Chapter – 2

UML - Unified Modelling Language

Objectives

At the end of this Chapter, the students will be able to:

- Understands goals of Unified Modeling Language.
- Understand various symbols and representations used in modeling software development by unified process.
- Understand UML Tool for construction of Use Case, Collaboration, sequence and State diagrams.
- Apply the concepts for the design of new software.

2.0. Introduction

The software process is the way we produce software. It incorporates the methodology with its underlying software life-cycle model and techniques, the tools we use, and most important of all, the individuals building the software.

As methodology is one component of a software process. The primary object-oriented methodology today is the Unified Process. The Unified "Process" is actually a methodology, but the name Unified Methodology already had been used as the name of the first version of the Unified Modeling Language (UML). The three precursors of the Unified Process (OMT, Booch's method, and objectory) are no longer supported, and the other object-oriented methodologies have had little or no following. As a result, the Unified Process is usually the primary choice today for object-oriented software production. The Unified Process is an excellent object-oriented methodology in almost every way.

The Unified Process is not a specific series of steps that, if followed, will result in the construction of a software product. In fact, no such single "one size fits all" methodology could exist because of the wide variety of types of software products. For example, there are many different application domains, such as insurance, aerospace, and manufacturing. Also, a methodology for rushing a COTS package to market ahead of its competitors is different from one used to construct a high-security electronic funds transfer network. In addition, the skills of software professionals can vary widely.

Instead, the Unified Process should be viewed as an adaptable methodology. That is, it is modified for the specific software product to be developed. Some features of the Unified Process are inapplicable to small- and even medium-scale software. However, much of the Unified Process is used for software products of all sizes.

An Object-Oriented Paradigm can be discussed in detail with notations and representations using Unified Modeling Language (UML).

2.1. Unified Modeling Language

The Unified Modeling Language (UML) is an Object-Oriented language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling (UML Document Set, 2001). Rational Software and its partners developed the UML. It is the successor to the modeling languages found in the Booch (Booch 1994), OOSE/Jacobson, OMT and other methods.

By offering a common development language, UML relieves developers of the proprietary ties that are so common in software engineering and can make developing systems difficult and expensive. Major companies such as IBM, Microsoft, and Oracle are brought together under the UML umbrella. Due to the fact that UML uses simple, innate notation, even users with limited software engineering skills can understand UML models. In fact, many of the language's supporters claim that UML's simplicity is its chief benefit. If developers, customers and implementers can all understand a UML diagram, they are more likely to agree on the intended functionality, thereby increasing their chances of creating an application that really solves a problem.

UML, a visual modeling language, is not intended to be a visual programming language. The UML notation is useful for graphically portraying an object-oriented analysis and design model. It not only allows you to specify the requirements of a system and capture the design decisions, but it also enhances communication amongst all the relevant people involved in the development task. The emphasis in modeling should be on analysis and design.

UML has several diagrams that you can use to model a system, but the minimum that would be required are:

- A class diagram to show the objects within the system and any link between them. This is useful tool for analysis or design phases.

- A use case diagram to help capture what the system does and who interacts with it. This is most useful for showing the purpose of the system.
- A sequence diagram to capture the events taking place in the system and examine how the system behaves.
- A data model to capture the data. This is usually more beneficial to the implementation stage of the lifecycle model.

Different models in Unified Modeling Language (UML)

- User model view. This view represents the system (product) from the user's (called "actors" in UML) perspective.
- Structural model view. Data and functionality is viewed from inside the system. That is, static structure (classes, objects, and relationships) is modeled.
- Behavioral model view. This part of the analysis model represents the dynamic or behavioral aspects of the system.
- Implementation model view. The structural and behavioral aspects of the system are represented as they are to be built.
- Environment model view. The structural and behavioral aspects of the environment in which the system is to be implemented are represented.

UML analysis modeling focuses on the first two views of the system.
UML design modeling addresses the three other views.

2.1.1. Goals of UML

The primary goals in the design of the UML were:

1. Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modeling language.
5. Encourage the growth of the OO tools market.
6. Support higher-level development concepts such as collaborations, frameworks, patterns and components.
7. Integrate best practices.

Some of the benefits of UML are:

1. Your software system is professionally designed and documented before it is coded. You will know exactly what you are getting, in advance.

2. Since system design comes first, reusable code is easily spotted and coded with the highest efficiency. You will have lower development costs.

3. Logic 'holes' can be spotted in the design drawings. Your software will behave as you expect it to. There are fewer surprises.

4. The overall system design will dictate the way the software is developed. The right decisions are made before you are married to poorly written code. Again, your overall costs will be less.

5. UML lets us see the big picture. We can develop more memory and processor efficient systems.

6. When we come back to make modifications to your system, it is much easier to work on a system that has UML documentation. Much less 'relearning' takes place. Your system maintenance costs will be lower.

7. If you should find the need to work with another developer, the UML diagrams will allow them to get up to speed quickly in your custom system. Think of it as a schematic to a radio. How could a tech fix it without it?

8. If we need to communicate with outside contractors or even your own programmers, it is much more efficient.

Using the Unified Modeling Language will result in lower overall costs, more reliable and efficient software, and a better relationship with all parties involved. Software documented with UML can be modified much more efficiently.

2.1.2 Building Blocks in UML

There are three types of building blocks in UML. They are: (1) Entities, (2) Relationships among the entities, and (3) Diagrams that depict the relationship among the entities.

2.1.2.1 UML Entities

Entities can be structural, behavioral, grouping, or annotational. Table 2.1 gives the names of the various entities. Table 2.2 briefly describes the entities, and shows their UML symbols.

