IMPLEMENTATION MODEL: MAPPING DESIGNS TO CODE

Introduction

Interaction diagrams and DCDs provide sufficient detail to generate code for the domain layer of objects.

The UML artifacts created during the design work, the **interaction diagrams and DCDs,** will be used as input to the code generation process.

The UP defines the Implementation Model that contains the implementation artifacts such as the source code, database definitions, JSP/XML/HTML pages etc.

Programming and the Development Process

There is prototyping or designing **while programming** as well!!

Modern development tools provide an excellent environment to quickly explore alternate approaches, and some (or even lots) **design-while-programming** is usually worthwhile.

Visual modeling before programming is helpful

The creation of code in an object-oriented programming language—such as Java or C#—is **not part of OOA/D**; it is an end goal.

The artifacts created in the UP Design Model provide some of the information necessary to generate the code.

A strength of OOA/D and OO programming—when used with the UP—is that they **provide an end-to-end roadmap** from requirements through to code.

Creativity and Change during Implementation

Some decision-making and creative work, are accomplished during design work.

However, the **programming work is not a trivial code generation step**—quite the opposite.

Realistically, the results generated during design are an incomplete first step; **during programming and testing, numerous changes will be made** and detailed problems will be uncovered and resolved.

The **design artifacts provide a resilient core** that scales up with elegance and robustness to meet the new problems encountered during programming.

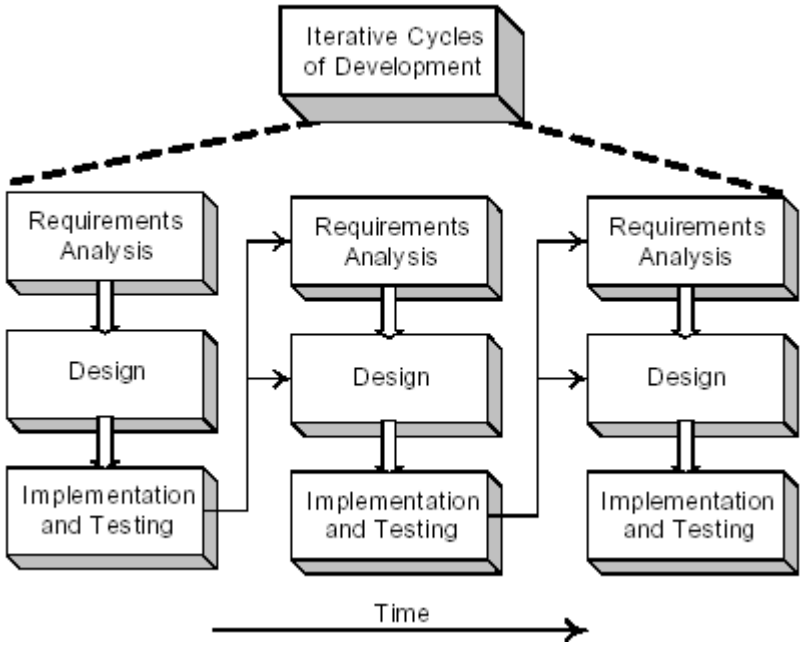
Hence, expect and plan for change and deviation from the design during programming.

Code Changes and the Iterative Process

A strength of an iterative and incremental development process is that the **results of a prior iteration can feed into the beginning of the next iteration**.

Thus, **subsequent analysis and design results are continually being refined and enhanced** from prior implementation work.

For example, when the code in iteration N deviates from the design of iteration N (which it inevitably will), the final design based on the implementation can be input to the analysis and design models of iteration N+l.



An early activity within an iteration is to synchronize the design diagrams; the earlier diagrams of iteration N will not match the final code of iteration N, and they need to be synchronized before being extended with new design results.

Code Changes, CASE Tools, and Reverse-Engineering

It is desirable for the diagrams generated during **design to be semi-automatically updated to reflect changes in the subsequent coding work**.

Ideally this should be done with a CASE tool (like Rational Rose) that can read source code and automatically generate, for example, package, class, and sequence diagrams.

This is an aspect of reverse-engineering—the activity of generating diagrams from source (or sometimes, executable) code.

Mapping Designs to Code

Implementation in an object-oriented programming language requires writing source code for:

