Course Material

On

Object Oriented Modeling and Design

Module-1

1.1 Introduction to UML

The Unified Modeling Language (UML) is a standardized visual language designed for modeling the structure and behavior of software systems. It provides a comprehensive set of diagrams and notation techniques to facilitate the visualization, specification, construction, and documentation of software artifacts.

UML is distinct from programming languages, as it focuses on creating detailed blueprints of software systems rather than executing code. It is versatile and can be applied to a variety of system types, from enterprise-level applications and web-based systems to real-time embedded systems, making it a crucial tool for developers and analysts in designing and understanding complex software projects.

1.2 Importance of modeling

"A model is a simplification of reality."

The statement "A model is a simplification of reality" underscores the fundamental purpose of a model: to distill and represent complex, real-world systems or phenomena into more manageable and understandable forms.

Models are abstractions that focus on essential aspects of reality while omitting or simplifying less critical details. This simplification allows individuals to analyze, study, and make decisions about complex systems without being overwhelmed by their full complexity.

"Modeling is a proven and well-accepted engineering technique."

The statement "Modeling is a proven and well-accepted engineering technique" highlights that modeling has been established as a reliable and widely adopted method in engineering for managing complexity and enhancing understanding across various disciplines.

Why do we model?

"We build models so that we can better understand the system we are developing."

Through modeling, we achieve four aims.

1. Models help us to **visualize** a system as it is or as we want it to be.
2. Models permit us to **specify** the structure or behavior of a system.
3. Models give us a template that guides us in **constructing** a system.
4. Models **document** the decisions we have made.

Why do we model the large and complex systems?

"We build models of complex systems because we cannot comprehend such a system in its entirety."

1.3 Principles of modeling

The use of modeling has a rich history in all the engineering disciplines. That experience suggests four basic principles of modeling.

1. The choice of what models to create has a profound influence on how a Problem is attacked and how a solution is shaped.
2. Every model may be expressed at different levels of precision.
3. The best models are connected to reality.
4. No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

1.4 Object-Oriented Modeling

In software, there are several ways to approach a model. The two most common ways are from an algorithmic perspective and from an object-oriented perspective.

- *Algorithmic perspective*: Is the traditional view of software development. In this approach, the main building block of all software is the procedure or function.

  This view leads developers to focus on issues of control and the decomposition of larger algorithms into smaller ones.

  As requirements change and the system grows, systems built with an algorithmic focus turn out to be very hard to maintain.

  Language dependent

  Algorithm/Process dependent

- *Object-oriented perspective*: Is the contemporary view of software development. With Object-oriented perspective (diagrammatic representation) it is very easy to visualize the system.

No language dependent

No process dependent

1.5 Conceptual model of the UML

To understand conceptual model of UML first we need to clarify what is a conceptual model? And why a conceptual model is at all required?

- A conceptual model can be defined as a model which is made of concepts and their relationships.
- A conceptual model is the first step before drawing a UML diagram. It helps to understand the entities in the real world and how they interact with each other.

As UML describes the real time systems it is very important to make a conceptual model and then proceed gradually. Conceptual model of UML can be mastered by learning the following three major elements:

- UML building blocks
- Rules to connect the building blocks
- Common mechanisms of UML

- **Building blocks of the UML:** The vocabulary of the UML encompasses three kinds of building:Things, Relationships and Diagrams.

1. Things are the abstractions that are first-class citizens in a model; relationships tie these things together; diagrams group interesting collections of things.

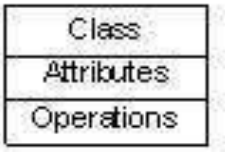
Things are the most important building blocks of UML. Things can be:

i. Structural things
ii. Behavioral things
iii. Grouping things
iv. An notational things

These things are the basic object-oriented building blocks of the UML. You use them to write well-formed models.

i. **Structural Things:** *Structural things* are the **nouns** of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things.