Structural entity		Behavioral entity	Grouping entity	Annotational entity
Conceptual	**physical**			
Class Interface Collaboration Use Case Active Class	Component Node	Interaction State machine Activity	Package	Note

Table 2.1: The Entities in UML

2.1.2.2 Relationships among Entities

A relationship is defined between two entities to build a model. It can be of four types:

1. Dependency (A semantic relationship)
2. Association (A structural relationship)
3. Generalization (A generalization/specialization relationship)
4. Realization (A semantic relationship)

Table 2.3 gives the description of the relationships and their UML symbols.

2.1.2.3 Diagrams in the UML

UML specifies nine diagrams to visualize relationships among the entities of a system. The diagrams are directed graphs in which nodes indicate entities and arcs indicate relationships among the entities. The nine diagrams are the following:

1. Class Diagram
2. Object Diagram
3. Use Case Diagram
4. Sequence Diagram
5. Collaboration Diagram
6. State-chart Diagram
7. Activity Diagram
8. Component Diagram and
9. Deployment Diagram
10. Package Diagram

Table 2.4 indicates which diagrams are useful in which view of the software architecture.

Entity	Description	UML symbol
Class	A template for a set of objects.	Polygon Area getArea ( ) Rotate ( )
Interface	A set of operations specifying a service of a class or component.	Customer
Collaboration	Interaction among entities to provide a service.	Customer Order
Use Case	A set of sequence of actions that an actor and the system do.	Process Order
Active Class	A class whose objects own processes or threads and hence can initiate a control activity required in concurrent systems.	Polygon area getArea ( ) rotate ( )
Component	A replaceable entity—packaging classes, interfaces, collaborations, and standard deployment components—which conforms to and provides the realization of a set of interfaces.	agent.java
Node	An element that exists at run time, executes components, and is a computational resource (with some memory and processing capability).	Sales
Interaction	A set of messages exchanged among a set of objects.	compute
State Machine	Sequences of states an object or an interaction goes through during its lifetime in response to events.	Renewing a Book
Package	A mechanism for organizing elements of structural, behavioural, and grouping entities into a set.	Borrow
Note	Comments and constraints attached to an element or a collection of elements of a UML model.	This is an instance of the employee class

Table 2.2: Entity Descriptions and their UML Symbols

Relationship	Description	UML symbol
Dependency	A semantic relationship between an independent entity and a dependent entity—a change in the former causes a semantic change in the latter.	
Association	A structural relationship describing a set of links—a set of connections—among objects	1 * teacher student
Generalization	A generalization/specialization relationship in which objects of a child inherit the structure and behaviour of a parent.	
Realization	A semantic relationship between classifiers (*i.e.*, between interfaces and classes and between use cases and their collaborations) so that a contract specified by one is carried out by the other.	

Table 2.3: Relationship Description and their UML Symbols

Architectural view	Use case		Design		Process		Implementation		Deployment	
Diagrams	Static	Dynamic	Static	Dynamic	Static	Dynamic	Static	Dynamic	Static	Dynamic
Class			x		x					
Object			x		x					
Use Case	x									
Sequence		x		x		x		x		x
Collaboration		x		x		x		x		x
Statechart		x		x		x		x		x
Activity		x		x		x		x		x
Component							x			
Deployment									x	

Table 2.4: Use of Diagrams in the Architectural Views of Software Systems

2.1.3. Structural Elements & Types of UML Diagrams

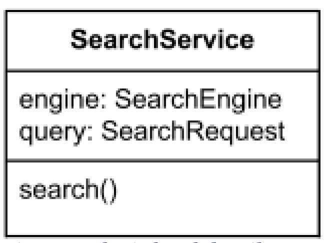
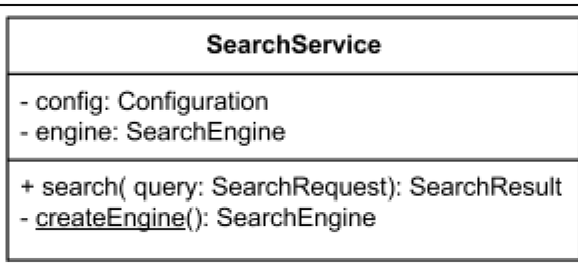
Each UML diagram is designed to let developers and customers view a software system from a different perspective and in varying degrees of abstraction. UML diagrams commonly created in visual modeling tools include:

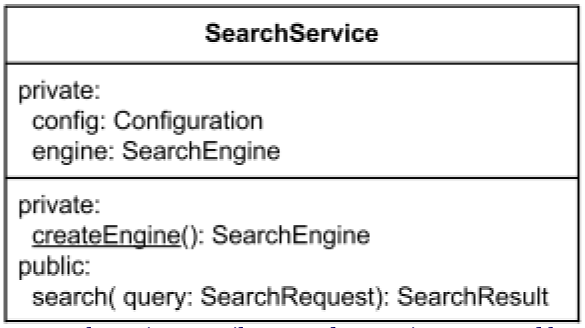
2.1.3.1. Class Diagrams

Class Diagram models class structure and contents using design elements such as classes, packages and objects. It also displays relationships such as containment, inheritance, associations and others.

A class is a category or group of things that have similar attributes and common behaviors.

