MAPPING DESIGN TO CODE

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

- class and interface definitions
- method definitions

Implementation is discussed in Java

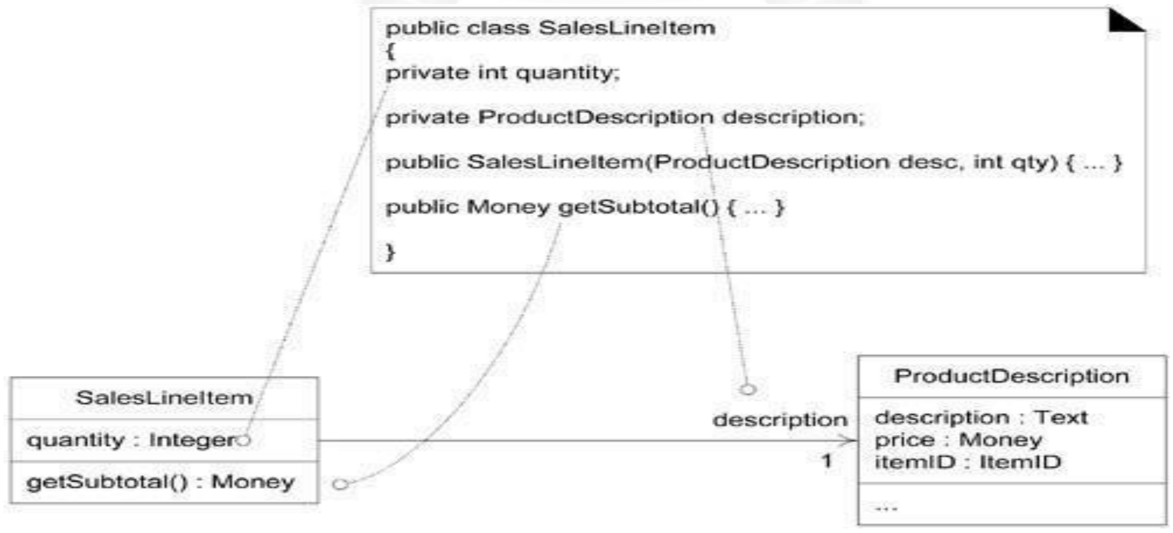
1. Creating Class Definitions from Design Class Diagrams(DCD)
2. Creating Methods from Interaction Diagrams
3. Collection Classes in Code
4. Exceptions and Error Handling
5. Order of Implementation
6. Test-Driven or Test-First Development

Creating Class Definitions from DCDs

DCDs depict the class or interface name, superclasses, operation signatures, and attributes of a class. If the DCD was drawn in a UML tool, it can generate the basic class definition from the diagrams.

Defining a Class with Method Signatures and Attributes

From the DCD, a mapping to the attribute definitions (Java fields) and method signatures for the Java definition of SalesLineItem is straightforward.



```
public class SalesLineItem
{
private int quantity;

private ProductDescription description;

public SalesLineItem(ProductDescription desc, int qty) { ... }

public Money getSubtotal() { ... }

}
```

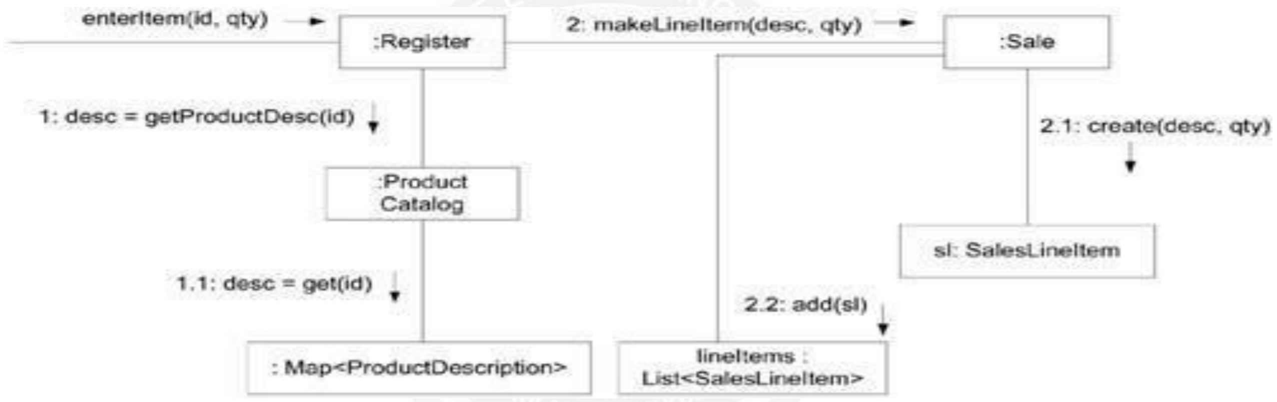
SalesLineItem in Java.