- class and interface definitions
- method definitions

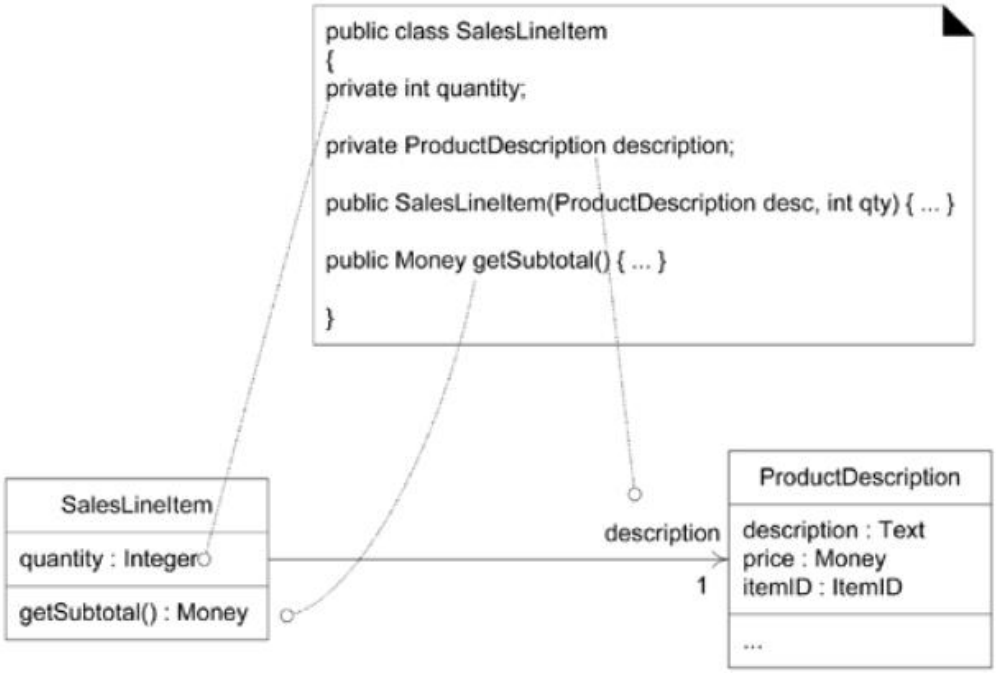
Creating Class Definitions from DCDs

DCDs depict the class or interface name, superclasses, method signatures, and simple attributes of a class.

This is sufficient to create a basic class definition in an object-oriented programming language.

Defining a Class with Methods and Simple Attributes

From the DCD, a mapping to the basic attribute definitions and method signatures for the Java definition of `SalesLineItem` is straightforward

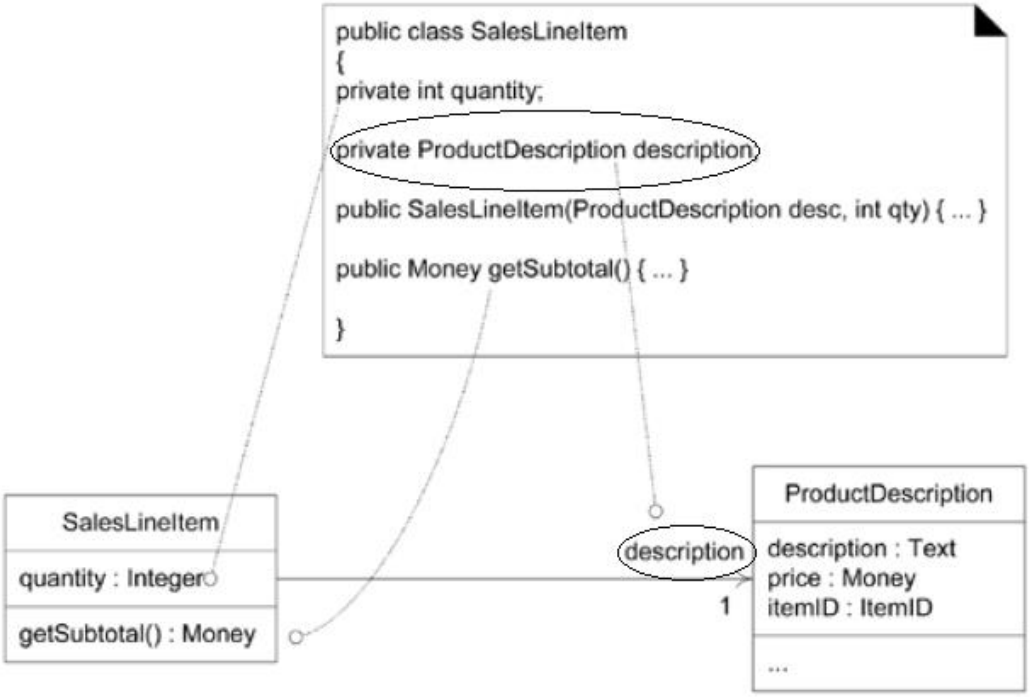


The Java Constructor public `SalesLineItem(ProductDescription desc,int qty)` is created from the `create(desc,qty)` message sent to `SalesLineItem` in the `enterItem` interaction diagram

Adding Reference Attribute

A reference attribute is an attribute that refers to another complex object, **not to a primitive type such as a String, Number**, and so on. (i.e. and object as a attribute of another object)

For example, a `SalesLineItem` has an association to a `ProductDescription`, with navigability to it. This is a reference attribute in class `SalesLineItem` that refers to a `ProductDescription` instance although we have added an instance field to the definition of `SalesLineItem` to point to a `ProductDescription`, it is **not explicitly declared as an attribute in the attribute section of the class box.**