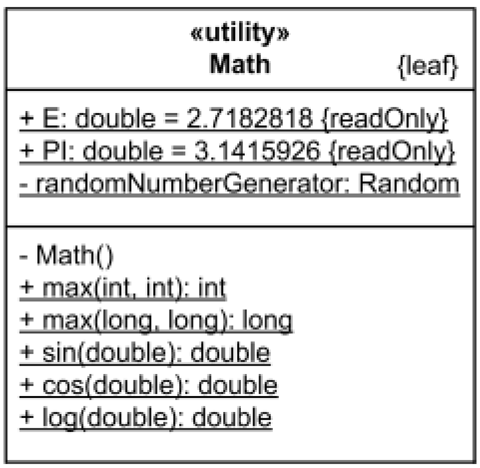
Class diagrams describe three different perspectives when designing a system, conceptual, specification, and implementation. These perspectives become evident as the diagram is created and help solidify the design.

Notation	Description
Class	
 ***Class*** *Customer - details suppressed.*	A **class** is a **classifier** which describes a set of objects that share the same **features** **constraints** **semantics (meaning)**. A class is shown as a solid-outline rectangle containing the class name, and optionally with compartments separated by horizontal lines containing features or other members of the classifier.
 ***Class*** *SearchService - analysis level details*	When class is shown with three **compartments**, the middle compartment holds a list of **attributes** and the bottom compartment holds a list of **operations**. Attributes and operations should be **left justified** in **plain**face, with the first letter of the names in **lower** case.
 ***Class*** *SearchService - implementation level details. The createEngine is* ***static*** *operation.*	Middle compartment holds attributes and the bottom one holds operations.



Class *SearchService - attributes and operations grouped by visibility.*

Attributes or operations may be grouped by **visibility**. A visibility keyword or symbol in this case can be given once for multiple features with the same visibility.



Math is ***utility*** *class - having static attributes and operations (underlined)*

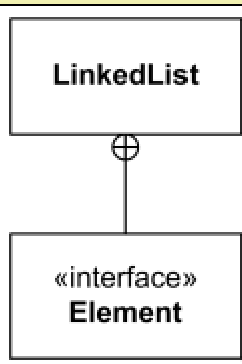
Utility is class that has only class scoped **static attributes and operations**. As such, utility class usually has no instances.

Abstract Class

Class *SearchRequest is* ***abstract class****.*

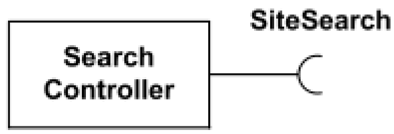
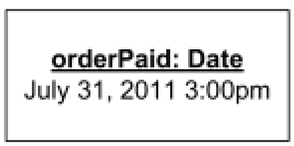
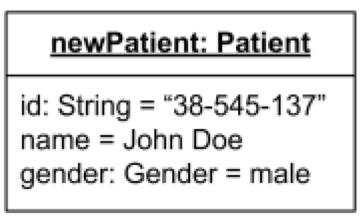
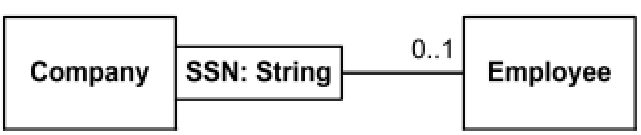
Abstract class was defined in **UML 1.4.2** as class that can't be directly instantiated. No object may be a direct instance of an abstract class.
UML 2.4 mentions abstract class but provides no definition. We may assume that in UML 2.x **abstract class**does not have complete declaration and "typically" can not be instantiated.
The name of an **abstract class** is shown in **italics**.

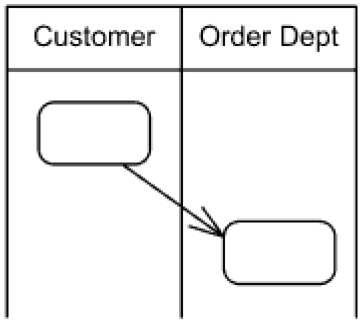
Nested Classifiers



Class LinkedList is nesting the Element interface. The Element is

A class or interface could be used as a **namespace** for various **classifiers** including other classes, interfaces, use cases, etc. This **nesting of classifier** limits the visibility of the classifier defined in the class to the scope of the namespace of the containing class or interface.
In obsolete **UML 1.4.2** a declaring class and a class in its namespace may be shown connected by a line, with an "anchor" icon on the end connected to a declaring class (namespace). An anchor icon is a cross inside a circle.
UML 2.x specifications provide no explicit notation for the nesting by classes. Note, that UML's 1.4 "anchor" notation is still used in one example in UML 2.4.x for **packages** as an "alternative membership notation".

in scope of the LinkedList namespace.	
 ***Interface** SiteSearch is **used** (required) by SearchController.*	The **usage dependency** from a classifier to an interface is shown by representing the interface by a half-circle or socket, labeled with the name of the interface, attached by a solid line to the classifier that **requires** this interface.
Object	
 *Anonymous instance of the Customer class.*	**Object** is an **instance** of a **class** or an **interface**. Object is not a UML element by itself. Objects are rendered as**instance specifications**, usually on **object diagrams**. Class **instance** (object) could have no name, be anonymous.
 *Instance newPatient of the unnamed or unknown class.*	In some cases, class of the instance is unknown or not specified. When instance name is also not provided, the notation for such an anonymous instance of an unnamed classifier is simply underlined colon - :.
 *Instance orderPaid of the Date class has value July 31, 2011 3:00 pm.*	If an instance has some value, the value specification is shown either after an equal sign ('=') following the instance name, or without the equal sign below the name.
 *Instance newPatient of the Patient class has slots with values specified.*	**Slots** are shown as **structural features** with the feature name followed by an equal sign ('=') and a value specification. Type (classifier) of the feature could be also shown.
Association Qualifier	
 *Given a company and a social security number (SSN) at most one employee could be found.*	A **qualifier** is a property which defines a partition of the set of associated instances with respect to an instance at the qualified end. Qualifiers are used to model **hash maps** in Java, **dictionaries** in C#, **index tables**, etc. where fast access to linked object(s) is provided using qualifier as a hash key, search argument or index. A **qualifier** is shown as a small rectangle attached to the end of an association between the final path segment and the symbol of the classifier that it connects to. The qualifier rectangle is part of the **association**, not part of the classifier. A qualifier may not be suppressed. In the case in which the target multiplicity is 0..1, the qualifier value is unique with respect to the qualified object, and designates at most one associated object.

