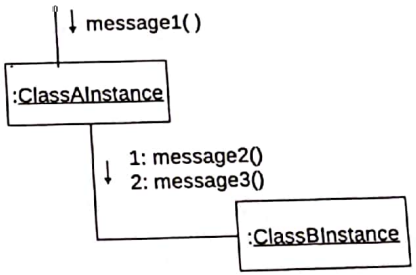
Bidur Devkota

From Textbook: *Applying UML and patterns* C. Larman

Interaction Diagram Notation

Sequence & Collaboration Daigrams

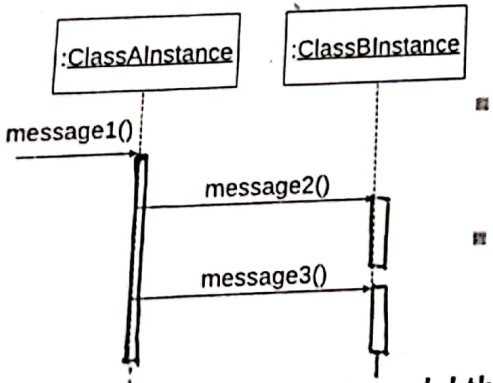
Introduction



- *Interaction diagrams* illustrate how objects interact via messages.
- ***Collaboration*** *diagrams* illustrate object interactions in a graph or network format.

Collaboration Diagrams → model the structural organization of objects

Introduction



- ***Sequence*** *diagrams* illustrate interactions in a kind of fence format.
- Set of all operation contracts defines system behavior.
- We will create an interaction diagram for each operation contract.

Sequence Diagrams → model the time ordering of messages among objects.

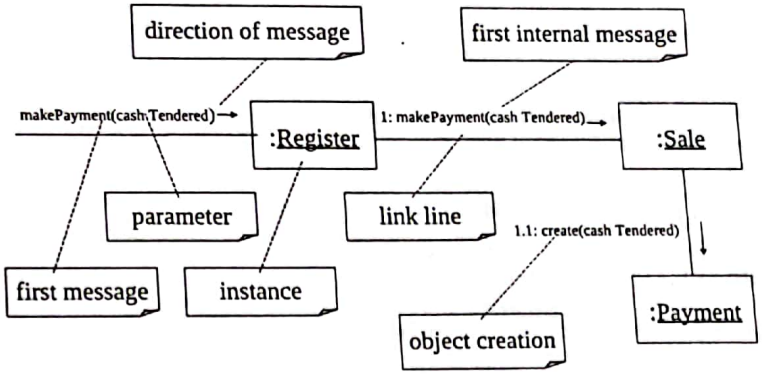
Introduction

- ***Sequence*** *diagrams:*
  - Strength: shows time ordering of events in simple notation
  - Weakness: Must add new objects to the right side (space constraint)
- ***Collaboration*** *Diagrams:*
  - Strength: space economical as objects can be added in any direction. Better to show complex branching,iteration and concurrent behavior
  - Weakness: difficult to visualize sequence of messages , more complex notation.

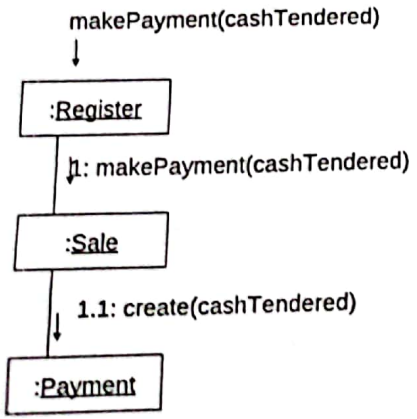
Sequence vs. Collaboration

Sequence Diagrams	Collaboration Diagrams
The sequence diagram represents the UML, which is used to visualize the sequence of calls in a system that is used to perform a specific functionality.	The collaboration diagram also comes under the UML representation which is used to visualize the organization of the objects and their interaction.
The sequence diagram are used to represent the sequence of messages that are flowing from one object to another.	The collaboration diagram are used to represent the structural organization of the system and the messages that are sent and received.
The sequence diagram is used when time sequence is main focus.	The collaboration diagram is used when object organization is main focus.
The sequence diagrams are better suited of analysis activities.	The collaboration diagrams are better suited for depicting simpler interactions of the smaller number of objects.

Example Collaboration Diagram: makePayment

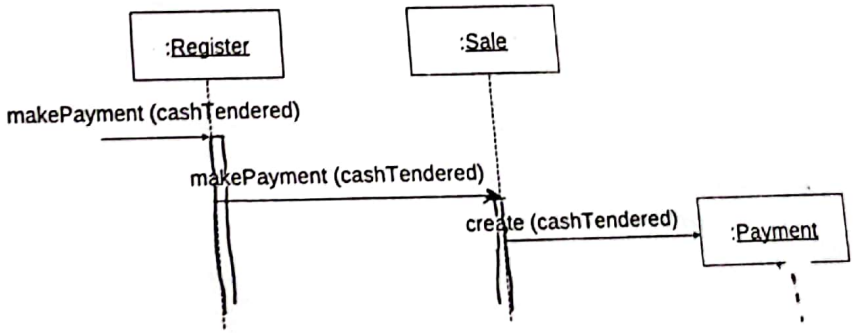


How to Read the makePayment Collaboration Diagram



1. The message makePayment is sent to an instance of Register. The sender is not identified.
2. The Register instance sends the makePayment message to a Sale instance.
3. The Sale instance creates an instance of a Payment.

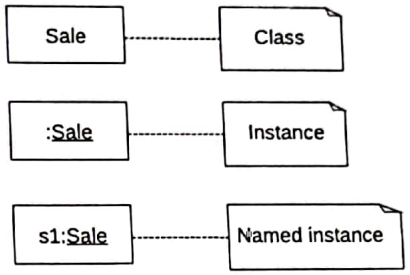
Example Sequence Diagram: makePayment



Interaction diagrams are valuable

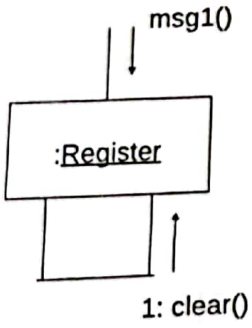
- Assignment of responsibility
- creation of use cases, domain models, and other artifacts is easier than this task
- Making interaction diagrams is deciding on details of object design is a creative step in OOAD.
- Coding patterns, principles,and idioms can be applied for improving quality.

Illustrating Classes and Instances



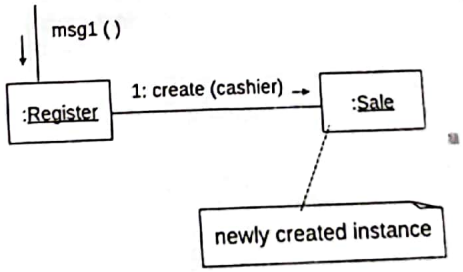
- To show an instance of a class, the regular class box graphic symbol is used, but the name is underlined. Additionally a class name should be preceded by a colon.
- An instance name can be used to uniquely identify the instance.

Messages to "self" or "this"



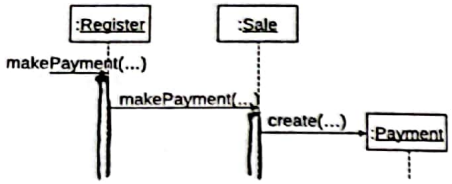
- A message can be sent from an object to itself.
- This is illustrated by a link to itself, with messages flowing along the link.
- multiple messages, and messages both ways, can flow along the same single link.
- Sequence number is used to show message ordering

Creation of Instances



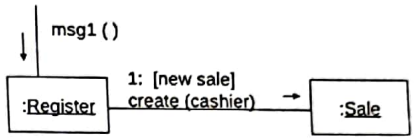
- The language independent creation message is create, being sent to the instance being created.
- The create message may include parameters, indicating passing of initial values.

Creation of Instances



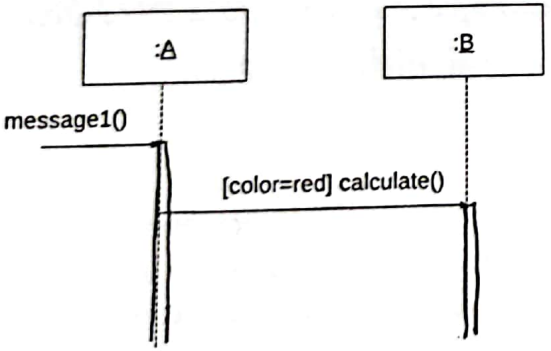
- An object lifeline shows the extend of the life of an object in the diagram.
- Note that newly created objects are placed at their creation height.