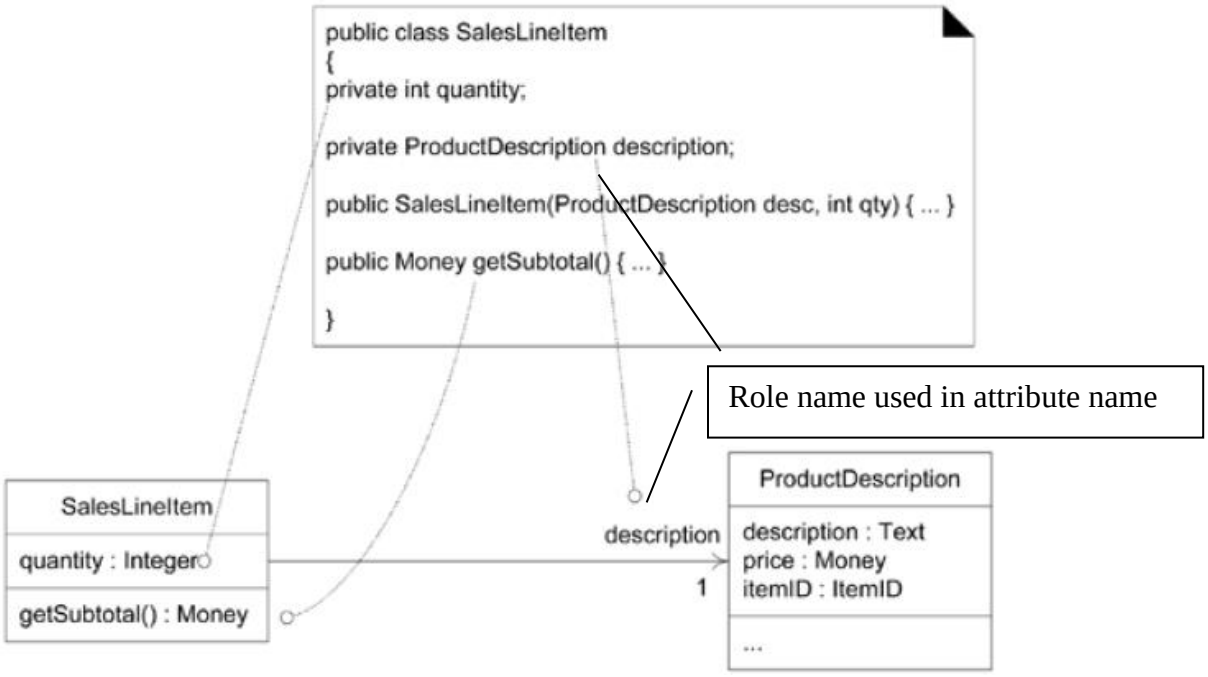
There is a **suggested attribute visibility indicated by the association and navigability (description)** —which is **explicitly defined as an attribute during the code generation phase**. In this case, an object A sees another object B, so that object A can pass message or invoke methods of that object B

Reference Attributes and Role Names

A role name is a name that identifies the role and often provides some semantic context as to the nature of the role.

If a role name is present in a class diagram, use it as the basis for the name of the reference attribute during code generation.

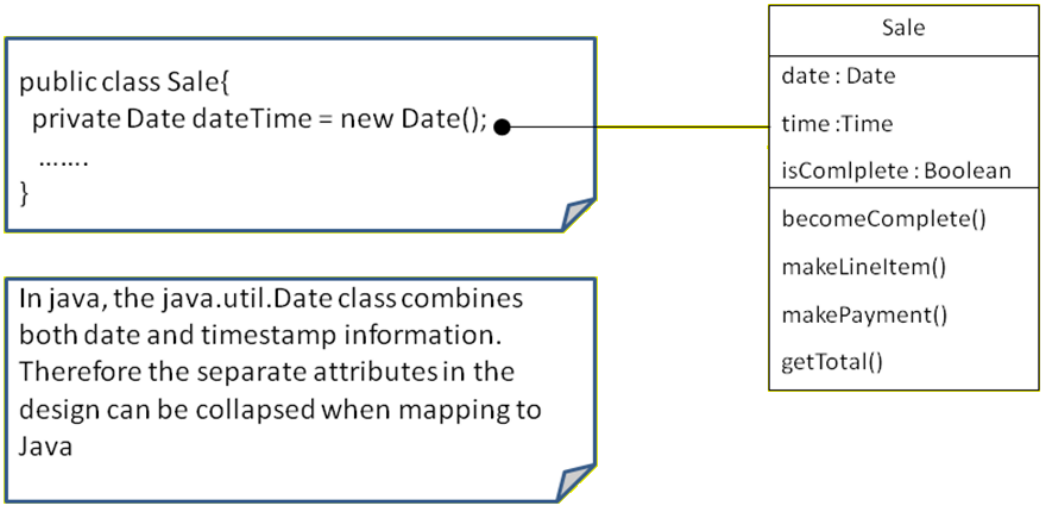
In other words, the role name in the class diagram `description` becomes the name of the reference attribute `description` of the class `SalesLineItem`



Mapping Attributes

In some cases one must consider the mapping of attributes from the design to the code in different languages.

