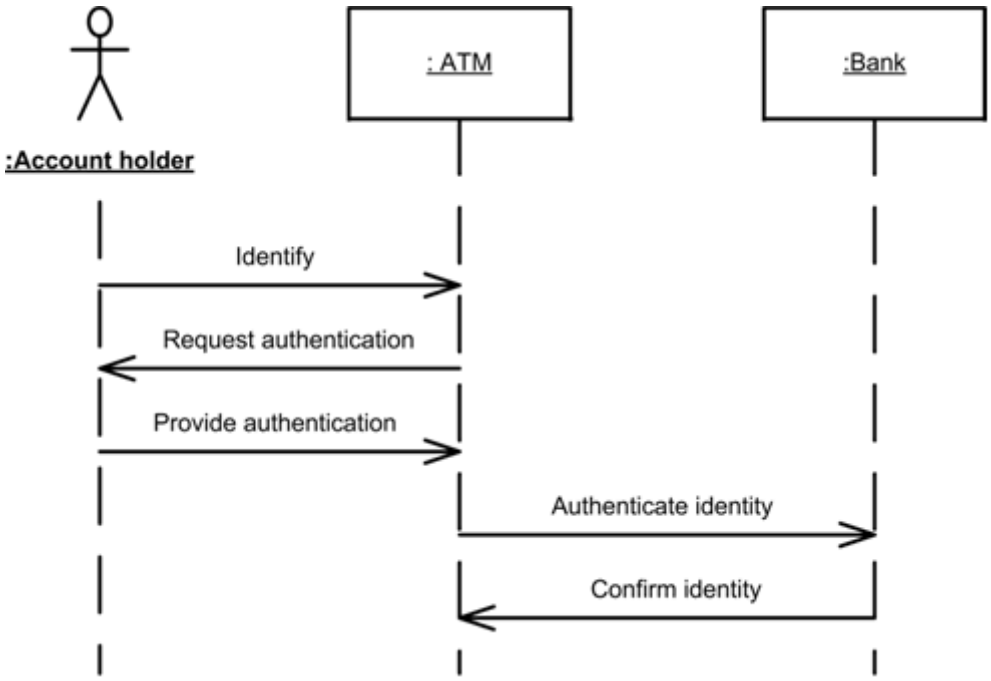
## UNIT IV
## PART A

**1. Define SSD. Mention its use.**

A system sequence diagram (SSD) is a picture that shows, for a particular scenario of use case, the events that external actors generate their order and inter system events. All systems are treated as a black box. The emphasis of the diagram is the events that cross the system boundary from actors to systems.




**Example for a System sequence diagram**

**2.Define System events.**

The system sequence diagrams shows system Events or I/O messages relative to the system. Input system events imply the system has standalone system operations to handle the events, just as an Object Oriented message (a kind of event or signal) is handled by an Object oriented method (a kind of operation).

**3.Define System Behaviour.**

System Behaviour is a description of what a system does, without explaining how it does it. One part of the description is the system sequence diagram. Other parts include the use case and system operation contracts.

**4.How to name system events and operations?**

System events (and their associated system operations) should be expressed at the level of intent (abstract level) rather than in terms of the physical input medium or interface widget level. It also improves clarity to start the name of a system event with a verb, since it emphasizes the command orientation of these events.

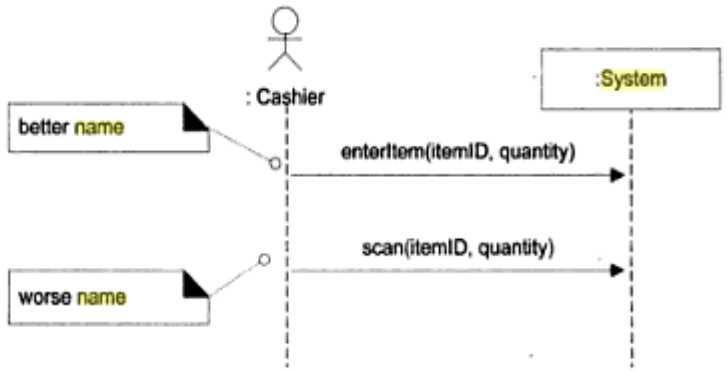



Figure 10.4 Choose event and operation names at an abstract level.

**5.List out the frame operators in sequence diagram.**

The common frame operators:

- Alt : alternate fragment for mutual exclusion
- Loop : loop fragment while guard is true
- Opt : Optional fragment that executes if guard is true
- Par: parallel fragments that execute in parallel.
- Region: critical region within which only one thread can run.

**6.Give the strength and weakness of sequence and collaboration diagram.**

| Type | Strengths | Weaknesses |
|---|---|---|
| Sequence | Clearly shows sequence or time ordering of messages<br>Large set of detailed notation option | Forced to extend to the right when adding new objects.<br>Consumes horizontal space |
| | Space economical flexibility to add new objects in two dimensions | More difficult to see sequence of messages<br>Fewer notation options |

**7.Define system operation.**

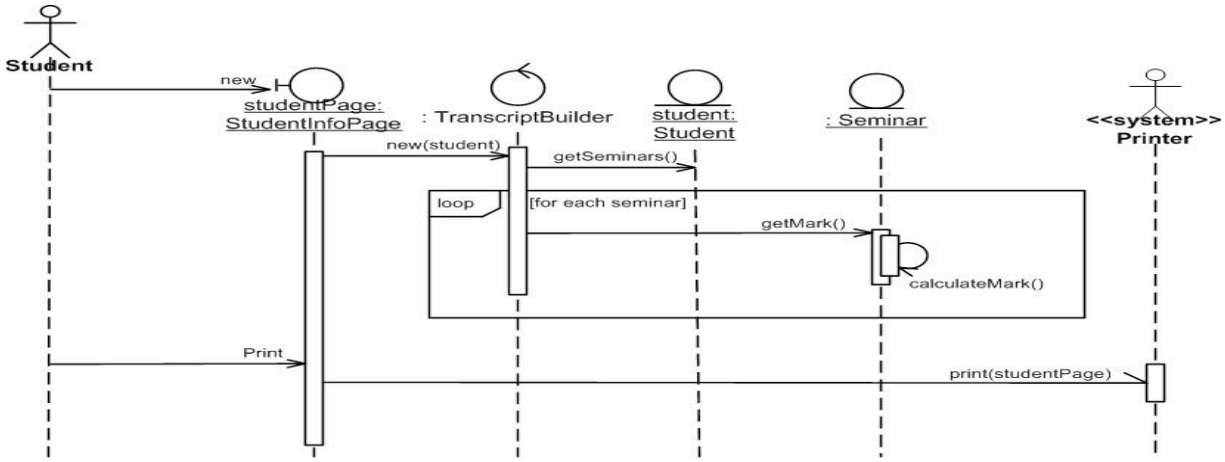
Operation that the system as a black box component offers in its public interface is called system operation. System operations can be identified while sketching System sequence diagrams. The system sequence diagram show system events or I/O messages relative to the system.

**8.How to create instance in collaboration diagram.**

A message 'create' can be used to create an instance in a collaboration diagram. If another name is used, the message may be annotated with a UML stereotype, like <<create>>. The create message may include parameters indicating the passing of initial messages.

**9.What do you mean by synchronous and asynchronous call?**

An asynchronous message call does not wait for a response. They are used in multi threaded environments such as .Net and Java so that the new thread executions can be created and initiated. When a task is being executed asynchronously, there is no need to wait for it to finish, before starting with another task. In Synchronous message calls, the task has to be completed before starting another task.




**Example for Asynchronous and synchronous call**

**10.Define classifier.**

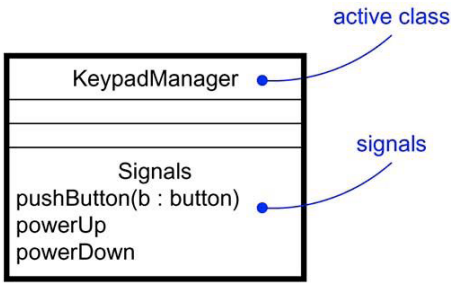
A UML classifier is a "model element that describes behavioural and structure features". Classifiers can also be specialized. They are a generalization of many of the elements of the UML, including classes, interfaces, use cases and actors. In class diagram, the two most common classifiers are regular classes and interfaces.

**11.How to show methods in class diagram?**

A UML method is the implementation of an operation. If constraints are defined, the method must satisfy them. A method may be illustrated in class diagrams with a UML note symbol stereotype with <<method>>.

**12.Define Active class.**

An active object runs on and controls its own thread of execution. Active classes are just Classes which represents an independent flow of control. Active class share the same properties as all the other classes. When an active object is created, the associated flow of control is started. When the object is destroyed the associated flow of control is terminated.




**13.Justify why class diagram is called static object modelling.**

The UML class diagram does not have any dynamic elements and all the representations are static. The classes and the methods in the classes do not change with respect to any external criteria. The characteristics and the methods of a class are standard and constant and cannot be changed. Therefore, the class diagram is called as static object modelling.

**14.List the relationships used in class diagram.**

The various relationships used in class diagrams are:

- Association with multiplicities.
- Interface implementation
- Inheritance
- Dependency
- Composition over Aggregation

- Qualified Association
- Qualified association
- Association Class

15.**Define singleton class with an example.**

When exactly one instance of a class is allowed, it is called a "singleton" class. In UML diagram, such a class can be marked with a "I" in the upper right corner of the name component.

**16.What is Logical Architecture?**

The logical architecture is the large scale organization of the software classes into packages, namespaces, subsystems and layers. It's called the logical architecture because there's no decision about how these elements are deployed across different operating system processes or across physical computers in a network. An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

**17.What are the types in layered architecture? List the layers in OO system.**

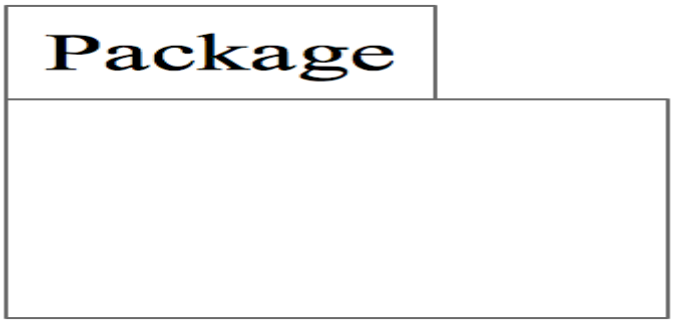
- Strict layered architecture
- Relaxed layered architecture
- Layers in OO architecture:
- User Interface
- Application Logic and Domain Objects
- Technical Services

**18.Define package and draw the UML notation for package.**

A UML package diagram provides a way to group elements. It can group anything: classes, other packages, use cases. UML package diagrams are often used to illustrate the logical architecture of a system -the layers, the subsystems, packages.

The package name is placed on the tab if the package shows the inner members or on the main folder if not. Dependency or coupling is shown by the UML – dependency line – a dashed line with arrow pointing towards depended on package. Fully qualified names are represented in UML for example as java :: util:: date

**UML notation for package**




**19.Give the benefits of using layers.**

Relaxed complexity is encapsulated and decomposable.

Lower layers contain reusable functions

Some layers (primarily the domain and technical services) can be distributed.

Development by team is aided because of the logical segmentation

Some layers can be replaced with new implementations.

**20.Relationship between domain layer and domain model.**

The domain layer is part of the software and the domain model is part of the conceptual perspective analysis. By creating a domain layer with inspiration from the domain model, a lower representation gap between the real world domain and the software design is achieved.

**21.Define tiers, layers and partition.**

Tier in architecture is a logical layer, not a physical node. The layers of architecture are said to represent the vertical slices, while partitions represent a horizontal division of relatively parallel subsystems of a layer.

**Ex: The technical services layer may be divided into partitions such as security and reporting.**

**22.Define model view separation principle.**

The model view separation principle states that model objects should not have direct knowledge of view objects, at least as view objects. Ex: a register or sale object should not directly send a message to a GUI window object process Sale Frame, asking it to display something.