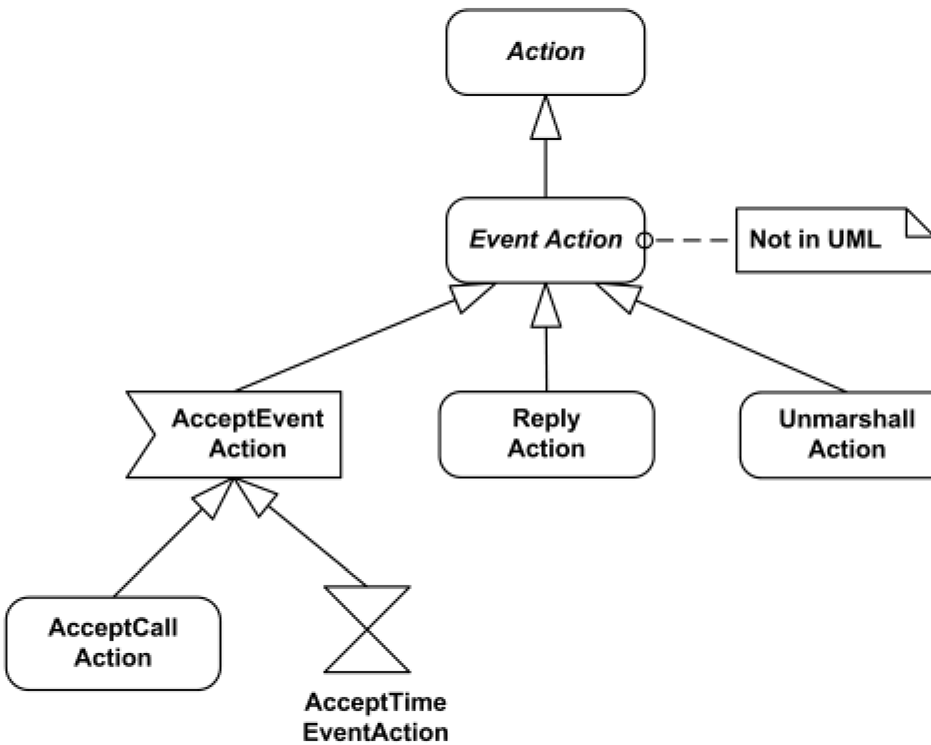


Activity partitions Customer and Order Dept as vertical swimlanes.

Action

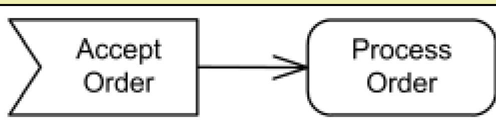
Actions are notated as round-cornered rectangles. The **name** of the action or **other description** of it may appear in the symbol.

Event Action



***Event actions** overview diagram.*

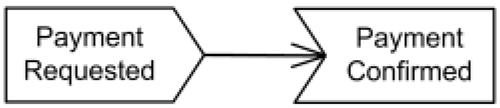
Accept Event Action

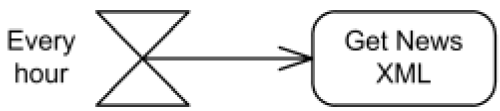


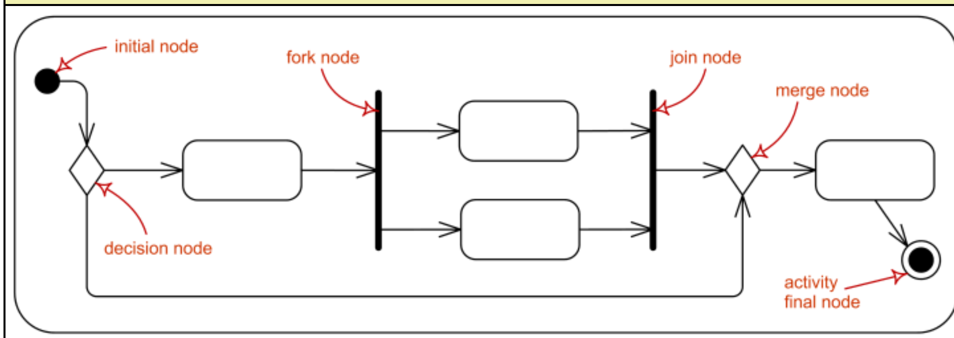
*Acceptance of the **Accept Order** signal causes an invocation of a **Process Order** action. The **accept event action** Accept Order is enabled on entry to the*

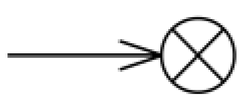
Accept event action is notated with a concave pentagon.

