CS 332
OBJECT-ORIENTED PROGRAM DESIGN AND ANALYSIS

Lecture 9
Design Phase Collaboration Diagrams and Design Class Diagram

Objectives

- Notion of collaboration diagrams
- UML notation for collaboration diagram
- The nature of the design phase
- Patterns for assigning responsibilities to objects
- Use Patterns to create collaboration diagrams

Interaction Diagrams

- A contract for system operations describes ***what*** the system operation does, but it does not show a solution of ***how*** software objects are going to work collectively to fulfill the contract of the operation.
- The later (**how**) is specified by an ***interaction diagrams*** in UML.
- **A major task of the design phase is to create the interaction diagrams for the system operations.**

Interaction Diagrams

- **The UML defines two kinds of interaction diagrams**, either of which can be used to express similar or identical messages interactions:
  - —**Collaboration diagrams**
  - —**Object sequence diagrams**
- Each type has strengths and weaknesses as shown below:
  - —When drawing diagrams to be published on pages of narrow width, collaboration diagrams have the advantage of allowing vertical expansion for new objects
  - —additional objects in a sequence diagram must extend to the right, which is limiting.
  - —On the other hand, collaboration diagram examples make it harder to easily see the sequence of messages.

Sequence versus Collaboration Diagram

Type	Strengths	Weaknesses
Sequence Diagram	– Clearly shows sequence or time ordering of messages – Simple notation	– Forced to extend to the right when adding new objects; consumes horizontal space
Collaboration Diagram	– Space economy – flexibility to add new objects in two dimensions – Better to illustrate complex branching, iteration, and concurrent behaviour	– Difficult to see sequence of messages – More complex notation

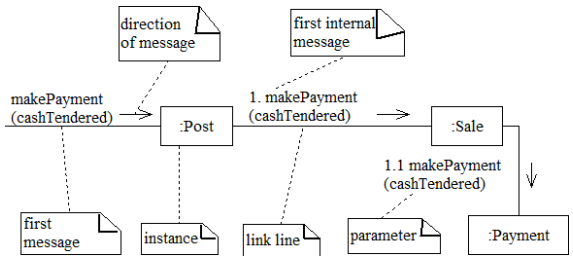
System Versus Object Sequence Diagrams

System Sequence Diagrams	Object Sequence Diagrams
– illustrates the interaction between the whole system and external actors,	– shows the interactions between objects of the system.
– shows only system's external events and thus identifies system operations	– identifies operations of objects.
– are created during the analysis phase,	– are models created and used in the design phase.

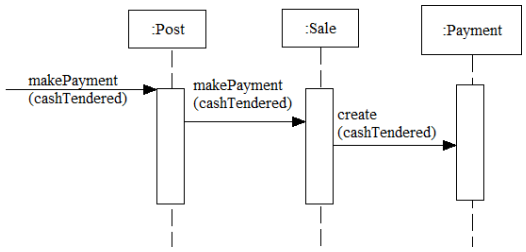
Collaboration Diagrams

- A ***collaboration diagram*** is a graph showing a number of objects and the links between them, which in addition shows the messages that, are passed from one object to another.
- As both collaboration diagrams and object sequence diagrams can express similar constructs
- **Different people may prefer one to another**
- **We shall mainly discuss and use collaboration diagrams.**

Collaboration Diagram for "makePayment(cashTendered)" Operation



Sequence Diagram for "makePayment(cashTendered)" Operation



Represents the following pseudo program

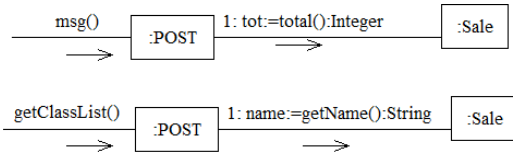
:POST accepts a call of it method makePayment()
(from an Interface Object);
:POST calls for the method makePayment() of :Sale;
:Sale calls the constructor of class Payment to create a new :Payment

UML Notation for Collaboration Diagrams

1. Representing a return value

—Some message sent to an object requires a value to be returned to the sending object. A return value may be shown by preceding the message with a return value variable name and an assignment operator ('**:=**'). The type of the return value may operationally be shown. The standard syntax for messages is:

return := message(parameter : parameterType) : returnType

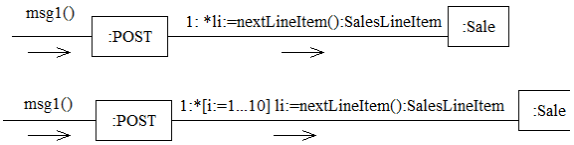


UML Notation for Collaboration Diagrams

2. Representing iteration

—An object may repeatedly send a message to another object a number of times.

