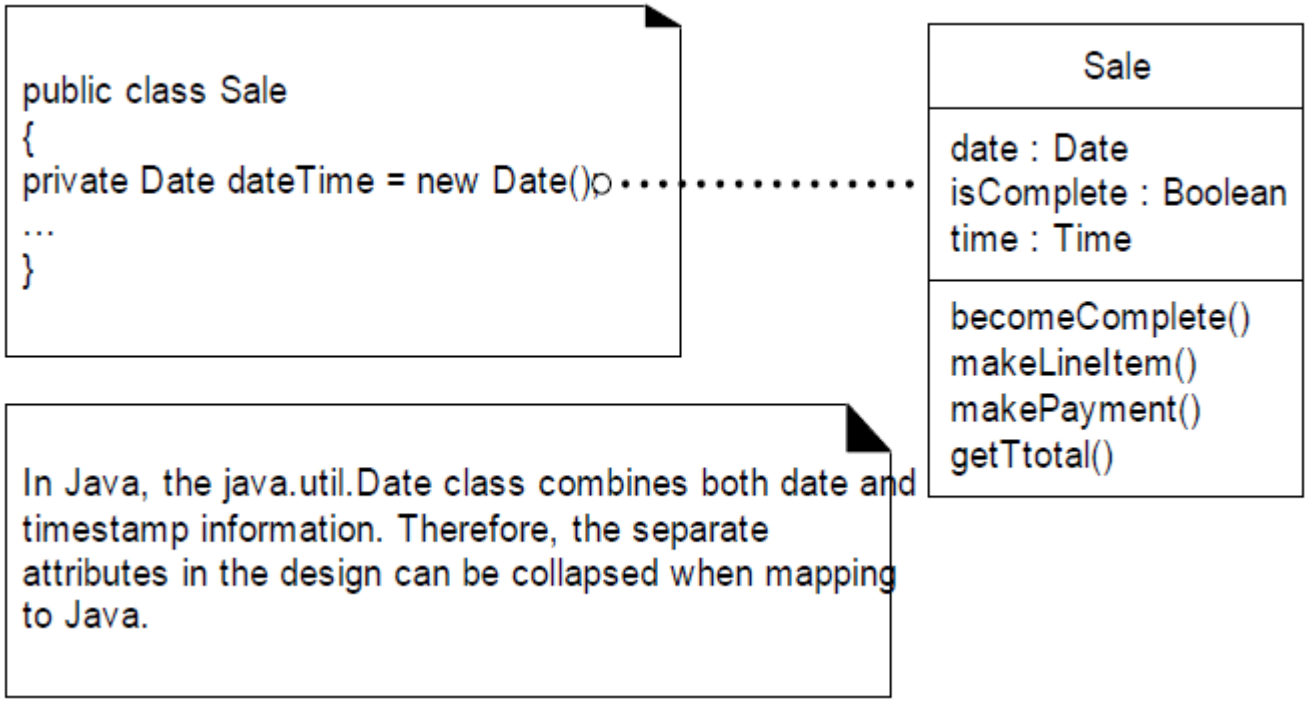


Reference Attributes and Role Names

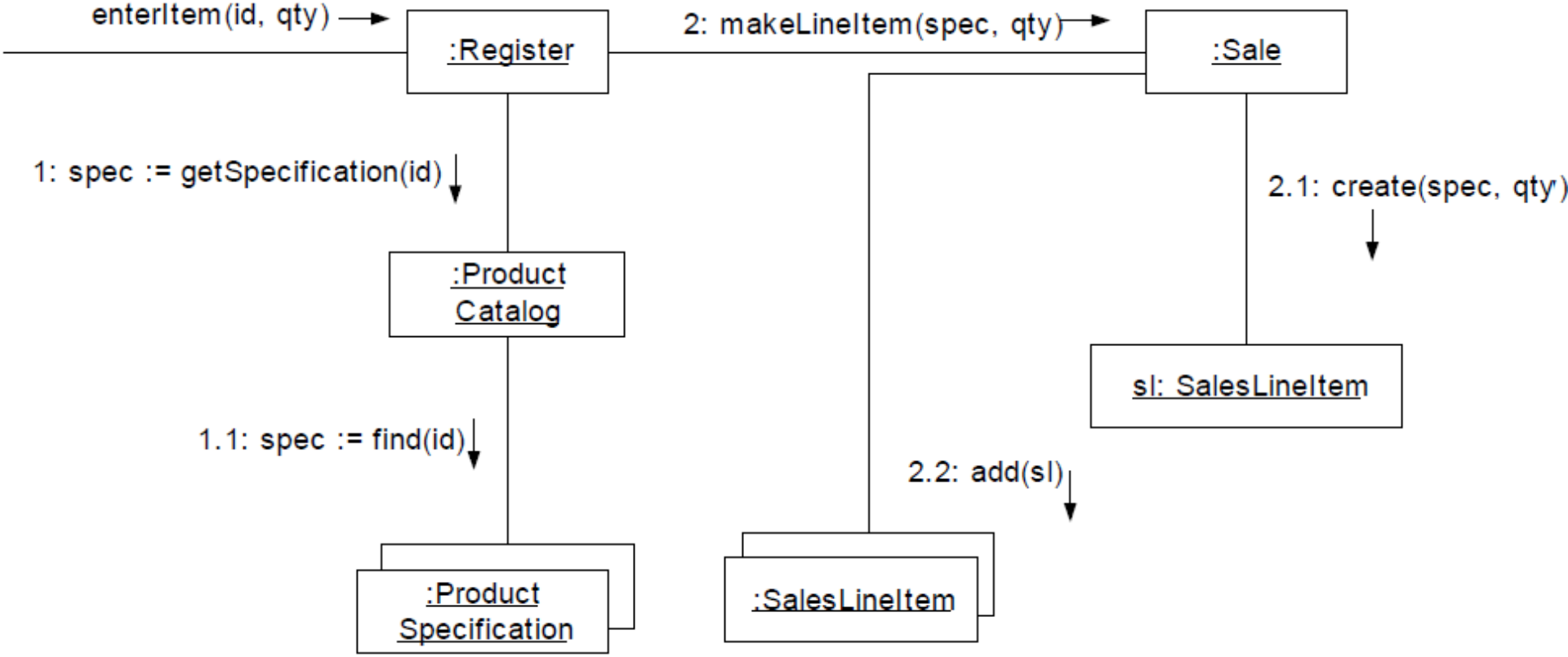
- Role names may be used to generate instance variable names.



