Object-Oriented Software Analysis and Design

School of Computer Science
University of Windsor

Designing for Visibility

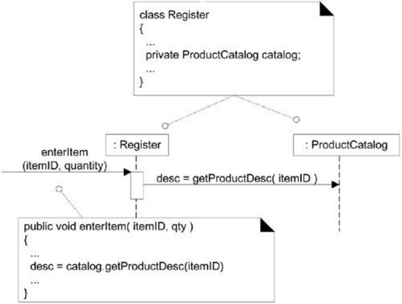
Visibility Between Objects

- The designs created for the system operations (`enterItem`, and so on) illustrate messages between objects.
- For a sender object to send a message to a receiver object, the sender must be visible to the receiver—the sender must have some kind of reference or pointer to the receiver object.

Visibility Between Objects (contd.)

For example,

the `getProductDescription` message sent from a `Register` to a `ProductCatalog` implies that the `ProductCatalog` instance is visible to the `Register` instance, as shown in the Figure below



```
class Register
{
  ...
  private ProductCatalog catalog;
  ...
}
```

```
public void enterItem( itemID, qty )
{
  ...
  desc = catalog.getProductDesc(itemID)
  ...
}
```

Figure: *Visibility from the `Register` to `ProductCatalog` is required.*

What is Visibility?

- In common usage, visibility is the ability of an object to "see" or have a reference to another object.
- More generally, it is related to the issue of scope: Is one resource (such as an instance) within the scope of another?
- For an object `A` to send a message to an object `B`, `B` must be visible to `A`.
- There are four common ways that visibility can be achieved from object `A` to object `B`:
  1. **Attribute visibility** `B` is an attribute of `A`.
  2. **Parameter visibility** `B` is a parameter of a method of `A`.
  3. **Local visibility** `B` is a (non-parameter) local object in a method of `A`.
  4. **Global visibility** `B` is in some way globally visible.

What is Visibility? (contd.)

For example,

to create an interaction diagram in which a message is sent from a `Register` instance to a `ProductCatalog` instance, the `Register` must have visibility to the `ProductCatalog`.
A typical visibility solution is that a reference to the `ProductCatalog` instance is maintained as an attribute of the `Register`.

Attribute Visibility

- Attribute visibility from `A` to `B` exists when `B` is an attribute of `A`.

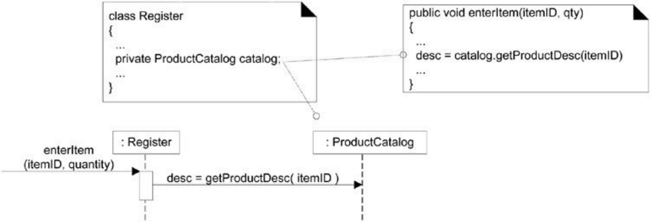
- It is a relatively permanent visibility because it persists as long as `A` and `B` exist.

- This is a very common form of visibility in object-oriented systems.

Attribute Visibility (contd.)

To illustrate,

in a Java class definition for `Register`, a `Register` instance may have attribute visibility to a `ProductCatalog`, since it is an attribute (Java instance variable) of the `Register`.



```
class Register
{
  ...
  private ProductCatalog catalog;
  ...
}
```

```
public void enterItem(itemID, qty)
{
  ...
  desc = catalog.getProductDesc(itemID)
  ...
}
```

Figure: *Attribute visibility.*

Parameter Visibility

- Parameter visibility from `A` to `B` exists when `B` is passed as a parameter to a method of `A`. It is a relatively temporary visibility because it persists only within the scope of the method.

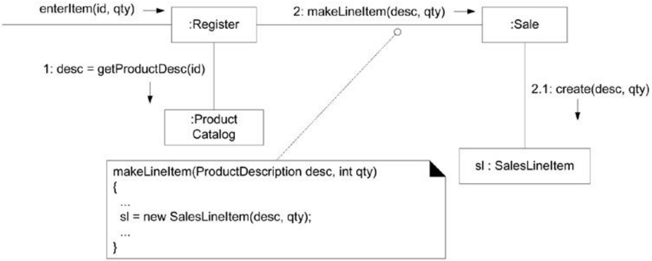
- After attribute visibility, it is the second most common form of visibility in object-oriented systems.

Parameter Visibility (contd.)

To illustrate,

when the `makeLineItem` message is sent to a `Sale` instance, a `ProductDescription` instance is passed as a parameter.

