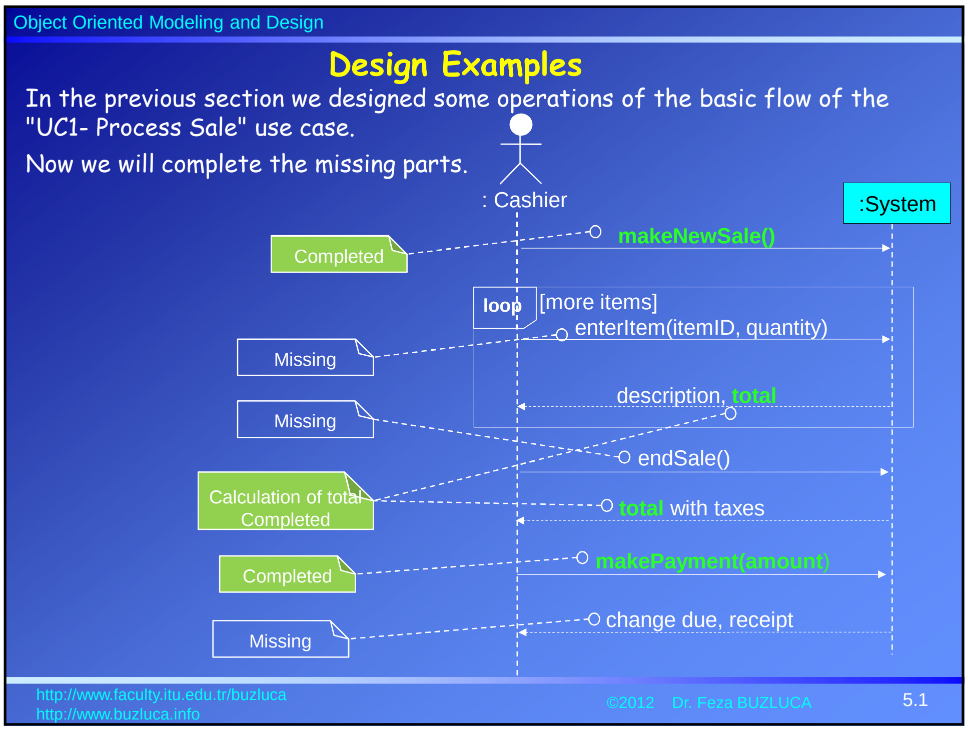
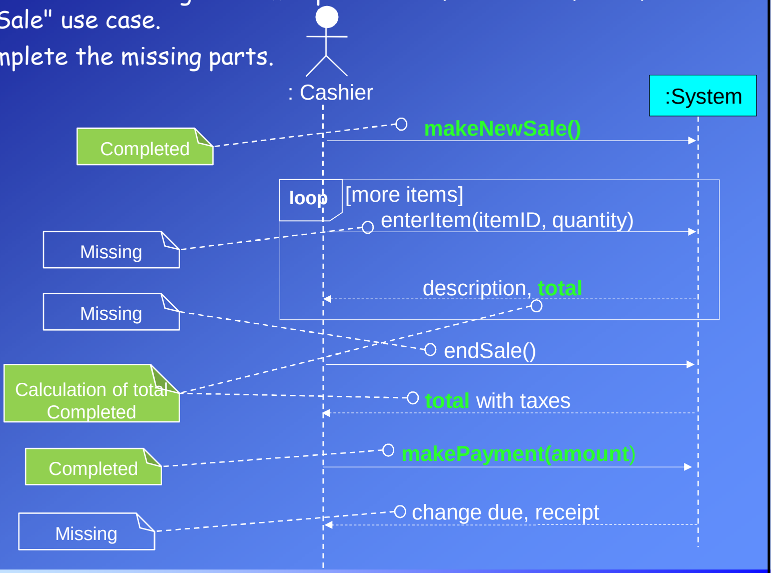
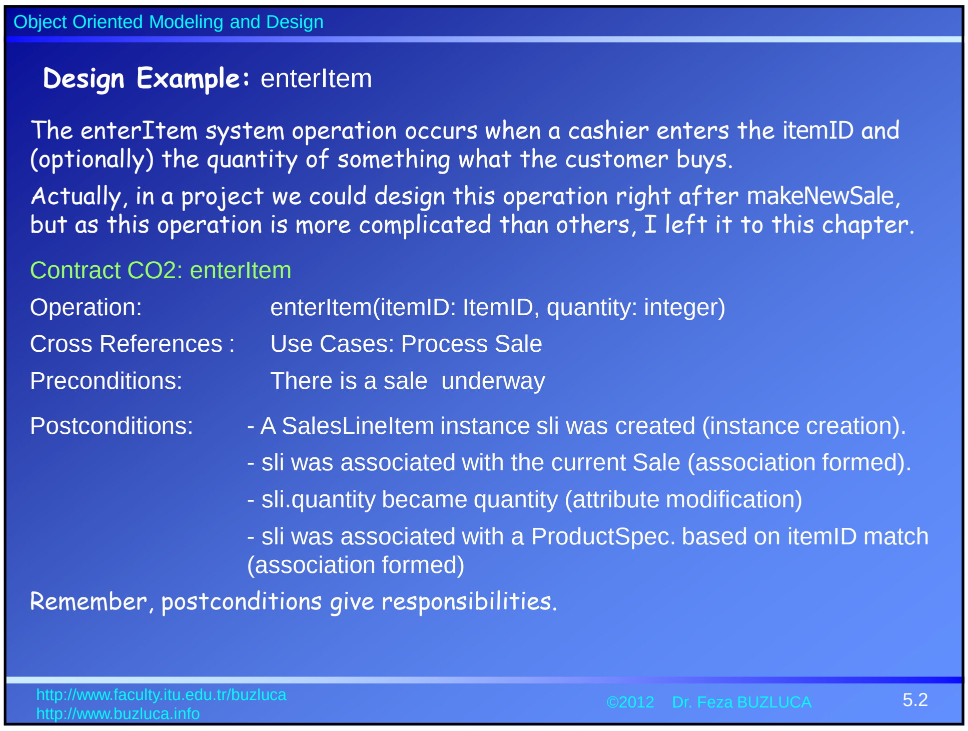


