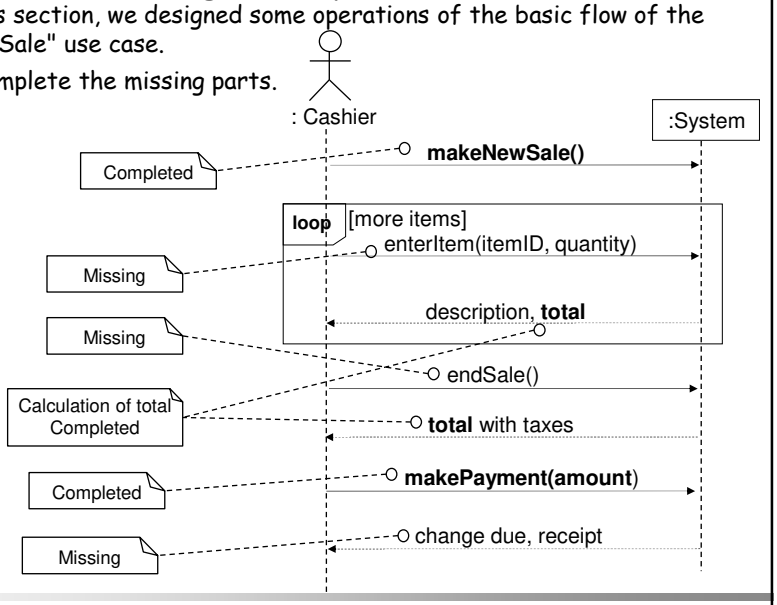
Design Examples

In the previous section, we designed some operations of the basic flow of the "UC1- Process Sale" use case.

Now we will complete the missing parts.



Design Example: enterItem

The enterItem system operation occurs when a cashier enters the itemID and (optionally) the quantity of an item the customer buys.

Actually, we could design this operation right after makeNewSale, but as this operation is more complicated than others, I left it to this chapter.

Remember the contract of the operation.

Contract CO2: enterItem

Operation:	enterItem(itemID: ItemID, quantity: integer)
Cross References:	Use Cases: Process Sale
Preconditions:	There is a sale underway
Postconditions:	- A SalesLineItem instance sli was created (instance creation). - sli was associated with the current Sale (association formed). - sli.quantity became quantity (attribute modification) - sli was associated with a ProductSpec. based on itemID match (association formed)

Remember, postconditions give responsibilities.

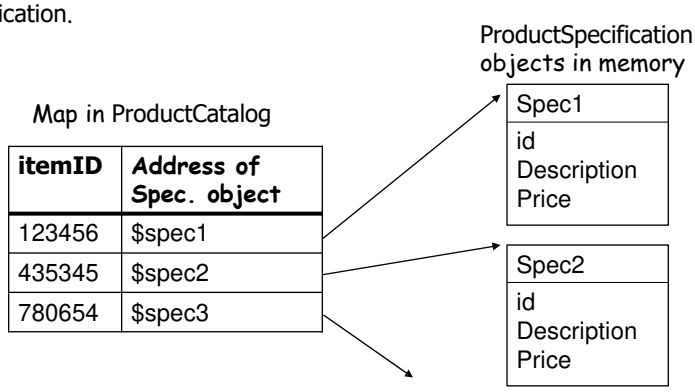
Responsibilities and assignments:

- **Choosing the Controller:**
  We will continue to use Register as the facade controller.

- **Displaying Item Description and Price:**
  Because of the principle of Model-View Separation, it is not the responsibility of non-GUI objects.
  Therefore, we ignore the design of the display at this time.

- **Creating a New SalesLineItem:**
  Analysis of the Domain Model reveals that a Sale contains SalesLineItem objects.
  Therefore, by Creator, a makeLineItem message is sent to a Sale for it to create a SalesLineItem.
  The Sale creates a SalesLineItem and then stores the new instance in its permanent collection.