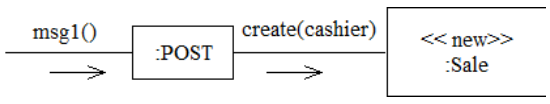
—This is indicated by prefixing the message with a start ('*****').



UML Notation for Collaboration Diagrams

3. Representing creation of instances

—The UML creation message is *create* which is independent of programming languages, shown being sent to the instance being created. Optionally, the newly created instance may include a <<*new*>> symbol. A *create* message can optionally take parameters when some attributes of the object to be created need to be set an initial value.



UML Notation for Collaboration Diagrams

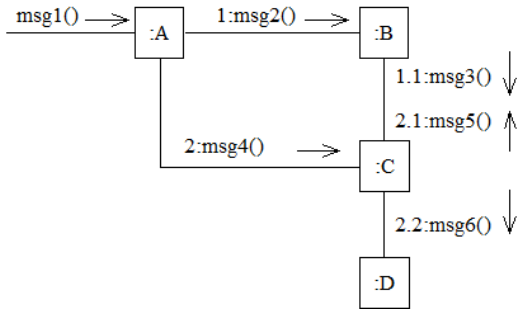
4. Representing message number sequencing

—The order of messages is illustrated with *sequence numbers*, as shown in Figure. The numbering scheme is:

a) The first message is not numbered. Thus, msg1() is unnumbered.

b) The order and nesting of subsequent messages is shown with a legal numbering scheme in which nested messages have appended to them a number. Nesting is denoted by pre-pending the incoming message number to the outgoing message number.

Sequence Numbering



UML Notation for Collaboration Diagrams

5. Representing conditional messages

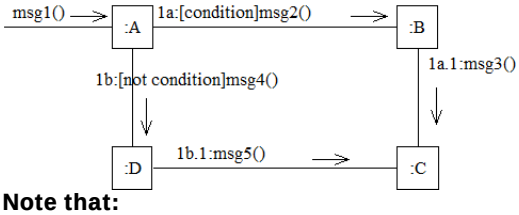
—Sometimes, a message may be *guarded* and can be sent from one object to another only when a condition holds.

—At a point during the execution of an object, a choice of several messages, guarded by different conditions, will be sent.

—In a sequential system, an object can send one message at a time and thus these conditions must be mutually exclusive.

—When we have mutually exclusive conditional messages, it is necessary to modify the sequence expressions with a conditional path letter.

Mutually Exclusive Messages



- **Note that:**

—Either 1a or 1b could execute after msg1() , depending on the conditions.

—The sequence number for both is 1, and a and b represent the two paths.

—This can be generalized to any number of mutually exclusive Conditional Messages.

UML Notation for Collaboration Diagrams

6. Representing multiobjects

—We call a logical set of instances/objects as a *multiobject*.

—A multiobject is an instance of a *container class* each instance of which is a set of instances of a given class (or type).

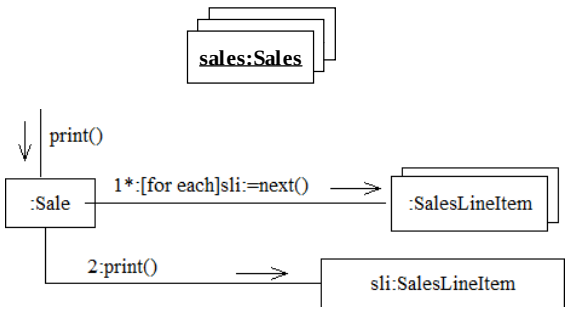
—E.g. *SetOfSales* is a class each instance of which is a set of sales.