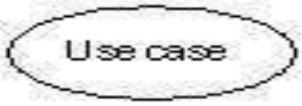
Class: a *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces. Graphically, a class is rendered as a rectangle.



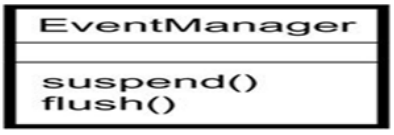
Interface: Interface defines a set of operations which specify the responsibility of a class or component. An interface therefore describes the externally visible behavior of that element. An interface might represent the complete behavior of a class or component or only a part of that behavior.

Collaboration: C*ollaboration* defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. Therefore, collaborations have structural, as well as behavioral, dimensions. A given class might participate in several collaborations.

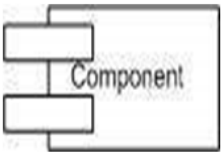
Use case: Use case represents a set of actions performed by a system for a specific goal. A use case is used to structure the behavioral things in a model



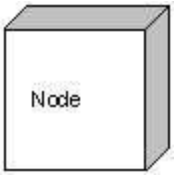
Active Class: an *active class* is a class whose objects own one or more processes or threads and therefore can initiate control activity. An active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements.



Component: A *component* is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.

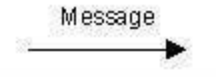


Node: N*ode* is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability.



ii. Behavioral things: A behavioral thing consists of the dynamic parts of UML models. Following are the behavioral things:

Interaction: Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.



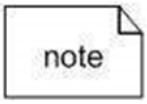
State machine: State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change.

iii. Grouping things: Grouping things are the organizational parts of UML models. There is only one grouping thing available:

Package: Package is the only one grouping thing available for gathering structural and behavioral things.

iv. Annotational things: Annotational things are the explanatory parts of UML models. These are the comments we may apply to illuminate, describe and capture remarks of UML model elements.

Note: is the only one Annotational thing available. A note is used to render comments, constraints etc of an UML element.

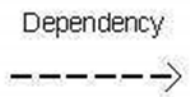


2. Relationship: It is another most important building block of UML. It shows how elements are associated with each other and this association describes the functionality of an application.

There are four kinds of relationships available.

Dependency:

Dependency is a semantic relationship between two things in which change in one element may affects the other one.



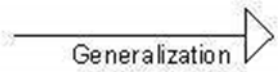
Association:

Association is structural relationship that describes a set of links that connects elements of an UML model. It also describes how many objects are taking part in that relationship. We can have unidirectional association as well as bi-directional association

Generalization:

Generalization is a specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). In this way, the child shares the structure and the behavior of the parent

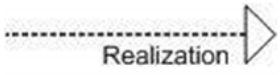
Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent.



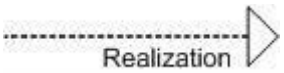
Realization:

Realization is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out.

Graphically, a realization relationship is rendered as a cross between a generalization and a dependency relationship



Graphically, a realization relationship is rendered as a cross between a generalization and a dependency relationship



3. UML Diagrams

A *diagram* is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).

UML includes the following nine diagrams and the details are described in the following chapters.

- Class diagram
- Object diagram
- Use case diagram
- Sequence diagram
- Collaboration diagram
- Activity diagram
- Statechart diagram
- Deployment diagram
- Component diagram

➢ **Rules of the UML:** The UML's building blocks can't simply be thrown together in a random fashion. Like any language, the UML has a number of rules that specify what a well-formed model should look like. A *well-formed model* is one that is semantically self-consistent and in harmony with all its related models.