Mapping Date and Time to Java

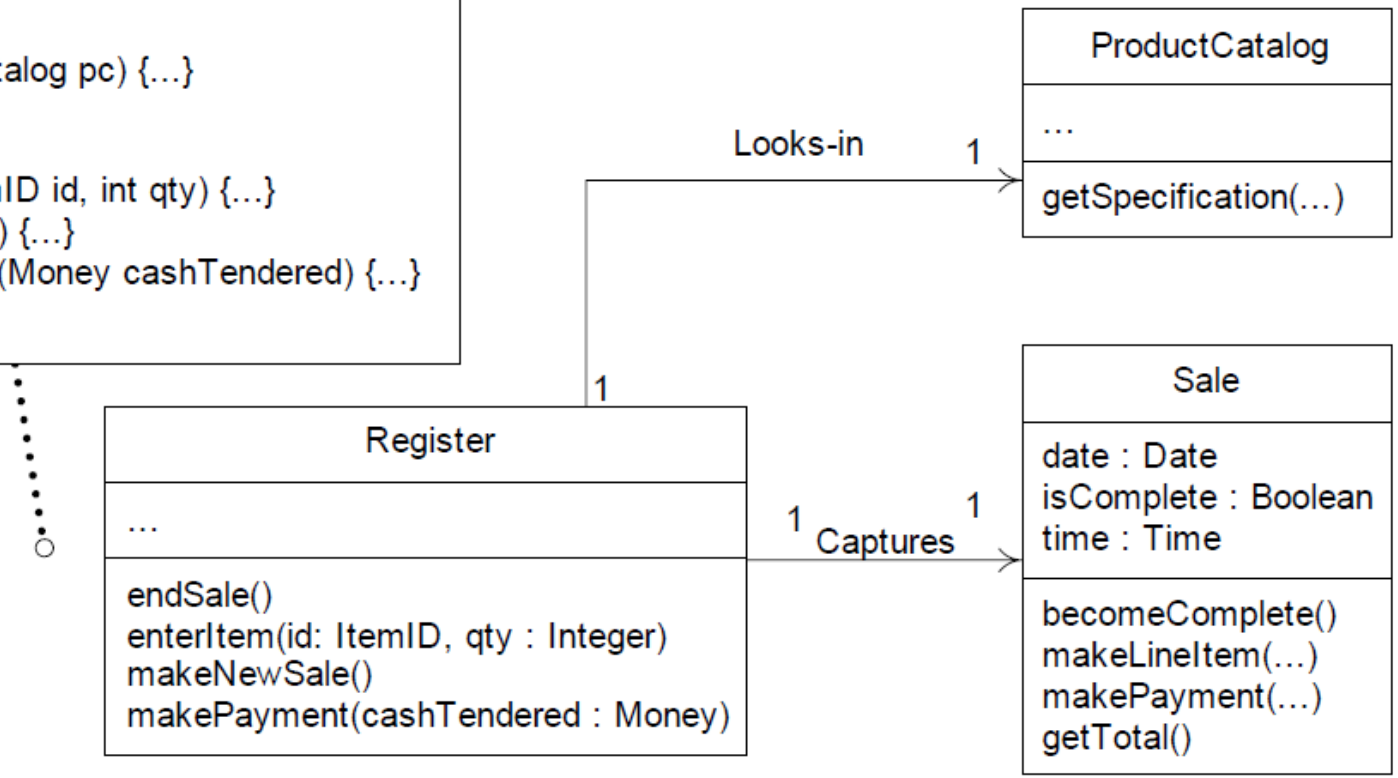


Creating Methods from Interaction Diagrams

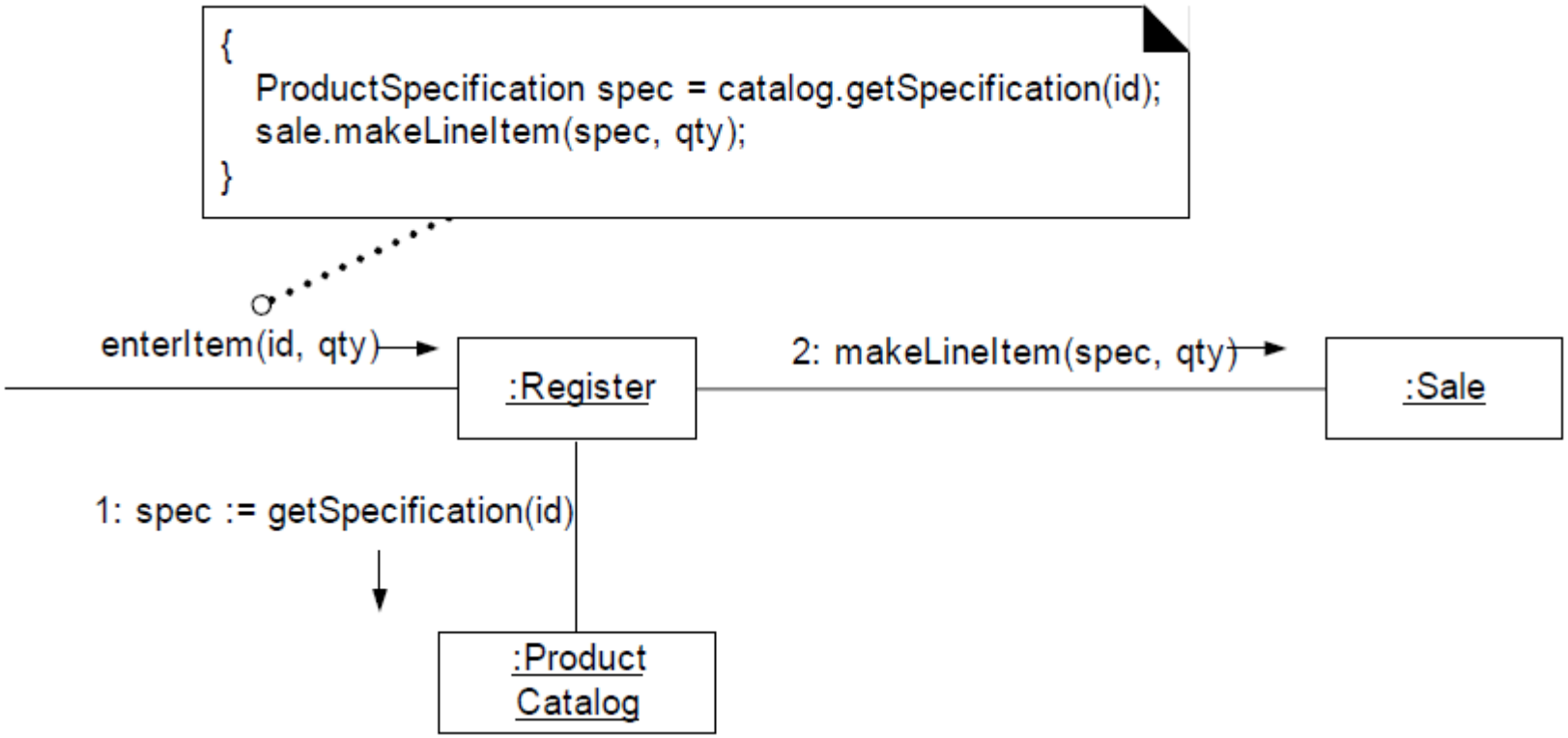