—Each multiobject is usually implemented as a group of instances stored in a *container* or a *collection* object,

—In a collaboration diagram, a multiobject represents a set of objects at the "many" end of an association.

—In UML, a multiobject is represented as a stack icon as illustrated below.

UML Notation for Collaboration Diagrams



Overview of Design Phase

- During the design phase, there is a shift in emphasis from application domain concepts toward software objects, and from what toward how.
- The objects discovered during analysis serve as the skeleton of the design, but the designer must choose among different ways to implement them.
- In particular, the system operations identified during the analysis must be assigned to individual objects or classes

Overview of Design Phase

- Complex operations must be decomposed into simpler internal operations which are assigned to (or carried out by) further objects.
- **The major task in the design is to create the collaboration diagrams for the system operations identified in the requirement analysis phase**
- **The most important and difficult part in the generation of collaboration diagrams is the assignment of responsibilities to objects**
- Therefore, the rest of this chapter will mainly discuss the general principles for responsibility assignment, which are structured in a format called **patterns**.

Artifacts Needed for creating collaboration diagrams

- Conceptual model:
  - —the designer may choose to define software classes corresponding to concepts. Objects of these classes participate in interactions illustrated in the interaction diagrams.
- System operations contracts:
  - —the designer identifies the responsibilities and postconditions that the interaction diagrams must fulfill.
- Essential (or real) use cases:
  - —the designer may collect information about what tasks the interaction diagrams fulfill, in addition to what is in the contracts (remember that contracts do not say very much about the UI-outputs).

GRASP: Patterns for Assigning Responsibilities

- Deciding what methods belong where and how objects should interact is terribly important and trivial
- Assigning responsibility is a crucial/critical step in developing object-oriented systems.
- **Responsibilities:**
  - —A **responsibility** is a contract or obligation of an object.
  - —Responsibilities are related to the obligations of objects in term of their behaviour.
  - —The purpose is to help methodically apply fundamental principles for assigning responsibilities to objects.
  - —Within the UML artifacts, a common context where these responsibilities (implemented as methods) are considered is during the creation of interaction diagrams.

Types of Responsibilities

- ***Doing responsibilities***: these are about the actions that an object can perform including
  - —doing something itself
  - —initiating an action or operation in other objects
  - —controlling and coordinating activities in other objects
- ***Knowing responsibilities:*** these are about the knowledge an object maintains:
  - —knowing about private encapsulated data
  - —knowing about related objects
  - —Knowing about things it can derive or calculate

Steps to Create Collaboration Diagrams

- Start with the responsibilities which are identified from the use cases, conceptual model, and system operations' contracts.
- Assign these responsibilities to objects, then
  - —Decide what the objects need to do to fulfill these responsibilities in order to identify further responsibilities which are again assigned to objects.
  - —Repeat these steps until the identified responsibilities are fulfilled and a collaboration diagram is completed.

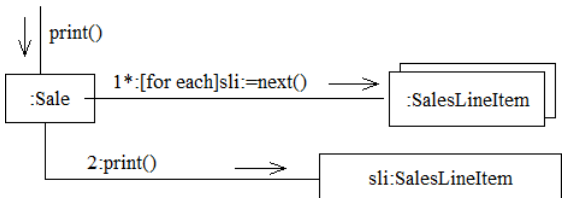
Create Collaboration Diagrams

- Responsibilities of an object are implemented by using *methods* of the object which either act alone or collaborate with other methods and objects. For example,
  - —the *Sale* class might define a method that is performing printing of a *Sale* instance; say a method named *print*. To fulfill that responsibility,
  - —the *Sale* instance may have to collaborate with other objects, such as sending a message to SalesLineItem objects asking them to print themselves.

Create Collaboration Diagrams

- Within UML,
  - —responsibilities are assigned to objects when creation a collaboration diagram and
  - —the collaboration diagram represents both of the assignment of responsibilities to objects and the collaboration between objects for their fulfillment.
- Example
  - —*Sale* objects have been given a responsibility to print themselves, which is invoked with a *print* message.
  - —The fulfillment of this responsibility requires collaboration with *SalesLineItem* objects asking them to print.