Within the scope of the `makeLineItem` method, the `Sale` has parameter visibility to a `ProductDescription`.

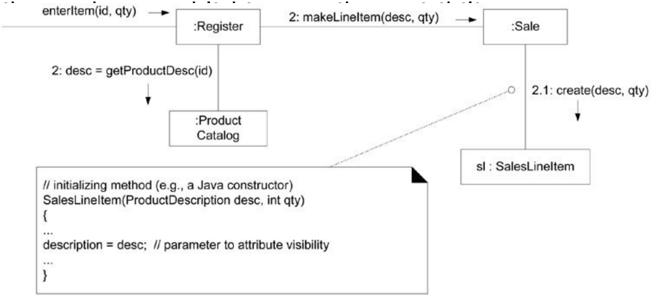


```
makeLineItem(ProductDescription desc, int qty)
{
  ...
  sl = new SalesLineItem(desc, qty);
  ...
}
```

Figure: *Parameter visibility.*

Parameter Visibility (contd.)

It is common to transform parameter visibility into attribute visibility. When the `Sale` creates a new `SalesLineItem`, it passes the `ProductDescription` in to its initializing method (in C++ or Java, this would be its constructor). Within the initializing method, the parameter is assigned to an attr



```
// initializing method (e.g., a Java constructor)
SalesLineItem(ProductDescription desc, int qty)
{
...
description = desc;  // parameter to attribute visibility
...
}
```

Figure: *Parameter to attribute visibility.*

Local Visibility

- Local visibility from `A` to `B` exists when `B` is declared as a local object within a method of `A`.

- It is a relatively temporary visibility because it persists only within the scope of the method.

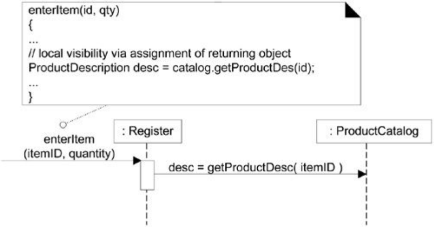
- After parameter visibility, it is the third most common form of visibility in object-oriented systems.

- Two common means by which local visibility is achieved are:
  1. Create a new local instance and assign it to a local variable.
  2. Assign the returning object from a method invocation to a local variable.

- As with parameter visibility, it is common to transform locally declared visibility into attribute visibility.

Local Visibility (contd.)

An example of the second variation (assigning the returning object to a local variable) can be found in the `enterItem` method of c



```
enterItem(id, qty)
{
...
// local visibility via assignment of returning object
ProductDescription desc = catalog.getProductDes(id);
...
}
```

Figure: *Local visibility.*

Global Visibility

- Global visibility from `A` to `B` exists when `B` is global to `A`.

- It is a relatively permanent visibility because it persists as long as `A` and `B` exist.

- It is the least common form of visibility in object-oriented systems.

- One way to achieve global visibility is to assign an instance to a global variable, which is possible in some languages, such as C++, but not others, such as Java.

- The preferred method to achieve global visibility is to use the **Singleton** pattern.

Mapping Designs to Code

Introduction

- The UML artifacts created during the design work - the **interaction diagram**s and **DCD**s - will be used as input to the code generation process.
- In UP terms, there exists an **Implementation Model**:
  - This is all the implementation artifacts, such as the **source code**, **database definitions**, **JSP/XML/HTML pages**, and so forth. Thus, the code being created in this phase can be considered part of the **UP Implementation Model**.

Programming and Iterative, Evolutionary Development

- The creation of code in an OO language - such as Java or C# - is not part of OOA/D—it's an **end goal**.
- The artifacts created in the **Design Model** provide some of the information necessary to generate the code.
- A strength of use cases plus OOA/D plus OO programming 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** was accomplished during design work.
- However, in general, the programming work is not a trivial code generation step - quite the opposite!
- Realistically, the results generated during design modeling are an **incomplete first step**; during programming and testing, myriad changes will be made and detailed problems will be uncovered and resolved.
- The **ideas and understanding** (not the diagrams or documents!) generated during OO design modeling will provide a **great base that scales up with elegance and robustness** to meet the new problems encountered during programming.
- **Expect and plan for lots of change and deviation** from the design during programming.