**23.What are the various GoF design patterns?**

- Adapter
- Factory
- Singleton
- Strategy
- Composite
- Façade
- Observer

**24.What is the use of GoF design pattern?**

The GoF design patterns help to resolve and simplify the key software development processes such as,

- Flexibility
- Extensibility
- Dependability
- Predictability
- Scalability
- Efficiency

**25.What are the interactive diagrams? List out the components involved in interaction diagrams. What is the use of interaction diagrams?**

The UML interaction diagrams are used to illustrate how objects interact via messages. They are used for dynamic object modelling. There are two common types: sequence and communication interaction diagrams

The components that are involved in sequence diagrams are:

- Lifeline boxes
- A found message
- Synchronous message
- Asynchronous message
- Execution specification bar
- Frame with guard expression

The components that are involved in communication interaction diagrams are:

- Objects
- Links
- Messages
- Sequence numbers
- Conditional messages
- Looping or iteration messages

**PART B**

1. **Explain interaction diagrams with suitable examples. (AU APR/MAY 2011, NOV/DEC 2011 and NOV/DEC 2012)**

**Sequence and Collaboration Diagrams:**

The term *interaction diagram,* is a generalization of two more specialized UML diagram types; both can be used to express similar message interactions:

collaboration diagrams

sequence diagrams

both types will be used, to emphasize the flexibility in choice.

**Collaboration diagrams** illustrate object interactions in a graph or network format, in which objects can be placed anywhere on the diagram.

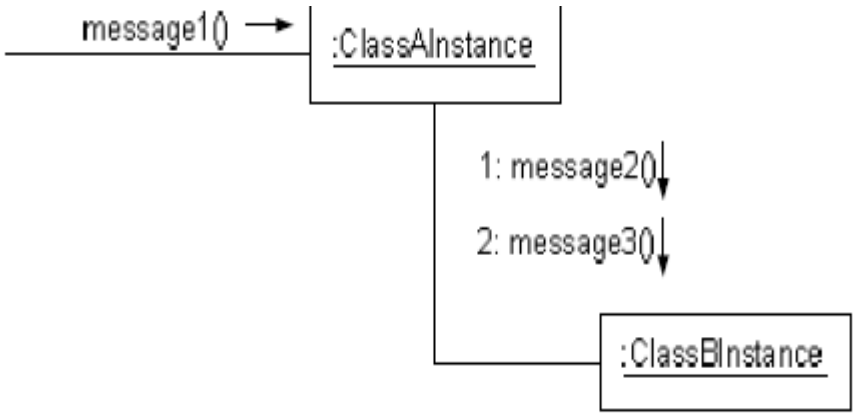



Figure 15.1 Collaboration diagram

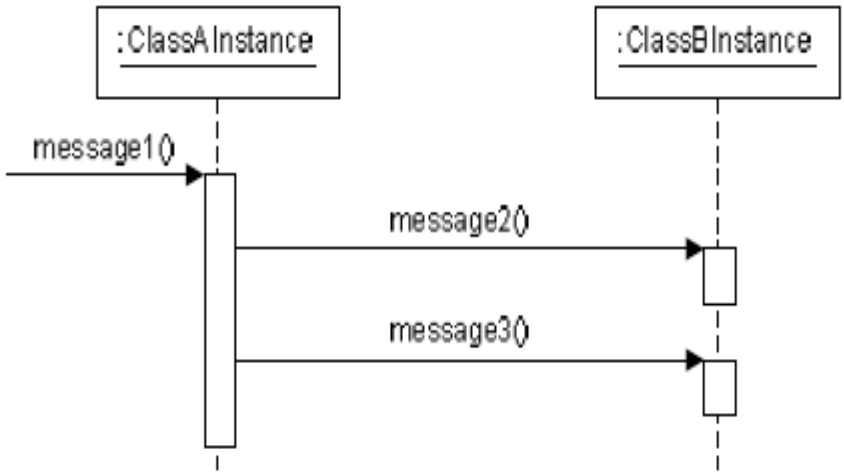



Figure 15.2 Sequence diagram.

Example Collaboration Diagram: makePayment

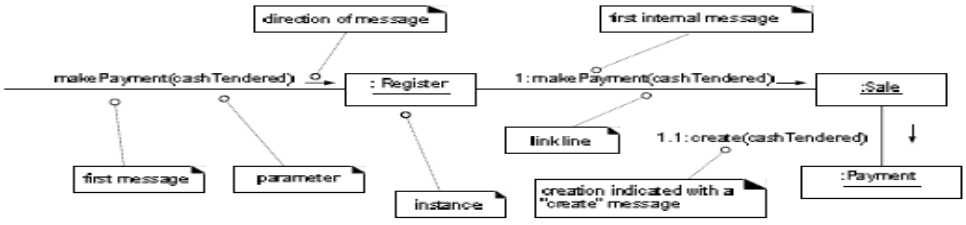



Figure 15.3 Collaboration diagram.

The collaboration diagram shown in Figure 15.3 is read as follows:

1. The message *makePayment is* sent to an instance of a *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 44

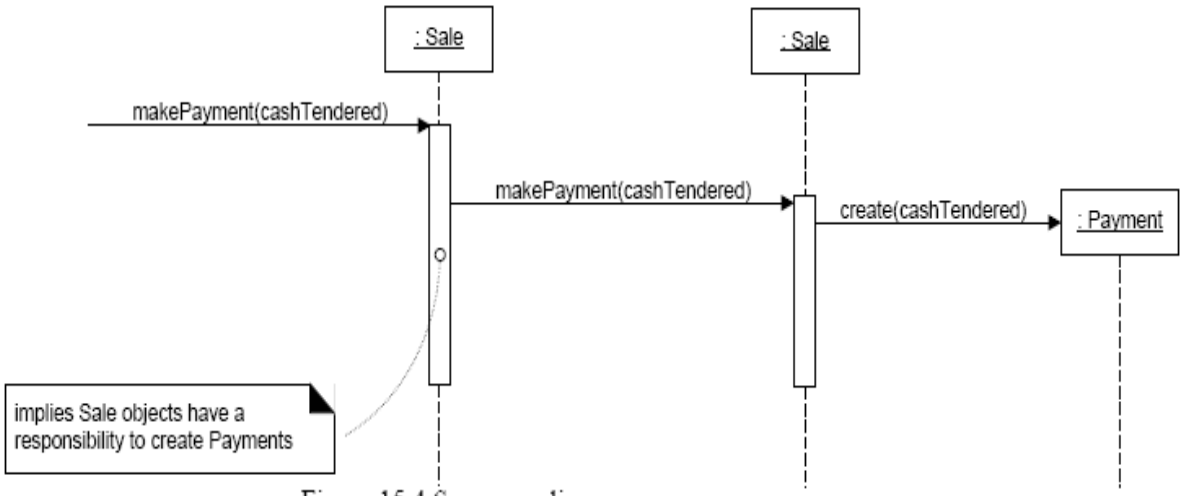



Figure 15.4 Sequence diagram.

The sequence diagram shown in Figure 15.4 has the same intent as the prior collaboration diagram.

2. **Compare sequence vs. collaboration diagram with suitable examples. (AU MAY/JUNE 2012)**
   **Refer Ans 1**

3. **Illustrate with an example, the relationship between sequence and use case diagrams.(AU APR/MAY 2011)**
   An SSD shows system events for a scenario of a use case, therefore it is generated from inspection of a use case.
   Use cases describe how external actors interact with the software system we are interested in creating. During this interaction an actor generates events to a system, usually requesting some operation in response. For example, when a cashier enters an item's ID, the cashier is requesting the POS system to record that item's sale. That request event initiates an operation upon the system. It is desirable to isolate and illustrate the operations that an external actor requests of a system, because they are an important part of understanding system behavior. The UML includes **sequence diagrams** as a notation that can illustrate actor interactions and the operations initiated by them.
   **A system sequence diagram** (SSD) is a picture that shows, for a particular scenario of a use case, the events that external actors generate, their order, and inter-system events. All systems are treated as a black box; the emphasis of the diagram is events that cross the system boundary from actors to systems.
   **System Events and the System Boundary**
   To identify system events, it is necessary to be clear on the choice of system boundary. For the purposes of software development, the system boundary is usually chosen to be the software (and possibly hardware) system itself; in this context, a system event is an external event that directly stimulates the software .

Consider the *Process Sale* use case to identify system events. First, we must determine the actors that directly interact with the software system. The customer interacts with the cashier, but for this simple cash-only scenario, does not directly interact with the POS system—only the cashier does. Therefore, the customer is not a generator of system events; only the cashier is.

**Naming System Events and Operations**

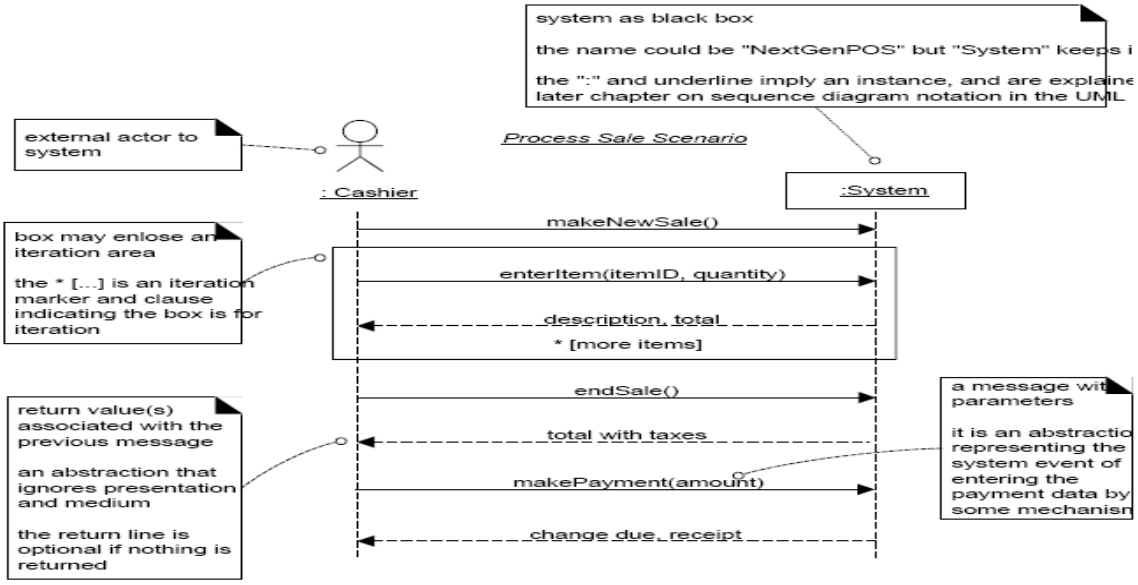
System events (and their associated system operations) should be expressed at the level of intent rather than in terms of the physical input medium or interface widget level. It also improves clarity to start the name of a system event with a verb (add..., enter..., end..., make...), since it emphasizes the command orientation of these events.Thus "enteritem" is better than "scan" (that is, laser scan) because it captures the intent of the operation while remaining abstract and noncommittal with respect to design choices about what interface is used to capture the system event.