Create Collaboration Diagrams



General Principles in Assigning Responsibilities

- **GRASP: Patterns of General Principles in Assigning Responsibilities**
- **GRASP** patterns: **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns
- Patterns are best principles for assigning responsibilities to objects.
- Patterns are guide to the creation of software methods.
- They are best principles for assigning responsibilities to objects.
- They are solutions to common occurring problems.

GRASP Patterns

Pattern Name:	the name given to the pattern
Solution:	description of the solution of the problem
Problem:	description of the problem that the pattern solves

- Most simply, a pattern is a named problem/solution pair that can be applied to new context, with advice on how to apply it in novel situations.
- GRASP patterns include the following five basic patterns:
  1. Expert,
  2. Creator,
  3. Low Coupling
  4. High Cohesion and
  5. Controller

GRASP 1: Expert

- This is a very simple pattern and is used more than any other pattern in assignment of responsibilities.
- Expert class is that class that has information necessary to fulfill the responsible.

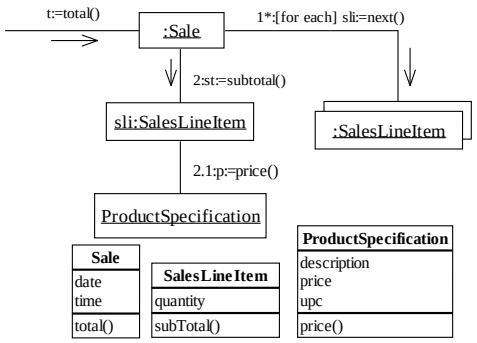
Pattern Name	Expert
Solution	Assign a responsibility to the information expert — the class that has the information necessary to fulfill the responsibility.
Problem	What is the most basic principle by which responsibilities are assigned in OOD?

GRASP 1: Expert

- In the POST application, some class needs to know the grand total of a sale
- When we assign responsibilities we had better to state the responsibility clearly
  - —Who should be responsible for knowing the grand total of the sale
- By expert, we should look for that class of objects that has the information
  - —All the *salesLineItem* instances of *sale* instance, and
  - —The *sum* of their subtotal

    That is needed to determine the total

GRASP 1: Expert



GRASP 1: Expert

- Expert leads to designs where a software object does those operations which are normally done to the real-world (or domain) object it represents; this is called the "**DO it Myself**" strategy.
- Fundamentally, objects do things related to information they know.
- The use of the Expert pattern maintain encapsulation, since objects use their own information to fulfill responsibilities.

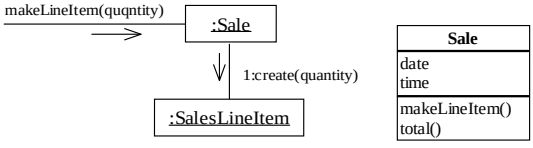
GRASP 2: Creator

- The creation of objects is one of the most common activities in an OO system.
- It asks the question "who should be responsible for creating instances of a particular class?"

Pattern Name	Creator
Solution	Assign class B the responsibility to create an instance of class A if one of these is true: 1. B aggregates A 2. B contains A 3. B records A 4. B closely uses A 5. B has the initializing data for A Thus B is an expert with respect to creating A objects
Problem	What should be responsible for creating a new instance of some class?

GRASP 2: Creator

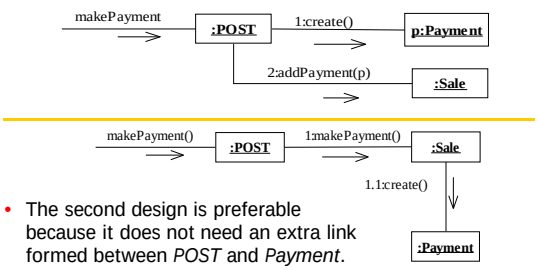
- Since a *Sale* contains (aggregates) many *SalesLineItem* objects, the Creator pattern suggests *Sale* is a good candidate to be responsible for creating *SalesLineItem* objects.
- This leads to a design of object interactions in a collaboration diagram.



GRASP 4: Low Coupling

- Coupling is a measure of how strongly one class is **connected to**, **has knowledge of**, or **relies on other classes**.
- A class with low (or weak) coupling is not dependent on too many other classes.
- When we assign a responsibility to a class, we would like to assign responsibilities in a way so that coupling between classes remains low.
- A class with high (or strong) coupling relies upon many other classes.