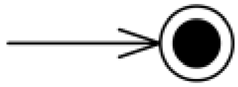
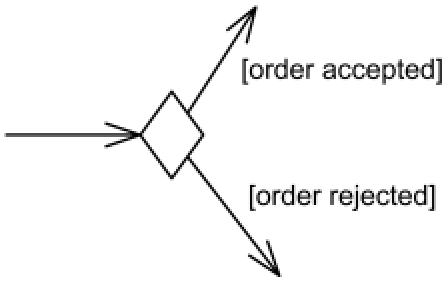
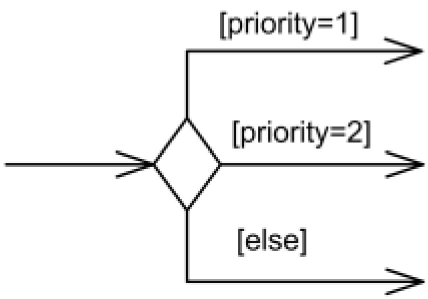
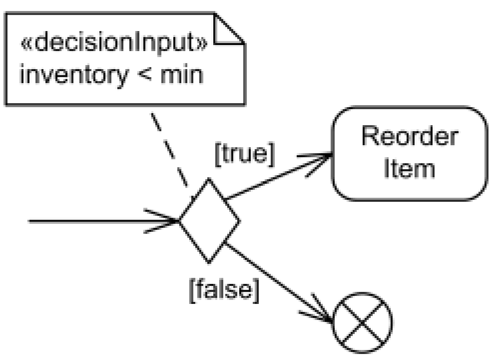
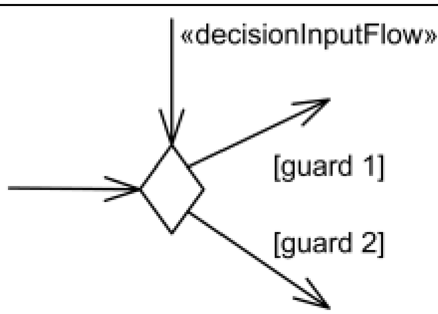
If an accept event action has **no incoming edges**, then the action starts when the containing activity or structured node does, whichever most immediately contains the action. In addition, an accept event action with no incoming edges remains enabled after it accepts an event. It does not terminate after

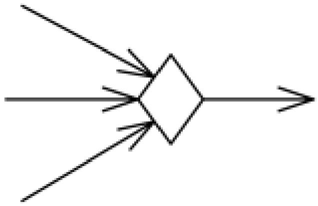
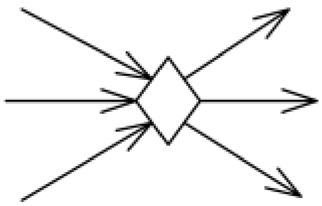
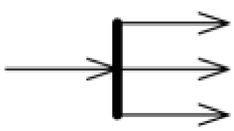
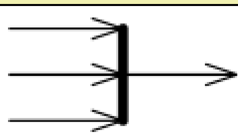
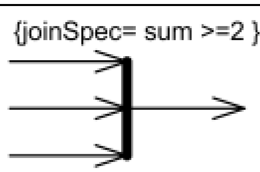
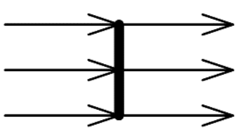
<table>
<tr><td>activity containing it, thereforeno input arrow is shown.</td><td>accepting an event and outputting a value, but continues to wait for other events.
An action whose trigger is a signal event is informally called accept signal action. It corresponds to send signal action.</td></tr>
<tr><td colspan="2">Accept Event Action with incoming edges</td></tr>
<tr><td>

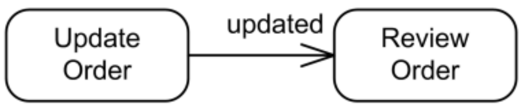
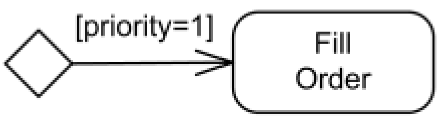
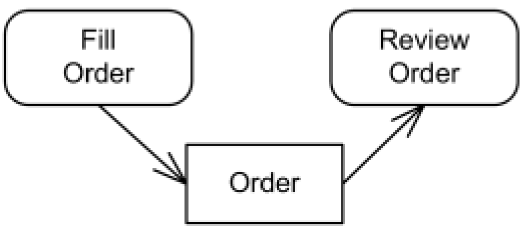
Payment Requested signal is sent. The activity then waits to receive Payment Confirmed signal.
Acceptance of thePayment Confirmed is enabled only after the request for payment is sent; no confirmation is accepted until then.</td><td>Accept event action could have incoming edges. In this case the action starts after the previous action completes.</td></tr>
<tr><td colspan="2">Wait Time Action</td></tr>
<tr><td>

The Every Hour accept time event action generates an output every hour. There are no incoming edges to this time event action, so it is enabled as long as its containing activity or structured node is.</td><td>If the event is a time event occurrence, the result value contains the time at which the occurrence happened. Such an action is informally called a wait time action.
Accept time event action (aka informal: wait time action) is notated with an hour glass.</td></tr>
<tr><td colspan="2">Control Nodes</td></tr>
<tr><td colspan="2">

Activity control nodes overview.</td></tr>
<tr><td colspan="2">Initial Node</td></tr>
<tr><td>
Activity initial node.</td><td>Initial node is a control node at which flow starts when the activity is invoked. Activity may have more than one initial node. Initial nodes are shown as a small solid circle.</td></tr>
<tr><td colspan="2">Flow Final Node</td></tr>
<tr><td>
Flow final node.</td><td>Flow final node is a control final node that terminates a flow. The notation for flow final node is small circle with X inside.</td></tr>
<tr><td colspan="2">Activity Final Node</td></tr>
</table>