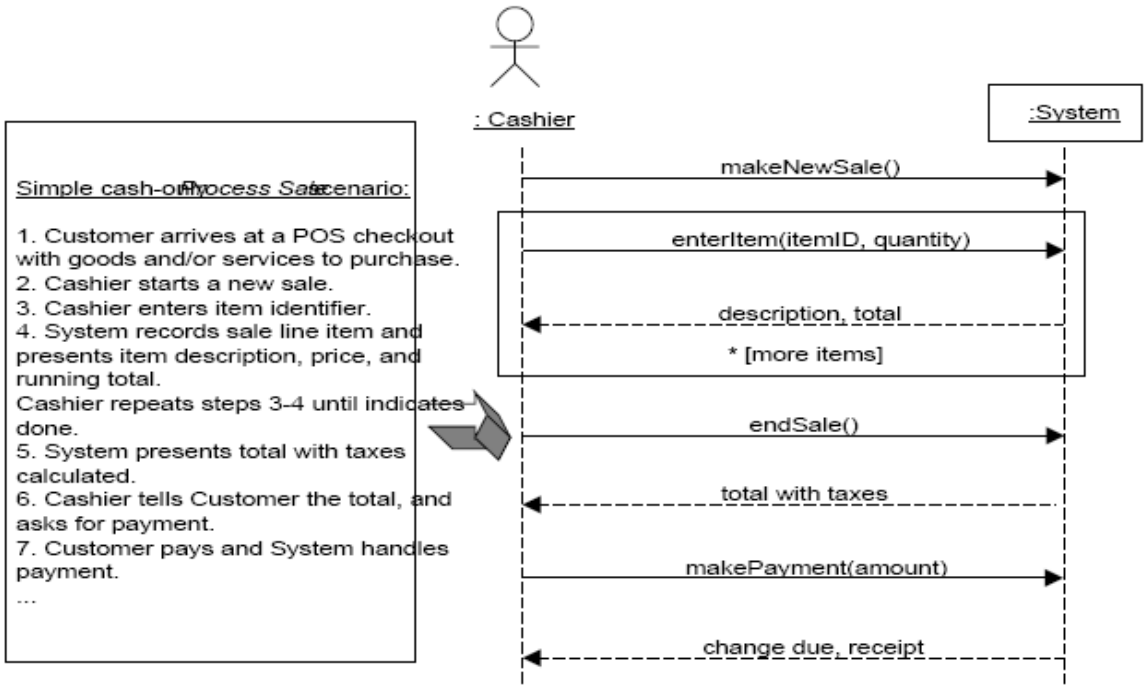
**Showing Use Case Text**

It is sometimes desirable to show at least fragments of use case text for the scenario,to clarify or enhance the two views .The text provides the details and context; the diagram visually summarizes the interaction.

**SSDs and the Glossary**

The terms shown in SSDs (operations, parameters, return data) are terse. These may need proper explanation so that during design work it is clear what is coming in and going out. If this was not explicated in the use cases, the Glossary could be used. However, as always when discussing the creation of artifacts other than code (the heart of the project), be suspicious. There should be some truly meaningful use or decision made with the Glossary data, otherwise it is simply low-value unnecessary work.



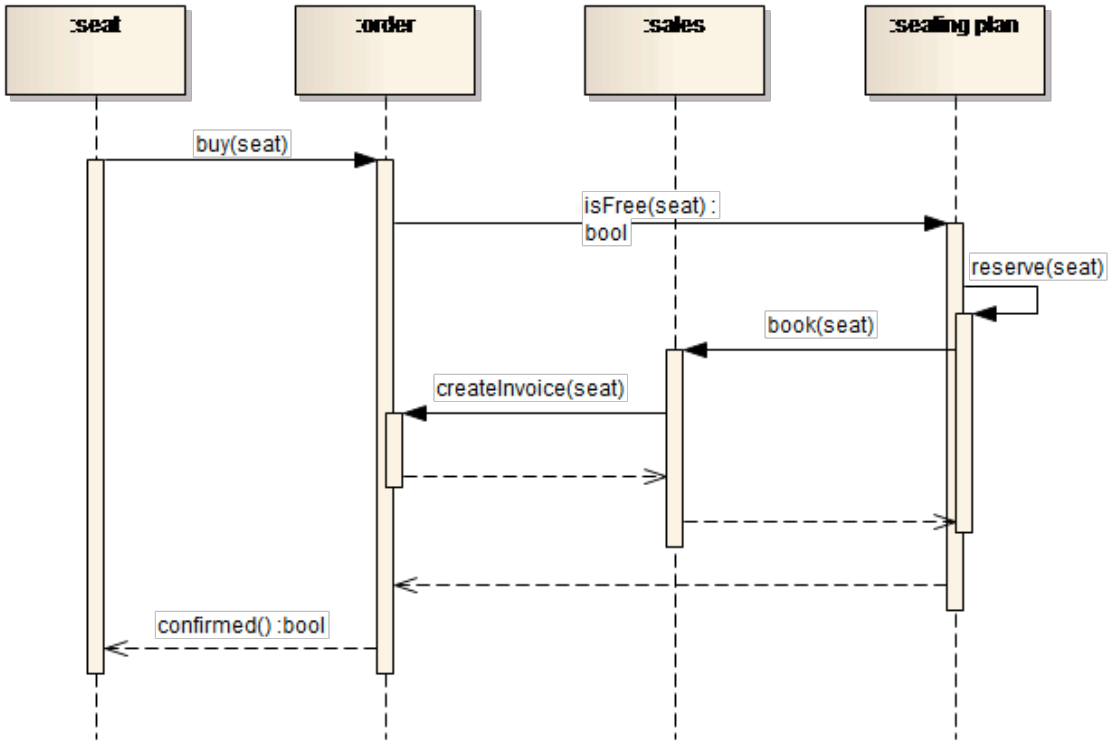



**4. Explain in detail about UML sequence diagrams.(AU MAY/JUNE 2013)**

## System Sequence Diagram 1

‰ System sequence diagram a picture that shows, for one particular scenario of a use case,the events that external actors generate, their order, and inter-system events.

All systems are treated as a black box; the emphasis of the diagram is events that cross the system boundary from actors to systems.




**POS Operation Contract**

**‰ Contract CO2**: enterItem

**Operation:** enterItem(itemID: ItemID, quantity: integer)

**Cross References:** Use Cases: Process Sale

**Preconditions:** There is a sale underway.

**Postconditions:**

- ŠA SalesLineItem instance sli was created (instance creation).
- Šsli was associated with the current Sale (association formed).
- Šsli.quantity became quantity (attribute modification).
- Šsli was associated with a ProductDescription, based on itemID match (association formed).

**Sections of a Contract**

**‰ Operation:** Name of operation, and parameters

‰ **Cross References:** Use cases this operation can occur within

‰ **Preconditions:** Noteworthy assumptions about the state of the system or objects in the Domain Model before execution of the operation.

**‰ Postconditions:** The state of objects in the Domain Model after completion of the operation.

**System Operations 1**

‰ **System operations**

- The system as a black box component offers in its public interface.
- can be identified while sketching SSDs
- SSDs show system events or I/O messages relative to the system.

Postconditions 1

**‰ Postconditions**

- Describe changes in the state of objects in the domain model.
- Domain model state changes include instances created, associations formed or broken, and attributes changed.
- Postconditions are not actions to be performed.

**5. (i) Explain in detail about tiers, layers and partition (8)**

(ii) Explain in detail about model view separation partition (8)

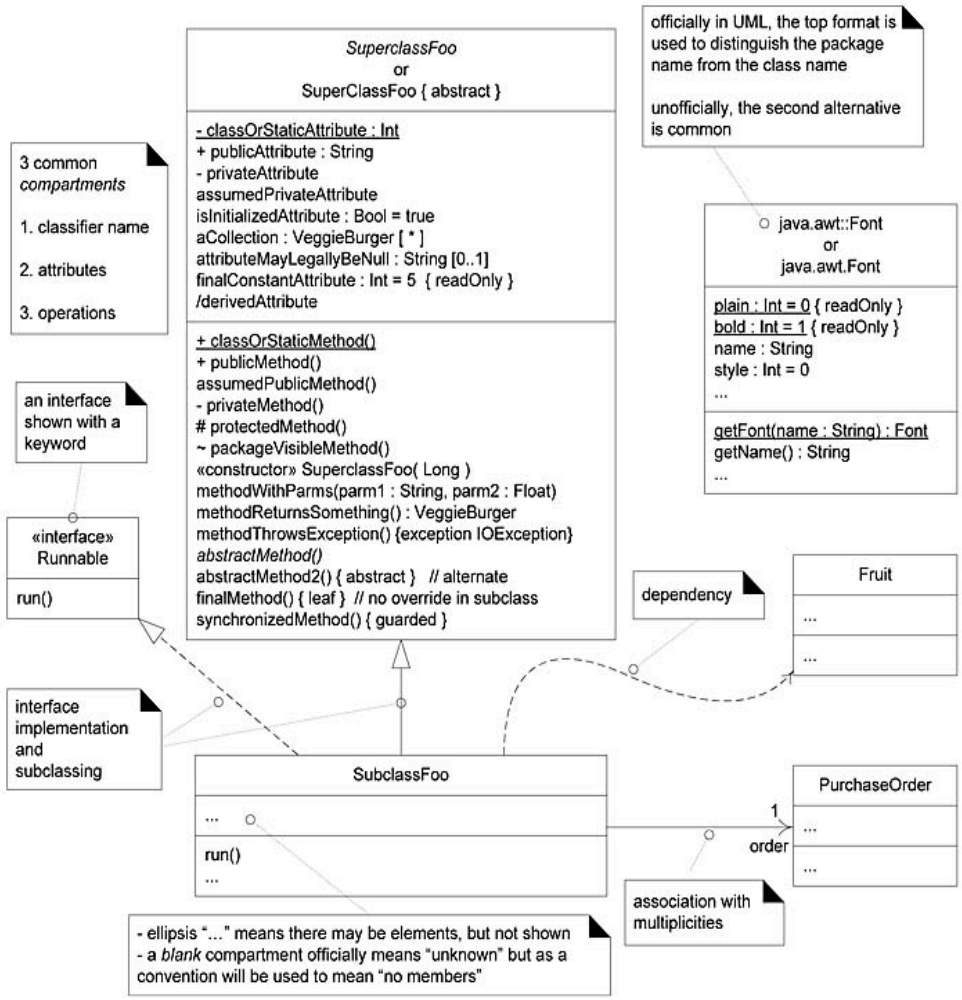
Refer Portal Class notes:

6. **(i) Describe the UML notation for class diagram with an example.(8)**

**(ii) Explain the concept of link, association and inheritance (8). (AU MAY/JUNE 2012)**

The UML includes class diagrams to illustrate classes, interfaces and their associations. They are used for static object modeling.

***UML Notation for Class diagram***




## *COMMON UML CLASS DIAGRAM NOTATION*

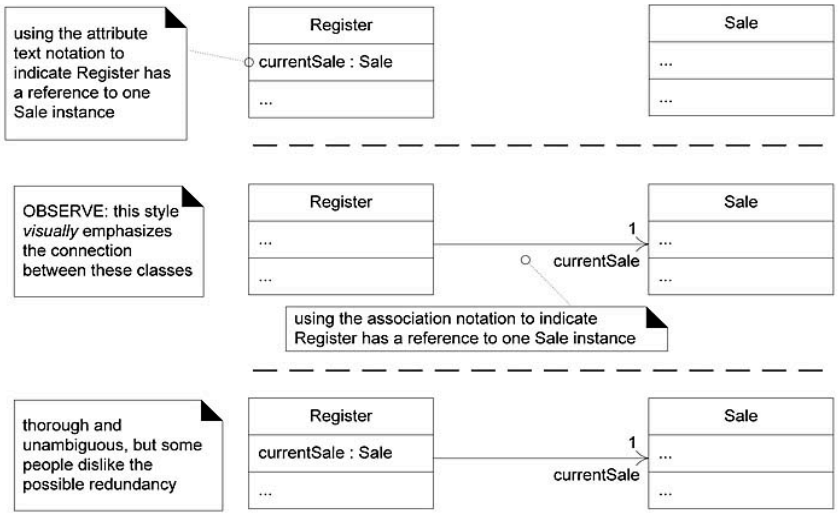
Most elements are optional (e.g.: +/- visibility, parameters, compartments) modelers draw, show or hide them depending on context and the needs of the reader or UML tool.