The addition in the source code of the Java constructor SalesLineItem(…). It is derived from the create(desc, qty) message sent to a SalesLineItem in the enterItem interaction diagram. This indicates, in Java, that a constructor supporting these parameters is required.

<u>Creating Methods from Interaction Diagrams</u>

The sequence of the messages in an interaction diagram translates to a series of statements in the method definitions. The enterItem interaction diagram illustrates the Java definition of the enterItem method. For this example, we will explore the implementation of the Register and its enterItem method.

A Java definition of the Register class.is given below :



The enterItem interaction diagram.

The enterItem message is sent to a Register instance; therefore, the enterItem method is defined in class Register.

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

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

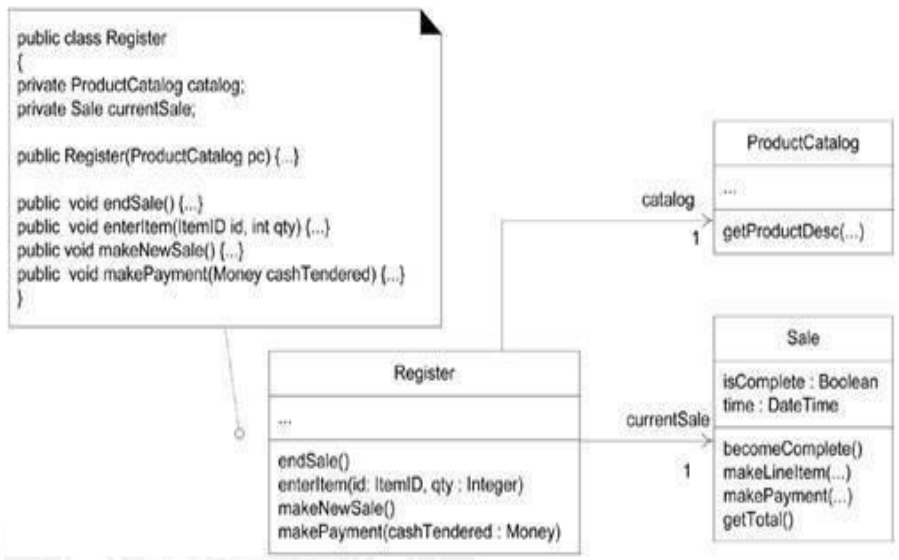
```
ProductDescription desc = catalog.getProductDescription(itemID);
```

Message 2: The makeLineItem message is sent to the Sale.

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

The Register class

The Register.enterItem Method

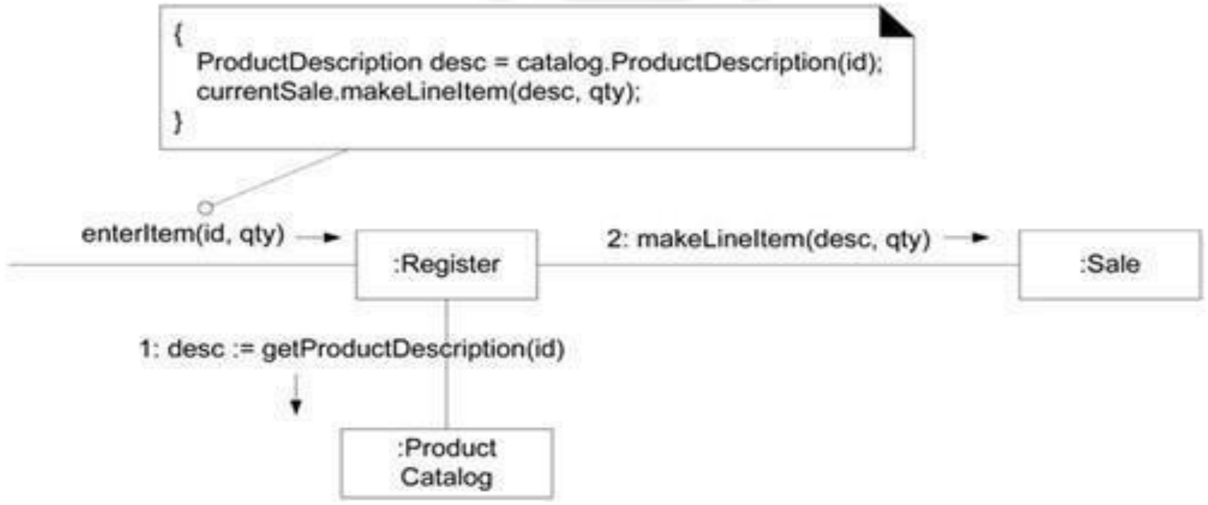


```
public class Register
{
private ProductCatalog catalog;
private Sale currentSale;

public Register(ProductCatalog pc) {...}

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

The enterItem method.