- **Finding a ProductSpecification:**
  The SalesLineItem needs to be associated with the ProductSpecification that matches the incoming itemID.
  We must retrieve a ProductSpecification, based on an itemID match.
  Analyzing the Domain Model reveals that the ProductCatalog logically contains all the ProductSpecifications.
  For example, the lookup can be implemented with a method called getSpecification.



- **Sending a message to a ProductCatalog to get the specification:**

Who will get the specification from the catalog; Register or Sale?

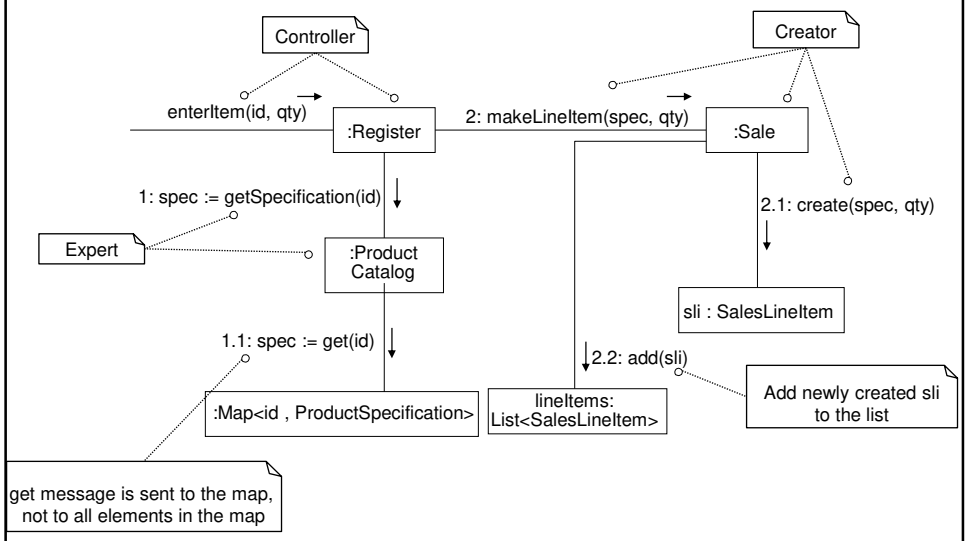
It is reasonable to assume that Register and ProductCatalog instances were created during the initialization of the system (Store) and that the Register object is permanently connected to the ProductCatalog object.

With that assumption (we must remember this during the design of initialization operations), we know that the Register can send the getSpecification message to the ProductCatalog.

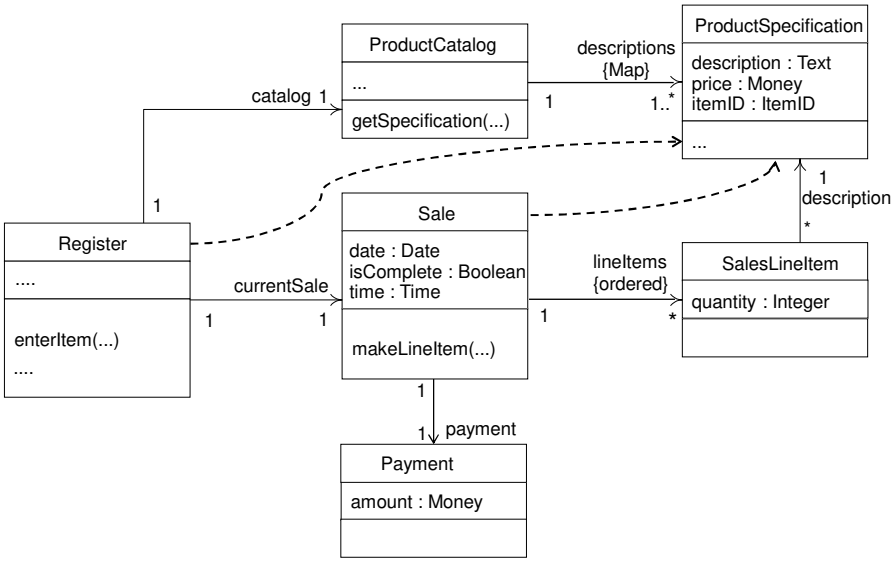
Another possibility is that the Sale sends the getSpecification message to the ProductCatalog.