Conditional Messages

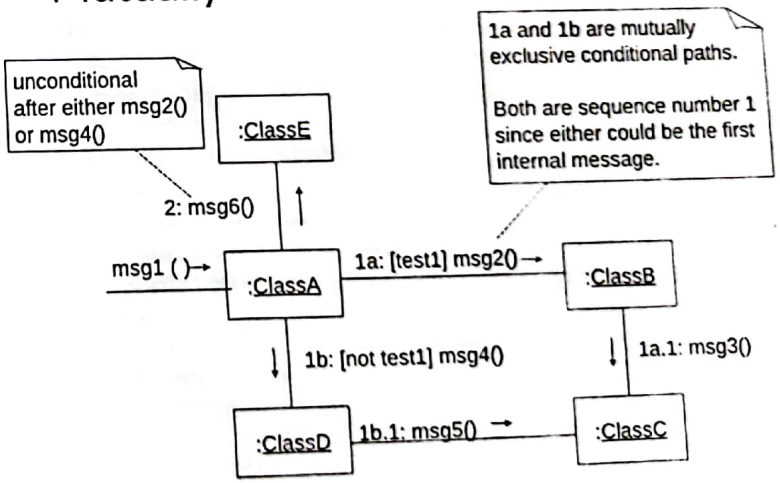


- A conditional message is shown by following a sequence number with a conditional clause in square brackets, similar to the iteration clause.
- The message is sent only if the clause evaluates to true.

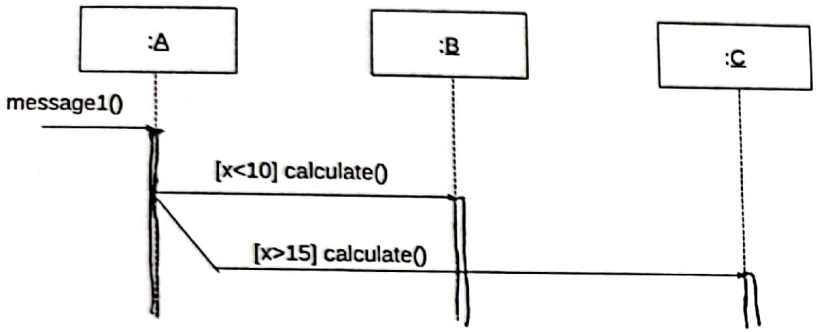
Conditional Messages



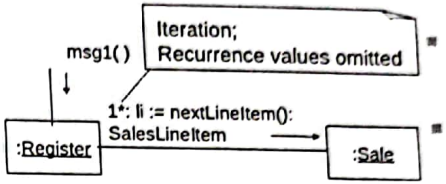
Mutually Exclusive Conditional Paths



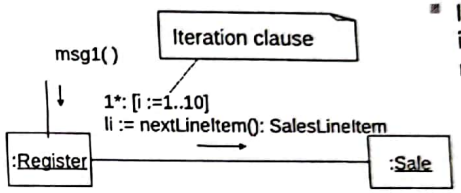
Mutually Exclusive Conditional Messages



Iteration or Looping



- Iteration is indicated by following the sequence number with a star *
- This expresses that the message is being sent repeatedly, in a loop, to the receiver.
- It is also possible to include an iteration clause indicating the recurrence values.



GRASP: Designing Objects with Responsibilities

- **GRASP**: General Responsibility Assignment Software Patterns
- to help one understand essential object design, and apply design reasoning in a methodical, rational, explainable way.
- **Patterns of assigning responsibilities**: This approach to understanding and using design principles

Responsibilities and Methods

- The focus of object design is to identify classes and objects, decide what methods belong where and how these objects should interact.
- Responsibilities are related to the obligations of an object in terms of its behavior.
- Two types of responsibilities:
  - Doing:
    - Doing something itself (e.g. creating an object, doing a calculation)
    - Initiating action in other objects.
    - Controlling and coordinating activities in other objects.
  - Knowing:
    - Knowing about private encapsulated data.
    - Knowing about related objects.
    - Knowing about things it can derive or calculate.

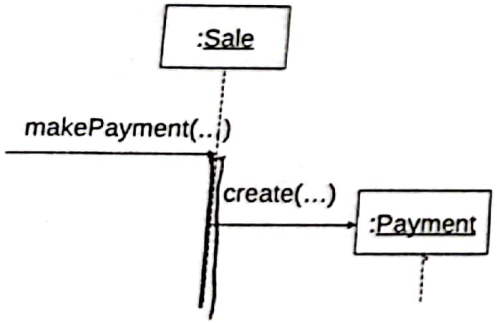
Responsibilities and Methods

- Responsibly: a contract or obligation of a classifier
- Responsibilities are assigned to classes during object design. For example, we may declare the following:
  - *"a Sale is responsible for creating SalesLineItems"* (doing)
  - *"a Sale is responsible for knowing its total"* (knowing)
- Responsibilities related to "knowing" are often inferable from the Domain Model (because of the attributes and associations it illustrates)

Responsibilities and Methods

- The translation of responsibilities into classes and methods is influenced by the granularity of responsibility.
  - For example, *"provide access to relational databases"* may involve dozens of classes and hundreds of methods, whereas *"create a Sale"* may involve only one or few methods.
- A responsibility is not the same thing as a method, but methods are implemented to fulfill responsibilities.
- To fulfil a responsibility, Methods either act alone, or collaborate with other methods and objects.

Responsibilities and Interaction Diagrams



- Within the UML artifacts, a common context where these responsibilities (implemented as methods) are considered is during the creation of interaction diagrams.
- Sale objects have been given the responsibility to create Payments, handled with the makePayment method.

Patterns

- A way of reusing idea and design of some problem-solution.
- "pattern" means a *repeating thing*.
- NOT to express new design ideas.
- To codify existing tried-and-true knowledge, idioms, and principles; the more honed and widely used, the better.
- GRASP Pattern:
  - a codification of widely used basic principles (NOT new things).
  - Describe fundamental principles of object design and responsibility assignment.
- All patterns ideally have suggestive names (merit: easy understanding, memory & communication)

Patterns

- It emphasizes on principles to guide choices in where to assign responsibilities.
- A pattern is a named description of a problem and a solution that can be applied to new contexts; it provides advice in how to apply it in varying circumstances. For example,
  - Pattern name: Information Expert
  - Problem: What is the most basic principle by which to assign responsibilities to objects?
  - Solution: Assign a responsibility to the class that has the information needed to fulfil it.

Information Expert (or Expert)

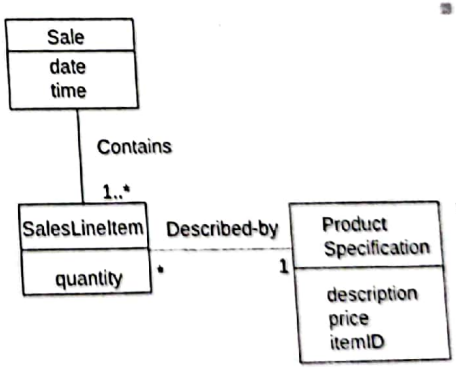
- Problem: what is a general principle of assigning responsibilities to objects?
- Solution: Assign a responsibility to the information expert - the class that has the information necessary to fulfill the responsibility.
- In the NextGen POS application, who should be responsible for knowing the grand total of a sale?
- By Information Expert we should look for that class that has the information needed to determine the total.
- **Real-world analogy:** we assign responsibility to the person who have the information necessary to fulfill a task

Information Expert

- Do we look in the Domain Model or the Design Model to analyze the classes that have the information needed?
  - If there are relevant classes in the Design Model, look there first.
  - Else, look in the Domain Model, and attempt to use (or expand) its representations to inspire the creation of corresponding design classes.

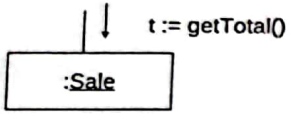
Information Expert

Who should be responsibe for determining grand total?



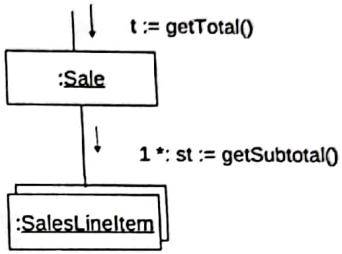
- It is necessary to know about all the SalesLineItem instances of a sale and the sum of the subtotals.
- A Sale instance contains these, i.e. it is an information expert for this responsibility.

Information Expert



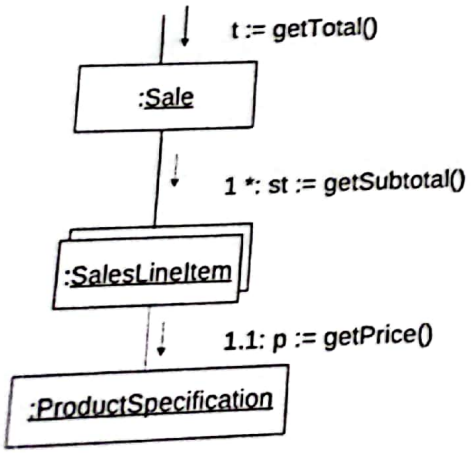
- This is a partial interaction diagram.