Register-enterItem Method

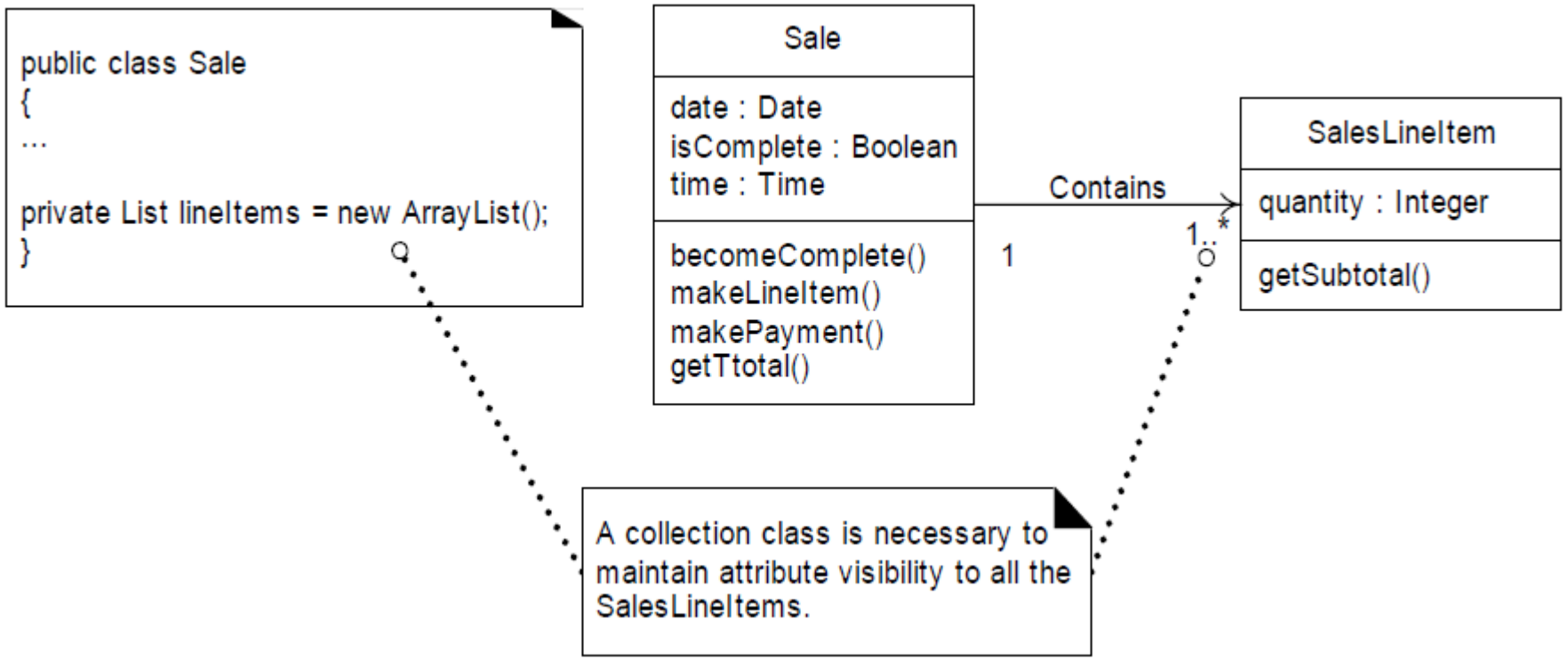
```
public class Register
{
private ProductCatalog catalog;
private Sale sale;

public Register(ProductCatalog pc) {...}

public  void endSale() {...}
public  void enterItem(ItemID id, int qty) {...}
public void makeNewSale() {...}
public  void makePayment(Money cashTendered) {...}
}
```