Which assignment is better? Coupling, cohesion . . .

Design of enterItem:



Partial design class diagram of the software system so far:



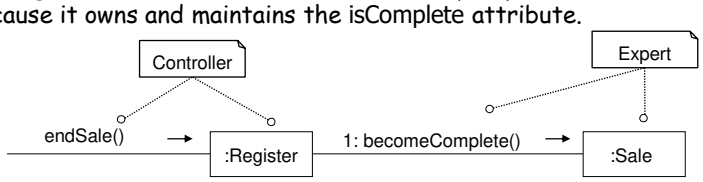
Design Example: endSale

The endSale system operation occurs when a cashier presses a button indicating the end of entering line items into a sale.

Contract CO3: endSale	
Operation:	endSale()
Cross References:	Use Cases: Process Sale
PreConditions:	There is a sale underway
PostConditions:	- Sale.isComplete became true (attribute modification)

Responsibilities and assignments:

- **Choosing the Controller:** We will continue to use Register as a controller.
- **Setting the Sale.isComplete Attribute:** By Expert, the Sale should set it because it owns and maintains the isComplete attribute.



Design Example: Calculating the balance

The "Process Sale" use case implies that the balance due from payment should be displayed somehow.

Because of the Model-View Separation principle, we do not concern ourselves with how the balance will be displayed or printed, but we must ensure that it is known.

Responsibility:

Who is responsible for knowing the balance?

To calculate the balance, we need the sale total and payment cash tendered. Therefore, Sale and Payment are partial Experts on solving this problem.

Solution 1:

If we assign the responsibility for knowing the balance to Payment, it needs visibility (coupling) to the Sale to ask the Sale for its total.

Since it does not currently know about the Sale (class diagram in 5.7), this approach would increase the overall coupling in the design (violates the Low Coupling pattern).

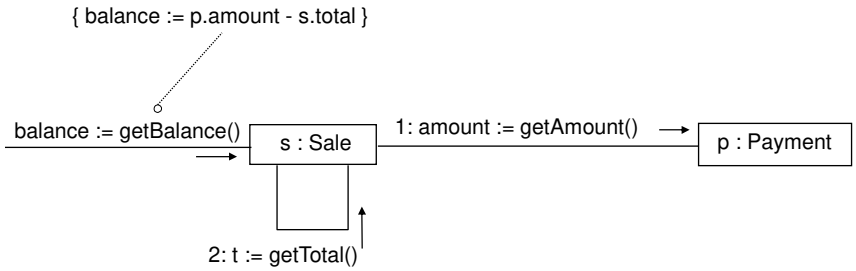
In this case, a new arrow from Payment to Sale would be necessary.

Solution 2:

If we assign the responsibility for knowing the balance to Sale, it needs visibility (coupling) to the Payment to ask it for its cash tendered.

Since the Sale already has visibility to the Payment (remember the design of makePayement in 4.21), the Sale creates the Payment.

Since this approach does not increase the overall coupling (in 5.7), it is preferable.



Design Example: Logging a Sale

The Process Sale use case:

8. System logs completed sale …

Responsibility:

Who is responsible for knowing all the logged sales and doing the logging?

Alternative 1:

With the goal of the low representational gap, we can expect a Store to know all the logged sales because they are strongly related to its finances.

Alternative 2:

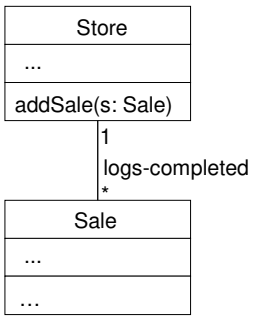
If Store has many other responsibilities, we can create a class such as SalesLedger.