Design Examples

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 something what the customer buys.

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

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:

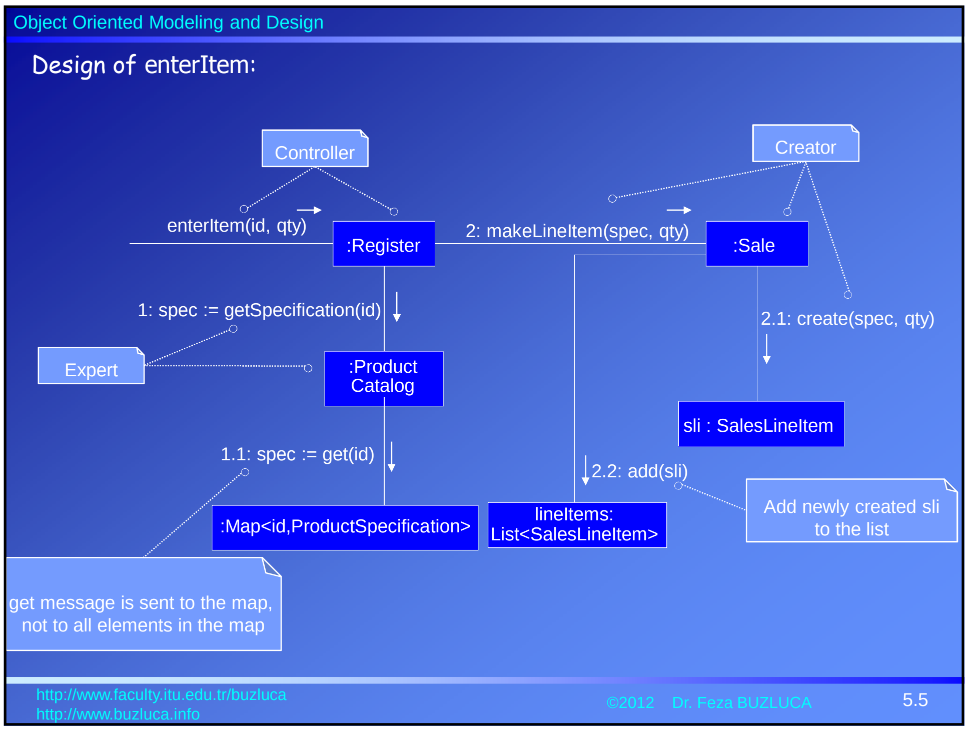
- ***Choosing the Controller*:**
  We will continue to use Register as a controller.