That is , in some languages attributes can be merged into one while mapping from design to code.



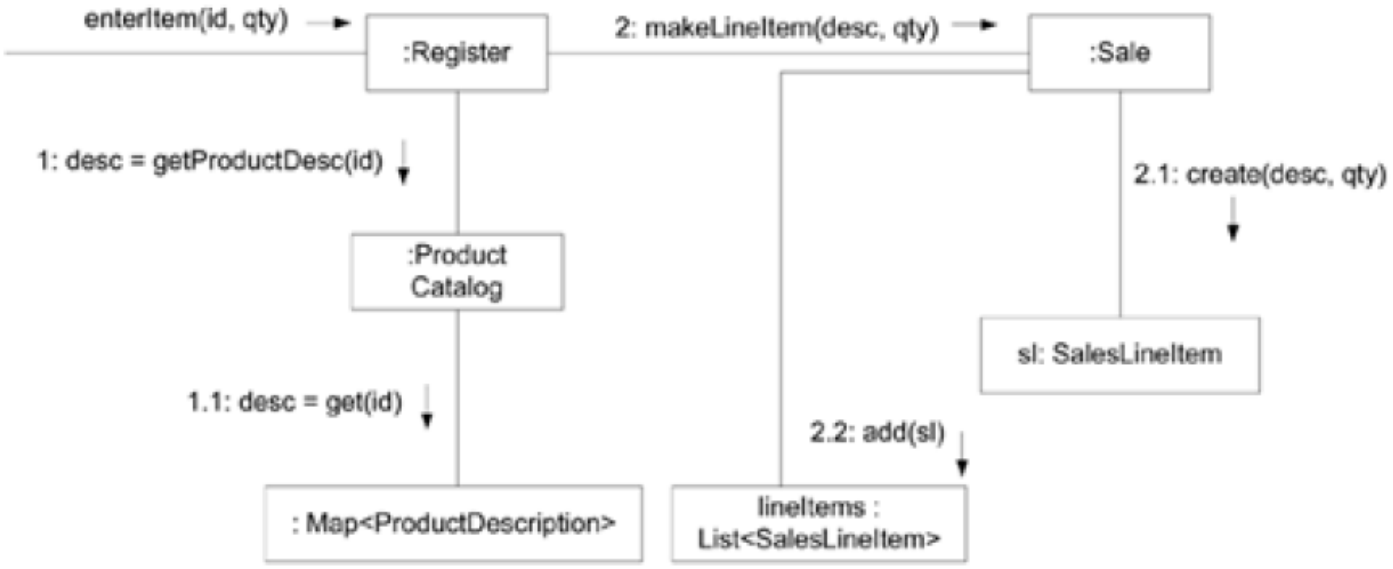
For example, a `Date` class in Java combines both data and time information hence while implementing the class a single variable `dateTime` can be used instead of separate attributes date and time

Creating Methods from Interaction Diagrams

An interaction diagram shows the messages that are sent in response to a method invocation. The sequence of these messages translates to a series of statements in the method definition.

Consider the `enterItem` communication diagram

The `enterItem` interaction diagram illustrates the Java definition of the `enterItem` method.



The `enterItem` message is sent to a Register instance so the enterItem method is defined in the Register class

```
public void enterItem(ItemID itemID, int qty)
```

Message 1: A `getProductDescription` message is sent to the `ProductCatalog` to retrieve `ProductDescription`