Using a SalesLedger object makes sense as the design grows and the Store becomes incohesive.

In this case, we would add SalesLedger to the Domain Model because a sales ledger is a concept in the real-world domain.

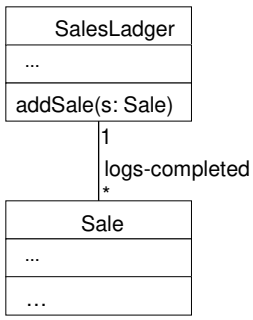
This kind of discovery and change during design work is to be expected.

Two alternatives for logging a sale:



Store can log sales.

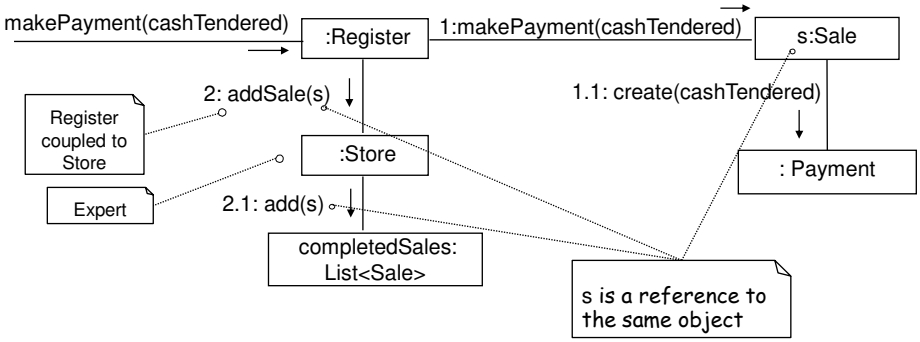
Acceptable if the Store has few responsibilities.



SalesLedger can log sales.

Suitable if the Store has many responsibilities and becomes incohesive.

In this example, we assign this responsibility to Store.



Design Example: Connecting the UI Layer to the Domain Layer

Remember, we put a controller object between the UI and domain layers to ensure low coupling.

However, in some cases, UI objects may send messages to domain objects directly.

For example, in the case of the enterItem message, we want the window to show the running total after each entry.

Solution 1: Add a getTotal method to the Register.

The UI sends the getTotal message to the Register, which delegates to the Sale.

Then the Register gets the result from Sale and passes it to the UI layer

This provides low coupling but may overload the Register, making it less cohesive.

Solution 2: An object in the UI gets the reference of the current Sale object from the Register.

When the UI requires the total, it directly sends messages to the Sale.

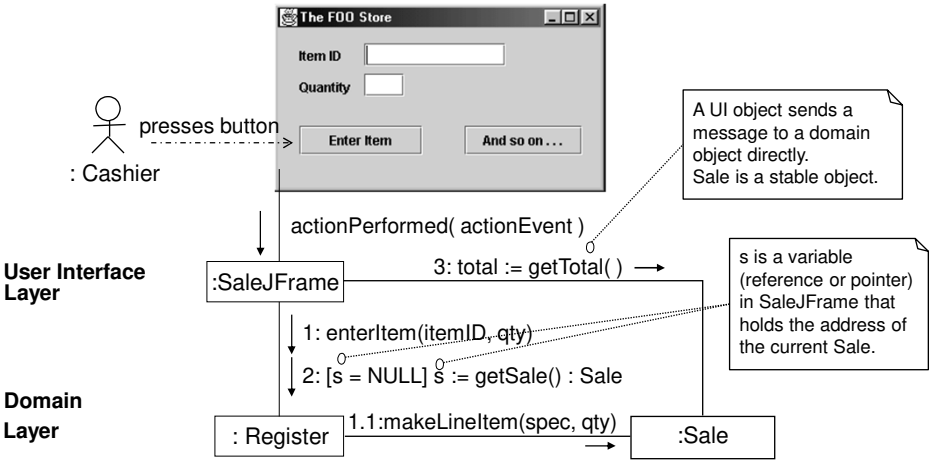
This design increases the coupling from the UI to the domain layer.