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

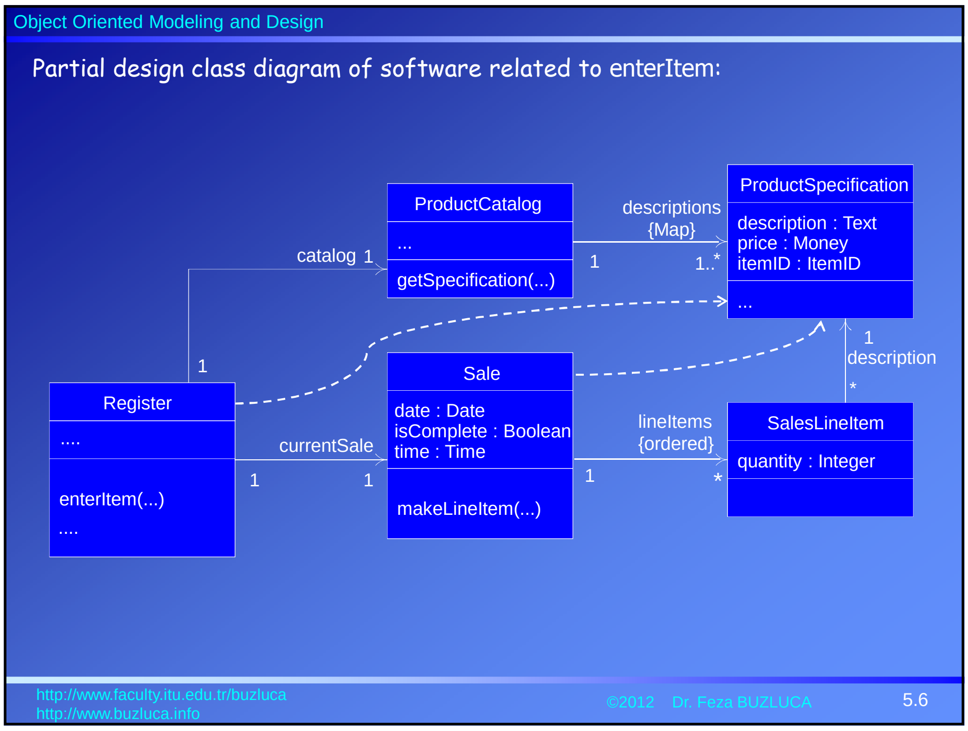
- **Finding a ProductSpecification:**
  The SalesLineItem needs to be associated with the ProductSpecification that matches the incoming itemID.
  We must retrieve a ProductSpecification, based on an itemID match.
  Analyzing the Domain Model reveals that the ProductCatalog logically contains all the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

- ***Sending message to a* ProductCatalog:**
  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 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:

Controller
Creator
enterItem(id, qty)
:Register
2: makeLineItem(spec, qty)
:Sale
1: spec := getSpecification(id)
2.1: create(spec, qty)
Expert
:Product Catalog
sli : SalesLineItem
1.1: spec := get(id)
2.2: add(sli)
:Map<id,ProductSpecification>
lineItems: List<SalesLineItem>
Add newly created sli to the list
get message is sent to the map, not to all elements in the map



Partial design class diagram of software related to enterItem:

Design Example: endSale

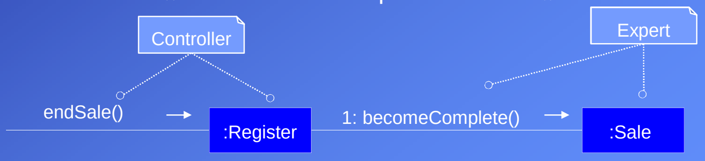
The endSale system operation occurs when a cashier presses a button indicating the end of entering line items into a sale.

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, since it owns and maintains the isComplete attribute..



Design Example: Calculating the balance

The Process Sale use case implies that the balance due from a payment 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.

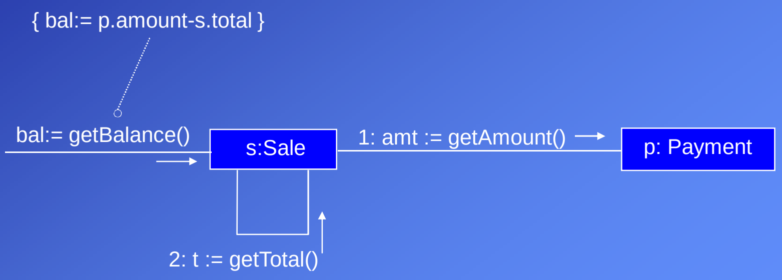
Since it does not currently know about the Sale, this approach would increase the overall coupling in the design (violates the Low Coupling pattern).

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 design of makePayement.

Sale creates the Payment), this approach does not increase the overall coupling and is therefore a preferable design.



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:

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

Alternative 2:

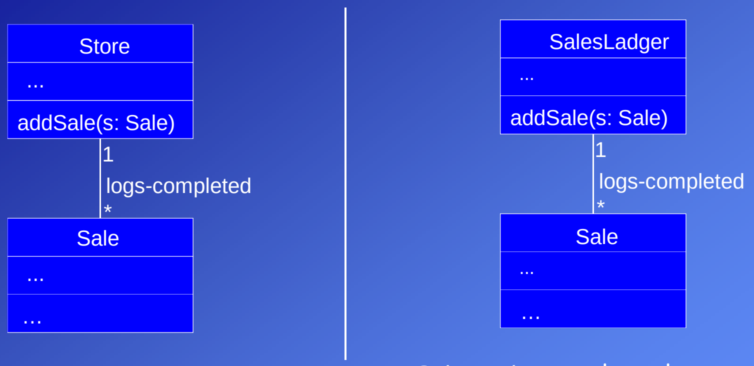
If Store has many other responsibilities we can create a class such as SalesLedger.