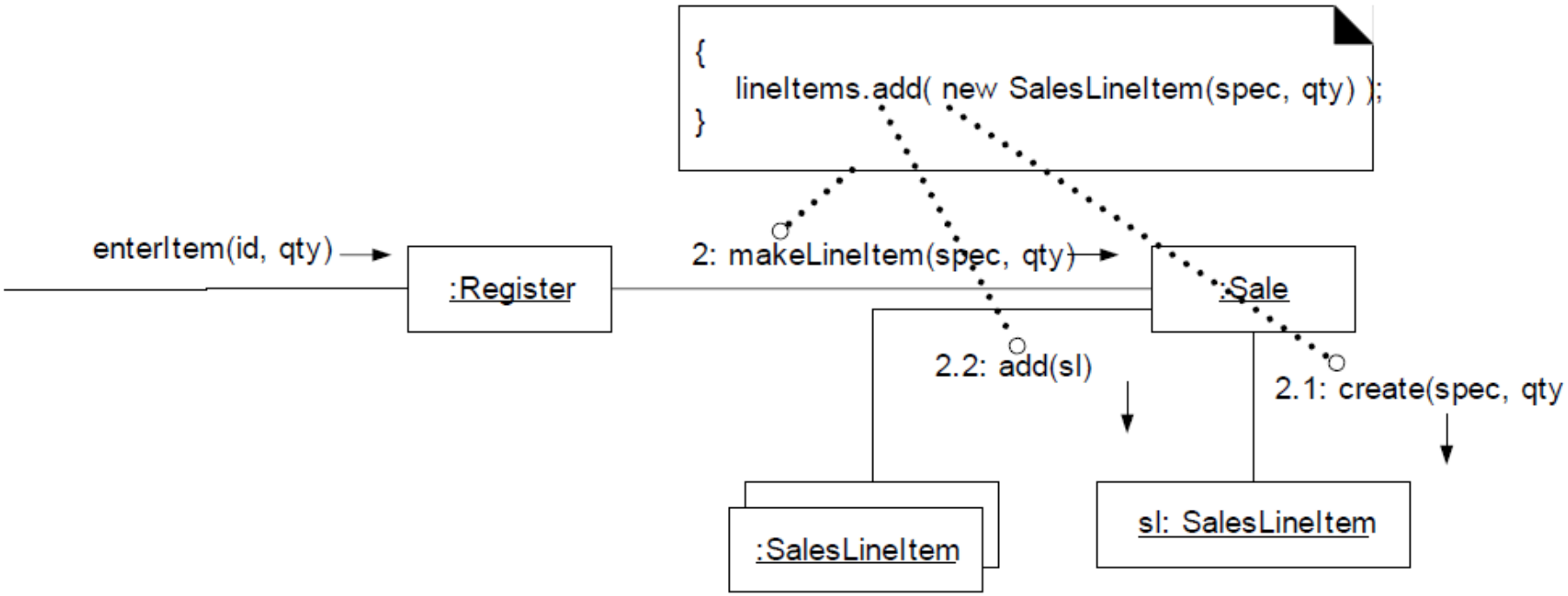
EnterItem Method



Adding Collection



Defining Sale – makeLineItem Method



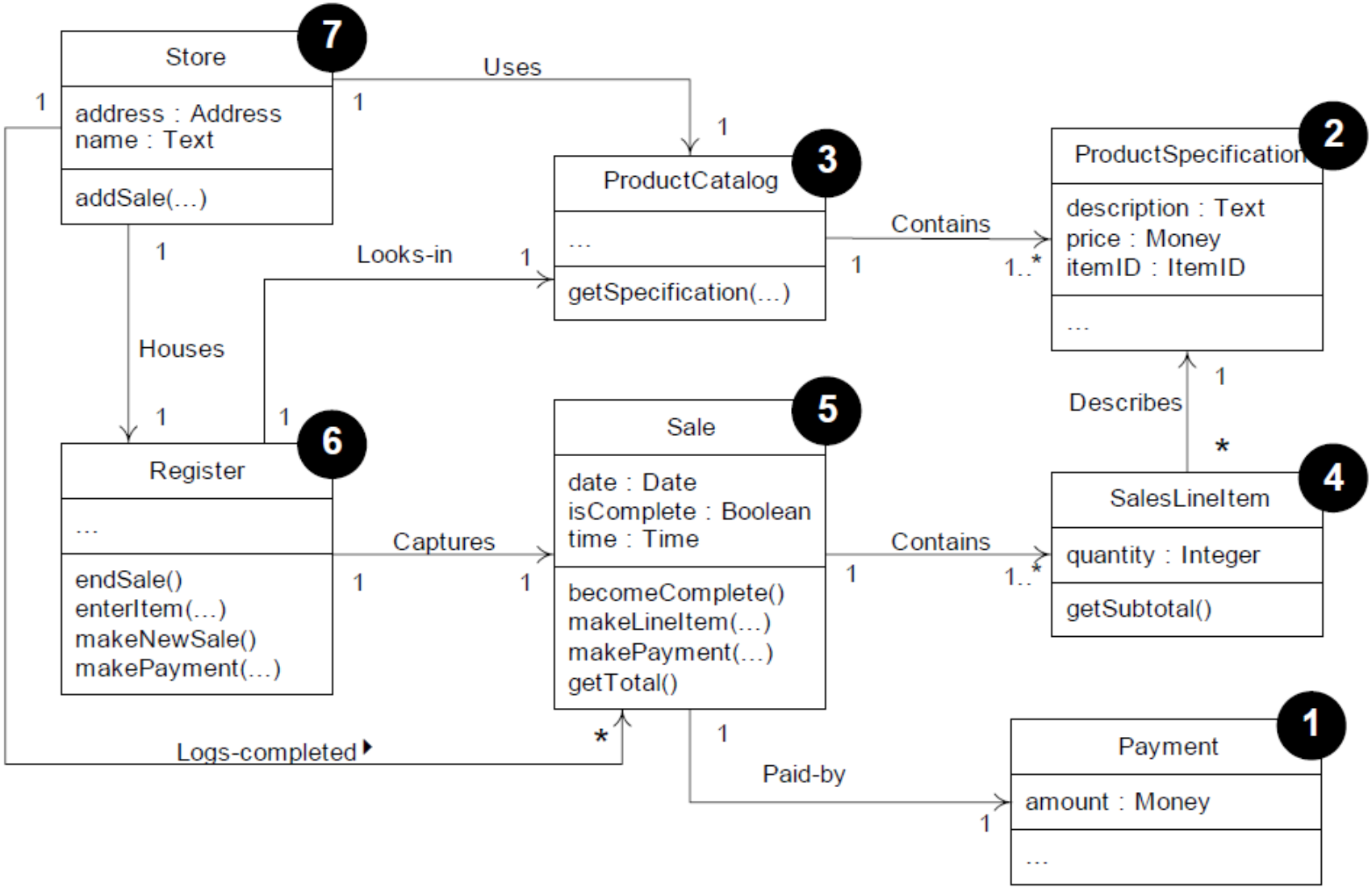
Order of Implementation

- What should be the order?

Order of Implementation

- Classes need to be implemented (and ideally, fully unit tested) from least-coupled to most-coupled.
- For example, possible first classes to implement are either *Payment* or *ProductSpecification;*
- Next are classes only dependent on the prior implementations— *ProductCatalog* or *SalesLineItem.*