However, coupling to the Sale is not a major problem if the Sale is a stable object.

This makes the Register more cohesive.

Connecting the UI Layer to the Domain Layer:

We assume that the Sale is a stable object.



Later we will see the *Observer (GoF)* pattern, which provides a solution to this problem in more complex systems.

Design Example: Initializing the System

For most of the systems, it is necessary to write a "Start-up" use case that includes system operations related to the starting up of the application.

What should happen when we start the program?

Although the "start-up" use case is the earliest one to execute, we delay its design until after all other system operations have been considered.

Do the initialization design last.

In a start-up, we create an **initial domain object** (or a set of initial domain objects).

The initial domain object is responsible for

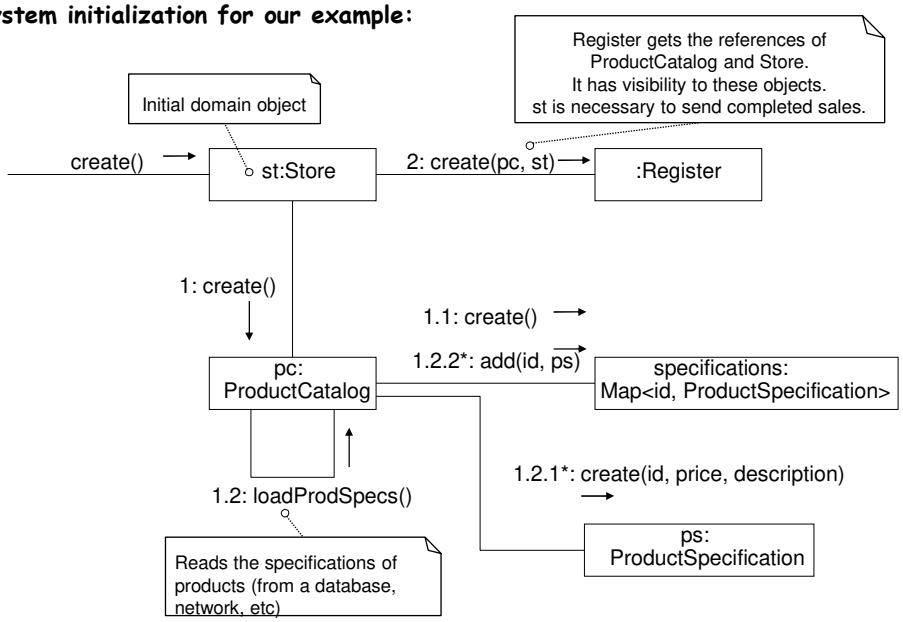
- the creation of its direct child domain objects (which must be created at the start up);
- ensuring the necessary visibility (connection) between related objects.

For example,

- Creating the Register and the ProductCatalog objects and establishing a connection between them.
- Establishing the connection between the UI and the controller object.

In our example, we chose the Store as the initial object.

System initialization for our example:



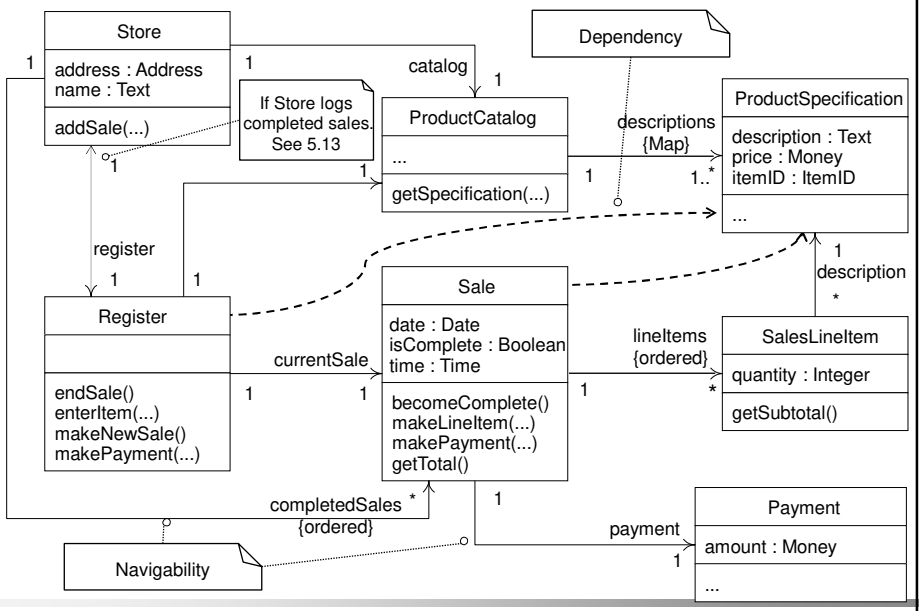
Initialization in Java:

```
public class Main                          // Java
{
public static main( String[] args)
{
  // Store is the initial domain object
  Store store = new Store();
  Register register = store.getRegister();          // register is created by Store
  ProcessSaleJFrame frame = new ProcessSaleJFrame(register);  // Frame is connected
  .....                                                       // to Register
 }
}
```

Initialization in C++:

```
int main( )                    // C++
{
  // Store is initial domain object
  Store store;
  Register *register = store.getRegister();
  ProcessSaleJFrame *frame = new ProcessSaleJFrame(register);
  .....
}
```

The Design Class Diagram reflecting our design decisions until now:

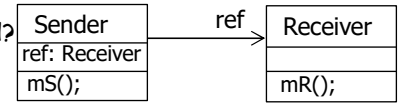


Visibility Between Objects

Visibility means that one object can "see" or have reference to another object.

To send a message to another object, the sender must have a reference or a pointer to the receiver object.

How can the Sender call Receiver's mR() method?



The sender must "see" the receiver.

obj.method(); ref.mR();
objPointer->method(); ref->mR();

During the design of a system as a set of interacting objects, it is necessary to ensure that the required visibility is achieved between objects to support message interaction.

Types of visibility:

There are four ways that visibility can be established from object A to object B:

- **Attribute visibility:** B is an attribute of A.
- **Parameter visibility:** B is a parameter of a method of A.
- **Local visibility:** B is a (non-parameter) local object in a method of A.
- **Global visibility:** B is in the global space of A.

Attribute visibility:

It exists from A to B when B is an attribute of A.

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

Example:

Objects of Register have attribute visibility to ProductCatalog.

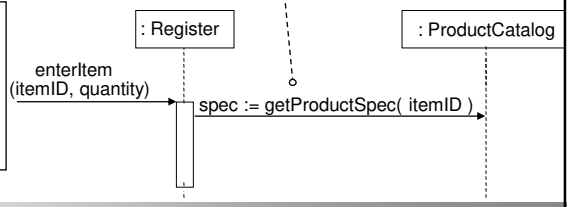
It is established during system initialization (see 5.17).

This visibility is necessary because, in the enterItem diagram, a Register needs to send the getProductSpec message to a ProductCatalog.

```
class Register
{
  .....
   private ProductCatalog catalog;
  .....
} // assignment in the constructor
```

```
public void enterItem(itemId,qty)
{
  .....
   spec= catalog.getProductSpec(itemID);
  .....
}
```

```
// Constructor of Register
public Register(pc: ProductCatalog, st:Store)
 {
  .....
   catalog=pc;        // attribute visibility
  ....
}
```



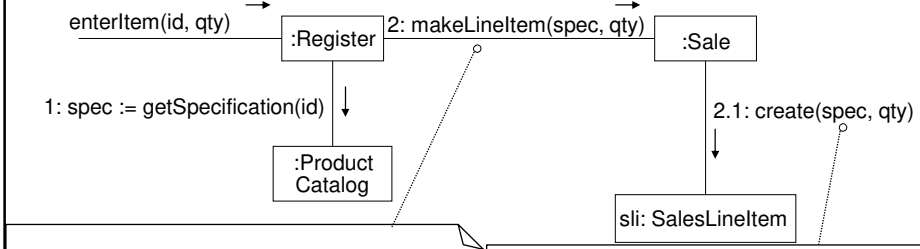
Parameter visibility:

It exists from A to B when B is passed as a parameter to a method of A.