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 as well since 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:



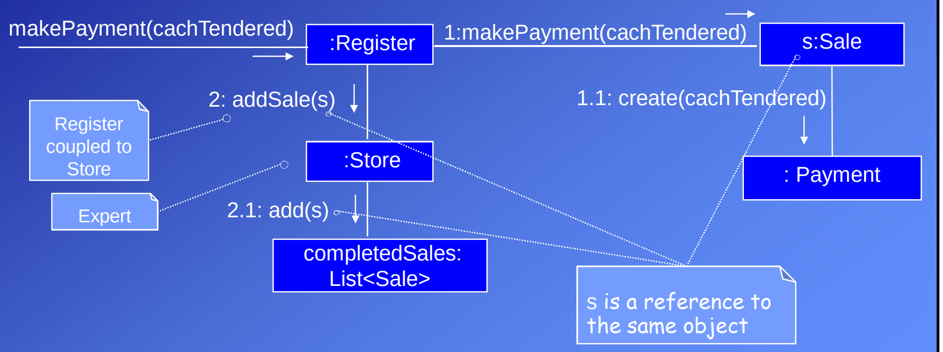
Store can log sales.

Acceptable if Store has few responsibilities.

SalesLedger can log sales.

Suitable if Store has many responsibilities and becomes incohesive.

In our example we assign this responsibility to Store.


makePayment(cachTendered)
:Register
1:makePayment(cachTendered)
s:Sale
2: addSale(s)
1.1: create(cachTendered)
Register coupled to Store
:Store
: Payment
Expert
2.1: add(s)
completedSales: List<Sale>
s is a reference to the same object

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 between them.

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 it 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.

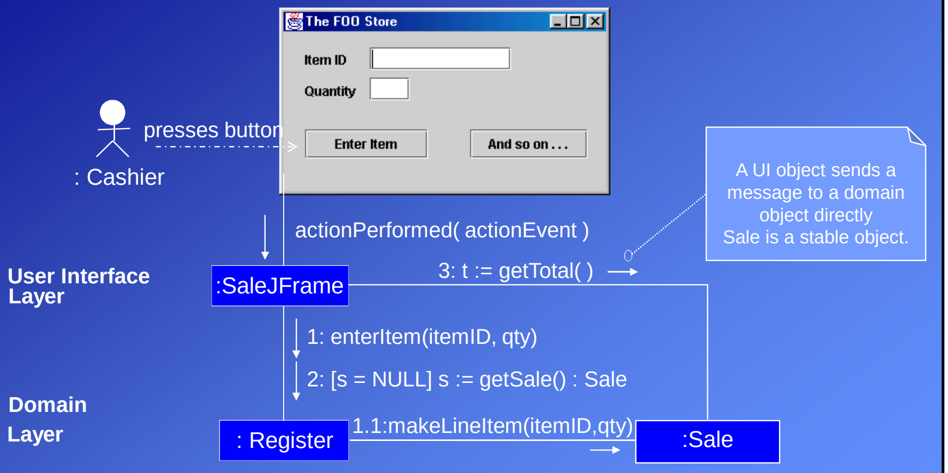
This design increases the coupling from the UI to the domain layer.

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, delay its design until after all other system operations have been considered.

Do the initialization design last.

In 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).

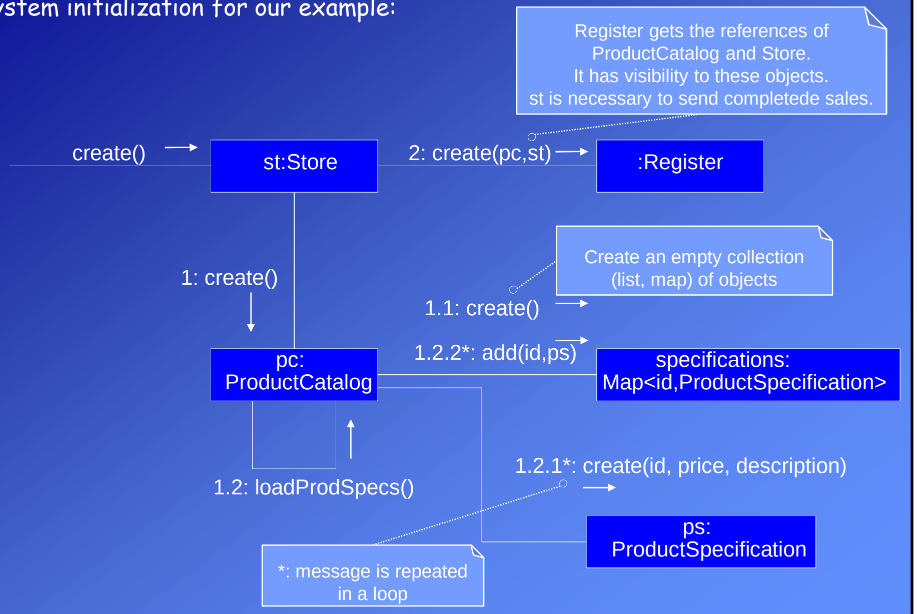
It also must ensure the necessary visibility (connection) between related objects.