```
{
  ProductDescription desc = catalog.ProductDescription(id);
  currentSale.makeLineItem(desc, qty);
}
```

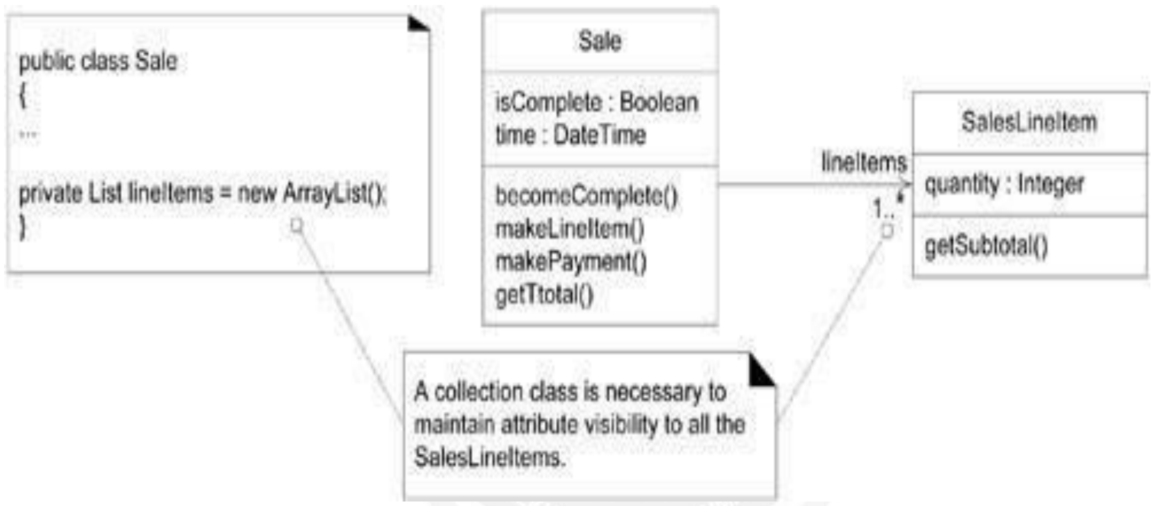
enterItem(id, qty)
:Register
2: makeLineItem(desc, qty)
:Sale
1: desc := getProductDescription(id)
:Product Catalog

Collection Classes in Code

The enterItem method.

One-to-many relationships are common. For example, a Sale must maintain visibility to a group of many SalesLineItem instances. In OO programming languages, these relationships are usually implemented with the introduction of a collection object, such as a List or Map, or even a simple array.

Adding a collection.



For example, the Java libraries contain collection classes such as ArrayList and HashMap, which implement the List and Map interfaces, respectively. Using ArrayList, the Sale class can define an attribute that maintains an ordered list of SalesLineItem instances.

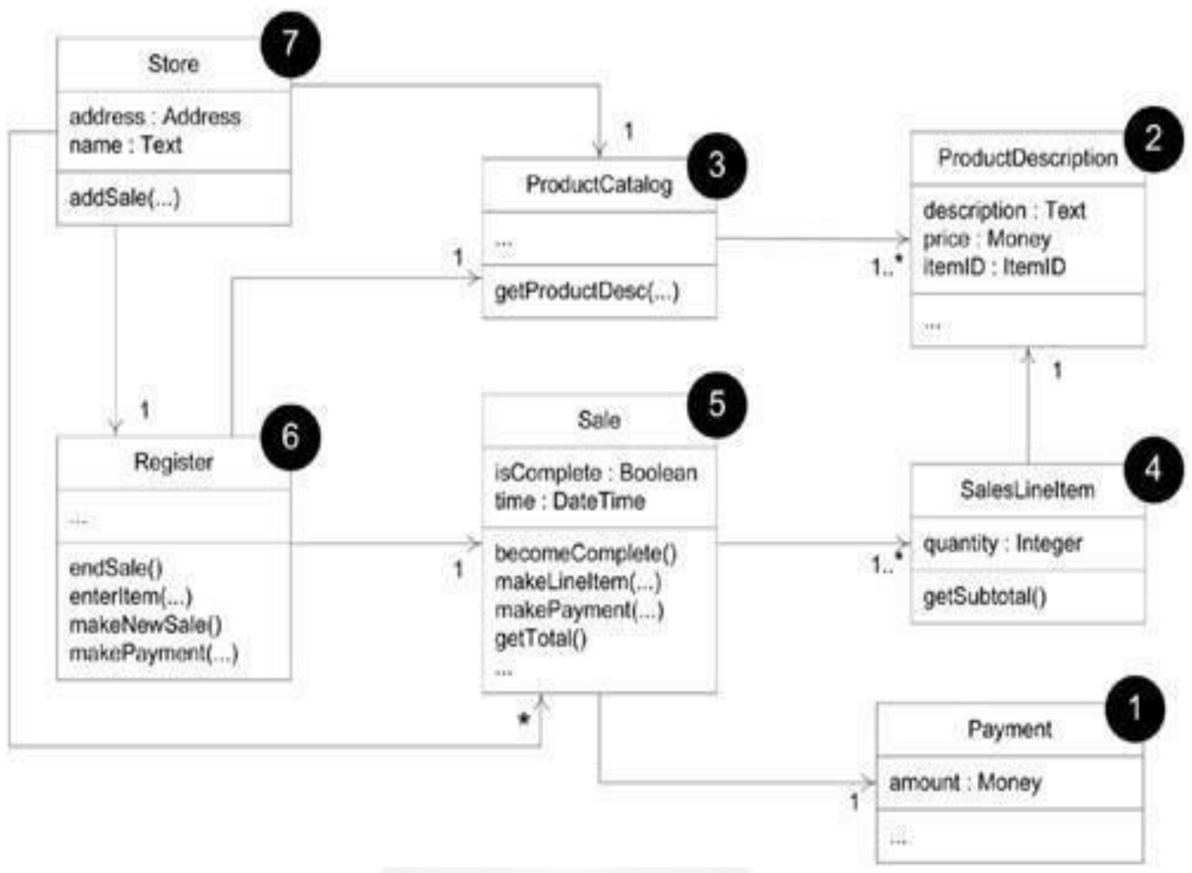
Exceptions and Error Handling

In terms of the UML, exceptions can be indicated in the property strings of messages and operation declarations .

Order of Implementation

Classes need to be implemented 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 implementationsProductCatalog or SalesLineItem

Possible order of class implementation and testing.



<u>Test-Driven or Test-First Development</u>

An excellent practice promoted by the iterative and agile XP method and applicable to the UP (as most XP practices are), is test-driven development (TDD).In OO unit testing TDD- style, test code is written before the class to be tested, and the developer writes unit testing code for nearly all production code.

The basic rhythm is

- to write a little test code,
- write a little production code
- make it pass the test
- write some more test code, and so forth.

Example

Suppose if we create TDD for the Sale class. Before programming the Sale class, we write a unit testing method in a SaleTest class that does the following:

Each testing method follows this pattern:

1. Create the fixture.
2. Do something to it (some operation that you want to test).

3. Evaluate that the results are as expected.

Example:

```
public class SaleTest extends TestCase
{
    // …
   // test the Sale.makeLineItem method

  public void testMakeLineItem()
  {
      // STEP 1: CREATE THE FIXTURE

      // -this is the object to test ,it is an idiom to name it
'fixture'

    Sale fixture = new Sale();

       // set up supporting objects for the test
    Money total = new Money( 7.5
    ); Money price = new Money(
    2.5 ); ItemID id = new
    ItemID( 1 );
    ProductDescription desc =
             new ProductDescription( id, price, "product 1" );

       // STEP 2: EXECUTE THE METHOD TO TEST

       // NOTE: We write this code **imagining** there
       // is a makeLineItem method. This act of imagination
       // test makeLineItem

    sale.makeLineItem( desc, 1 );
    sale.makeLineItem( desc, 2 );


       // STEP 3: EVALUATE THE RESULTS
```

```
        // there could be many assertTrue statements
        // for a complex evaluation
        // verify the total is 7.5
    assertTrue( sale.getTotal().equals( total ));
}
}
```

NextGen POS Program Solution

Class Store

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

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 ProductCatalog

```
public class ProductCatalog
{

 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 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 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 Register

```
public class Register
{
 private ProductCatalog catalog;
 private Sale currentSale;
 public Register( ProductCatalog catalog )
 {    this.catalog = catalog;   }

 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 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 );
  }
}
```