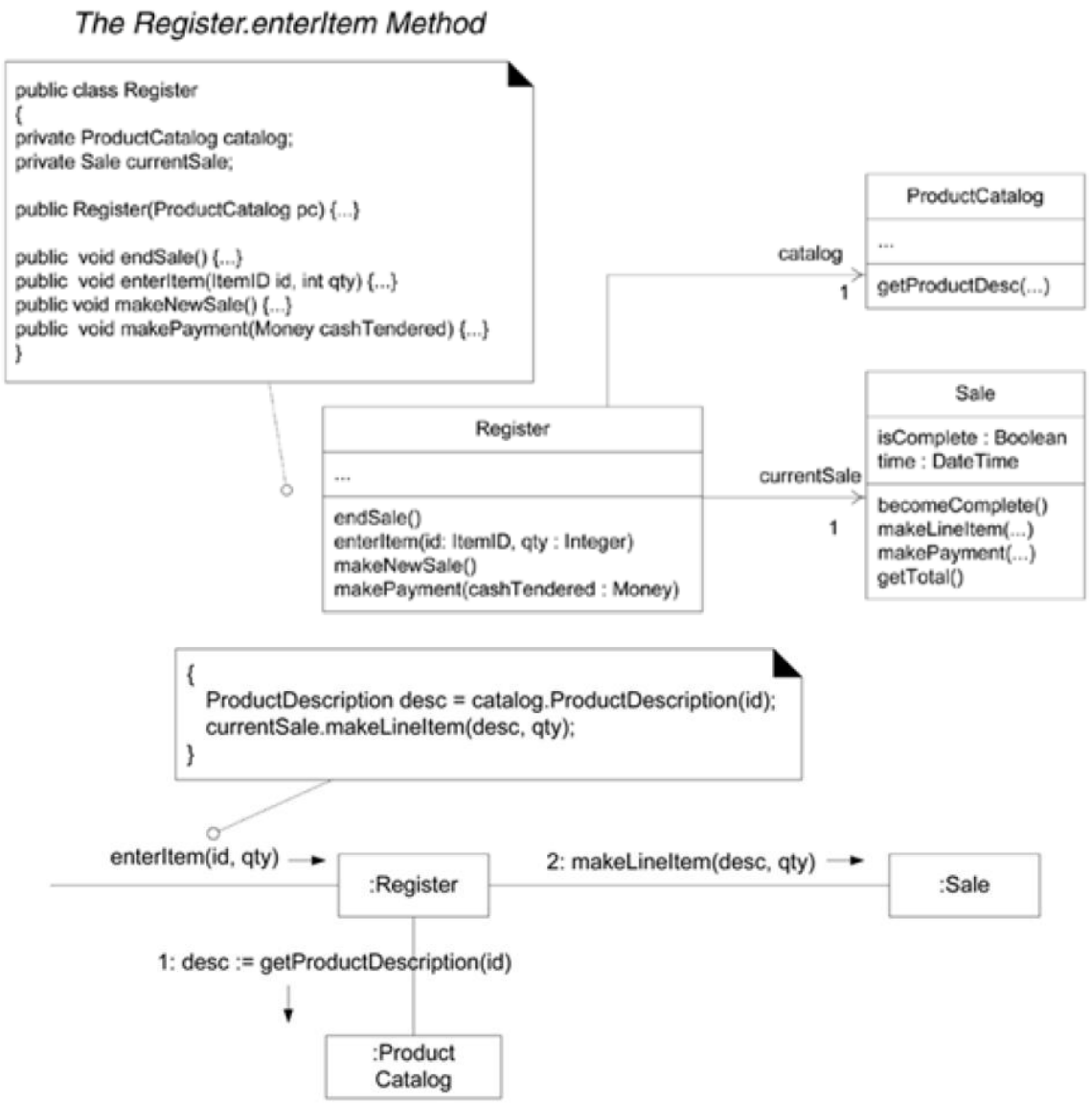
```
ProductDescription desc = catalog.getProductDescription(itemID);
```

Message 2: A `makeLineItem` message is sent to the `Sale`

```
currentSale.makeLineItem (desc,qty);
```

So it takes the form:

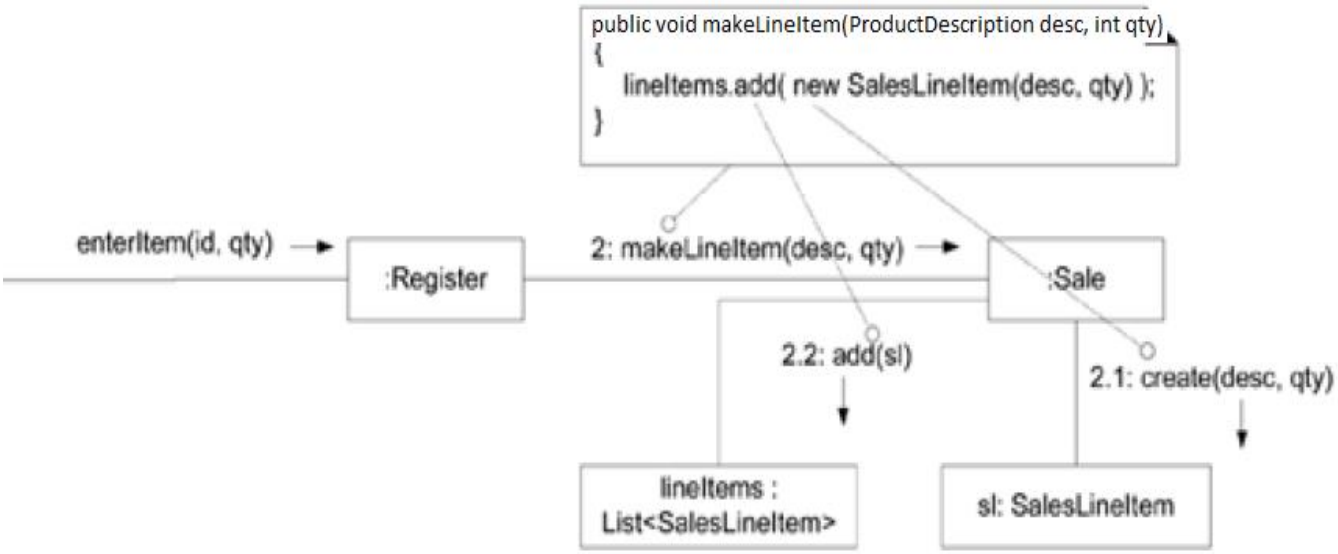
```
public void enterItem(ItemID itemID, int qty){
   ProductDescription desc = catalog.getProductDescription(itemID);
   currentSale.makeLineItem (desc,qty);
}
```



Defining the Sale—`makeLineItem` Method (`sale.makeLineItem`)

As a final example, the `makeLineItem` method of class `Sale` can also be written by inspecting the `enterItem` communication diagram.