Information Expert



- What information is needed to determine the line item subtotal?
  - quantity and price.
- SalesLineItem should determine the subtotal.
- This means that Sale needs to send getSubtotal() messages to each of the SalesLineItems and sum the results.

Information Expert



- To fulfil the responsibility of knowing and answering its subtotal, a SalesLineItem needs to know the product price.
- The ProductSpecification is the information expert on answering its price.

Information Expert

Class	Responsibility
Sale	Knows Sale total
SalesLineItem	Knows line item total
ProductSpecification	Knows product price

- To fulfil the responsibility of knowing and answering the sale's total, three responsibilities were assigned to three design classes
- The fulfillment of a responsibility often requires information that is spread across different classes of objects. This implies that there are many "partial experts" who will collaborate in the task.

Information Expert

Contradiction:

- Usually because of problems in coupling and cohesion
- E.g. who should be responsible for saving a Sale in a database?
  - information to be saved is in the Sale object. So Expert pattern suggests Sale object.
  - This means each class should have its own services to save itself in a database
  - This leads to problems in cohesion, coupling, and duplication as the Sale class must contain logic related to database handling.
  - violation of a basic architectural principle: design for a separation of major system concerns

Information Expert

Merit:

- Behavior is distributed across the classes that have the required information
  - Encouraging more cohesive "lightweight" class definitions
  - Easier to understand and maintain
- Information encapsulation is maintained.
  - As objects use their own information to fulfill tasks
  - Emphasizes Low coupling

Information Expert

Also known as:

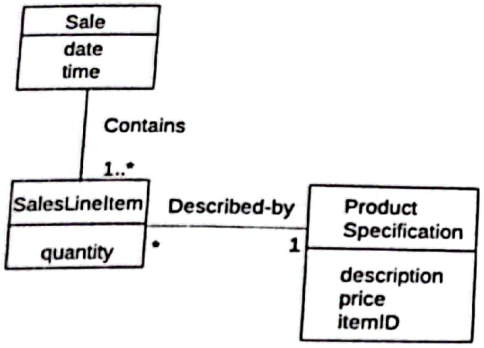
- "Place responsibilities with data,"
- "That which knows, does,"
- "Do It Myself,"
- "Put Services with the Attributes They Work On."

Creator

- Problem: Who should be responsible for creating a new instance of some class?
- Solution: Assign class B the responsibility to create an instance of class A if one or more of the following is true:
  1. B *aggregates* A objects. (E.g. Library aggregates Books)
  2. B *contains* A objects. (Shelf contains Book)
  3. B *records* instances of A objects. (Librarian records new Books)
  4. B *closely uses* A objects.(Librarian uses Books)
  5. B *has the initializing data* that will be passed to A when it is created (thus B is an Expert with respect to creating A).

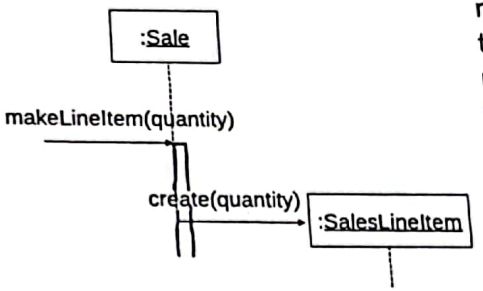
(Librarian records Book information like book name, id, cost..)

Creator



- In the POS application, who should be responsible for creating a SalesLineItem instance?
- Since a Sale contains many SalesLineItem objects, the Creator pattern suggests that Sale is a good candidate.

Creator



- This assignment of responsibilities requires that a makeLineItem method be defined in Sale.

Creator

- ***Contradiction:***

  Often, creation requires significant complexity, such as using recycled instances for performance reasons, conditionally creating an instance from one of a family of similar classes based upon some external property value, and so forth. it is advisable to delegate creation to a helper class called a **Factory.**

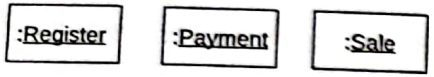
- ***Benefit:***

  Low Coupling - because the created class is likely already visible to the creator class

Low Coupling

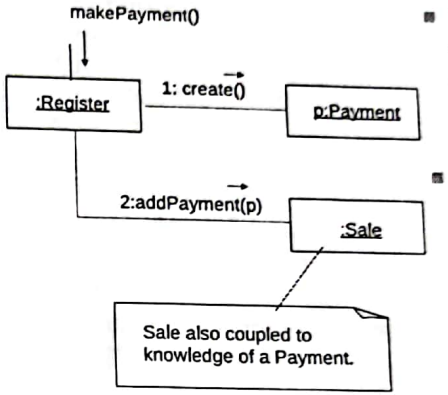
- Coupling: it is a measure of how strongly one element is connected to, has knowledge of, or relies upon other elements.
- A class with high coupling depends on many other classes (libraries, tools).
- Problems because of a design with high coupling:
  - Changes in related classes force local changes.
  - Harder to understand in isolation; need to understand other classes
  - Harder to reuse because it requires additional presence of other classes.
- Problem: How to support low dependency, low change impact and increased reuse?
- Solution: Assign a responsibility so that coupling remains low.

Low Coupling



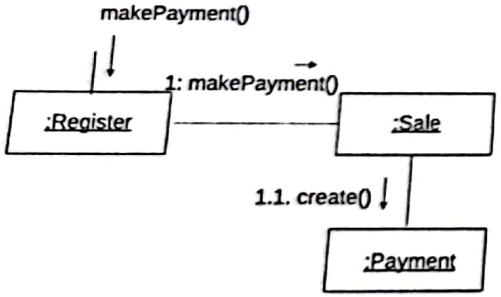
- Assume we need to create a Payment instance and associate it with the Sale.
- What class should be responsible for this?
- By Creator, Register is a candidate.

Low Coupling



- Register could then send an addPayment message to Sale, passing along the new Payment as a parameter.
- The assignment of responsibilities couples the Register class to knowledge of the Payment class.

Low Coupling



- An alternative solution is to create Payment and associate it with the Sale.
- No coupling between Register and Payment.

Low Coupling

- Some of the places where coupling occurs:
  - Attributes: X has an attribute that refers to a Y instance.
  - Methods: e.g. a parameter or a local variable of type Y is found in a method of X.
  - Subclasses: X is a subclass of Y.
  - Types: X implements interface Y.
- There is no specific measurement for coupling, but in general, classes that are generic and simple to reuse have low coupling.
- There will always be some coupling among objects, otherwise, there would be no collaboration.

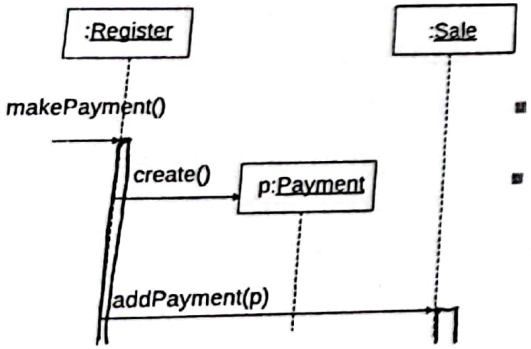
Low Coupling

- Superclass subclass have strong coupling
- The extreme case of Low Coupling is when there is no coupling between classes, which is not desirable because a central metaphor of object technology is a system of connected objects that communicate via messages.
- **Contradiction:** High coupling to stable elements and to pervasive elements is seldom a problem. For example, a Java J2EE application can safely couple itself to the Java libraries (java.util, and so on), because they are stable and widespread.

High Cohesion

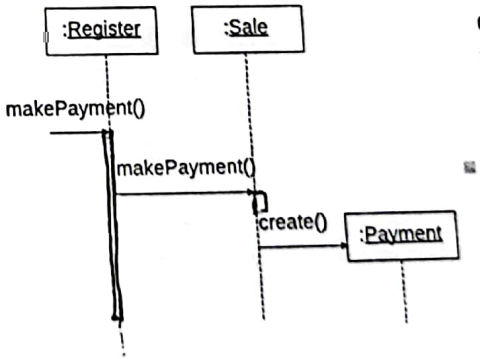
- Cohesion: it is a measure of how strongly related and focused the responsibilities of an element are.
- A class with low cohesion does many unrelated activities or does too much work.
- Problems because of a design with low cohesion:
  - Hard to understand.
  - Hard to reuse.
  - Hard to maintain.
  - Delicate, affected by change.
- Problem: How to keep complexity manageable?
- Solution: Assign a responsibility so that cohesion remains high.

High Cohesion



- Assume we need to create a Payment instance and associate it with Sale. What class should be responsible for this?
- By Creator, Register is a candidate.
- Register may become bloated if it is assigned more and more system operations.

High Cohesion



- An alternative design delegates the Payment creation responsibility to the Sale, which supports higher cohesion in the Register.
- This design supports high cohesion and low coupling.