***Concept of Link, Association and Inheritance:***

*Attribute-as-Association:*

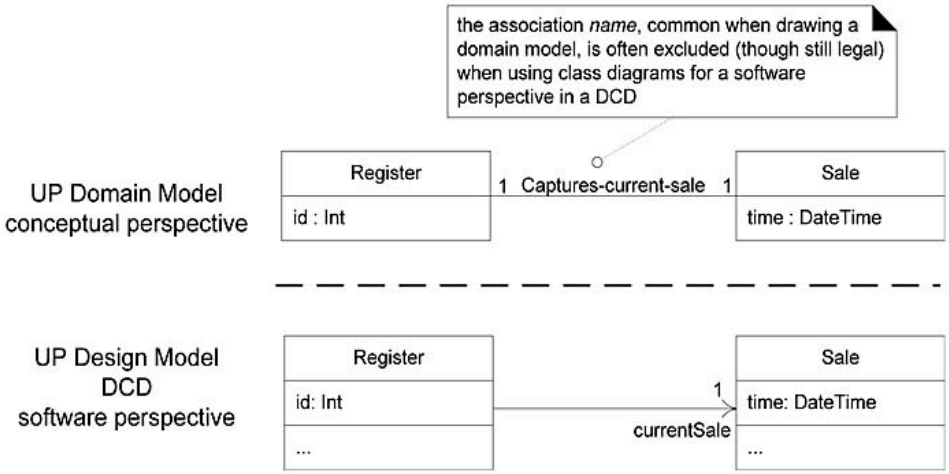
Attribute-as-Association line has the following style:

- ♣ A navigational arrow pointing from source (Register) to target (Sale), indicating a Register object has an attribute of one Sale.
- ♣ A multiplicity at the end, but not at the source.
- ♣ No association name.




***ATTRIBUTE TEXT VERSUS ASSOCIATION LINE NOTATION FOR A UML ATTRIBUTE***

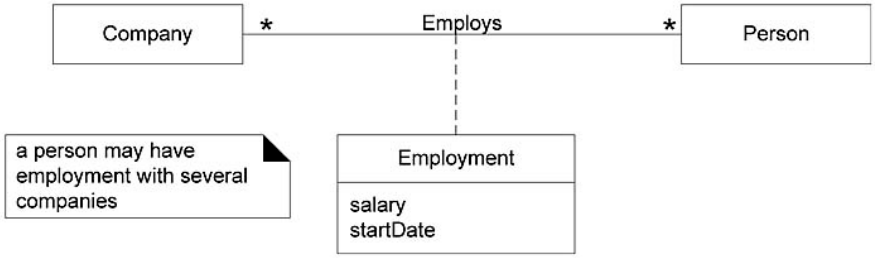
When using class diagrams for a domain model show the association names but avoid navigation arrows, as domain model is not perspective.




***IDIOMS IN ASSOCIATION NOTATION USAGE IN DIFFERENT PERSPECTIVE***

***Association Class:***

The association class allows treating association as a class, and modeling it with attributes and operations.




***ASSOCIATION CLASSES IN THE UML***

***Inheritance (Generalization):***

Generalization in the UML is shown with a solid line with a fat triangular arrow from the subclass to superclass. A taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the

general classifier. Thus the specific classifier indirectly has features of the more generic classifier.

7. **With a suitable example explain how to design a class. Give all the possible representation in a) class (such as name, attribute, visibility, methods, and responsibilities) (AU NOV/DEC 2011, NOV/DEC 2012)**

When it comes to system construction, a **class diagram** is the most widely used diagram. This diagram generally consists of **interfaces**, **classes**, **associations and collaborations**. Such a diagram would illustrate the object-oriented view of a system, which is static in nature. The object orientation of a system is indicated by a class diagram.

Since class diagrams are used for many different purposes, such as making stakeholders aware of requirements to highlighting your detailed design, you need to apply a different style in each circumstance.

The points that are going to be covered are indicated as follows:

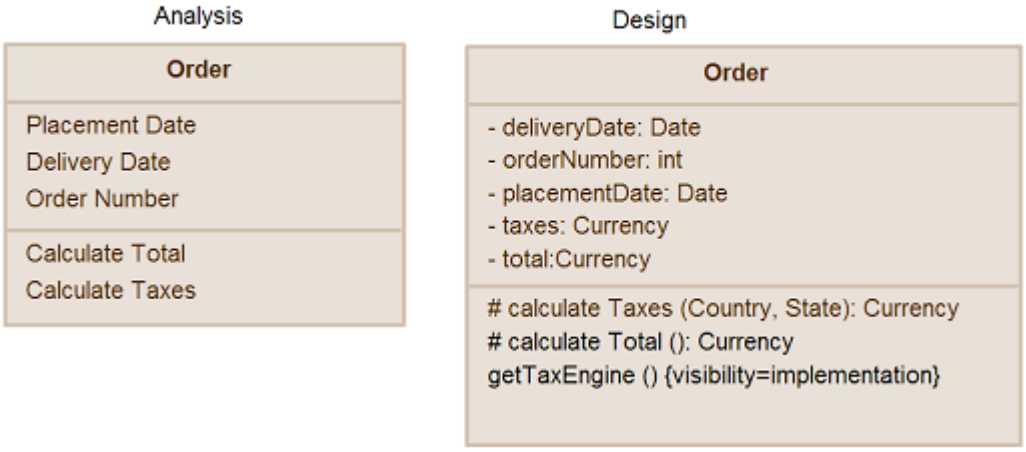
1. General issues
2. Classes
3. Interfaces
4. Relationships
5. Inheritance
6. Aggregation and Composition

***General Issues***

This section describes style guidelines that are relevant to various types of class diagrams.

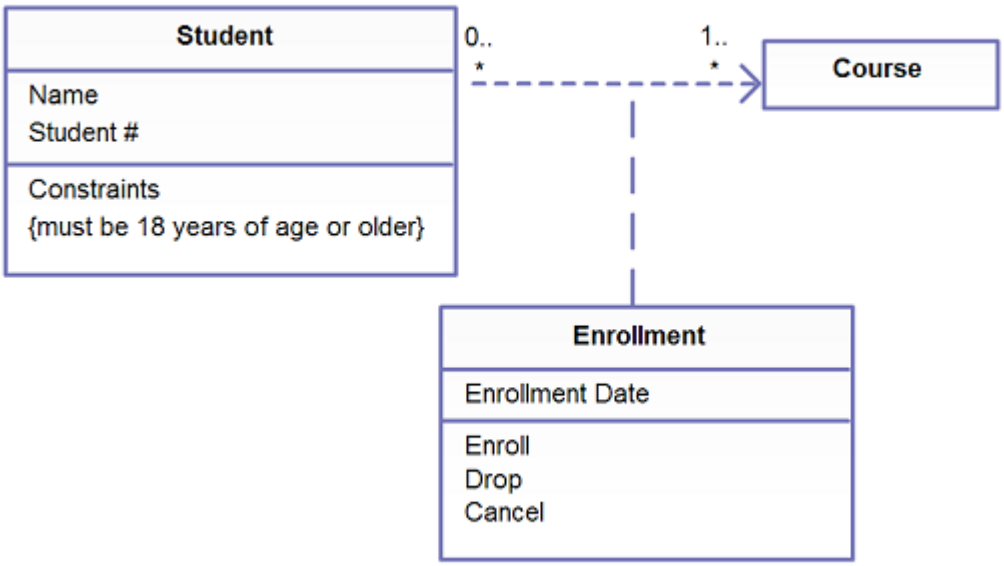
1. Show visibility only on design models
2. Assess responsibilities on domain class diagrams
3. Highlight language-dependent visibility with property strings
4. Highlight types only on design models
5. Highlight types on analysis models only when the type is an actual requirement

A - Analysis and design versions of a class




- - Design class diagrams should reflect language naming conventions. In the "Analysis and design version of a class" image you see that the design version of the Order class uses names that conform to common Java programming conventions such as placementDateand calculateTaxes().

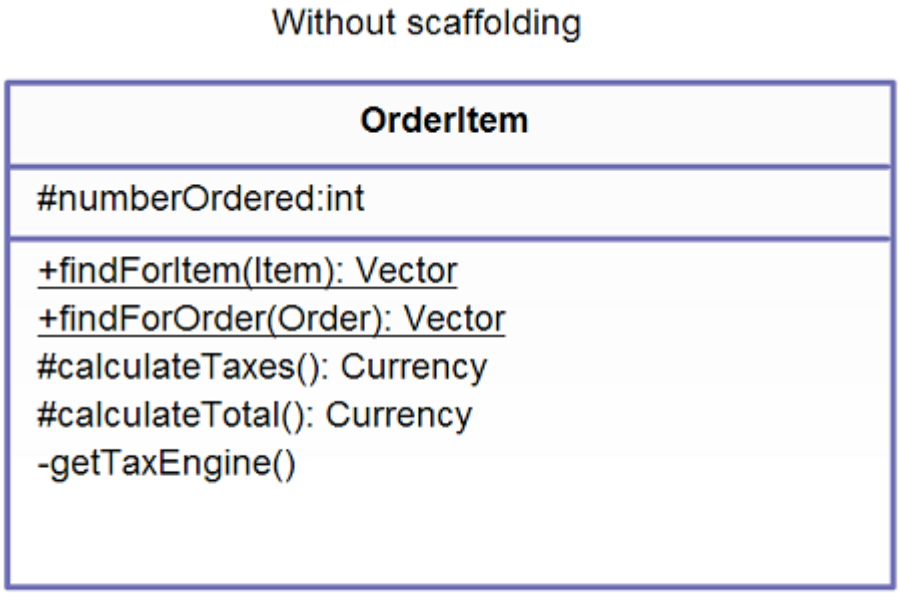
B - Modeling association classes




- - Model association classes on analysis diagrams. The image that shows the "Modelling association classes" indicates the association classes that are depicted as class attached via a dashed line to an association – the association line, the class, and the dashed line are considered one symbol in the UML.
- - Do not name associations that have association classes.
- - Center the dashed line of an association class.

***Class Style***

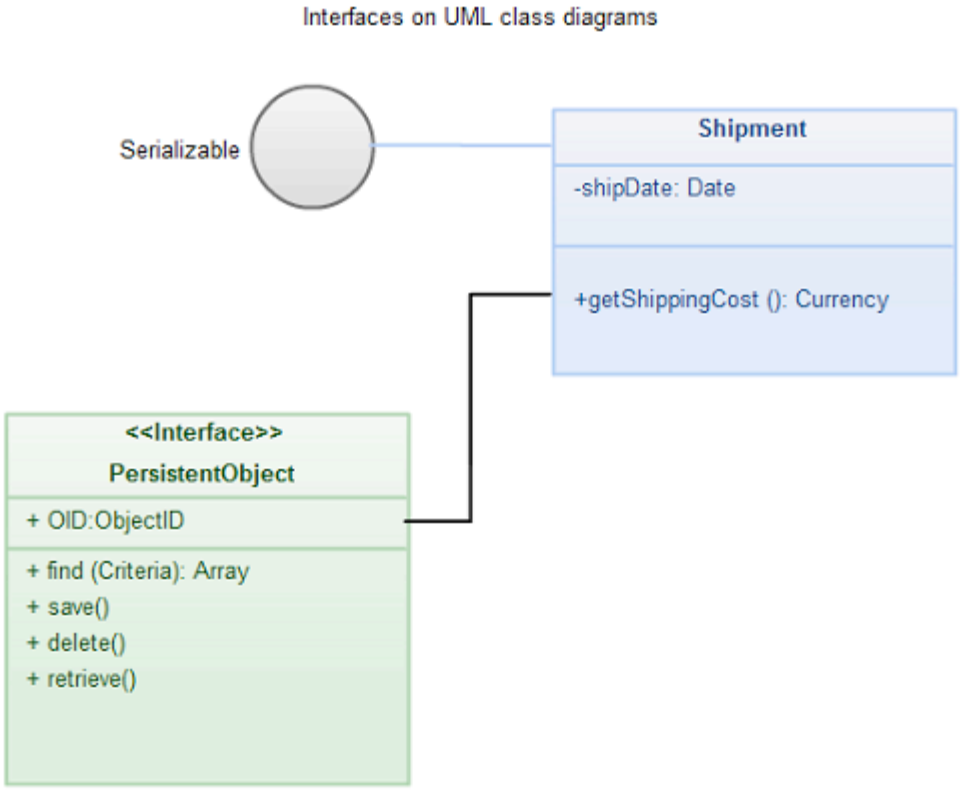
A class is basically a template from which objects are created. Classes define attributes, information that are relevant to their instances, operations, and functionality that the objects support. Some of the more important guidelines pertinent to classes are listed down below.