The UML has semantic rules for

- Name : What you can call things, relationships, and diagrams
- Scope: The context that gives specific meaning to a name
- Visibility: How those names can be seen and used by others
- Integrity: How things properly and consistently relate to one another
- Execution: What it means to run or simulate a dynamic model

Elided: Certain elements are hidden to simplify the view

Incomplete: Certain elements may be missing

Inconsistent: The integrity of the model is not guaranteed

The rules of the UML encourage youbut do not force youto address the most important analysis, design, and implementation questions that push such models to become well-formed over time

- **Common Mechanisms in the UML:**

A building is made simpler and more harmonious by the conformance to a pattern of common features. A house may be built in the Victorian or French country style largely by using certain architectural patterns that define those styles. The same is true of the

UML. It is made simpler by the presence of four common mechanisms that apply consistently throughout the language.

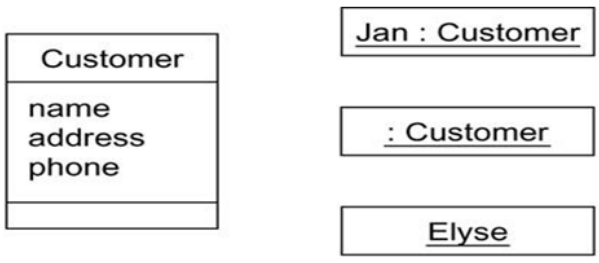
1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

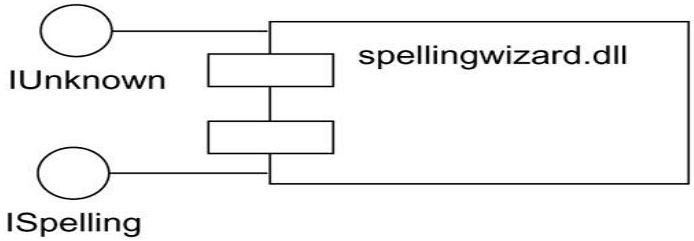
Specifications: The UML is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and semantics of that building block

Adornments: Most elements in the UML have a unique and direct graphical notation that provides a visual representation of the most important aspects of the element.

Common Divisions: In modeling object-oriented systems, the world often gets divided in at least a couple of ways.

- There is the division of class and object. A class is an abstraction; an object is one concrete manifestation of that abstraction.





Extensibility Mechanisms: The UML provides a standard language for writing software blueprints, but it is not possible for one closed language to ever be sufficient to express all possible nuances of all models across all domains across all time.

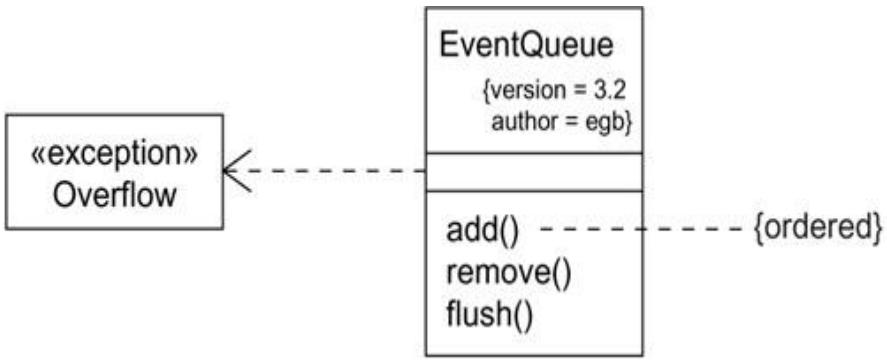
The UML's extensibility mechanisms include:

- Stereotypes: A *stereotype* extends the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem.

  Represents with << >>

- Tagged values: A *tagged value* extends the properties of a UML building block, allowing you to create new information in that element's specification. Represents with { }

- Constraints: A *constraint* extends the semantics of a UML building block, allowing you to add new rules or modify existing ones.



Visual Paradigm software tool for UML

Visual Paradigm is a software tool designed for software development teams to model business information system and manage development processes. Visual Paradigm supports key industry modeling languages and standards such as Unified Modeling Language (UML), SoaML, BPMN, XMI, etc.