High Cohesion

- Scenarios that illustrate varying degrees of functional cohesion

1. Very low cohesion: class responsible for many things in many different areas.
   - e.g.: a class responsible for interfacing with a data base and remote-procedure-calls.
2. Low cohesion: class responsible for complex task in a functional area.
   - e.g.: a class responsible for interacting with a relational database.

High Cohesion

3. High cohesion: class has moderate responsibility in one functional area and it collaborates with other classes to fulfill a task.
   - e.g.: a class responsible for one section of interfacing with a data base.

- Rule of thumb: a class with high cohesion has a relative low number of methods, with highly related functionality, and doesn't do much work. It collaborates and delegates.

Course Grained Remote Interface: (for distributed objects): E.g. instead of a remote object with three fine-grained operations setName, setSalary, and setHireDate, there is one remote operation setData which receives a set of data. This results in less remote calls, and better performance.

High Cohesion: benefits

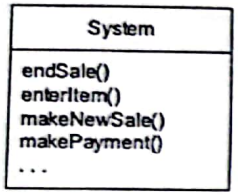
- Clarity and ease of comprehension of the design is increased.
- Maintenance and enhancements are simplified.
- Low coupling is often supported.
- The fine grain of highly related functionality supports increased reuse because a cohesive class can be used for a very specific purpose.

Controller

- Problem: Who should be responsible for handling an input system event?
- Solution: Assign the responsibility for receiving or handling a system event message to a class representing one of he following choices:
  - Represents the overall system.
  - Represents a use case scenario.
  - A Controller is a non-user interface object that defines the method for the system operation. Note that windows, applets, etc. typically receive events and delegate them to a controller.

Controller

- Example: NextGen application, there are several system operations, as illustrated below, showing the system itself as a class or component.
- Normally, a controller delegates work to other objects
- It coordinates or controls the activity. It does not do much work itself.



System operations associated with the system events.

Controller

Facade controller:

- representing the overall system, device, or a subsystem.
- choose some class name that suggests a cover, or facade, over the other layers of the application, and that provides the main point of service calls from the UI layer down to other layers.
- an abstraction of the overall physical unit, such as a Register, *TelecommSwitch, Phone, or Robot;*
- a class representing the entire software system, such as POSSystem,
- any other concept which the designer chooses to represent the overall system or a subsystem, for example, *ChessGame* if it was game software.
- *Facade controllers are suitable when there are not "too many" system events, or it is not possible for the user interface (UI) to redirect system event messages to alternating controllers*

Controller

Use case controller:

- *Bloated Controller :*
  - *with excessive responsibilities*
  - *placing the responsibilities in a facade controller leads to designs with low cohesion or high coupling*
- *A use-case controller is a good when there are many system events across different processes;*
  - *it factors their handling into manageable separate classes*
  - *provides a basis for knowing and reasoning about the state of the current scenario in progress.*

Controller

Bloated Controller :

- There is only a single controller class receiving all system events in the system, and there are many of them. This sometimes happens if a facade controller is chosen.
- The controller itself performs many of the tasks necessary to fulfill the system event, without delegating the work. This usually involves a violation of Information Expert and High Cohesion.
- A controller has many attributes, and maintains significant information about the system or domain, which should have been distributed to other objects, or duplicates information found elsewhere.

Controller

Bloated Controller : Solution

- Add more controllers—a system does not have to have only one. Instead of facade controllers, use use-case controllers.
- Design the controller so that it primarily delegates the fulfillment of each system operation responsibility on to other objects.
- For e.g., consider an application with many system events, such as an airline reservation system. It may contain the following controllers:

Use-case controllers
MakeReservationHandler
ManageSchedulesHandler
ManageFaresHandler

Controller

Interface Layer Does Not Handle System Events

- The interface objects (like, window objects) and the interface layer should **not** handle system events.
- Handle system operations to application layer objects *(NOT interface objects)* for increased reuse potential.
  - system operation— represents part of a business process logic which should not be tied with interfaces.
- For e.g. in *NextGenPOS*, Java JFrame is used to displays sale information and captures cashier operations

Controller

Interface Layer Does Not Handle System Events

- For e.g. in *NextGenPOS*, Java JFrame is used to displays sale information and captures cashier operations

- The figure adheres with Controller pattern:
  - *SaleJFrame* class (interface) passes the *enterItem()* message to the *Register* object (Controller).
  - The interface class did not get involved in processing the operation or deciding how to handle *enterItem()*.
- The interface window only delegated it to another layer.

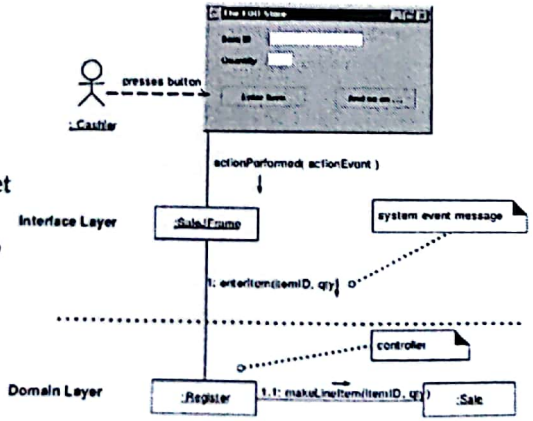


Figure 16.17 Desirable coupling of interface layer to domain layer.

Controller

Interface Layer Does Not Handle System Events

- The design adheres with Controller pattern:
  - relationship between the JFrame interface and Controller and other objects

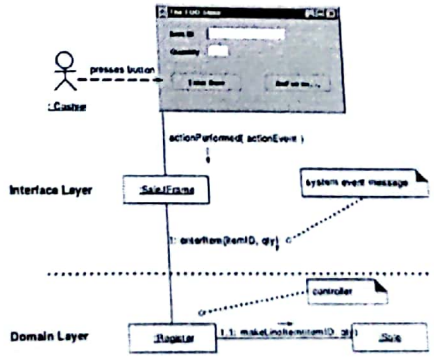


Figure 16.17 Desirable coupling of interface layer to domain layer.

- This design do **NOT** follow Controller pattern (Fig 16.18).

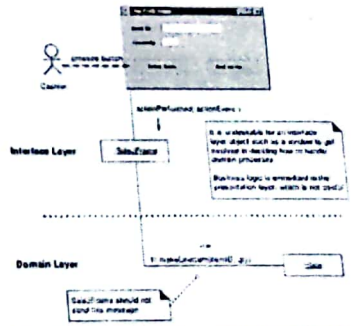


Figure 16.18 Less desirable coupling of interface layer to domain layer.

Controller

Interface Layer Does Not Handle System Events

- If the interface objects (like, window objects) handle system events decreases re-usability

- If an interface layer object (e.g. *SaleJFrame*) handles a system operation then business process logic would be contained in an interface object.
- Interface object has low opportunity for reuse because of its coupling to a particular interface and application.
- The design in Figure 16.17 in the preceding slide supports more reusability than this design (Figure 16.18)

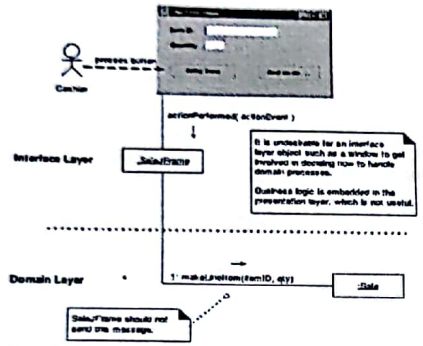


Figure 16.18 Less desirable coupling of interface layer to domain layer.

126

Use Case Realizations

Use Case Realizations

System
makeNewSale() addLineItem(itemID, quantity) endSale() makePayment()

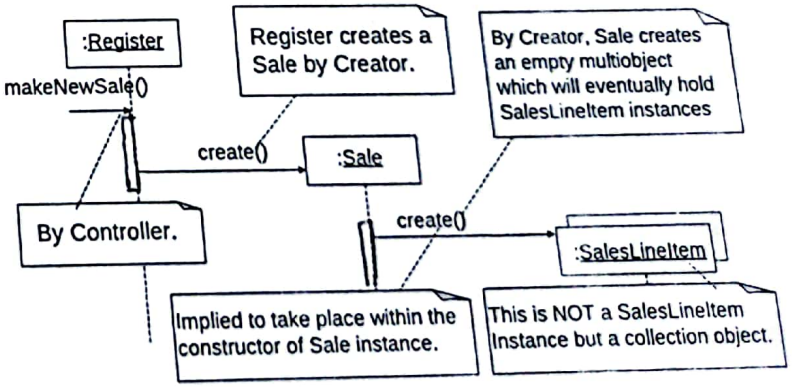
Contract CO1: makeNewSale
Operation: makeNewSale ()
Cross References: Use Cases: Process Sale.
Pre-conditions: none.
Post-conditions:
- A Sale instance s was created. (instance creation)
- s was associated with the Register (association formed)
- Attributes of s were initialized

- A use-case realization describes how a use case is realized in terms of collaborating objects.
- UML interaction diagrams are used to illustrate use case realizations.
- Recall Process Sale: from main scenario we identified a number of system events (operations)
- Each system event was then described by a contract.