1. Put common terminology for names
2. Choose complete singular nouns over class names
3. Name operations with a strong verb
4. Name attributes with a domain-based noun
5. Do not model scaffolding code. Scaffolding code refers to the attributes and operations required to use basic functionality within your classes, such as the code required to implement relationships with other classes. The image above depicts the difference between the OrderItem class without scaffolding code
6. Never show classes with just two compartments
7. Label uncommon class compartments
8. Include an ellipsis ( … ) at the end of incomplete lists
9. List static operations/attributes before instance operations/attributes
10. List operations/attributes in decreasing visibility
11. For parameters that are objects, only list their type
12. Develop consistent method signatures
13. Avoid stereotypes implied by language naming conventions
14. Indicate exceptions in an operation's property string. Exceptions can be indicated with a UML property string, an example of which is shown in the above image.

***Interfaces***

An interface can be defined as collection of operation signature and/or attribute definitions that ideally defines a cohesive set of behaviors. Interfaces are implemented, "realized" in UML parlance, by classes and components. In order to realize an interface, a class or component should use the operations and attributes that are defined by the interface. Any given class or component may use zero or more interfaces and one or more classes or components can use the same interface.

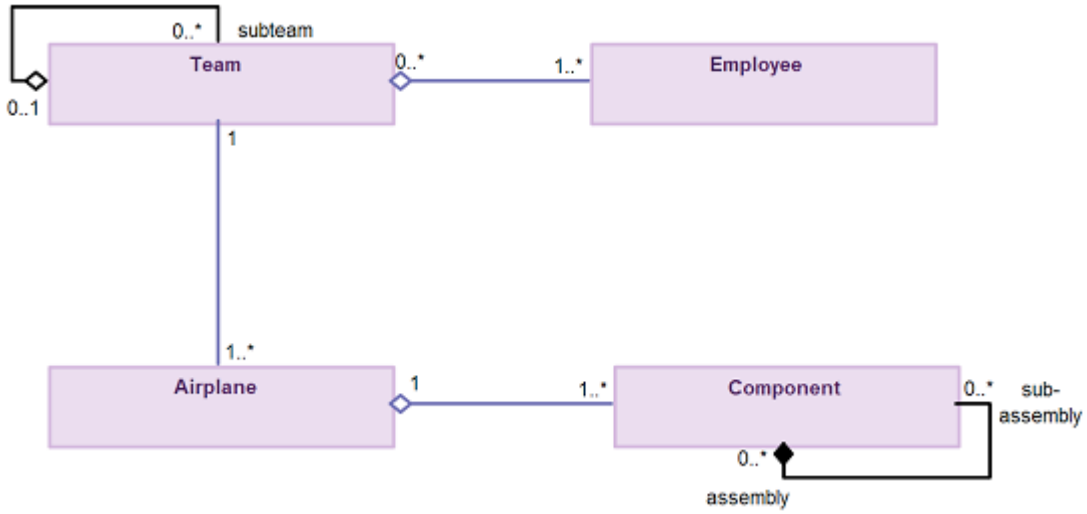



1. Interface definitions must reflect implementation language constraints. In the above image, you see that a standard class box has been used to define the interface PersistentObject (note the use of the <<interface>> stereotype).
2. Name interfaces according to language naming conventions
3. Apply "Lollipop" notation to indicate that a class realizes an interface
4. Define interfaces separately from your classes
5. Do not model the operations and attributes of an interface in your classes. In the above image, you'll notice that the shipment class does not include the attributes or operations defined by the two interfaces that it realizes
6. Consider an interface to be a contract

***Aggregation and Composition***

As it is known an object is made up of other objects. If you were to consider as examples where an airplane consists of wings, a fuselage, engines, flaps, landing gear and so on. A delivery

shipment would contain one or more packages and a team consists of two or more employees. These are all examples of the concept of aggregation that illustrates "is part of" relationships. An engine is part of a plane, a package is part of a shipment, and an employee is part of a team. Aggregation is a specialization of association, highlighting an entire-part relationship that exists between two objects. Composition is a much potent form of aggregation where the whole and parts have coincident lifetimes, and it is very common for the whole to manage the lifecycle of its parts. If you were to consider a stylistic point of view, aggregation and composition are both specializations of association where the guidelines for associations do apply.




1. You should be interested in both the whole and the part
2. Depict the whole to the left of the part
3. Apply composition to aggregates of physical items

***Inheritance***

Inheritance models "is a" and "is like" relationships, enabling you to rather conveniently reuse data and code that already exist. When "A" inherits from "B" we say that "A" is the subclass of "B" and that "B" is the superclass of "A." In addition to this, we have "pure inheritance" when "A" inherits all of the attributes and methods of "B". The UML modeling notation for inheritance is usually depicted as a line that has a closed arrowhead, which points from the subclass right down to the superclass.

1. Plus in the sentence rule for inheritance
2. Put subclasses below superclasses
3. Ensure that you are aware of data-based inheritance
4. A subclass must inherit everything

***Relationship***

At this particular juncture, the term "relationships" will encompass all UML concepts such as aggregation, associations, dependencies, composition, realizations, and inheritance. In other words, if it's a line on a UML class diagram, it can be considered as a relationship. The following guidelines could be considered as "best practices" and and effort should be made to adhere to them at all times.

**Figure A**

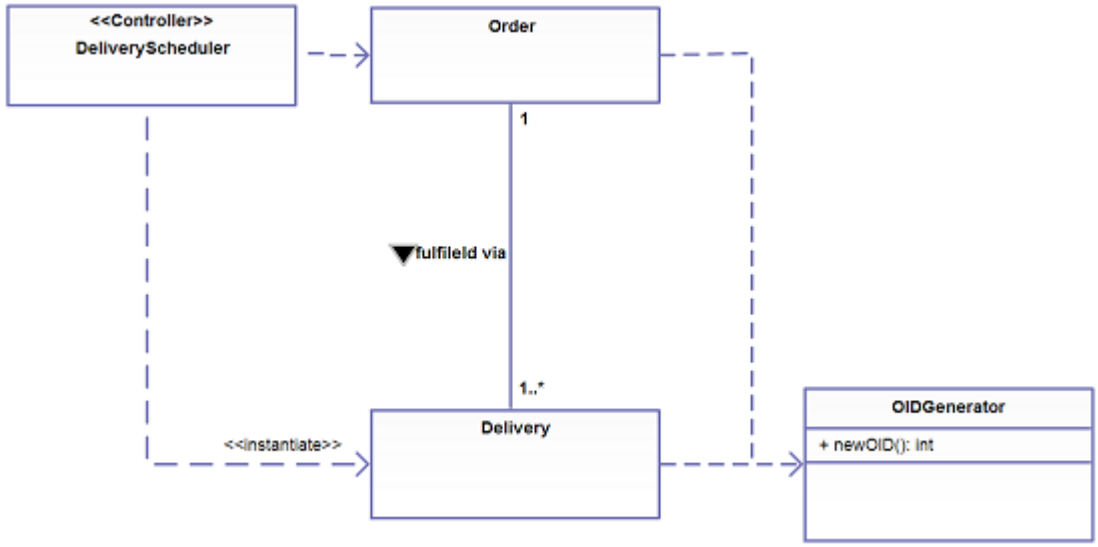



**Figure B**

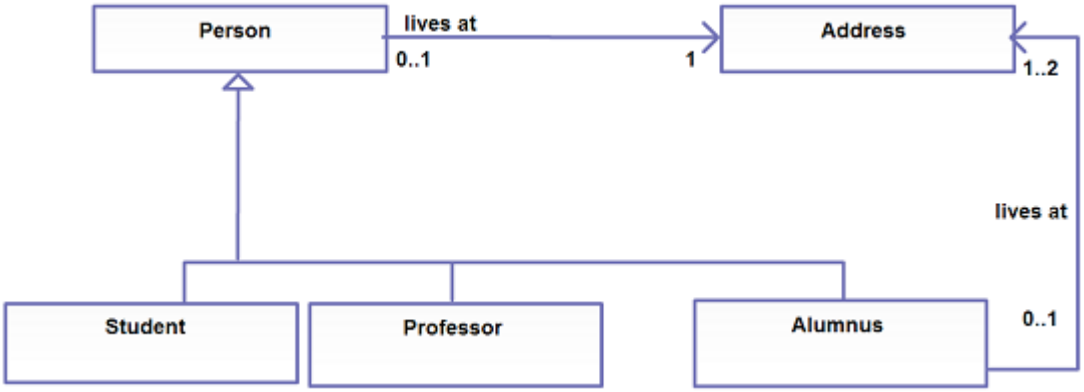
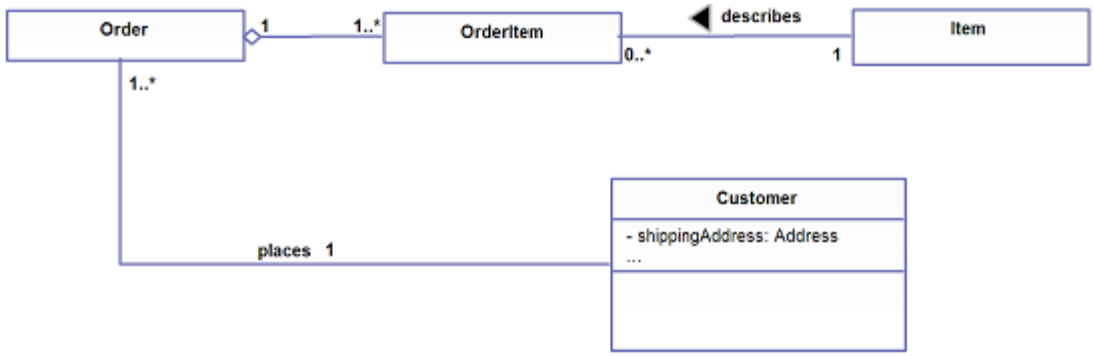



**Figure C**

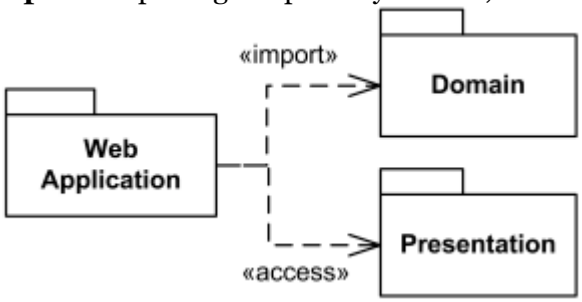



1. Ensure that you model relationships horizontally
2. Collaboration means a need for a relationship
3. Model a dependency when a relationship is in transition

4. Depict similar relationships involving a common class as a tree. In **Figure A** you see that both Delivery and Order have a dependency on OIDGenerator. Note how the two dependencies are drawn in combination in "tree configuration", instead of as two separate lines, to reduce clutter in the diagram.
5. As a rule it is best to always indicate the multiplicity
6. Avoid a multiplicity of "*" to avoid confusion
7. Replace relationships by indicating attribute types. In **Figure C** you see that the customer has a shippingAddress attribute of type Address – part of the scaffolding code to maintain the association between customer objects and address objects.
8. Never model implied relationships
9. Never model every single dependency
10. Center names on associations
11. Write concise association names in active voice
12. Indicate directionality to clarify an association name
13. Name unidirectional associations in the same direction
14. Word association names left-to-right
15. Indicate role names when multiple associations between two classes exist
16. Indicate role names on recursive associations

8. **Brief notes on**
    - **Keywords(4)**
    - **Stereotypes, profiles, tags(4)**
    - **Interfaces (4)**
    - **Active classes and dependency(4)**

A keyword is shown near the dashed arrow to identify which kind of package import is intended. The predefined keywords are **«import»** for a **public** package import, and **«access»** for a **private** package import. By default, the value of visibility is public.