Installing Visual Paradigm on Windows

Using installer (.exe)

1. Execute the downloaded Visual Paradigm installer file.
2. Click Next to proceed to the License Agreement page.
3. Read through the license agreement carefully.
4. Specify the directory for installing Visual Paradigm.
5. Specify the name of the Start Menu folder that will be used to store the shortcuts. Keep **Create shortcuts for all users** checked if you want the shortcut to be available in all the user accounts in the machine. Click **Next** to proceed.
6. In the **File Association** page, keep **Visual Paradigm Project (*.vpp)** checked if you want your system able to open the project file upon direct execution (i.e. double click). Click **Next** to start the file copying process.
7. Upon finishing, you can select whether to start Visual Paradigm or not. Keep **Visual Paradigm** selected and click **Finish** will run Visual Paradigm right away.

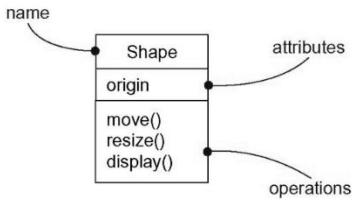
Perform the steps below to create a UML diagram in Visual Paradigm.

1. Select Diagram > New from the application toolbar.
2. In the New Diagram window, select Diagram.
3. Click Next.
4. Enter the diagram name and description. The Location field enables you to select a model to store the diagram.
5. Click OK.

Advanced Structural Modeling, Class, and Object Diagrams

Class

- A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics.
- A class implements one or more interfaces.
- The UML provides a graphical representation of class



Graphical Representation of a Class in UML

Terms and Concepts

Names

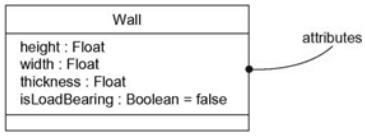
Every class must have a name that distinguishes it from other classes.
A name is a textual string that name alone is known as a simple name;
a path name is the class name prefixed by the name of the package in which that class lives.

Simple Name

Path Name

Attributes

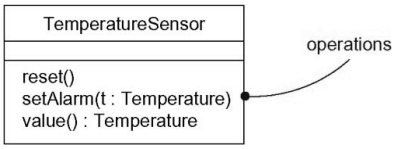
- An attribute is a named property of a class that describes a range of values that instances of the property may hold.
- A class may have any number of attributes or no attributes at all.
- An attribute represents some property of thing you are modeling that is shared by all objects of that class
- You can further specify an attribute by stating its class and possibly a default initial value



Attributes and Their Class

Operations

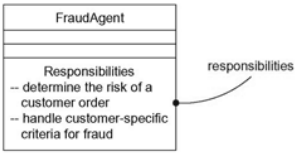
- An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior.
- A class may have any number of operations or no operations at all
- Graphically, operations are listed in a compartment just below the class attributes
- You can specify an operation by stating its signature, covering the name, type, and default value of all parameters and a return type



Organizing Attributes and Operations

Responsibilities

- A Responsibility is a contract or an obligation of a class
- When you model classes, a good starting point is to specify the responsibilities of the things in your vocabulary.
- A class may have any number of responsibilities, although, in practice, every well-structured class has at least one responsibility and at most just a handful.
- Graphically, responsibilities can be drawn in a separate compartment at the bottom of the class icon



Common Modeling Techniques

Modeling the Vocabulary of a System

- You'll use classes most commonly to model abstractions that are drawn from the problem you are trying to solve or from the technology you are using to implement a solution to that problem.
- They represent the things that are important to users and to implementers
- To model the vocabulary of a system

  - Identify those things that users or implementers use to describe the problem or solution.
  - Use CRC cards and use case-based analysis to help find these abstractions.
  - For each abstraction, identify a set of responsibilities.
  - Provide the attributes and operations that are needed to carry out these responsibilities for each class.