GRASP 4: Low Coupling



- The second design is preferable because it does not need an extra link formed between *POST* and *Payment*.
- This assignment of responsibility is also justifiable as it is reasonable to think that a *Sale* closely uses a *Payment*.

GRASP 4: Low Coupling

- Classes with high (or strong) coupling are undesirable; they suffer from the following problems:
  - —Changes in related classes force local changes, i.e. changes in this class.
  - —Changes in this class force the changes in many other related classes.
  - —Harder to understand in isolation.
  - —Harder to reuse as its reuse requires the use of the related class.

Pattern Name	Low Cohesion
Solution	Assign a responsible so that coupling remains low
Problem	How to support low dependency an increased reuse?

GRASP 4: Low Coupling

- Coupling may not be that important if reuse is not a goal, and there is not absolute measure of when coupling is too high.
- While high coupling makes it difficult to reuse components of the system, a system in which there is very little or no coupling between classes is rare and not interesting.

GRASP 3: High Cohesion

- Cohesion or coherence is the strength of dependencies within a subsystem.
- Cohesion is a measure of how strongly related and focused the responsibilities of a class.
- It is the internal “glue” with which a subsystem is constructed.
- A component is cohesive if all its elements are directed towards a task and the elements are essential for performing the same task.
- If a subsystem contains unrelated objects, coherence is low. High cohesion is desirable.

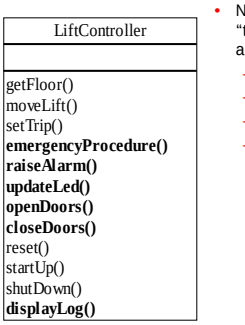
GRASP 3: High Cohesion

- A class with low cohesion is undesirable as it suffers from the following problems:
  - —hard to comprehend;
  - —hard to reuse;
  - —hard to maintain;
- It is often the case that a class takes on a responsibility and delegates parts of the work to fulfill the responsibility to other classes.
- This nature is captured by the high cohesion pattern.

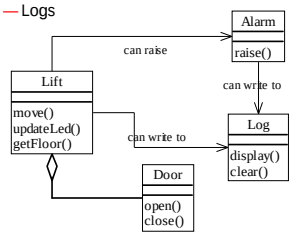
GRASP 3: High Cohesion

- A class with high cohesion has highly related functional responsibilities, and does not do tremendous amount of work.
- Such classes have a small number of methods with simple but highly related functionality.
- In a good object-oriented design, each class should not do too much work.
  - —Assign few methods to a class and delegate parts of the work to fulfill the responsibility to other classes.
- The following “liftController” class is not well designed. A class does a lot of work.

GRASP 3: High Cohesion



- Note that a class should represent one "thing" from the real would. Key separate abstractions from liftController are:
  - An Alarm
  - Lift
  - Doors
  - Logs



GRASP 3: High Cohesion

- The benefits from the use of the High Cohesion Pattern include:
  - Clarity and ease of comprehension of the design is increased.
  - Maintenance and enhancements are simplified.
  - Low coupling is often supported.
  - Supports reuse.

Pattern Name	High Cohesion
Solution	Assign a responsible so that cohesion remains high
Problem	How to keep complexity manageable?

GRASP 5: Controller

- Who handles a system event?
- Assign the responsibility for handling a system event message to a class representing one of these choices:
  - Represents the overall system, device, or a subsystem (facade controller).
  - Represents a use case scenario within which the system event occurs (use-case or session controller)
- A **controller is a non-user interface object responsible in handling a system input events**, and the controller defines the method for the system operation corresponding to the system input event.

GRASP 5: Controller

- One possible solution is to add/introduce a new class, and make it sit between the actor and the business classes.
- The name of this controller is usually called **<name>Handler**.
- Handler reads the commands from the user and then decides which classes the messages should be directed to.
- The handler is the only class that will be allowed to read and display.