 *Activity final node.*	**Activity final node** is a control final node that stops all flows in an **activity**. Activity final is new in UML 2.0. Activity final nodes are shown as a solid circle with a hollow circle inside. It can be thought of as a goal notated as "bull's eye," or target.
Decision	
 ***Decision node** with two **outgoing edges** with guards.*	**Decision node** is a **control node** that accepts tokens on one or two **incoming edges** and selects one **outgoing edge** from one or more outgoing flows. The notation for a decision node is a diamond-shaped symbol.
 ***Decision node** with three **outgoing edges** and [else] guard.*	For **decision points**, a predefined guard "**else**" may be defined for at most one outgoing edge.
 ***Decision node** with **decision input behavior**.*	**Decision** can have **decision input behavior**. Decision input behaviors were introduced in UML to avoid redundant recalculations in guards. In this case each data token is passed to the **behavior** before guards are evaluated on the outgoing edges. The output of the behavior is available to each guard. **Decision input behavior** is specified by the keyword **«decisionInput»** and some decision behavior or condition placed in a **note symbol**, and attached to the appropriate decision node.
	**Decision** may also have **decision input flow**. In this case the tokens offered on the decision input flow that are made available to the guard on each outgoing edge determine whether the offer on the regular incoming edge is passed along that outgoing edge. A **decision input flow** is specified by the keyword **«decisionInputFlow»** annotating that flow.

Decision node *with* ***decision input flow****.*	
Merge	
 ***Merge node*** *with three incoming edges and a single outgoing edge.*	**Merge node** is a control node that brings together multiple incoming **alternate flows** to accept single outgoing flow. There is no joining of tokens. Merge **should not** be used to synchronize **concurrent flows**. The notation for a merge node is a diamond-shaped symbol with two or more edges entering it and a single activity edge leaving it.
Merge and decision combined	
 ***Merge node*** *and* ***decision node*** *combined.*	The functionality of **merge node** and **decision node** can be combined by using the same node symbol.
Fork	
 ***Fork node*** *with a single activity edge entering it, and three edges leaving it.*	**Fork node** is a control node that has one incoming edge and multiple outgoing edges and is used to split incoming flow into multiple **concurrent** flows. The notation for a **fork node** is a line segment with a single activity edge entering it, and two or more edges leaving it.
Join Node	
 ***Join node*** *with three activity edges entering it, and a single edge leaving it.*	**Join node** is a control node that has multiple incoming edges and one outgoing edge and is used to synchronize incoming concurrent flows. The notation for a join node is a line segment with several activity edges entering it, and only one edge leaving it.
 ***Join node*** *with* ***join specification*** *shown in curly braces.*	Join specifications are shown in curly braces near the join node as **joinSpec=...**.
Join and fork combined	
 *Combined* ***join node*** *and* ***fork node****.*	The functionality of **join node** and **fork node** can be combined by using the same node symbol.

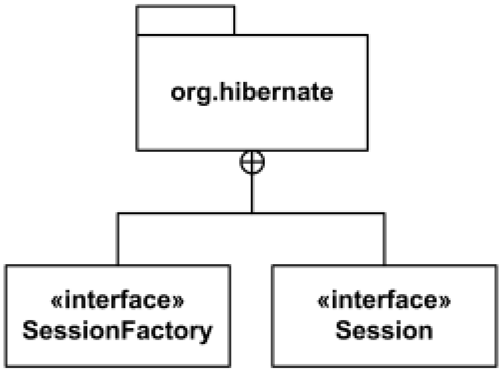
Activity Edge	
 ***Activity edge*** *connects Fill Order and Review Order.*	**Activity edge** could be **control edge** or **data flow edge** (aka **object flow edge**). Both are notated by an open arrowhead line connecting activity nodes.
 ***Activity edge "updated"*** *connects Update Order and Review Order.*	**Activity edge** can be named, however, edges are not required to have unique names within an activity. If the edge has a name, it is notated near the arrow.
 *Fill Order when priority is 1*	The **guard** of the activity edge is shown in square brackets that contain the guard. The guard must evaluate to true for every token that is offered to pass along the edge.
Object Flow Edge	
 *Data flow of Orders between Fill Order and Review Order actions*	**Object flow edges** are activity edges used to show data flow between action nodes. Object flow edges have no specific notation.

2.1.3.6. State Machine Diagrams

State Machine Diagram displays the sequences of states that an object of an interaction goes through during its life in response to received stimuli, together with its responses and actions. At any given time, an object is in particular state. State diagrams represent these states and their changes during time. Every state diagram starts with symbol that represents start state, and ends with symbol for the end state. For example every person can be a newborn, infant, child, adolescent, teenager or adult. State diagrams are used to describe the behavior of a system.
State diagrams describe all of the possible states of an object as events occur. Each diagram usually represents objects of a single class and track the different states of its objects through the system.

When to Use: State Diagrams

Use state diagrams to demonstrate the behavior of an object through many use cases of the system. Only use state diagrams for classes where it is necessary to understand the behavior of the object through the entire system. Not all classes will require a state diagram and state diagrams are not useful for describing the collaboration of all objects in a use case. State



Package org.hibernate contains interfaces SessionFactory and Session.

The elements that can be referred to within a package using **non-qualified** names are:

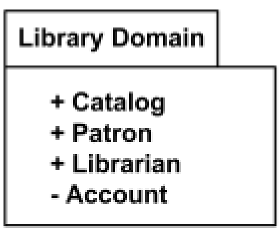
owned elements,

imported elements, and

elements in enclosing (outer) namespaces.

Owned and imported elements may have a **visibility** that determines whether they are available outside the package.

If an element that is owned by a package has **visibility**, it could be only **public** or **private** visibility. Protected or package visibility is not allowed. The visibility of a package element may be indicated by preceding the name of the element by a visibility symbol ("+" for public and "-" for private).