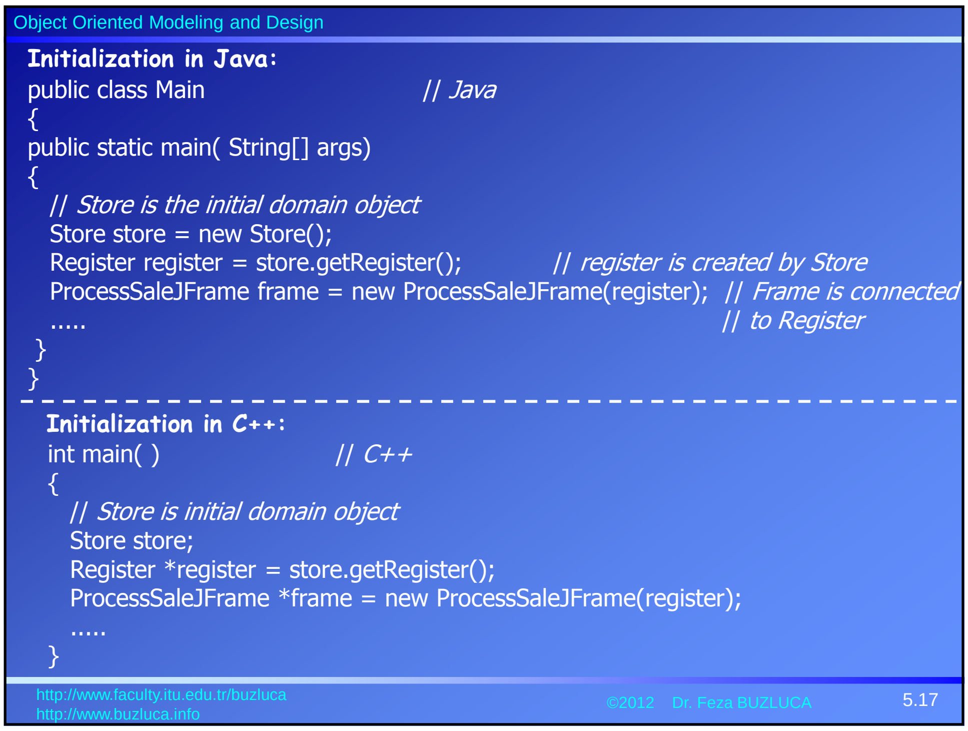
For example the connection between the UI and the controller object,

or the connection between the Register and the ProductCatalog.

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

System initialization for our example:


Register gets the references of ProductCatalog and Store.
It has visibility to these objects.
st is necessary to send completede sales.
create()
st:Store
2: create(pc,st)
:Register
1: create()
Create an empty collection (list, map) of objects
1.1: create()
1.2.2*: add(id,ps)
pc: ProductCatalog
specifications: Map<id,ProductSpecification>
1.2.1*: create(id, price, description)
1.2: loadProdSpecs()
ps: ProductSpecification
*: message is repeated in a loop



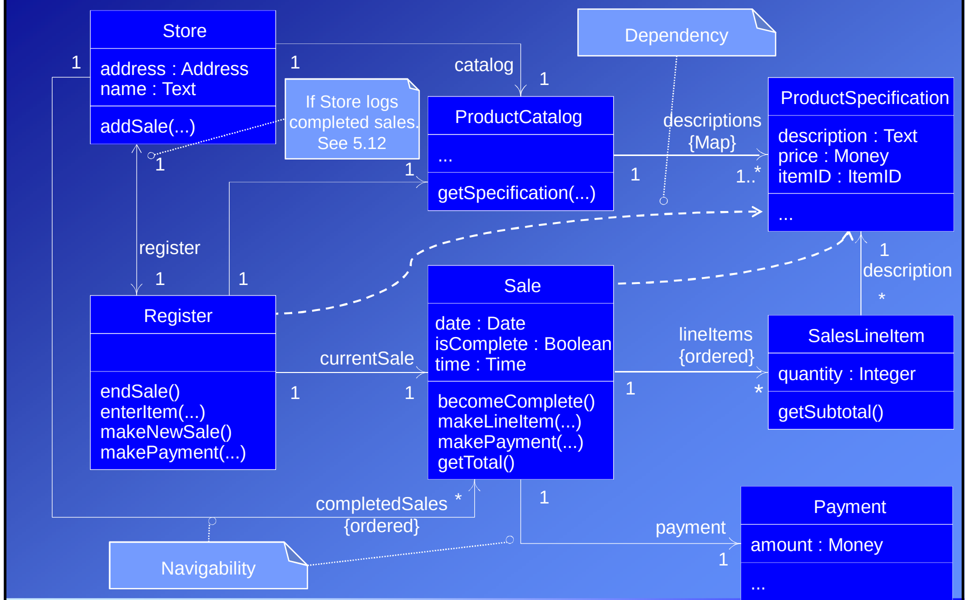
Initialization in Java:

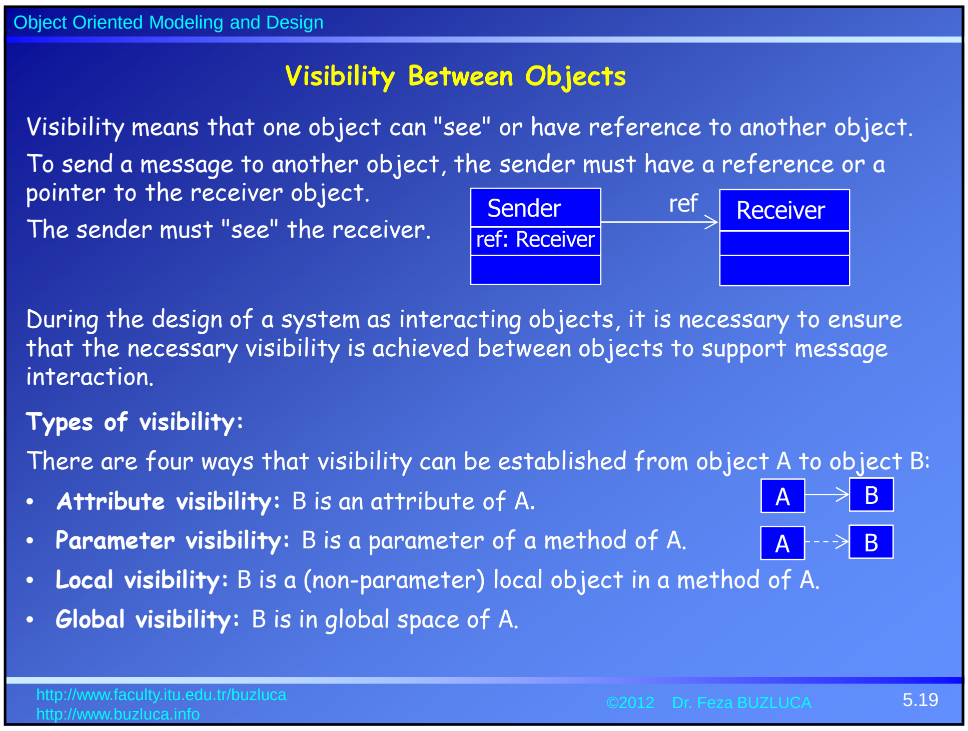
```
public class Main                           // Java
{
public static main( String[] args)
{
  // Store is the initial domain object
  Store store = new Store();
  Register register = store.getRegister();           // register is created by Store
  ProcessSaleJFrame frame = new ProcessSaleJFrame(register);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

The sender must "see" the receiver.

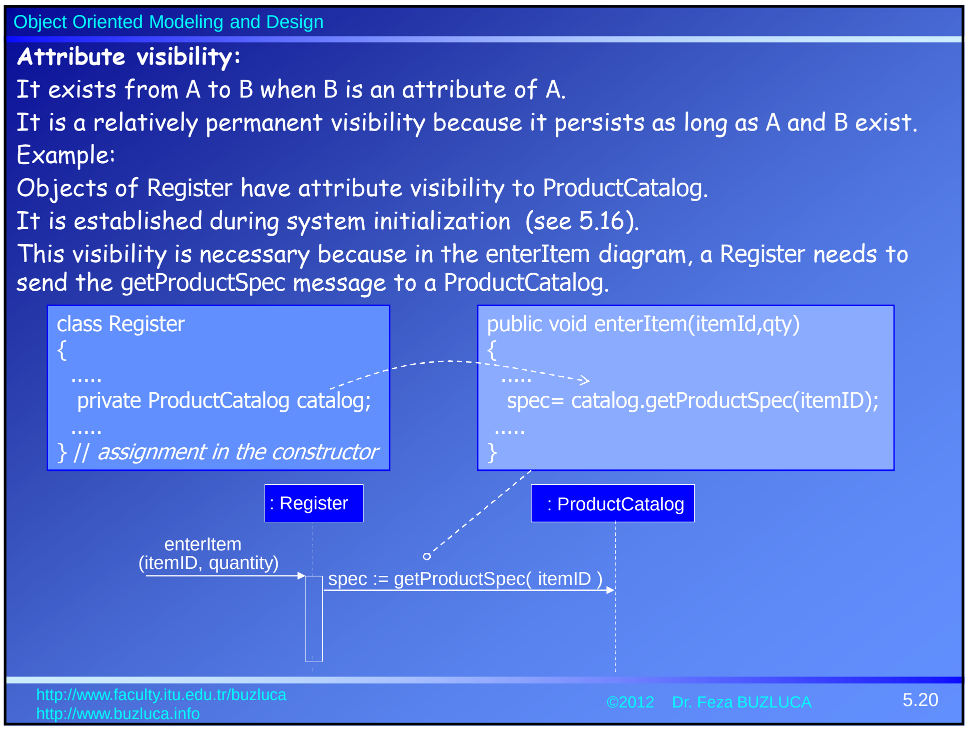
Sender
ref: Receiver
ref
Receiver

During the design of a system as interacting objects, it is necessary to ensure that the necessary 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 global space of A.



Attribute visibility:

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

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

Example:

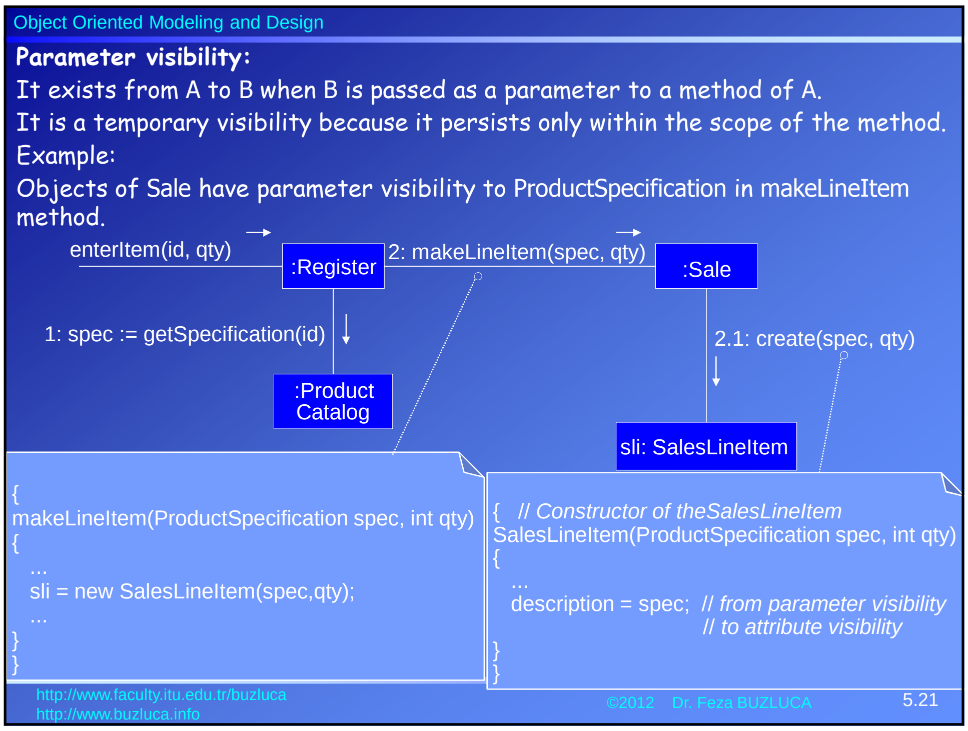
Objects of Register have attribute visibility to ProductCatalog.

It is established during system initialization (see 5.16).

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);
  .....
}
```