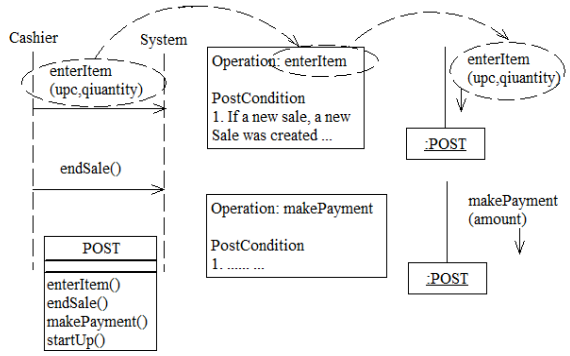
GRASP 5: Controller

Pattern Name	Controller
Solution	Assign the responsibility for handling a system (input) event to a class representing one of the following choices – Represents the "overall system" (*facade controller*). – Represents the overall business or organization (*facade controller*). – Represents something in the real-world that is active (for example, the role of a person) that might be involved in the task (*role controller*). – Represents an artificial handler of all system (input) events of a use case, usually named "< UseCaseName> Handler" (*use-case controller*).
Problem	Who should be responsible for handling a system input event?

A Design of POST

- This section applies the GRASP patterns to assign responsibilities to classes, and to create collaboration diagrams for POST
- We consider ***Buy Items with Cash*** and ***Start Up*** use cases in this design
- Guidelines for making a collaboration diagram:
  - Create a separate diagram for each system operation which has identified and whose contracts are defined.
  - If the diagram gets complex, split it into smaller diagrams.
  - Using the contract responsibilities and post-conditions, and use case description as a starting point, design a system of interacting objects to fulfill the tasks. Apply the GRASP to develop a good design.

Buy Items with Cash



Buy Items with Cash and *Start Up*

- With **Buy Items with Cash** and **Start Up** use cases, we have identified four system operations ***enterItem***, ***endSale***, ***makePayment*** and ***startUp***.
- According to our guidelines we should construct at least four collaboration diagrams.
- According to the Controller pattern, the *POST* class could be chosen as controller for handling these operations.

POST
enterItem() endSale() makePayment() startUp()

Collaboration Diagram for *enterItem*

- we need to look at the contract of *enterItem* what *POST* needs to do to fulfill this responsibility.

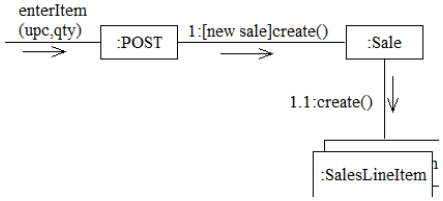
Post-conditions:	—If a new sale, a Sale was created (*instance creation*). —If a new Sale, the *Sale* was associated with the *POST* (*association formed*). —A *SalesLineItem* was created (*instance creation*). —The *SalesLineItem.quantity* was set to quantity (*attribute modification*). —The *SalesLineItem* was associated with a *ProductSpecification*, based on *UPC* match (*association formed*).

Creating a New Sale

- The post-conditions of *enterItem* indicate a responsibility for creation an object *Sale*.
- The **Creator pattern** suggests *POST* is a reasonable candidate creator of the *Sale*, as *POST* records the *Sale*.
- Having *POST* create the *Sale*, the *POST* can easily be associated with it over time.

Creating a New SalesLineItem

- When the Sale is created, it must create an empty collection to record all the future *SaleLineItem* that will be added.
- This collection will be maintained by the *Sale* instance, which implies by Creator that the *Sale* is a good candidate for creating it (SalesLineItem).



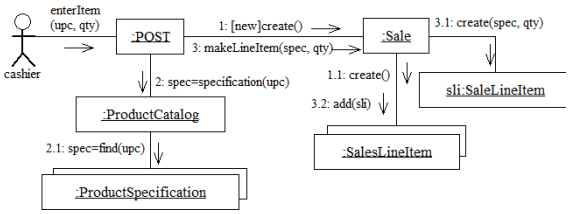
Finding a ProductSpecification

- Newly created SalesLineItem is required to be associated with ProductSpecification that matches with upc
- This need the parameters to the makeLineItem message sent to the Sale include ProductSpecification instance denoted by sec, which matches upc
- We need to retrieve the ProductSpecification before the message makeItem(spec,qty) is sent to Sale
- From Expert pattern, ProductCatalog contains all the productSpecification, hence good candidate for looking up the ProductSpecification