**Stereotypes**

A **stereotype** is one of three types of extensibility mechanisms in the Unified Modeling Language (UML), the other two being tags and constraints. They allow designers to extend the vocabulary of UML in order to create new model elements, derived from existing ones, but that have specific properties that are suitable for a particular problem domain or otherwise specialized usage. The nomenclature is derived from the original meaning of stereotype, used in printing. For example, when modeling a network you might need to have symbols for representing routers and hubs. By using stereotyped nodes you can make these things appear as primitive building blocks.

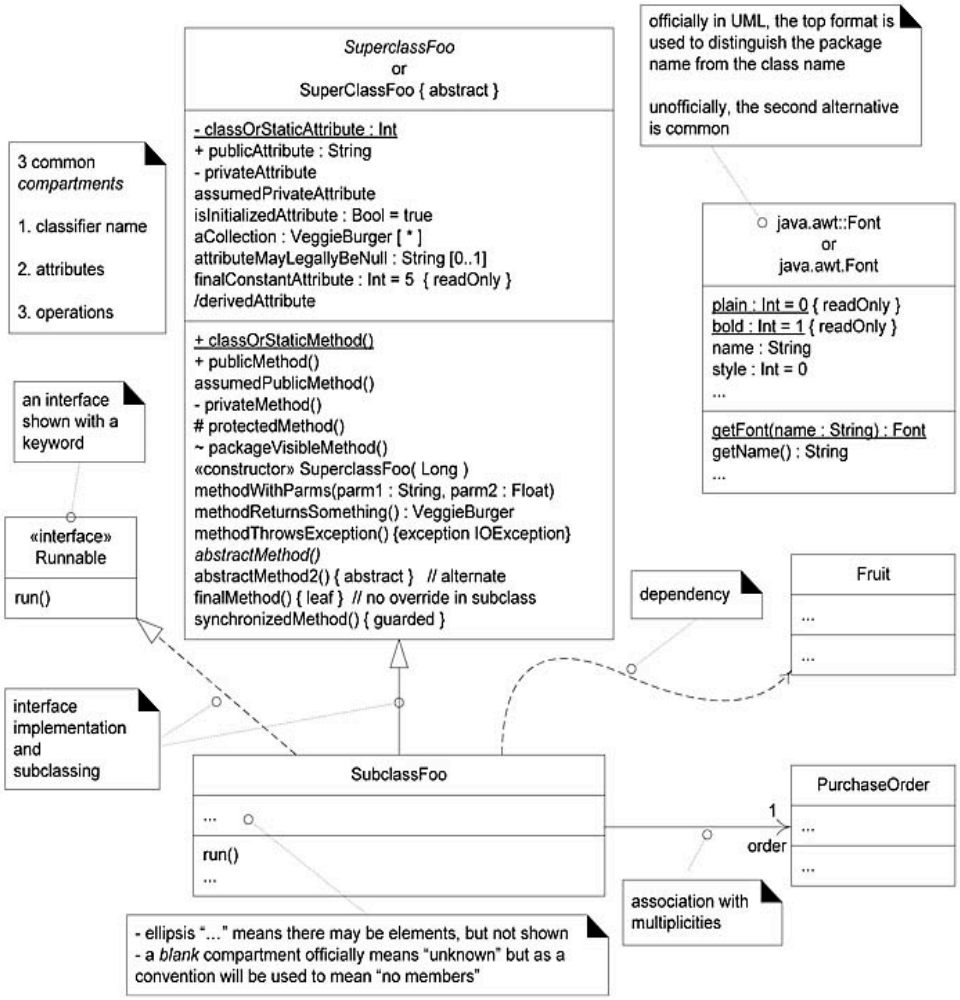
Graphically, a stereotype is rendered as a name enclosed by guillemets (« » or, if guillemets proper are unavailable, << >>) and placed above the name of another element. In addition or alternatively it may be indicated by a specific icon. The icon image may even replace the entire UML symbol. For instance, in a class diagram stereotypes can be used to classify method behavior such as «constructor» and «getter». Despite its appearance, «interface» is not a stereotype but a classifier.[1]

One alternative to stereotypes, suggested by Peter Coad in his book *Java Modeling in Color with UML: Enterprise Components and Process* is the use of colored archetypes. The archetypes indicated by different-colored UML boxes can be used in combination with stereotypes. This added definition of meaning indicates the role that the UML object plays within the larger software system.

**9. Explain in detail about the various types of relations in a class diagram.**

The UML includes class diagrams to illustrate classes, interfaces and their associations. They are used for static object modeling.

***UML Notation for Class diagram***




***COMMON UML CLASS DIAGRAM NOTATION***

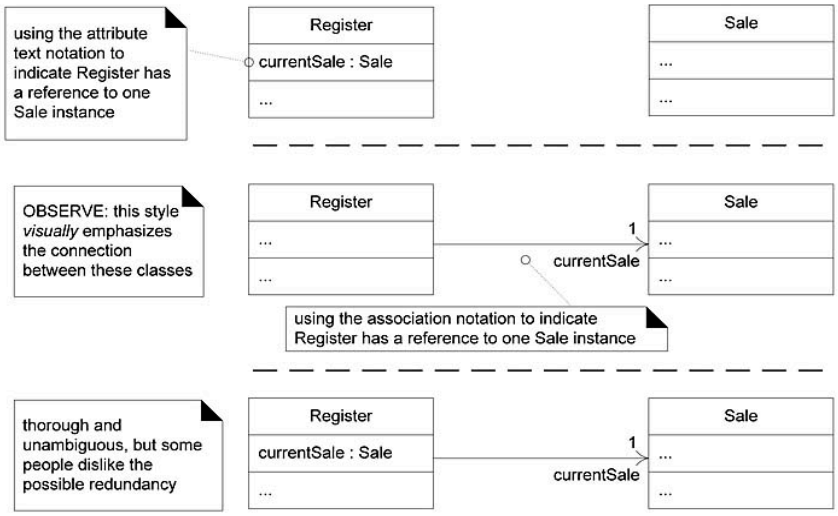
Most elements are optional (e.g.: +/- visibility, parameters, compartments) modelers draw, show or hide them depending on context and the needs of the reader or UML tool.

***Concept of Link, Association and Inheritance:***

*Attribute-as-Association:*

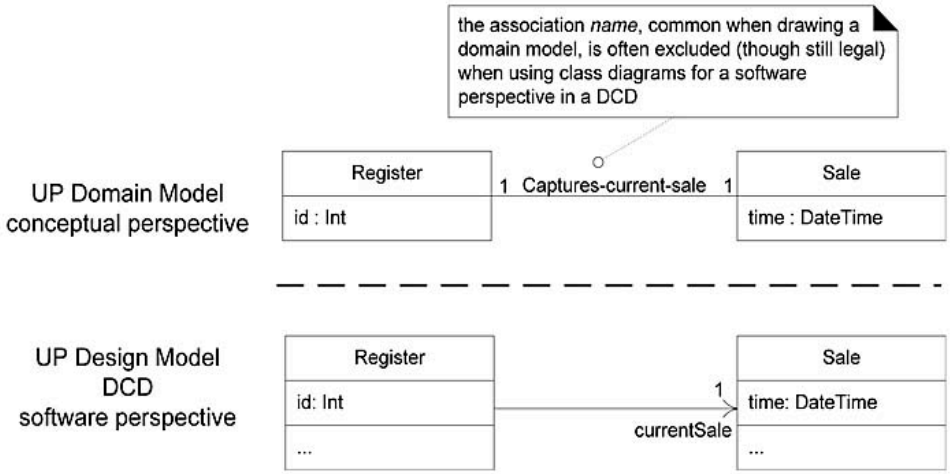
Attribute-as-Association line has the following style:

- ♣ A navigational arrow pointing from source (Register) to target (Sale), indicating a Register object has an attribute of one Sale.
- ♣ A multiplicity at the end, but not at the source.
- ♣ No association name.




***ATTRIBUTE TEXT VERSUS ASSOCIATION LINE NOTATION FOR A UML ATTRIBUTE***

When using class diagrams for a domain model show the association names but avoid navigation arrows, as domain model is not perspective.




***IDIOMS IN ASSOCIATION NOTATION USAGE IN DIFFERENT PERSPECTIVE***

**10. Explain about use case realization with GoF design pattern?**

**Adapter Pattern**

The polymorphism pattern (GRASP) example covered in the earlier lecture and its solution is more specifically an example of the GoF Adapter pattern.

**Context/Problem:** How to resolve incompatible interfaces, or provide a stable interface to similar components which have different interfaces?

**Solution:** Convert the original interface of a component into another interface, through an intermediate *adapter* object.

The POS system needs to support several kinds of external third-party services, including tax calculators, credit authorization services, inventory systems, accounting systems, etc. Each has a different API which we don't have any control.

A solution is to add a level of indirection with objects that adapt the varying external interfaces to a consistent interface used within the application.

***Using an Adapter***

A particular adapter instance will be instantiated for the chosen external service, such as SAP for accounting. This instance will adapt the *postSale* request to the external interface.

***Relationship with GRASP patterns***

The above application of the Adapter pattern is a specialization of the GRASP building blocks. It offers Protected Variations from changing external interfaces or third-party packages through the use of Indirection object that applies interfaces and Polymorphism.

***Domain Model Changes during Design***

The *getTaxes* operation returns a list of *TaxLineItem*s. The tax line items are associated with a Sale. The TaxLineItem class will now be both a software (design) class as well as a domain concept.

***Related Patterns***

An adapter that hides an external system may also be considered a *Façade* object, as it wraps access to the system with a single object. Strong motivation to call it an adapter since the wrapping object provides adaptation to varying external interfaces.

**Factory Pattern**

Consider the problem discussed in the Adapter pattern: who creates the adapters? How to determine which class of adapter to create?

If some domain object creates them, the responsibilities of the domain object are going beyond pure application logic and into concerns related to connectivity with external software components.

**Goal**: Design to maintain a separation of concerns. If a domain object is chosen, a lower cohesion results.

Apply the (concrete) Factory pattern, in which a Pure Fabrication "factory" object is defined to create objects. Thus, we

Separate the responsibility of complex creation into cohesive helper objects

Hide potentially complex creation logic.

**Context/Problem:** Who should be responsible for creating objects when there are special considerations such as complex creation logic, a desire to separate the creation responsibilities for better cohesion, etc.?

**Solution:** Create a pure fabrication object called a *Factory* that handles the creation.

The code for getTaxCalculatorAdapter()

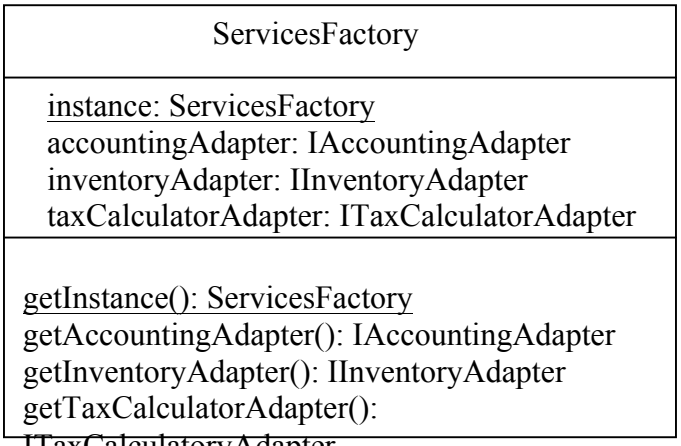
```
{
        if (taxCalculatorAdapter == null)
        {
          String className = System.getProperty("taxcalculator.class.name");
          TaxCalculatorAdapter = (ITaxCalculatorAdapter)
Class.forName(className).newInstance();
        }
        return taxCalculatorAdapter;
}
```

- **Singleton Pattern**

  A new problem with the *ServicesFactory* approach: who creates the factory itself, and how it accessed?
  - Only one instance of the factory is needed with the application
  - The methods of this factory may be called from anywhere in the code. How to get visibility to this single instance?