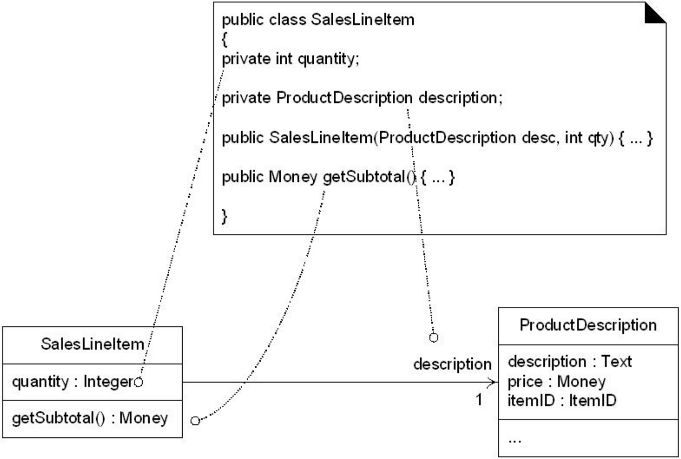
Mapping Designs to Code

- Implementation in an object-oriented language requires writing source code for:
  - class and interface definitions
  - method definitions

Creating Classes from DCDs

- DCDs depict the **class or interface name**, **superclasses**, **operation signatures**, and **attributes** of a class.
- This is sufficient to create a basic class definition in an OO language.
- 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



```
public class SalesLineItem
{
private int quantity;

private ProductDescription description;

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

public Money getSubtotal() { ... }

}
```

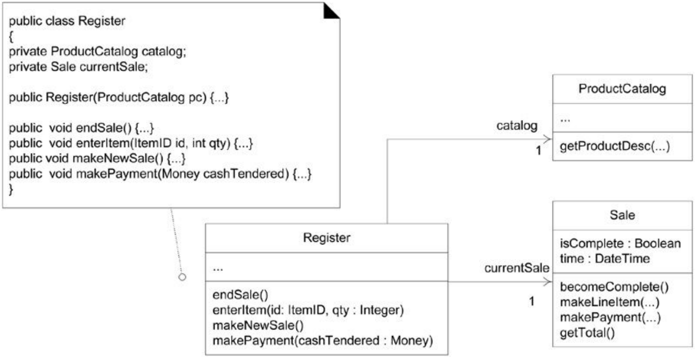
Creating Methods from Interaction Diagrams

- The sequence of the messages in an interaction diagram **translates** to a series of statements in the method definitions.

- In the following example, we will explore the implementation of the `Register` and its `enterItem` method.

Creating Methods from Interaction Diagrams (contd.)

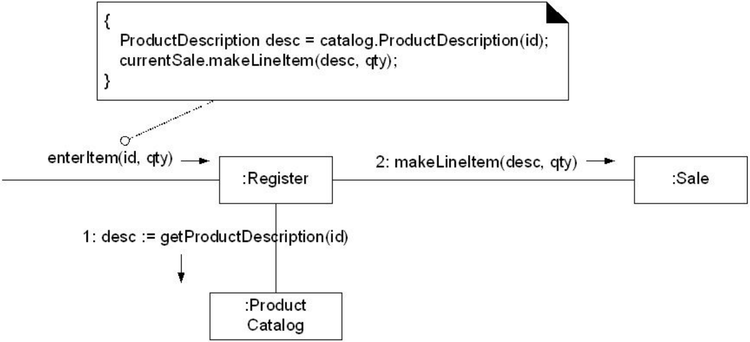
The Register.enterItem Method



```
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) {...}
}
```

Figure: The `Register` class.

Creating Methods from Interaction Diagrams (contd.)



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

enterItem(id, qty) →

:Register

2: makeLineItem(desc, qty) →

:Sale

1: desc := getProductDescription(id)

↓

:Product Catalog

Figure: The `enterItem` method

Creating Methods from Interaction Diagrams (contd.)

- 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`.

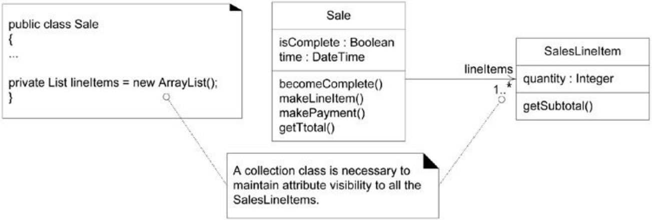
  ```
  currentSale.makeLineItem(desc, qty);
  ```

Collection Classes in Code

- One-to-many relationships are common.