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

Example:

Objects of Sale have parameter visibility to ProductSpecification in the makeLineItem method.




```
{
makeLineItem(ProductSpecification spec, int qty)
{
  ...
  sli = new SalesLineItem(spec,qty);
  ...
}
}
```

```
{   // Constructor of theSalesLineItem
SalesLineItem(ProductSpecification spec, int qty)
{
  ...
  description = spec;  // from parameter visibility
                       // to attribute visibility
}
}
```

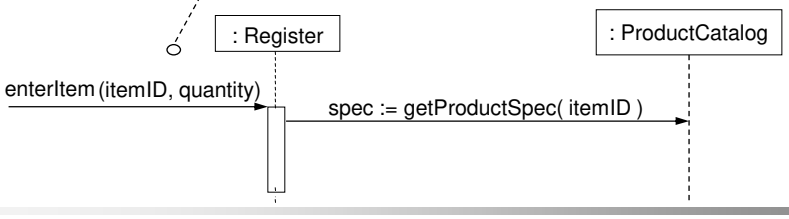
Local visibility:

It exists from A to B when B is declared as a local object within a method of A.
It is temporary visibility because it persists only within the scope of the method.

Example:

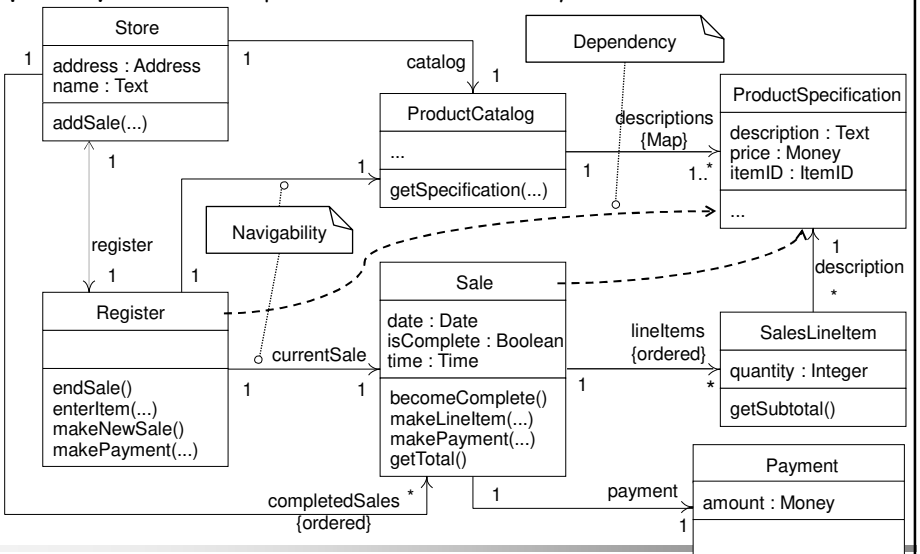
Objects of Register have local visibility to ProductSpecification in the enterItem method.

```
public void enterItem(itemId,qty)
{
    //local visibility to ProductSpecification (spec is a local object)
  ProductSpecification spec = catalog.getProductSpec(itemID);
 .....
}
```



Navigability and Dependency in UML Class Diagrams of the design model

Navigability refers to attribute visibility, and solid line arrows Show it.
Dependency refers to the parameter and local visibility. Dashed line arrows Show it.



Details of UML Class Diagrams

If necessary, access modifiers of the class members and data types may be shown in class diagrams.

In most cases, class diagrams are used to indicate the design decisions.
Therefore, programming details are optional.