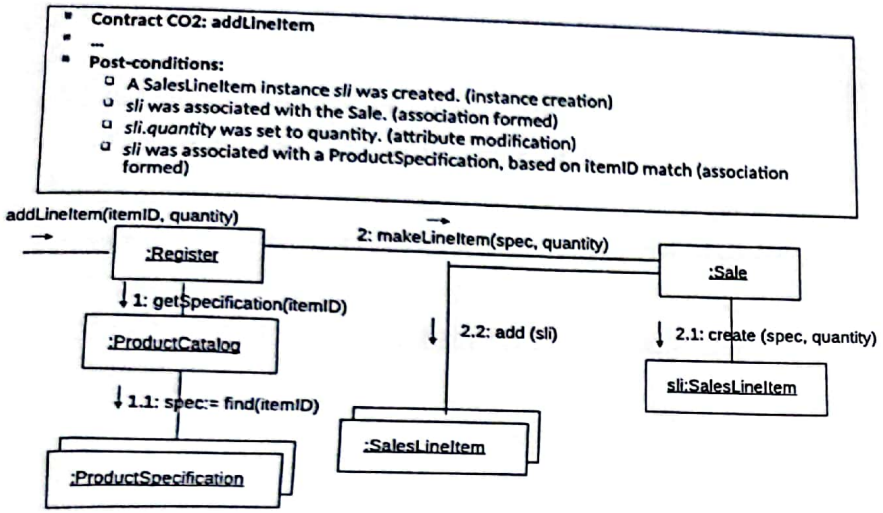
132

Object Design: makeNewSale

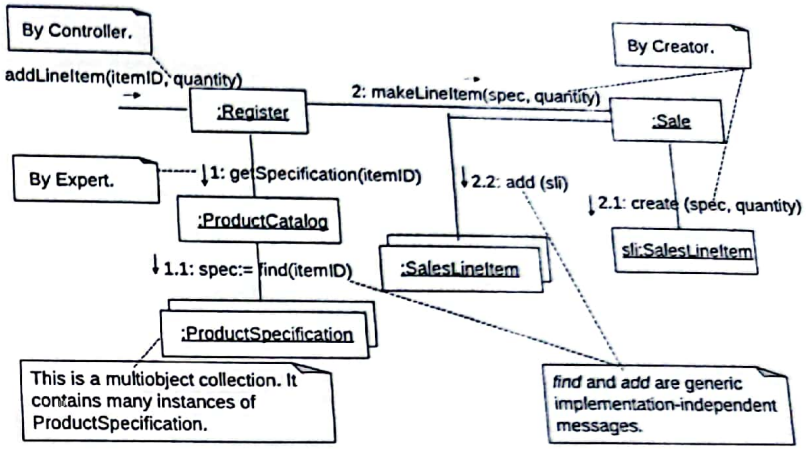
- We work through the postcondition state changes and design message interactions to satisfy the requirements.



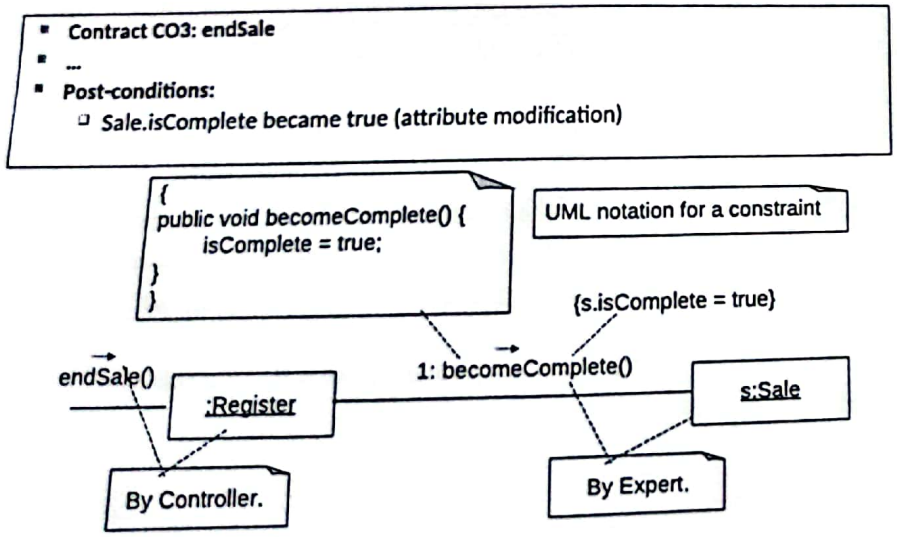
Object Design: addLineItem



Object Design: addLineItem



Object Design: endSale

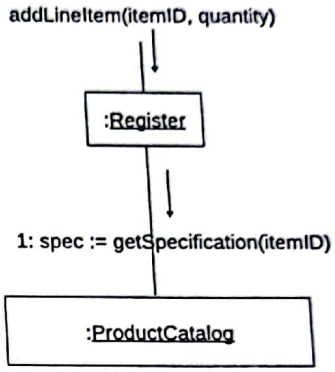


Design Model: Determining Visibility

Introduction

- Visibility: the ability of an object to "see" or have reference to another object.
- For a sender object to send a message to a receiver object, the receiver must be visible to the sender – the sender must have some kind of reference (or pointer) to the receiver object.

Visibility Between Objects

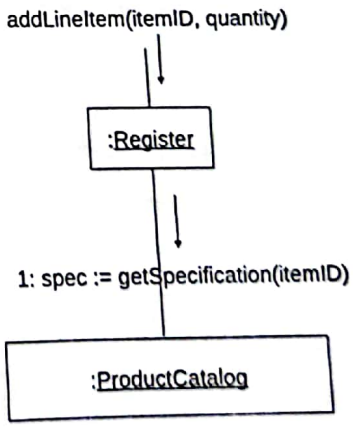


- The getSpecification message sent from a Register to a ProductCatalog, implies that the ProductCatalog instance is visible to the Register instance.

Visibility

- How do we determine whether one resource (such as an instance) is within the scope of another?
- Visibility can be achieved from object A to object B in four common ways:
  - 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 some way globally visible.

Visibility

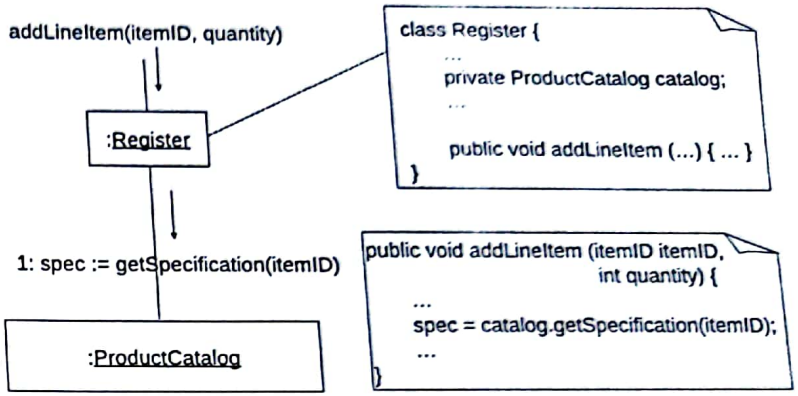


- The Regiser must have visibility to the ProductCatalog.
- A typical visibility solution is that a reference to the ProductCatalog instance is maintained as an attribute of the Register.

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.
- In the addLineItem collaboration diagram, Register needs to send message getSpecification message to a ProductCatalog. Thus, visibility from Register to ProductCatalog is required.

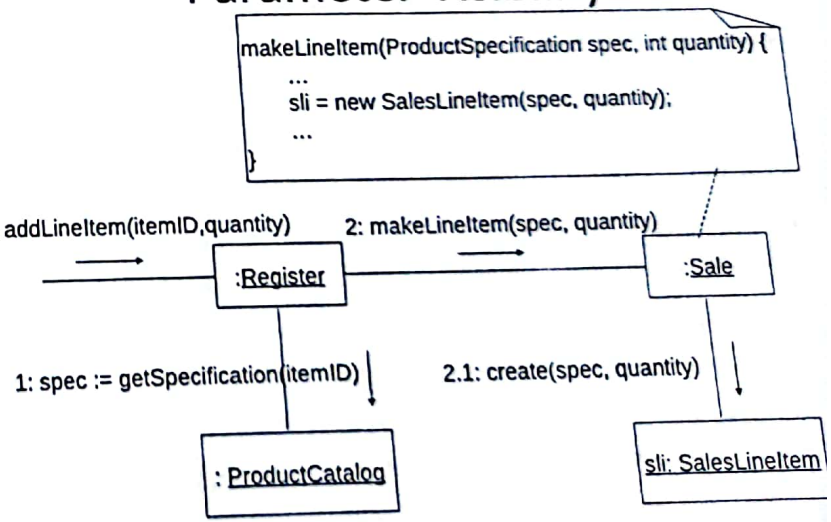
Attribute Visibility



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.
- When the makeLineItem message is sent to a Sale instance, a ProductSpecification instance is passed as a parameter.