Possible Order of Implementation



Sample Code

Class Payment

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

Sample Code

Class ProductSpecification

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

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

   public ItemID getItemlDO { return id;}

   public Money getPrice() { return price; }

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

Sample Code

Class ProductCatalog

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

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

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

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

Sample Code

Class SalesLineItem

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

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

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

Sample Code

Class Sale

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

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

   public void becomeComplete() { isComplete = true; }

   public boolean isComplete() { return isComplete; }

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

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

   public void makePayment( Money cashTendered )
   {
   payment = new Payment( cashTendered ); } }
```

Sample Code

Class Register

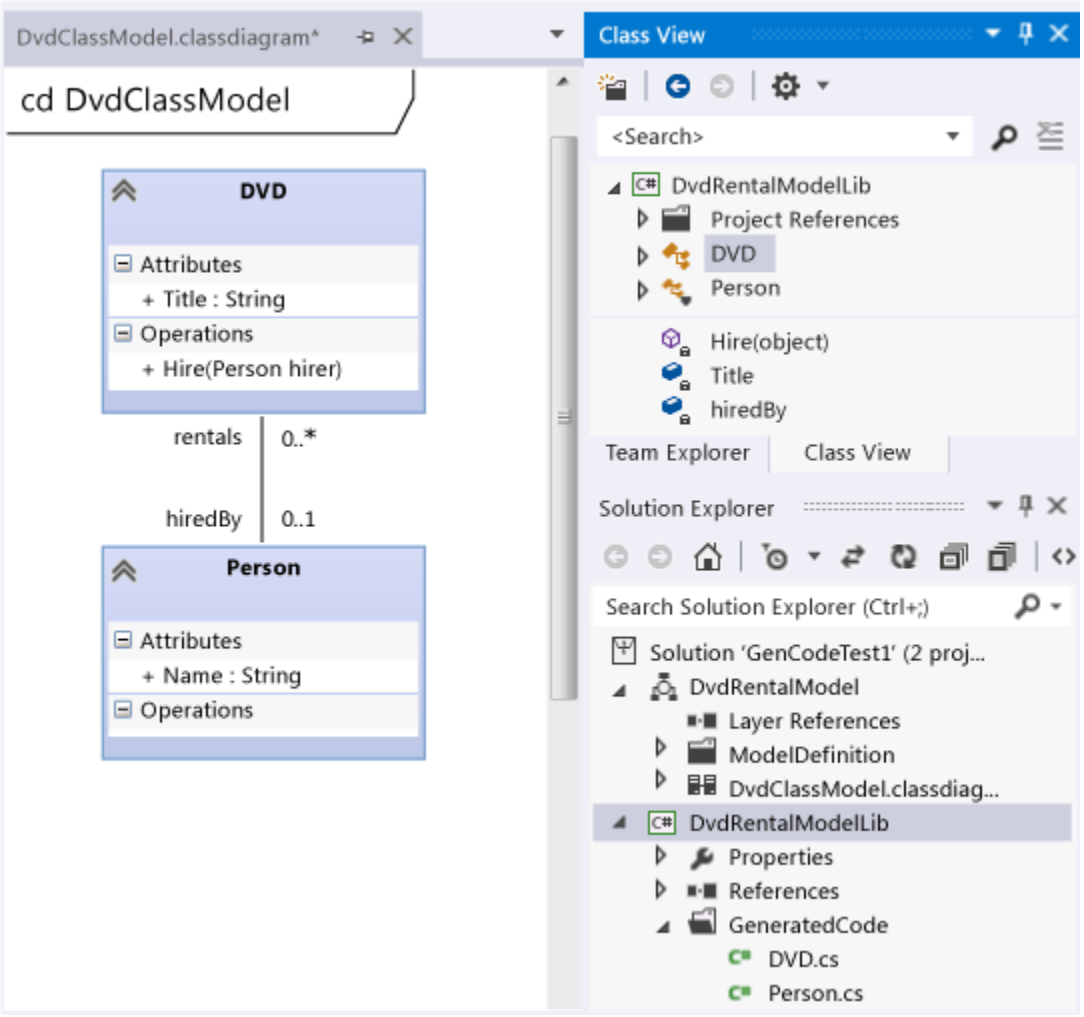
```
public class Register {
    private ProductCatalog catalog;
    private Sale sale;

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

    public void endSaleO {
        sale.becomeComplete();
     }

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

        sale.makeLineItem( spec, quantity ); }

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

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

}
```

Automatic Code Generation

There are many tools, that provide automatic code generation based on UML diagrams which saves lots of time.

- NET **code** from **UML** class diagrams in Visual Studio
- Eclipse **UML** Generators

Automatic Code Generation



Quiz

- Are interactions diagrams useful to create DCDs? If yes, how?
- What is navigability used for?
- How to determine the visibility of attributes and functions in DCDs?
- How to determine the order of the implementation of classes?

Actions

- Review Slides.
- Read Chapter 19 and 20
  - *Applying UML and Patterns*, Craig Larman