  For example,

  a `Sale` must maintain visibility to a group of many `SalesLineItem` instances, as shown in Figure



```
public class Sale
{
...
private List lineItems = new ArrayList();
}
```

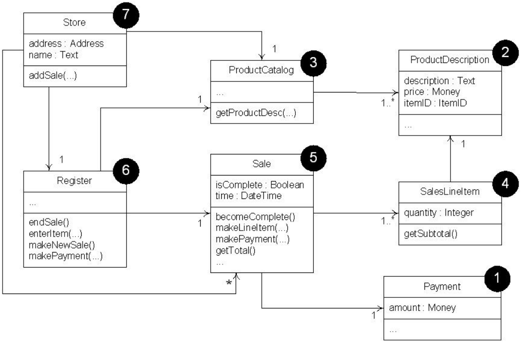
A collection class is necessary to maintain attribute visibility to all the SalesLineItems.

Figure: Adding a collection

Exceptions and Error Handling

- Exception handling has been ignored so far in the development of a solution. This was intentional to focus on the basic questions of responsibility assignment and object design.
- However, in application development, it's wise to consider the large-scale exception handling strategies during design modeling (as they have a large-scale architectural impact), and certainly during implementation.
- Briefly, in terms of the UML, exceptions can be indicated in the property strings of messages and operation declarations.

Order of Implementation

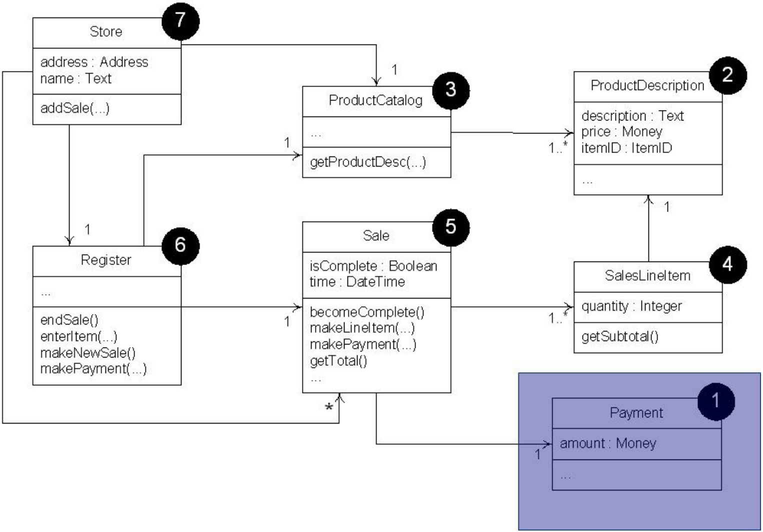


- Classes need to be implemented (and ideally, fully unit tested) **from least-coupled to most-coupled**.

  For example,

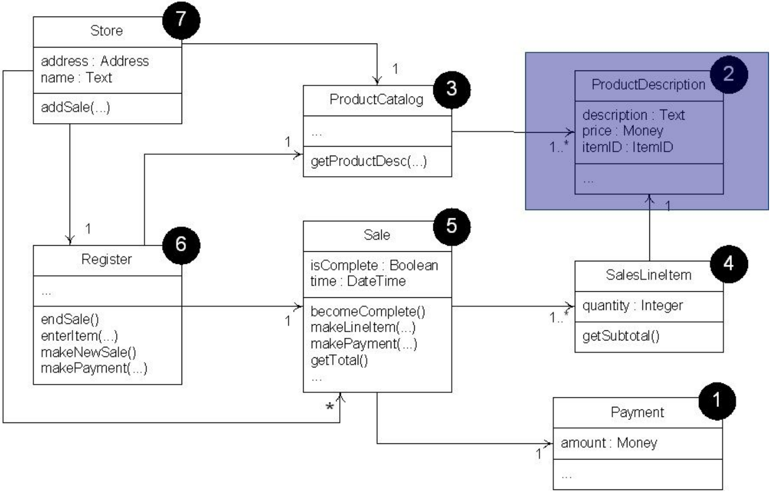
  possible first classes to implement are either `Payment` or `ProductDescription`; next are classes only dependent on the prior implementations `ProductCatalog` or `SalesLineItem`.