Here, in response to the method invocation (`makeLineItem(----)`), the **sale object creates a `saleLineObject`** and **adds that item**, which is done in a single method invocation **`makeItems.add (new SalesLineItem(spec,qty))`**



Collection Classes in Code

It is used to depict relationships line the one to many relationships

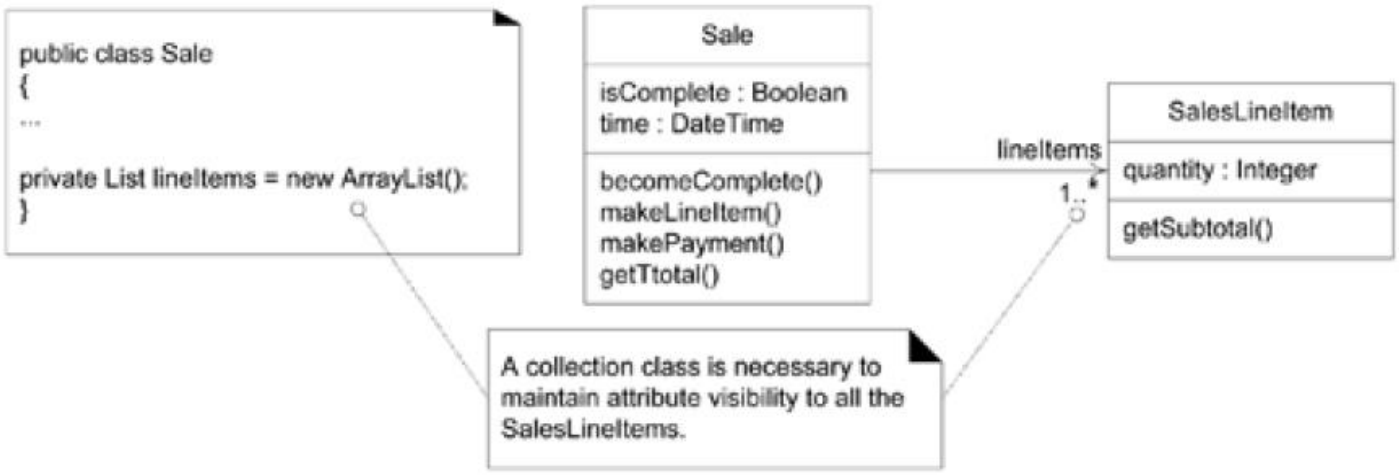
A Sale must maintain visibility to a group of may `SalesLineItem` instances

These relationships are usually maintained in OO programming languages with the collection of objects such as arrays, List or Map

Java libraries contain collection classes like ArrayList, HashMap that maintain list and map interfaces so using ArrayList the Sale can define an attribute that can maintain the ordered list of `SalesLineItem instances`

If an object `im`plements an interface, declare the variable in terms of interface , not in terms of concrete class

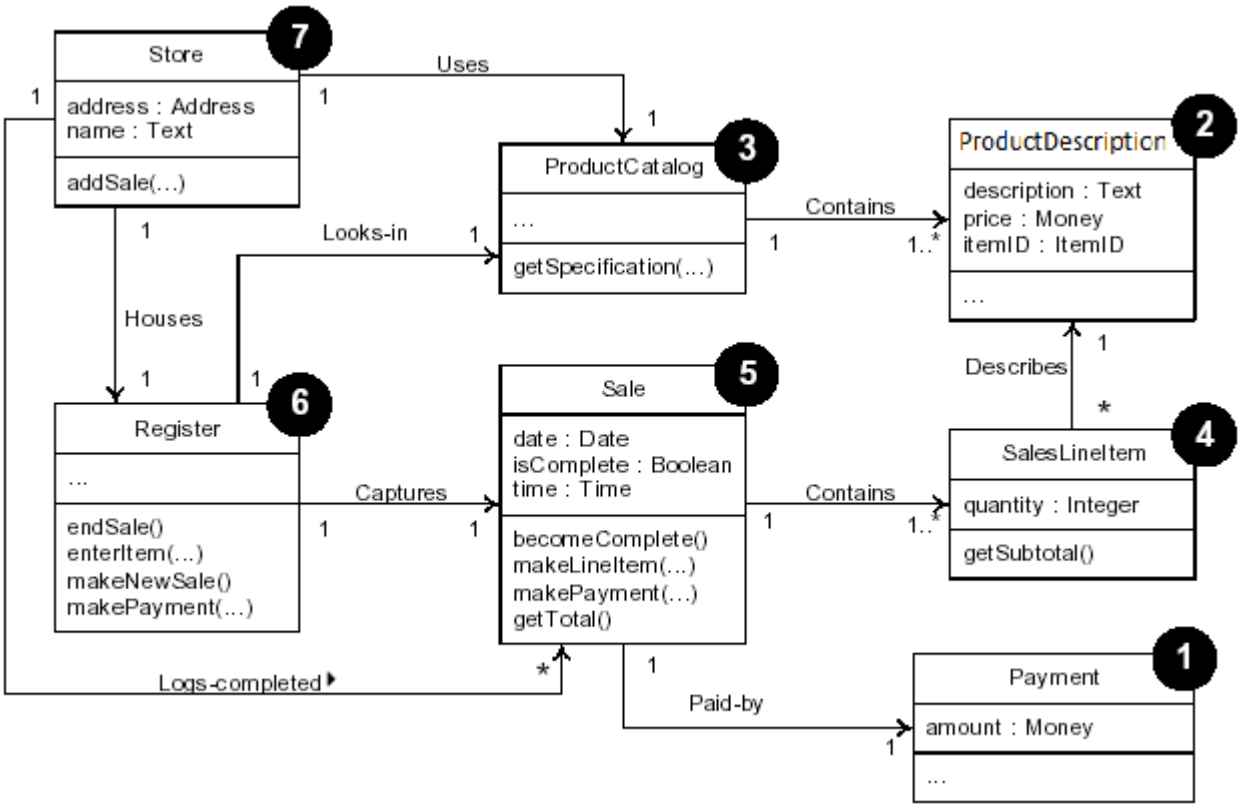
```
private List lineItems = new ArrayList();
```