Parameter Visibility



Parameter Visibility

```
// constructor
SalesLineItem(ProductSpecification spec,
              int quantity) {
    ...
    // parameter to attribute visibility
    productSpec = spec;
    ...
}
```

- When Sale crates a new SalesLineItem, it passes a ProductSpecification to its constructor.
- We can assign ProductSpecification to an attribute in the constructor, thus transforming parameter visibility to attribute visibility.

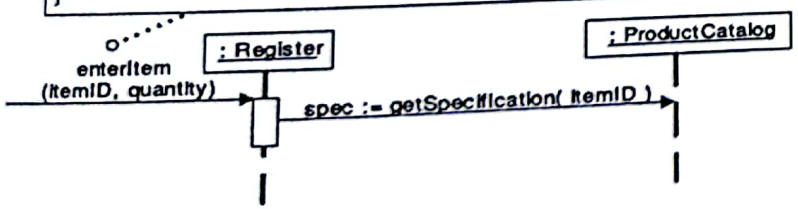
Local Visibility

- Locally declared visibility from A to B exists when B is declared as a local object within a method of A.
- It is relatively temporary visibility because it persists only within the scope of the method. Can be achieved as follows:
  1. Create a new local instance and assign it to a local variable.
  2. Assign the return object from a method invocation to a local variable.

```
enterItem(itemID, quantity) {
    ...
    ProductSpecification spec = catalog.getSpecification(itemID);
    ...
}
```


Local Visibility

```
{
enterItem(id, qty)
{
...
// local visibility via assignment of returning object
ProductSpecification spec = catalog.getSpecification(id);
...
}
}
```

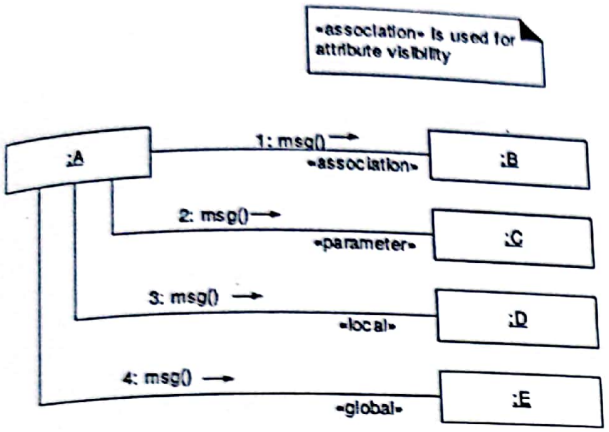


Global Visibility

- 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.
- One way to achieve this is to assign an instance to a global variable (possible in C++ but not in Java).
- Preferred method to achieve global visibility is to use the Singleton pattern.

Singleton pattern is a design solution where an application wants to have one and only one instance of any class.

UML: Implementation stereotypes for visibility



Design Model: Creating Design Class Diagrams

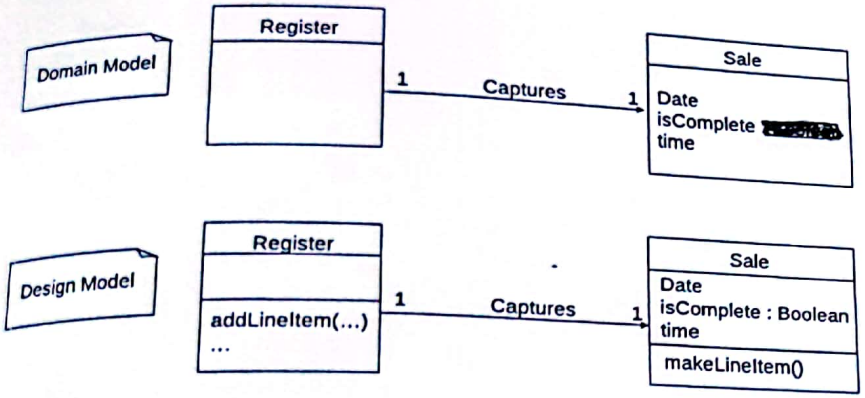
When to create DCDs

- Once the interaction diagrams have been completed it is possible to identify the specification for the software classes and interfaces.
- A class diagram differs from a Domain Model by showing software entities rather than real-world concepts.
- The UML has notation to define design details in static structure, or class diagrams.

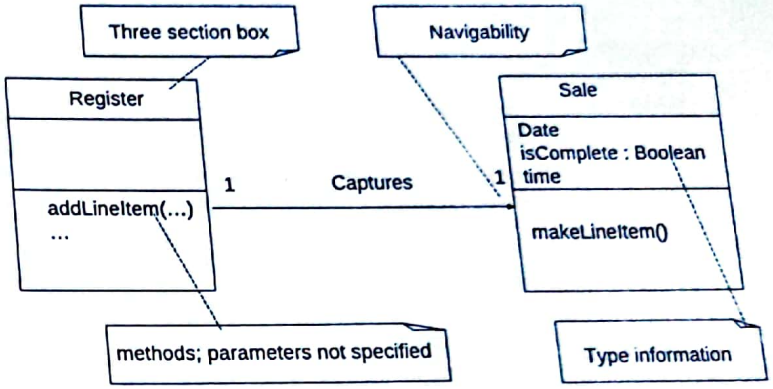
DCD and UP Terminology

- Typical information in a DCD includes:
  - Classes, associations and attributes
  - Interfaces (with operations and constants)
  - Methods
  - Attribute type information
  - Navigability
  - Dependencies
- The DCD depends upon the Domain Model and interaction diagrams.
- The UP defines a Design Model which includes interaction and class diagrams.

Domain Model vs. Design Model Classes

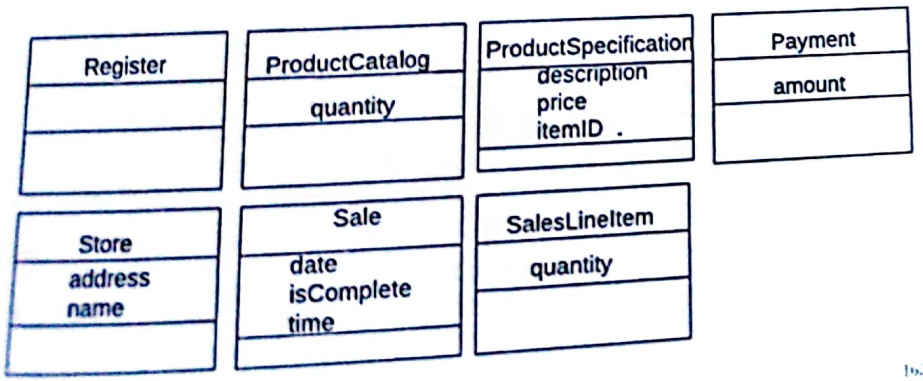


An Example DCD



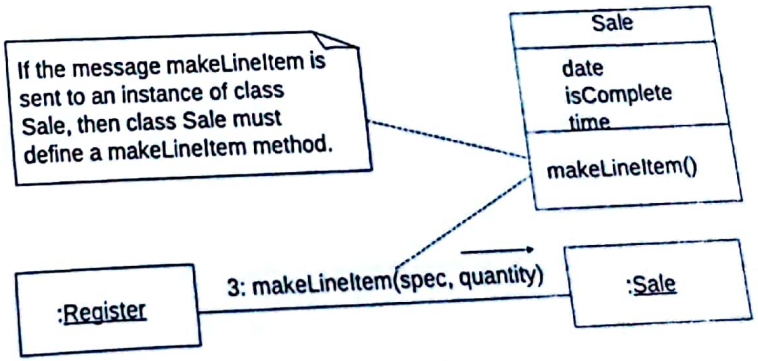
Creating a NextGen POS DCD

- Identify all the classes participating in the software solution. Do this by analyzing the interaction diagrams. Draw them in a class diagram.
- Duplicate the attributes from the associated concepts in the Domain Model.