All elements of Library Domain package are public except for Account

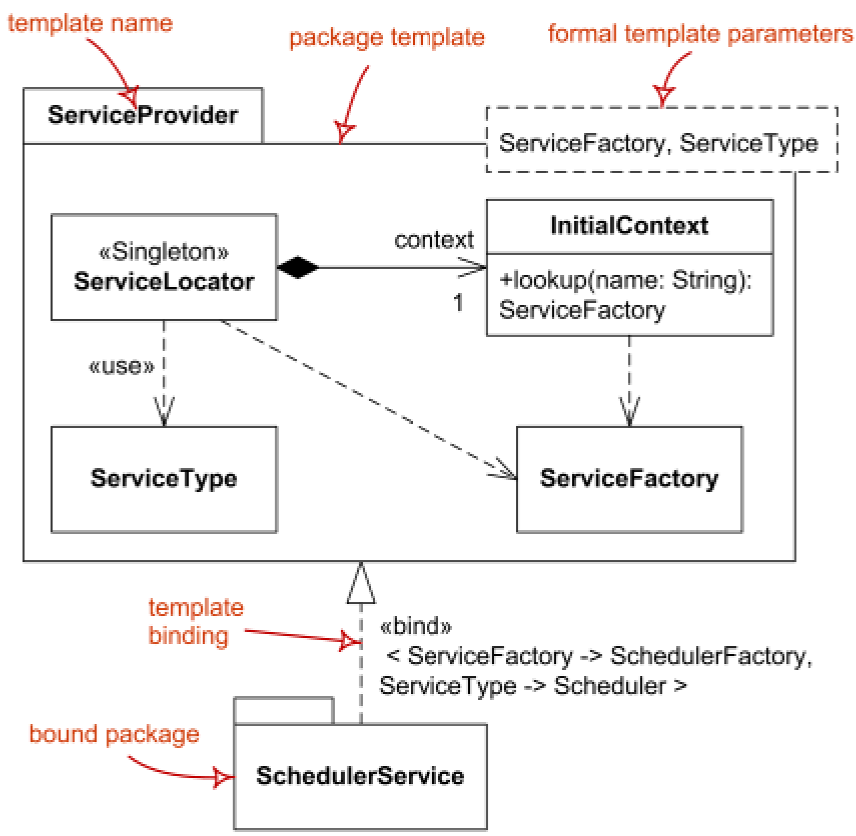
The public elements of a package are always accessible outside the package through the use of **qualified** names.

Package Template

Package can be used as a template for other packages. Note, that [UML 2.4.1 Specification] inconsistently calls it both **package template** and **template package**.

Packageable element can be used as a template parameter. A package template parameter may refer to any element owned or used by the package template, or templates nested within it.

A package may be bound to one or more template packages. When several bindings are applied the result of bindings is produced by taking the intermediate results and merging them into the combined result using package merge.



Package template Service Provider and bound package Scheduler Service.

Package Element

Packageable element is a named element that may be **owned** directly by a package.

Some examples of **packageable elements** are:

- Type
- classifier (--> type)
- class (--> classifier)
- use case (--> classifier)
- component (--> classifier)
- package
- constraint
- dependency
- event

Packageable element by itself has no notation, see specific subclasses.

Element Import

Element import is a directed relationship between an importing namespace and imported packageable element. It allows the element to be referenced using its name without a qualifier. An element import is used to selectively import individual elements without relying on a package import.

The name of the packageable element or its alias is to be added to the namespace of the importing namespace. It works by reference, which means that it is not possible to add features to the element import itself, but it is possible to modify the referenced element in the namespace from which it was imported.

It is possible to control whether the imported element can be further imported by other namespaces using either element or package imports. The visibility of the element import may be either the same or more restricted than that of the imported element. The visibility of an element import is either public or private.

The default visibility is the same as that of the imported element. If the imported element does not have a visibility, it is possible to add visibility to the element import. Default value is public.

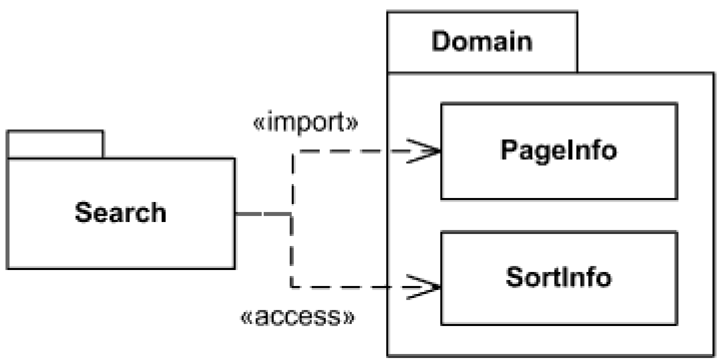
In case of a name clash with an outer name (an element that is defined in an enclosing namespace is available using its unqualified name in enclosed namespaces) in the importing namespace, the outer name is hidden by an element import, and the unqualified name refers to the imported element. The outer name can be accessed using its qualified name.

If more than one element with the same name would be imported to a namespace as a consequence of element imports or package imports, the elements are not added to the importing namespace and the names of those elements must be qualified in order to be used in that namespace. If the name of an imported element is the same as the name of an element owned by the importing namespace, that element is not added to the importing namespace and the name of that element must be qualified in order to be used.

Alias specifies the name that should be added to the namespace of the importing package in lieu of the name of the imported packagable element. The aliased name must not clash with any other member name in the importing package. By default, no alias is used.

An element import is shown using a dashed arrow with an open arrowhead from the importing namespace to the imported element. Note, that though it looks exactly
like dependency and usage relationships, it is a completely separate directed relationship.