Order of Implementation



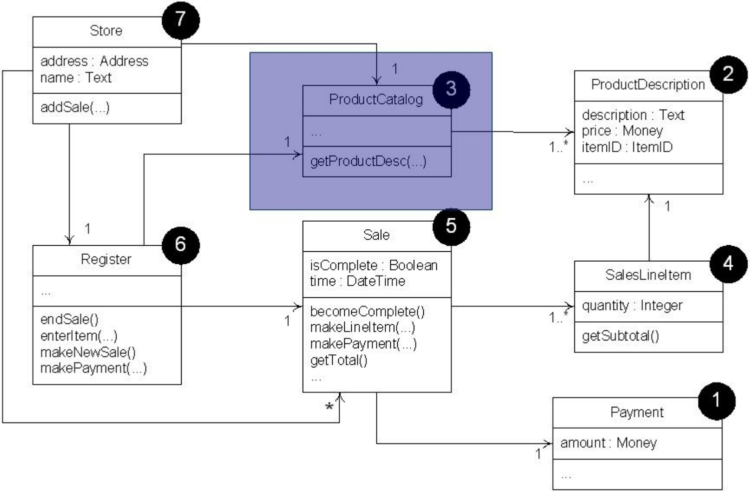
```
//Class Payment

public class Payment
{
   private Money amount;

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

Order of Implementation



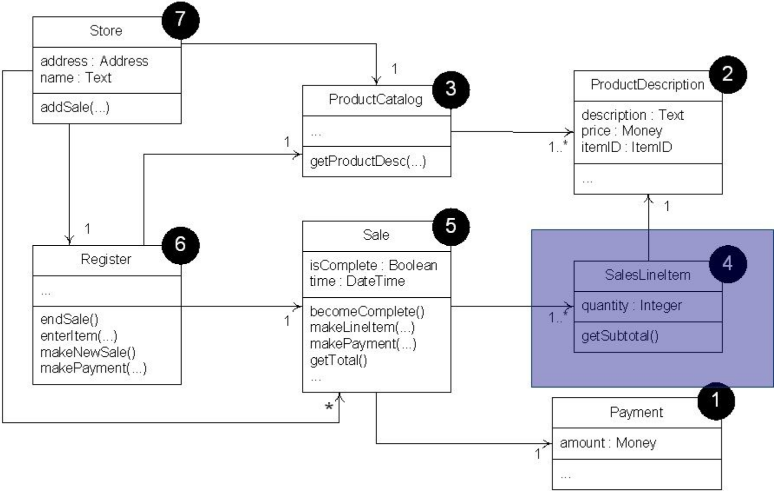
```
//Class ProductDescription
public class ProductDescription
{
    private ItemID id;
    private Money price;
    private String description;

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

    public ItemID getltemlDO { return id;}

    public Money getPrice() { return price; }

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

Order of Implementation



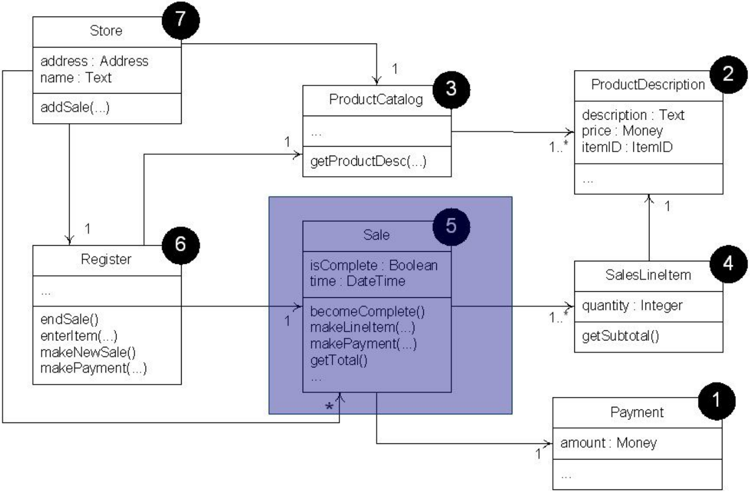
```
// Class ProductCatalog
public class ProductCatalog
{
    private Map<ItemID, ProductDescription>
        descriptions = new HashMap()<ItemID,
            ProductDescription>;
    public ProductCatalog() {
    // sample data
    ItemID idl = new ItemID( 100 );
    ItemID id2 = new ItemID( 200 );
    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 );
    }
}
```