Ordering of Classes

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 `ProductDescription`; next are classes only dependent on the prior implementations—`ProductCatalog` or `SalesLineItem`.



Exceptions in the UML

In the UML, an Exception is a specialization of a Signal, which is the specification of an asynchronous communication between objects. This means that in interaction diagrams, Exceptions are illustrated as asynchronous messages.

The UML has default syntax for operations but it doesn't include an official solution to show exceptions thrown by an operation

Three solutions:

1. The UML allows the operation syntax to be any other language, such as Java, in addition, some UML CASE tools allow display of operations explicitly in Java syntax :

   ```
   object get(key, class) throws DBUnavailableException, FatalException
   ```

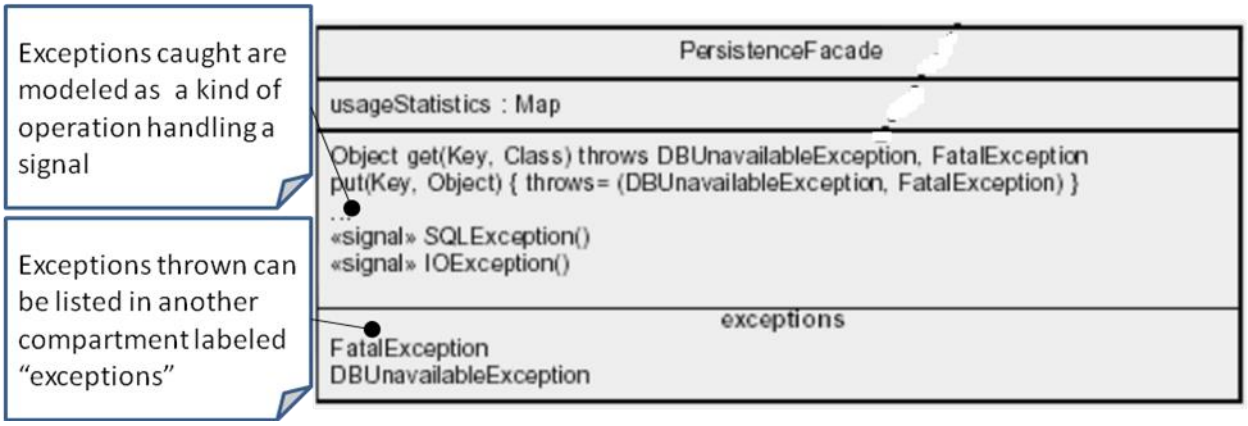
2. The default syntax allows the last element to be a "property string" this is a list of arbitrary property + value pairs such as