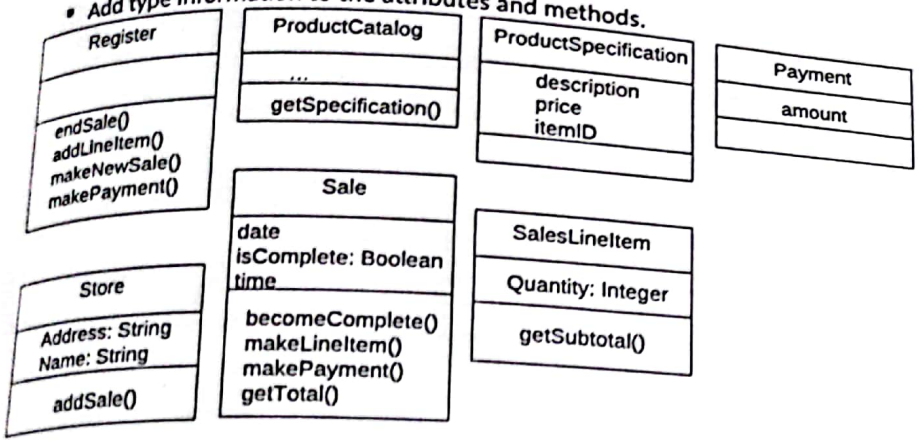
Creating a NextGen POS DCD

- Add method names by analyzing the interaction diagrams.
  - The methods for each class can be identified by analyzing the interaction diagrams.

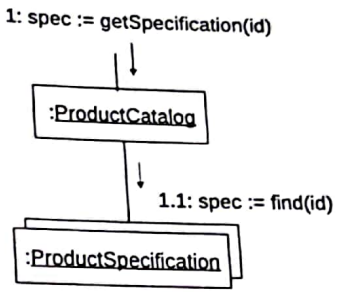


Creating a NextGen POS DCD

- Add type information to the attributes and methods.



Method Names -Multiobjects



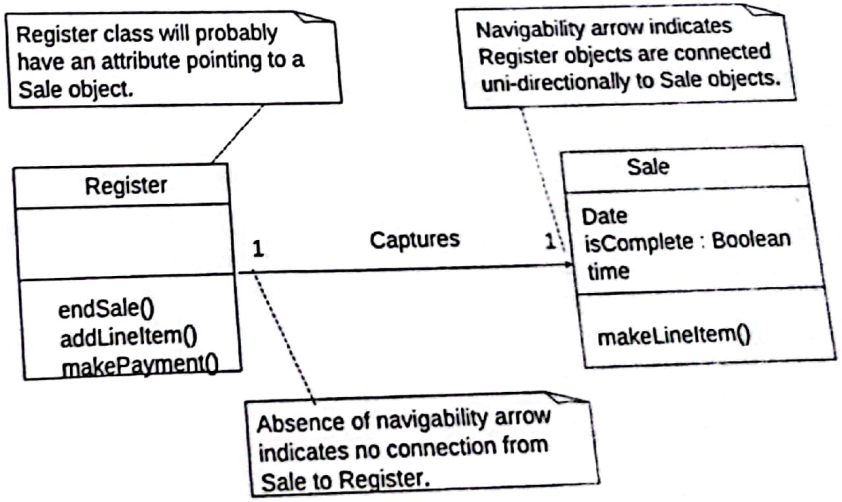
- The find message to the multiobject should be interpreted as a message to the container/ collection object(e.g. Java Map, smalltalk Dictionary)
- The find method is not part of the ProductSpecification class.
- Find method is part of the multiobject's interface, usually predefined library elements (like interface java.util.Map). Hence not shown in DCD

Associations, Navigability, and Dependency Relationships

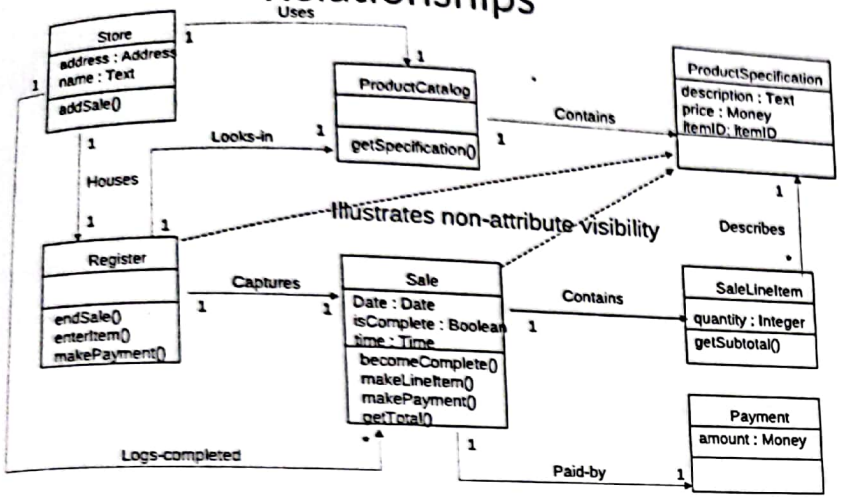
- Add the associations necessary to support the required attribute visibility.
  - Each end of an association is called a role.

  A role name explains how an object participates in the relationship
- Navigability is a property of the role implying visibility of the source to target class.
  - Attribute visibility is implied.
  - Add navigability arrows to the associations to indicate the direction of attribute visibility where applicable.
  - Common situations suggesting a need to define an association with navigability from A to B:
    - A sends a message to B.
    - A creates an instance of B.
    - A needs to maintain a connection to B
- Add dependency relationship lines to indicate non-attribute visibility (i.e. parameter, global and locally declared visibility).

Creating a NextGen POS DCD



Adding Navigability and Dependency Relationships



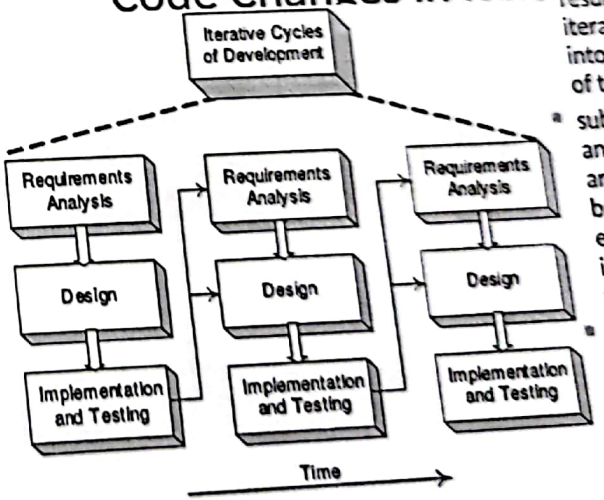
(From Book)
Refer figure 19.12 for UML class diagram member notation details

Implementation Model: Mapping Designs to Code

Mapping Designs to Code

- The interaction diagrams and DCDs— input to the code generation process.
- Implementation artifacts :
  - the source code,
  - database definitions,
  - JSP/XML/HTML pages,
  - and so on.
- The creation of code in an object-oriented programming language—such as Java is not part of OOA/D; it is an **end goal.**

Code Changes in Iterations

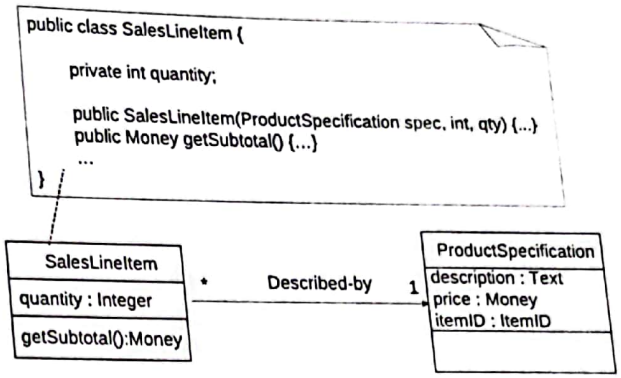


- results of a prior iteration can feed into the beginning of the next iteration
- subsequent analysis and design results are continually being refined and enhanced from prior implementation work
- Reverse engineering via CASE tools

Mapping design to code

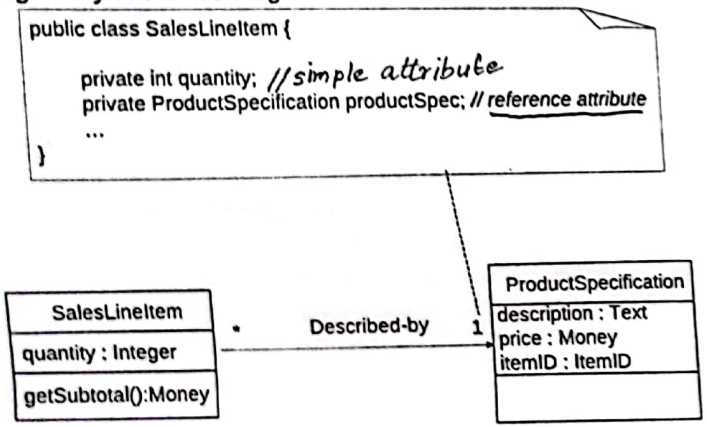
- Implementation in an object-oriented programming language requires writing source code for:
  - class and interface definitions
  - method definitions

Defining a Class with Methods and Simple Attributes



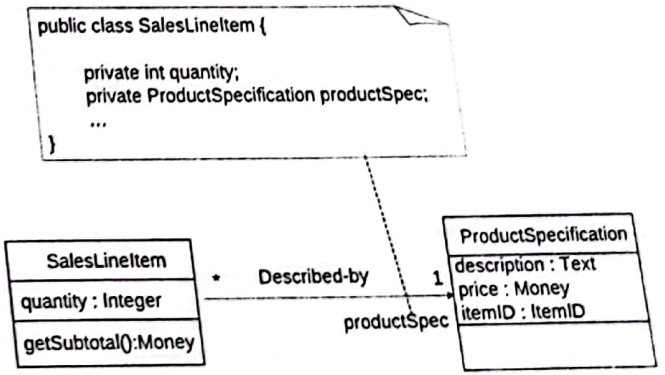
Adding Reference Attributes

Reference attributes: **implicitly** suggested by the **associations and navigability** in a class diagram.