  - **One Solution**: Pass the *ServicesFactory* instance around as a parameter to whenever a visibility is needed for it. This approach is inconvenient.
  - **Alternative approach**: The *Singleton* pattern. Sometimes, it is desirable to support global visibility or a single access point to a single instance of a class.

  **Context/Problem:** Exactly one instance of a class is allowed – it is a "singleton". Objects need a global and a single point of access.
  **Solution:** Define a static method which returns the singleton.




The code for the getInstance() method:

```
{
        public static synchronized ServicesFactory getInstance()
        {
          if (instance == null)
                instance = new ServicesFactory();
         return instance;
        }
}
```

The key idea is that class X defines a static method *getInstance* that itself provides a single instance of X. Now, a programmer has global visibility to this single instance:

```
public class Register
{
        public void initialize()
        {
                …
                accountingAdapter =
ServicesFactory.getInstance().getAccountingAdapter();
                …
        }
}
```

**UML for Singleton Access:**

This approach avoids explicitly showing the *getInstance* message to the class before sending a message to the singleton interface.

**Strategy Pattern**

Since the behavior of pricing varies by strategy (or algorithm), we create multiple *SalePricingStrategy* classes, each with a polymorphic *getTotal* method. Each *getTotal* method takes the *Sale* object as a parameter, so that the pricing strategy object can find the pre-discounted price from the *Sale*, and then apply the discounting rule.

A strategy object is attached to a context object – the object to which it applies the algorithm, e.g., Sale. When a *getTotal* message is send to a *Sale*, it delegates some of the work to its strategy object.

## UNIT V
## PART A

**1.Define testing.**

Testing is the process of using suitable test cases to evaluate and ensure the quality of a product by removing or sorting out the errors and discrepancies. It is also used to ensure that the product has not regressed (such as, breaking a feature that previously worked).Testing involves various types and levels based on the type of object/product under test. Testing can be described as a process used for revealing defects in software, and for establishing that the software has attained a specified degree of quality with respect to selected attributes.

**2.What is test driven development?**

Unit testing code is written before the code to be tested, and the developer writes unit testing code for all production code. Ie, the basic method is to write a test code, then write a little production code, make it pass the test, then write some more test code and so forth.

**3.What is refactoring?**

Refactoring is a structured, disciplined method to rewrite or restructure existing code without changing its external behaviour, applying small transformation steps combined with re-executing tests at each step. Refactoring is another extreme programming (XP) step applicable to all iterative methods.

**4.What are the issues in object oriented testing?**

Specify product requirements long before testing commences
Understand the users of the software, with use cases
Develop a testing plan that emphasizes “rapid cycle testing
Build robust software that is designed to test itself
Conduct formal technical reviews to assess test strategy and test cases

**5.What are the steps in object oriented testing?**

Because the OO analysis and design models are similar in structure and content to the resultant OO program, “testing” begins with the review of these models. Once code has been generated, OO testing begins “in the small” with class testing. As classes are integrated to form a subsystem, thread-based, use-based, and cluster testing are applied to fully exercise collaborating classes. Finally, use-cases are used to uncover errors at the software validation level.

**6.What is the need for testing a code?**

The programmers and the testers have to execute the program before it gets to the customer with the specific intent of removing all errors, so that the customer will not experience the frustration associated with a poor-quality product. In order to find the highest possible number of errors, tests must be conducted systematically and test cases must be designed using disciplined techniques.

**7.What is class testing?**

This deals with checking how the class (smallest testable unit) reacts to a scenario. These test integrations are for classes and they determine how a class reacts to being created and how each of its methods reacts to being called in a typical system scenario. The test should determine if the calling of any methods has undesirable side effects for other methods.

**8.What is random class testing?**

In random class testing, the classes or the methods of a class can be tested using random test cases in a random sequence. The test process need not follow a procedure or a finite set of test cases for the methods of a class.

**9.Explain object oriented system testing.**

System Testing is the process of removing the test harness and combining the integrated classes, and testing the system as a whole.

**10.Explain about object oriented integration testing.**

Integration testing is the process of testing the classes in combination with other classes. If everything works as expected then it is termed that the class has passed the test. In this test methodology, the test cases and classes that create errors or are faulty can be removed and replaced with correct and working classes.

**11.What is a test harness?**

A test harness is an environment into which a software component can be placed and tested using test cases. If a class under test does not interact with any other classes, then the test harness consist of a main program and the class under test. If the class interacts with other classes then the test harness consists of the main program, the class under test and dummy class to replace the other classes.

**12.Compare system testing and integration testing.**

In integration testing, the product is divided into smaller subsystems and tested separately. They may be combined with other subsystems and tested.

In system testing the test data and the classes are combined as one and tested as a whole to find out how the entire system works in the test or launch environment.

**13.What are test cases? When we say test case is effective?**

The usual approach to detecting defects in a piece of software is for the tester to select a set of input data and then execute the software with the input data under a particular set of conditions. The tester bundles this information into an item called test case.

- A greater probability of detecting defects
- A more efficient use of organizational resources
- A higher probability for test reuse
- Closer adherence to testing and project schedules and budgets
- The possibility for delivery of a higher-quality software product

**14.What is validation and verification?**

Validation is the process of evaluating a software system or component during, or at the end of the development cycle in order to determine whether it satisfies specified requirements.

Validation is usually associated with traditional execution based testing, that is exercising the code with test cases.

**15.How is class testing different from conventional testing?**

Conventional testing focuses on input-process-output, whereas class testing focuses on each method, then designing sequences of methods to exercise states of a class. In conventional testing methods the test cases are applied and the output is focused on ie, enter the input and wait for the valid and correct output. If there is a false output then it signifies the occurrence of an error.In class testing the methods of the classes under test is subjected to various test cases in a suitable test environment.

**16.Explain about thread based, cluster based, and use based testing in Integration testing.**

Thread-based testing – testing of all classes which are required to respond to one system input or event

Use-based testing – in this the independent classes are tested first and the dependent classes are tested later.

Cluster testing - groups of collaborating classes are tested for interaction errors

**17.Explain about GUI testing.**

Graphical user interface testing is the process o testing the look, presentation and the feel of the software under test. It involves using test cases so as to make the interface and the usage of the application under test, more easy to use.

**18.Why do conventional top down and bottom up integration testing methods have less meaning in an object oriented context?**

Basic object oriented software does not have a hierarchical control structure, hence top down approach and bottom up approach of testing is of less use in testing. Therefore, thread based, use based and cluster based testing methods are incorporated for performing integration testing.

**19.What is the test case design for object oriented software?**

White-box testing methods can be applied to testing the code used to implement class operations. In this the code and the methods used in the algorithms can be tested. Black-box testing methods are appropriate for testing OO systems. In black box testing only the interface and the structure of the code can be tested.

**20.When does testing of a product/scenario end?**

Practically, testing is a process that never ends. This can be expressed in three ways.

- The burden of ensuring quality to a product simply shifts from the developer to the tester and then to the customer.
- Testing is done when you run out of time or money.
- Use a statistical model:

Assume that errors decay logarithmically with testing time
Measure the number of errors in a unit period
Fit these measurements to a logarithmic curve

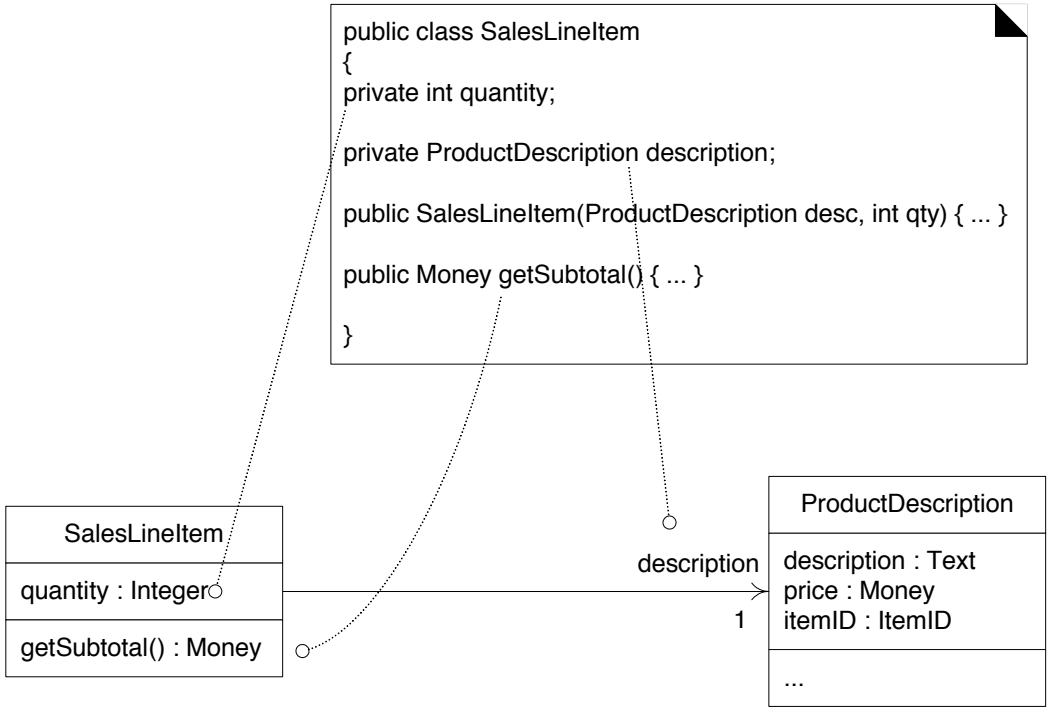
**PART B**

**1. Explain Mapping design to code process with examples.**

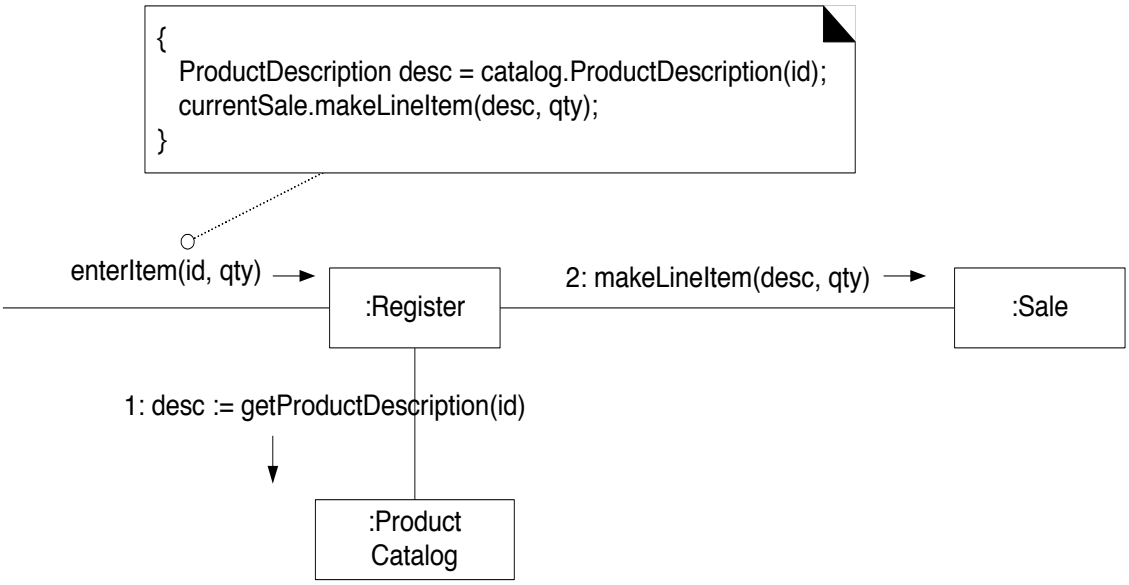
**MAPPING DESIGNS TO CODE:**

- Write source code for:
  - Class and interface definitions
  - Method definitions
- Work from OOA/D artifacts
  - Create class definitions for Domain Class Diagrams (DCDs)
  - Create methods from Interaction diagrams