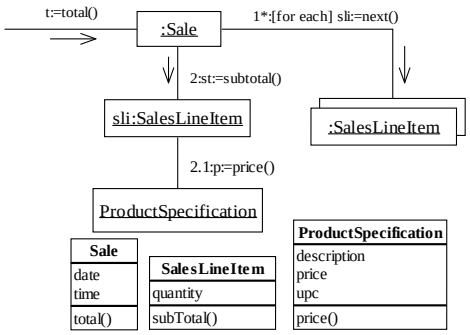
Visibility to a ProductCatalog

- From startUp() contract it shows POST was asociated with the ProductCatalog
- POST is responsible in sending message to ProductCatalog denoted by specification

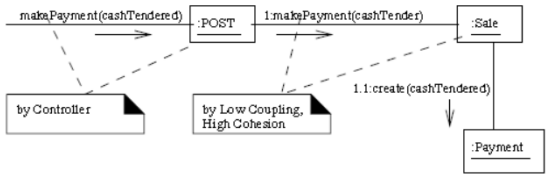
Collaboration Diagram for *enterItem*



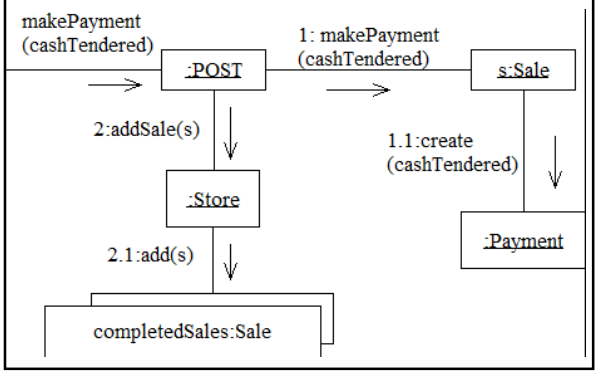
Collaboration Diagram for *endSale*



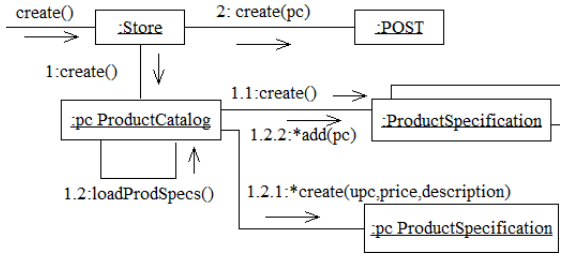
Collaboration Diagram for *makePayment*



Collaboration Diagram for *makePayment*



Collaboration Diagram for *StartUp*



Design Class Diagram

- During creation of a collaboration diagram, we record the methods corresponding to the responsibilities assigned to a class in the third section of the class box. These classes with methods are *software classes representing the conceptual classes in the conceptual* models. Then based on these identified software classes, the collaboration diagrams and the original conceptual model, we can create a design class diagram which illustrate the following information:
- Classes, associations and attributes
- Methods
- Attribute type information
- Navigability
- Dependencies

Steps in Making a Design Class Diagram

1. Identify all the classes participating in object interaction by analyzing the collaborations
2. Present them in a class diagram
3. Copy attributes from the associated concepts in the conceptual model
4. Add methods names by analyzing the interaction diagrams
5. Add type information to the attributes and methods

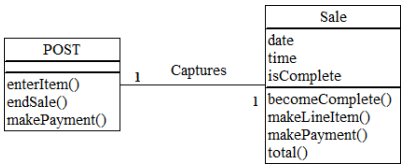
Steps in Making a Design Class Diagram

6. Add the association necessary to support the required attribute visibility
7. Add navigability arrow necessary to the associations to indicate the direction of the attribute visibility
8. Add dependency relationship lines to indicate non-attribute visibility

- More:
  - Enhance attributes by adding datatypes
  - Determine visibility e.g. public (+), Private (–) – concept for encapsulation

Add Associations and Navigability

- Each end of an association is called a role.
- In a design class diagram, the role may be decorated with a navigability arrow.
- **Navigability** is a property of the role which indicates that it is possible to navigate uni-directionally across the association from objects of the source to the target class.



DCD with Association and Navigability