Parameter visibility:

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

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

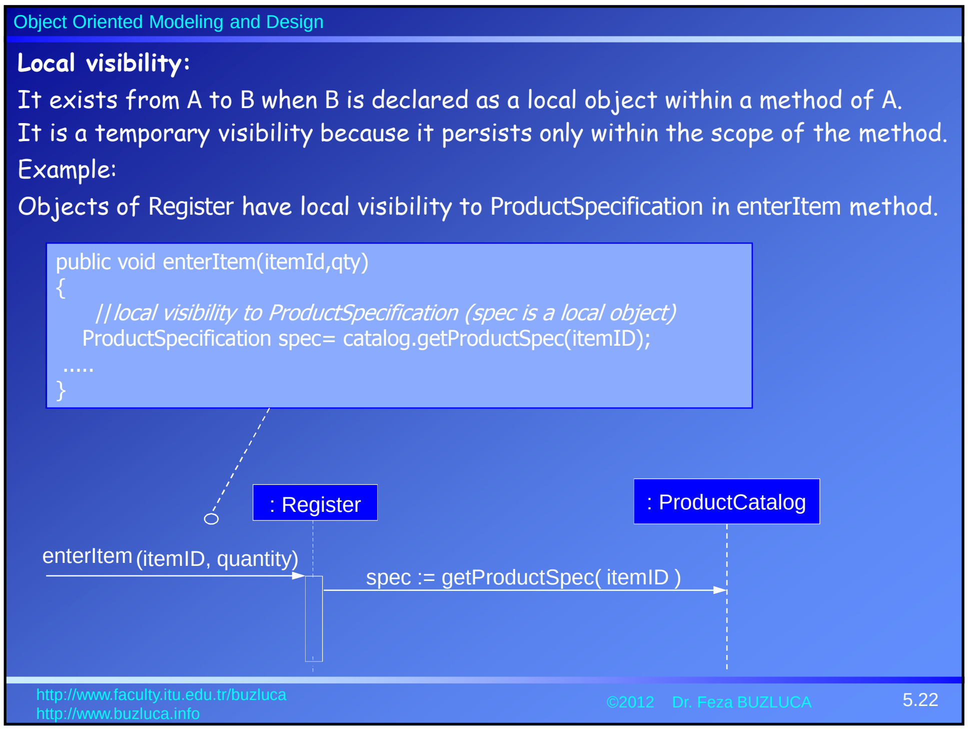
Example:

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

enterItem(id, qty)
:Register
2: makeLineItem(spec, qty)
:Sale
1: spec := getSpecification(id)
:Product Catalog
2.1: create(spec, qty)
sli: SalesLineItem

```
{
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 a temporary visibility because it persists only within the scope of the method.

Example:

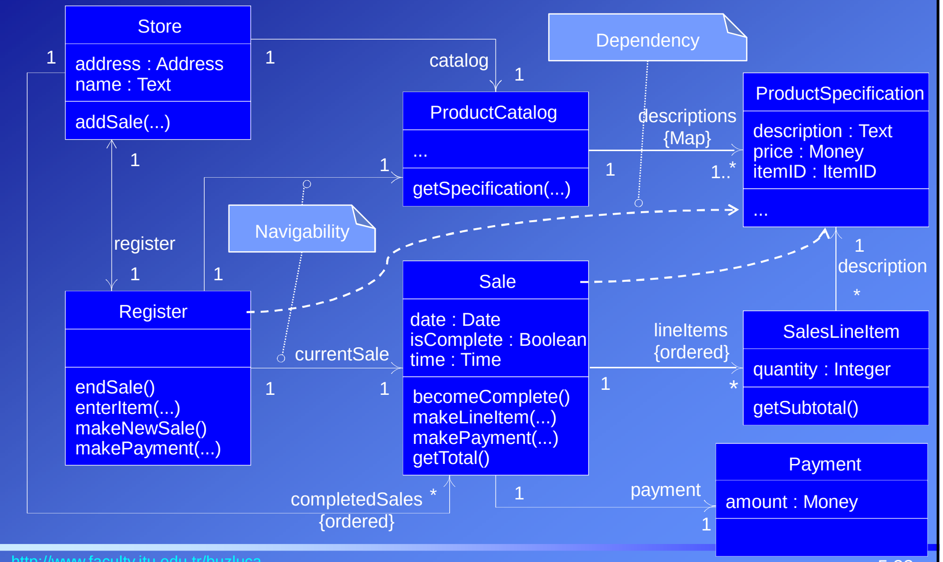
Objects of Register have local visibility to ProductSpecification in enterItem method.

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

: Register
: ProductCatalog
enterItem (itemID, quantity)
spec := getProductSpec(itemID)

In Design Class Diagrams **navigability** refers to attribute visibility and it is shown by solid line arrows.

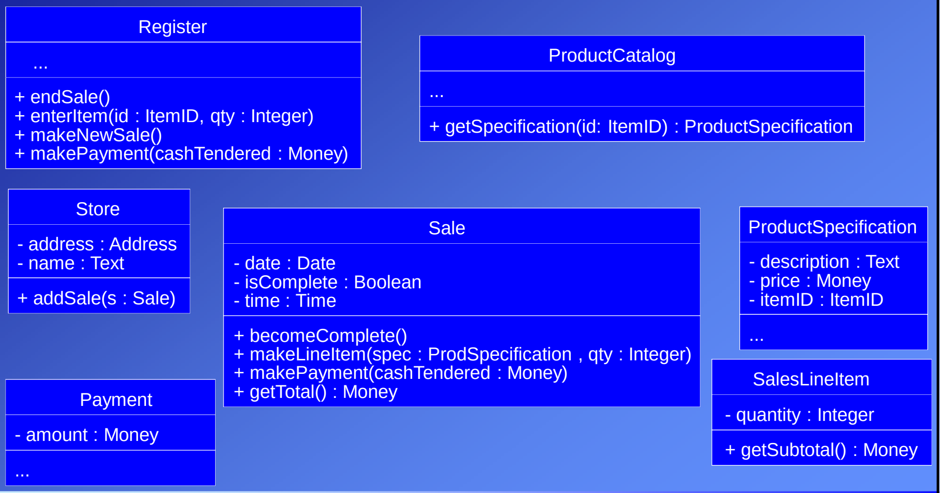
Dependency refers to parameter and local visibility. It is shown by dashed line arrows.



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

In most cases class diagrams are used to show the design decisions.

Therefore programming details are mostly not necessary.


Register
...
+ endSale()
+ enterItem(id : ItemID, qty : Integer)
+ makeNewSale()
+ makePayment(cashTendered : Money)
ProductCatalog
...
+ getSpecification(id: ItemID) : ProductSpecification
Store
- address : Address
- name : Text
+ addSale(s : Sale)
Sale
- date : Date
- isComplete : Boolean
- time : Time
+ becomeComplete()
+ makeLineItem(spec : ProdSpecification , qty : Integer)
+ makePayment(cashTendered : Money)
+ getTotal() : Money
ProductSpecification
- description : Text
- price : Money
- itemID : ItemID
...
Payment
- amount : Money
...
SalesLineItem
- quantity : Integer
+ getSubtotal() : Money