**FROM DCD TO JAVA CLASS**

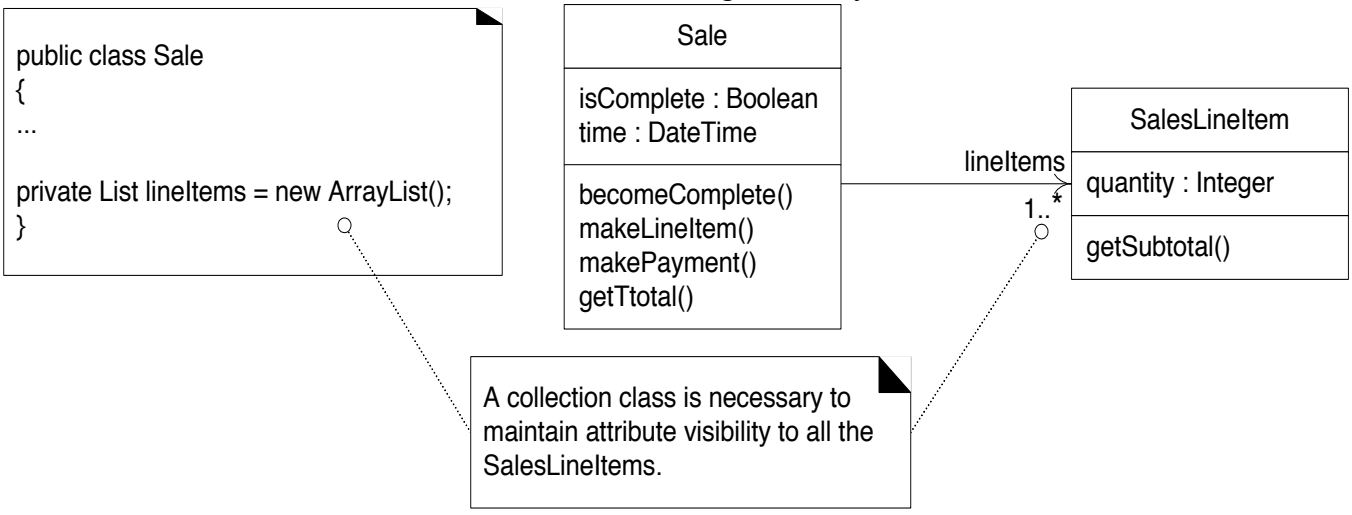


## FROM INTERACTION DIAGRAM TO METHOD




## Collection classes

What collection class has been added to the design and why?



2. **What is object relation mapping? Explain in detail.**

**Object-relational mapping** (**ORM**, **O/RM**, and **O/R mapping**) in computer science is a programming technique for converting data between incompatible type systems in object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language. There are both free and commercial packages available that perform object-relational mapping, although some programmers opt to create their own ORM tools.

In object-oriented programming, data management tasks act on object-oriented (OO) objects that are almost always non-scalar values. For example, consider an address book entry that represents a single person along with zero or more phone numbers and zero or more addresses. This could be modeled in an object-oriented implementation by a "Person object" with attributes/fields to hold each data item that the entry comprises: the person's name, a list of phone numbers, and a list of addresses. The list of phone numbers would itself contain "PhoneNumber objects" and so on. The address book entry is treated as a single object by the programming language (it can be referenced by a single variable containing a pointer to the object, for instance). Various methods can be associated with the object, such as a method to return the preferred phone number, the home address, and so on.

However, many popular database products such as structured query language database management systems (SQL DBMS) can only store and manipulate scalar values such as integers and strings organized within tables. The programmer must either convert the object values into groups of simpler values for storage in the database (and convert them back upon retrieval), or only use simple scalar values within the program. Object-relational mapping is used to implement the first approach.[1]

The heart of the problem is translating the logical representation of the objects into an atomized form that is capable of being stored in the database, while preserving the properties of the objects and their relationships so that they can be reloaded as objects when needed. If this storage and retrieval functionality is implemented, the objects are said to be persistent.

3. **List out the various issues in OO Testing.**

**Testing Steps:**

Determine what the test is supposed to measure
Decide how to carry out the tests
Develop the test cases
Determine the expected results of each test (test oracle)
Execute the tests Å Compare results to the test oracle

**Issues:**

Because object interaction is essential to O-O systems, integration testing must be more extensive
Inheritance makes testing more difficult by requiring more contexts (all sub classes) for testing an inherited module

4. **What is testing and what is the need for testing? Explain the different types of Testing.**

**Testing:**

1. Object-oriented systems are built out of two or more interrelated objects
2. Determining the correctness of O-O systems requires testing the methods that change or communicate the state of an object
3. Testing methods in an object-oriented system is similar to testing subprograms in process-oriented systems

**Need for Testing:**

1. A test case is a set of inputs to the system
2. Successfully testing a system hinges on selecting representative test cases
3. Poorly chosen test cases may fail to illuminate the faults in a system
4. In most systems exhaustive testing is impossible, so a white box or black box testing strategy is typically selected

**Types Of Testing:**

Black box testing
White box testing
Unit testing
System testing
Integration testing

**5. Explain in detail about Class testing with example.**

Object-Oriented Testing Techniques

Grey Box Testing

The different types of test cases that can be designed for testing object-oriented programs are called grey box test cases. Some of the important types of grey box testing are:

- **State model based testing** : This encompasses state coverage, state transition coverage, and state transition path coverage.
- **Use case based testing** : Each scenario in each use case is tested.
- **Class diagram based testing** : Each class, derived class, associations, and aggregations are tested.
- **Sequence diagram based testing** : The methods in the messages in the sequence diagrams are tested.

**6. Write in detail with an example about Object oriented integration testing.**

| | |
|---|---|
| integration testing | **has definition** Testing a system in a phased approach - first testing individual subsystems, then testingthe integrated system |
| | **has definition** Testing during or following the process of integrating a system |
| | **has advantage** when you find a problem, you can find the defect more easily because you have a better idea in which subsystem to look |
| | **is** better than big bang testing *for* large systems |
| | **is a subtopic of** 10.9 - Strategies for Testing Large Systems |
| | **is a kind of** testing performed by software engineers |
| testing | **can find** defects whose consequences are obvious but which are buried in complex code, and thus will be hard to detect when inspecting |
| | **involves** thinking of what could go wrong without actually studying the software. |

**Kinds of integration testing** :

- big bang testing (5 facts) - An inappropriate approach to integration testing in which you take the entire integrated system and test it as a unit
- incremental testing (2 kinds, 3 facts) - A integration testing strategy in which you test subsystems in isolation, and then continue testing as you integrate more and more subsystems.

**7. Explain briefly about GUI testing with an example.**

In software engineering, **graphical user interface testing** is the process of testing a product's graphical user interface to ensure it meets its specifications. This is normally done through the use of a variety of test cases.

Unlike a CLI (command line interface) system, a GUI has many operations that need to be tested. A relatively small program such as Microsoft WordPad has 325 possible GUI operations. In a large program, the number of operations can easily be an order of magnitude larger.

The second problem is the sequencing problem. Some functionality of the system may only be accomplished with a sequence of GUI events. For example, to open a file a user may have to first click on the File Menu, then select the Open operation, use a dialog box to specify the file name, and focus the application on the newly opened window. Increasing the number of possible operations increases the sequencing problem exponentially. This can become a serious issue when the tester is creating test cases manually.

Regression testing becomes a problem with GUIs as well. A GUI may change significantly, even though the underlying application does not. A test designed to follow a certain path through the GUI may then fail since a button, menu item, or dialog may have changed location or appearance.

These issues have driven the GUI testing problem domain towards automation. Many different techniques have been proposed to automatically generate test suites that are complete and that simulate user behavior.

Most of the testing techniques attempt to build on those previously used to test CLI (Command Line Interface) programs, but these can have scaling problems when applied to GUI's. For example, Finite State Machine-based modeling where a system is modeled as a finite state machine and a program is used to generate test cases that exercise all states — can work well on a system that has a limited number of states but may become overly complex and unwieldy for a GUI.

8. **What is System testing and explain the same with an example.**

Software may be part of a larger system. This often leads to "finger pointing" by other system dev teams

Finger pointing defence:

a. Design error-handling paths that test external information
b. Conduct a series of tests that simulate bad data
c. Record the results of tests to use as evidence

Types of System Testing:

d. Recovery testing: how well and quickly does the system recover from faults
e. Security testing: verify that protection mechanisms built into the system will protect from unauthorized access (hackers, disgruntled employees, fraudsters)
f. Stress testing: place abnormal load on the system

g. Performance testing: investigate the run-time performance within the context of an integrated system

All rules and methods of traditional systems testing are also applicable applicable to object-oriented.

A thread can be "re-defined" as a sequence of method executions linked by messages in the object network.

Use cases provide a more precise mechanism for writing test cases.

**9. What is the difference between integration and system testing?**

For Integration testing, test cases are developed with the express purpose of exercising the interfaces between the components
In system testing the complete system is configured in a controlled environment and test cases are created to simulate the real time scenarios that occurs in a simulated real life test environment.

The purpose of integration testing is to ensure distinct components of the application still work to accordance of the user requirements. The purpose of system testing is to validate an application and completeness in performing as designed and to test all functions of the system that is required in real life. Integration testing is the phase in software testing in which individual software modules are combined and tested as a group.
It occurs after unit testing and before system testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system

The basic difference in integration testing and system testing is about its approach and scale. In system testing the basic code level knowledge of various modules is not required. Also the most popular approach of system testing is Black Box testing.
So, Just to sum up, system testing consumes all of the "integrated" software modules that cleared integration testing and also the software system itself.The thought behind performing integration testing is to identify and issues between different code modules that are integrated together.System testing aims to detect defects both within the integrated modules and also within the system as a whole single unit

**10. What are the different issues involved in object oriented testing? Explain with examples.**

Testing in an OO context must address the basics of testing a base class and the code that uses the base class. Factors that affect this testing are inheritance and dynamic binding.

Therefore, some systems are harder to test (e.g., systems with inheritance of implementations harder than inheritance of interfaces) and OO constructs such as inheritance and dynamic binding may serve to hide faults

Use the OO Metrics techniques to assess the design and identify areas that need special attention in testing for adequate coverage. For example, depth of tree complicates testing and number of methods that have to be tested in a class and a lack of cohesion in methods means more tests to ensure there are no side effects among the disjoint methods.

**OO Testing Strategies**

While there are efforts underway to develop more automated testing processes from test models of the object model characteristics (for example states, data flows, or associations), testing is still based on the creation of

test cases and test data by team members using a structural (White Box Testing) and/or a functional (SeeBlack Box Testing) strategy.

**Strategies for Selecting Test Cases**

Preparing Test Cases and Scenarios discusses strategies for test cases. Two common strategies to select test cases are based on an assessment of "likely faults" and "likely usage." Each is discussed below.

**Likely faults** - These types of tests are based on practical experience of areas in which errors are most likely to occur (e.g., certain syntax in a particular programming language or boundaries like beginning and end conditions). Checklists may be developed for types of applications or development environments. These help formalize this process and ensure more consistency and coverage in developing these types of tests.

**Likely usage** - These tests test to exercise the system in the ways that the user of the system will be likely to use the system. Or they aim to test "completely" the elements of the system most likely to be used.

These strategies apply to each type of test (structural or functional).

Derive the plan and the test cases to test the system sufficiently for it to be effective, i.e., to conform to its acceptance criteria.

**Levels of Testing**

Within the development life cycle, testing can occur at different levels: unit test, integration test, and system or acceptance tests.

**Unit** - Here the unit is a class, often a "bigger" or more complex thing than a procedural unit which may be a single function or sub-routine. Complicated by inheritance, encapsulation, and dynamic binding. A unit test is more complicated and more effective in the overall system than with procedural unit tests. .

**Integration** - Focus on interactions among classes. Complicated by dynamic binding. Recommended that unit be integrated in an incremental fashion, a few at a time, not one “big bang” integration of the system units. This approach lets interface problems be detected as early as possible but may complicate the design of a test since the cluster of units may not correspond neatly to a functional grouping. When the integration units don’t comprise a functional grouping, the test may, of necessity, focus on structural issues.