Order of Implementation



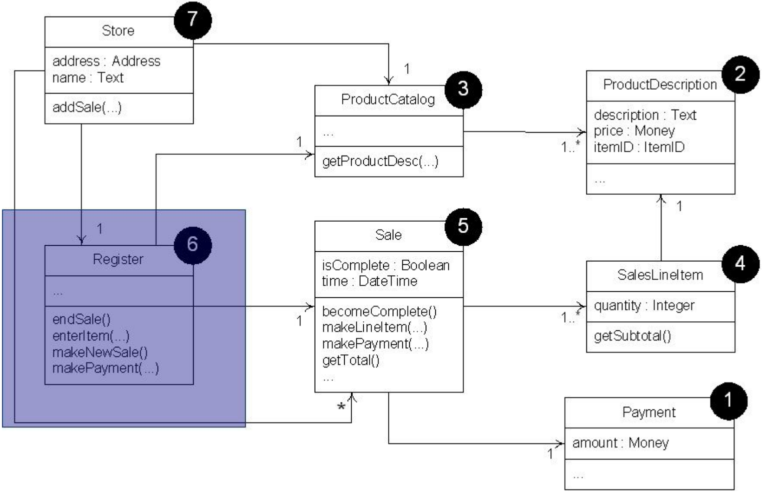
```
//Class SalesLineltem

public class SalesLineltem
{
    private int quantity;
    private ProductDescription description;

    public SalesLineltem (ProductDescription desc, int quantity )
    {
        this.description = desc;
        this.quantity = quantity;
    }

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

Order of Implementation



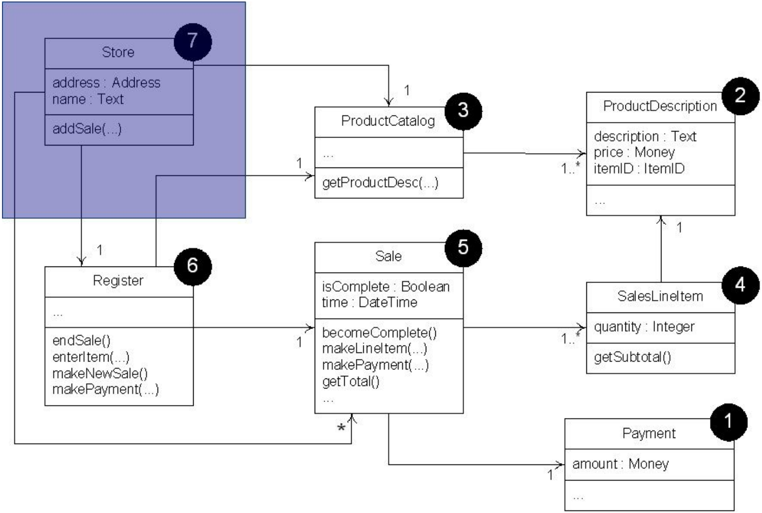
```
//Class Sale
public class Sale
{
   private List<SalesLineItem> lineItems = new
       ArrayList()<SalesLineItem>;
   private Date date = new Date();
   private boolean isComplete = false;
   private Payment payment;

   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 ) );
   }
//Class Sale (contd.)
   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 );
   }
}
```

Order of Implementation



```
//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 enterltem( ItemID id, int quantity )
    {
        ProductDescription desc = catalog.getProductDescription(
             id );
        currentSale.makeLineItem( desc, quantity );
    }
//Class Register (contd.)

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

    public void makePayment( Money cashTendered )
    {
        currentSale.makePayment( cashTendered );
    }
}
```

Order of Implementation



7 Store
address : Address
name : Text
addSale(...)

3 ProductCatalog
...
getProductDesc(...)

2 ProductDescription
description : Text
price : Money
itemID : ItemID
...

6 Register
...
endSale()
enterItem(...)
makeNewSale()
makePayment(...)

5 Sale
isComplete : Boolean
time : DateTime
becomeComplete()
makeLineItem(...)
makePayment(...)
getTotal()
...

4 SalesLineItem
quantity : Integer
getSubtotal()

1 Payment
amount : Money
...

```
//Class Store
public class Store
{
    private ProductCatalog catalog = new ProductCatalog();

    private Register register = new Register( catalog );

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

It's Quiz Time

1. For an object `A` to send a message to an object `B`, `B` must be visible to `A`. (True or False)

2. Attribute visibility from `A` to `B` exists when `B` is not an attribute of `A`. (True or False)

3. Classes need to be implemented (and ideally, fully unit tested) from most-coupled to least-coupled. (True or False)