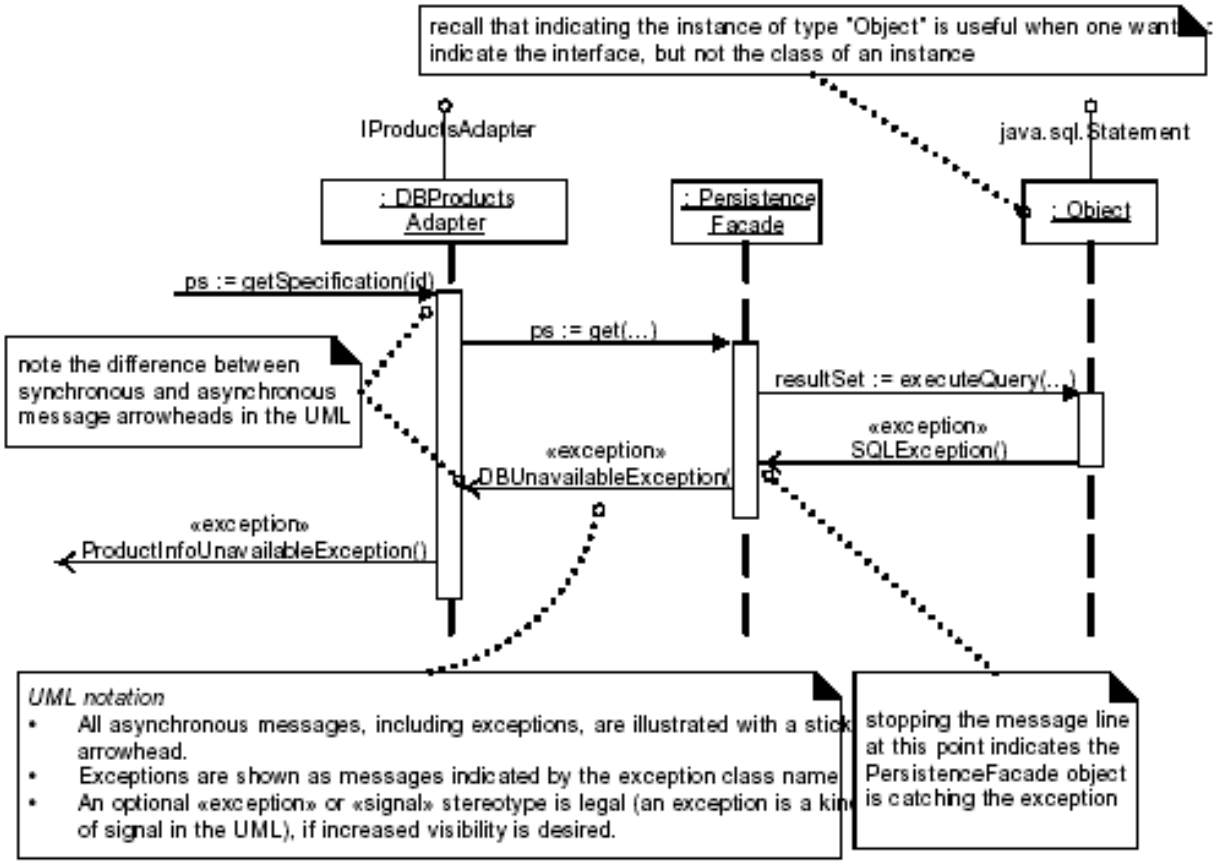
   ```
   { author = Craig, kids =(Hannah, Haley)  }
   ```

   thus

   ```
   put(Object id){throws = (DBUnavailableException, FatalException)}
   ```

3. Some UML CASE tools allow one to specify (in a special dialbox) the exceptions that an operation throws





Class Payment

```
// all classes are probably in a package named
// something like:
package com.foo.nextgen.domain;


public class Payment
{
   private Money amount;

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

Class ProductCatalog

```
public class ProductCatalog
{
   private Map<ItemID, ProductDescription>
         descriptions = new HashMap()<ItemID, ProductDescription>;

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

      ProductDescription desc;
      desc = new ProductDescription( id1, price, "product 1" );
      descriptions.put( id1, desc );
      desc = new ProductDescription( id2, price, "product 2" );
      descriptions.put( id2, desc );
   }

   public ProductDescription getProductDescription( ItemID id )
   {
      return descriptions.get( id );
   }
}
```

Class Register

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

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

   public void endSale()
   {
      currentSale.becomeComplete();
   }


public void endSale()
{
   currentSale.becomeComplete();
}

public void enterItem( ItemID id, int quantity )
{
   ProductDescription desc = catalog.getProductDescription( id );

   currentSale.makeLineItem( desc, quantity );
}

public void makeNewSale()
{
   currentSale = new Sale();
}
public void makePayment( Money cashTendered )
{
   currentSale.makePayment( cashTendered );
}
}
```

Class ProductDescription

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

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

   public ItemID getItemID() { return id;    }

   public Money getPrice() { return price; }
   public String getDescription() { return description; }
}
```

Class Sale

```
public class Sale
{
   private List<SalesLineItem> lineItems =
                           new ArrayList()<SalesLineItem>;
   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
      ( ProductDescription desc, int quantity )
   {
      lineItems.add( new SalesLineItem( desc, quantity ) );
   }
```

```
public Money getTotal()
{
   Money total = new Money();
   Money subtotal = null;

   for ( SalesLineItem lineItem : lineItems )
   {
      subtotal = lineItem.getSubtotal();
      total.add( subtotal );
   }
return total;
}

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

Class SalesLineItem

```
public class SalesLineItem
{
   private int    quantity;
   private    ProductDescription    description;

   public SalesLineItem (ProductDescription desc, int quantity )
   {
      this.description = desc;
      this.quantity = quantity;
   }
   public Money getSubtotal()
   {
      return description.getPrice().times( quantity );
   }
}
```

Class Store

```
public class Store
{
   private ProductCatalog catalog = new ProductCatalog();
   private Register register = new Register( catalog );

   public Register getRegister() { return register; }
}
```