The keyword «import» is shown near the dashed arrow if the visibility of import is public while the keyword «access» is used to indicate private visibility. If the imported element is not a package, the keyword may optionally be preceded by element, i.e., «element import».



Public import of PageInfo element and private import of SortInfo element from Domain package.

Elements that becomes available for use in an importing package through an element import may have a distinct color or be dimmed to indicate that they cannot be modified.

The aliased name may be shown after or below the keyword «import».

As an alternative to the dashed arrow, it is allowed to show an element import by having a text that uniquely identifies the imported element within curly brackets either below or after the name of the namespace. The syntax in this case could be described as (note, this is my modified version of the syntax):

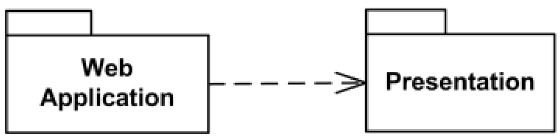
element-import ::= '{' ('element import' | 'element access') *qualified-name* ['as' *alias*] '}'

Package Import

Package import is a directed relationship between an importing namespace and imported package that allows the use of unqualified names to refer to the package members from the other namespace.

Importing namespace adds the names of the members of the imported package to its own namespace. Conceptually, a package import is equivalent to having an element to each individual member of the imported namespace, unless there is already a separately-defined element import.

A package import is shown using a dashed arrow with an open arrowhead from the importing namespace to the imported package.

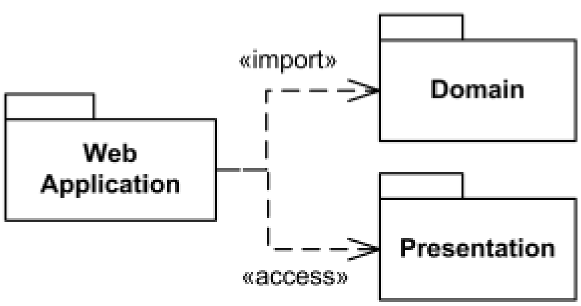


Web Application imports Presentation package.

Note, that though it looks exactly like dependency and usage relationships, it is a completely separate directed relationship.

The visibility of a package import could be either public or private. If the package import is public, the imported elements will be added to the namespace and made visible outside the namespace, while if it is private they will still be added to the namespace but without being visible outside.

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.



Private import of Presentation package and public import of Domain package

As an alternative to the dashed arrow, it is possible to show an package import by having a text that uniquely identifies the imported package within curly brackets either below or after the name of the namespace. The syntax in this case could be described as (note, this is my modified version of the syntax):

Package-import::= '{' ('import' | 'access') *qualified-name* '}'

Elements that becomes available for use in an importing package through a package import may have a distinct color or be dimmed to indicate that they cannot be modified.

Package Merge

A package merge is a directed relationship between two packages that indicates that content of one package is extended by the contents of another package.

Package merge is similar to generalization in the sense that the source element conceptually adds the characteristics of the target element to its own characteristics resulting in an element that combines the characteristics of both.

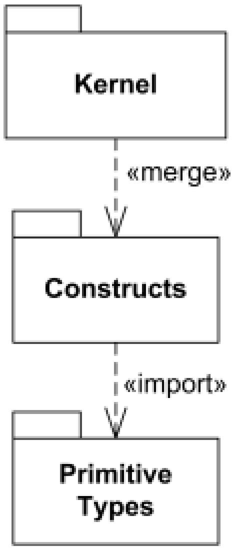
Package merge can be viewed as an operation that takes the contents of two packages and produces a new package that combines the contents of the packages involved in the merge.

This mechanism should be used when elements defined in different packages have the same name and are intended to represent the same concept. Most often it is used to provide different definitions of a given concept for different purposes, starting from a common base definition.

A given base concept is extended in increments, with each increment defined in a separate merged package. By selecting which increments to merge, it is possible to obtain a custom definition of a concept for a specific end.

Package merge is particularly useful in meta-modeling and is extensively used in the definition of the UML meta model.

Package merge is shown using a dashed line with an open arrowhead pointing from the receiving package to the merged package. Keyword «merge» is shown near the dashed line.



UML Kernel package merges constructs package which imports Primitive Types.

UML model describes what a system is supposed to do. It doesn't tell how to implement the system.

In summary an UML tool allows knowing how to build Use Case, Sequence, State and collaboration diagrams. UML uses unified technique for the design of software and understanding by all types of users in the development environment.

For More UML Diagrams and Information Refer to Appendix-A .

Summary

- Object Oriented Modeling method defines object, class, attributes, operations, messages, information hiding, inheritance and polymorphism.
- An UML tool allows knowing how to build Use Case, Sequence, State and collaboration diagrams.
- UML uses unified technique for the design of software and understanding by all types of users in the development environment.

Assignments:

1. Explain different diagrams for the design of software development?
2. What is UML? Where do we use UML and for what do we use UML?

Laboratory Exercises (Refer to Lab Manual for the help to do Lab Exercises)

1. Draw a neat State diagram and Sequence diagram for a Bank system?
2. Draw a neat Collaboration diagram for one single process in Railway Reservation system?

References:

[1]: http://www.scribd.com/doc/36554703/OOAD

[2]: http://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node43.html

[3]: http://www.ustudy.in/node/6448

[4]: Dennis A, Wixom B, Tegarden D. "Systems Analysis and Design with UML" , 2nd Edition, John Wiley & Sons

[5]: Fowler M, "UML Distilled", 2nd Edition, PHI/Pearson Education, 2002.

[6]: Schach Stephen R., "Introduction to Object Oriented Analysis and Design" 2nd Edition, Tata McGraw-Hill, 2003.

[7]: http://www.uml-diagrams.org/package-diagrams.html