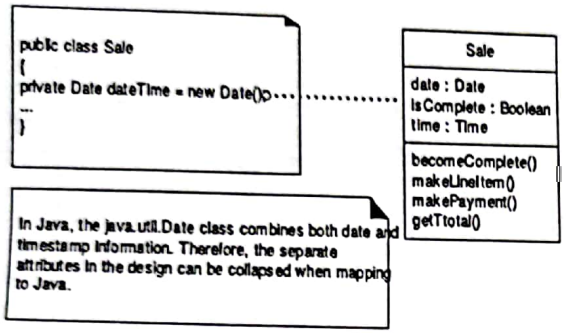


- Reference attribute is NOT primitive types like Number, string.
- It refers to complex Object.

Reference Attributes and Role Names

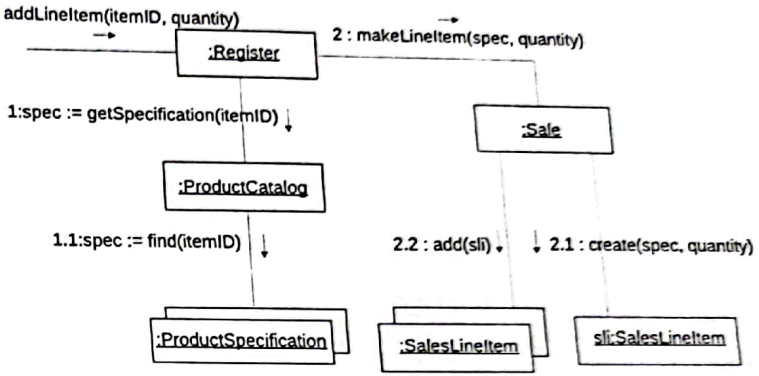


Mapping Attributes



- must consider the mapping of attributes from the design to the code in different languages.
- Example (above)Mapping date and time in Java

Creating Methods from Interaction Diagrams



The Register – addLineItem method

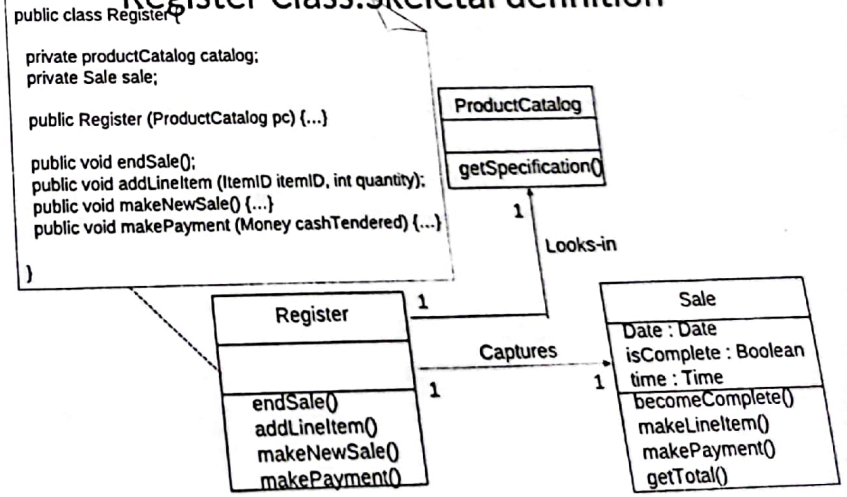
- The addLineItem collaboration diagram will be used to illustrate the Java definition of the addLineItem method.
- In which class does addLineItem belong to?
- The addLineItem message is sent to a Register instance, therefore the addLineItem method is defined in class Register.

  public void addLineItem(itemID itemID, int quantity);
- Message 1. A getSpecification message is sent to the productCatalog to retrieve a productSpecification

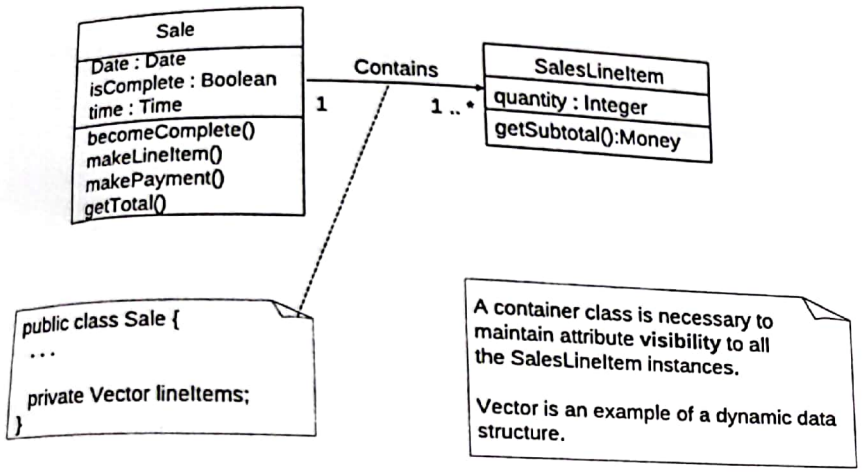
  productSpecification spec = catalog.getSpecification(itemID);
- Message 2: The makeLineItem message is sent to the Sale.

  sale.makeLineItem(spec, quantity);

Register Class:Skeletal definition

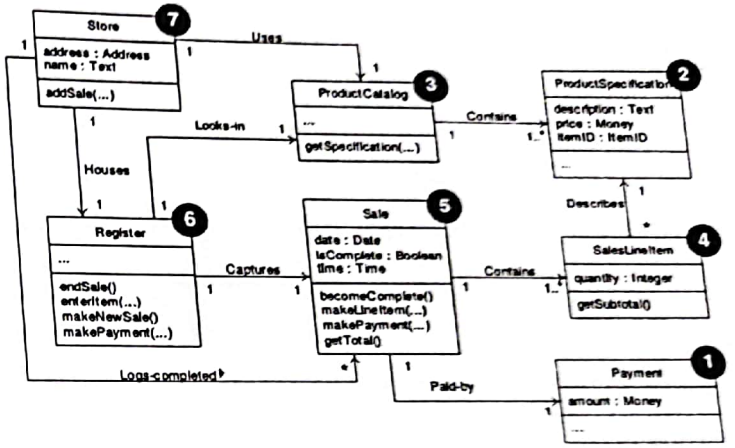


Container/Collection Classes



Order of Implementation & testing

- Least coupled to most coupled E.g. first implement *Payment*, Next dependent ones like: *SalesLineItem*.



Test First Programming

- the Extreme Programming (XP) method, applicable to the UP
- unit testing code is written before the code to be tested
- the developer writes unit testing code for all production code.
- Merits:
  - the unit tests actually get written
  - Programmer satisfaction
  - Clarification of interface and behaviour
  - Provable verification
  - Confidence to change things(as unit test code provides instant feedback)

Handling Failure

- Fault—the ultimate origin or cause of misbehavior. E.g. Programmer misspelled the name of a database.
- Error—a manifestation of the fault in the running system. Errors are detected (or not).

E.g. When calling the naming service to obtain a reference to the database (with the misspelled name), it signals an error.

- Failure—a denial of service caused by an error. E.g. The Products subsystem (and the NextGen POS) fails to provide a product information service.
- Throwing Exceptions

Pattern: **Name The Problem Not The Thrower**

Pattern: **Convert Exceptions** – avoid emitting lower level exceptions coming from lower subsystems or services. Rather, convert the lower level exception into one that is meaningful at the level of the subsystem. The higher level exception usually wraps the lower-level exception, and adds information, to make the exception more contextually meaningful to the higher level.

Pattern: **Centralized Error Logging/ Diagnostic Logger:** Use a Singleton-accessed central error logging object and report all exceptions to it.

Merits:

- Consistency in reporting.
- Flexible definition of output streams and format.

Pattern: **Error Dialog**-Use a standard Singleton-accessed, application-independent, non-UI object to notify users of errors

Some Problems with the Waterfall Lifecycle

- Design speculation and inflexibility.
- The waterfall lifecycle dictates that the architecture should be fully specified once the requirements are clarified and before implementation begins.
  - Since requirements usually change, the original design will not be reliable.
  - Lack of feedback on design until long after design decisions are made.