Modeling the Distribution of Responsibilities in a System

- Once you start modeling more than just a handful of classes, you will want to be sure that your abstractions provide a balanced set of responsibilities.
- To model the distribution of responsibilities in a system

  - Identify a set of classes that work together closely to carry out some behavior.
  - Identify a set of responsibilities for each of these classes.
  - Look at this set of classes as a whole, split classes that have too many responsibilities into smaller abstractions, collapse tiny classes that have trivial responsibilities into larger ones, and reallocate responsibilities so that each abstraction reasonably stands on its own.
  - Consider the ways in which those classes collaborate with one another, and redistribute their responsibilities accordingly so that no class within a collaboration does too much or too little.

Modeling Nonsoftware Things

- Sometimes, the things you model may never have an analog in software
- Your application might not have any software that represents them
- To model nonsoftware things

  - Model the thing you are abstracting as a class.
  - If you want to distinguish these things from the UML's defined building blocks, create a new building block by using stereotypes to specify these new semantics and to give a distinctive visual cue.
  - If the thing you are modeling is some kind of hardware that itself contains software, consider modeling it as a kind of node, as well, so that you can further expand on its structure.

Modeling Primitive Types

At the other extreme, the things you model may be drawn directly from the programming language you are using to implement a solution.
Typically, these abstractions involve primitive types, such as integers, characters, strings, and even enumeration types
To model primitive types

Relationships

In the UML, the ways that things can connect to one another, either logically or physically, are modeled as relationships.

Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships

In object-oriented modeling, there are three kinds of relationships that are most important:

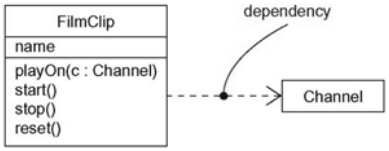
Dependencies
Generalizations
Associations

Dependency

A dependency is a using relationship that states that a change in specification of one thing may affect another thing that uses it but not necessarily the reverse.

Graphically dependency is rendered as a dashed directed line, directed to the thing being depended on.

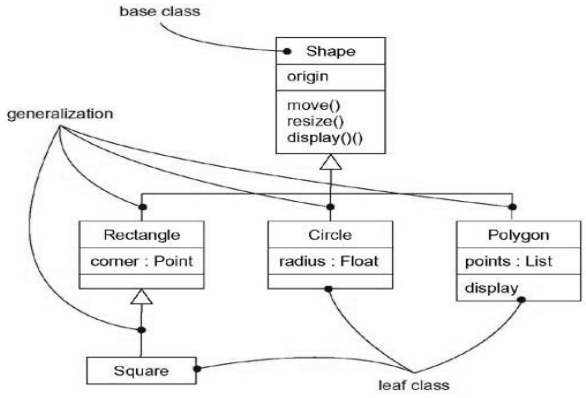
Most often, you will use dependencies in the context of classes to show that one class uses another class as an argument in the signature of an operation



Dependencies

Generalization

- A generalization is a relationship between a general thing (called the super class or parent)and a more specific kind of that thing (called the subclass or child).
- generalization means that the child is substitutable for the parent. A child inherits the properties of its parents, especially their attributes and operations
- An operation of a child that has the same signature as an operation in a parent overrides the operation of the parent; this is known as polymorphism.
- Graphically generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent



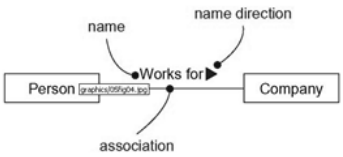
Generalization

Association

- An association is a structural relationship that specifies that objects of one thing are connected to objects of another
- An association that connects exactly two classes is called a binary association
- An associations that connect more than two classes; these are called n-ary associations.
- Graphically, an association is rendered as a solid line connecting the same or different classes.
- Beyond this basic form, there are four adornments that apply to associations

Name

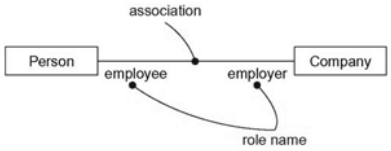
- An association can have a name, and you use that name to describe the nature of the relationship



Association Names

Role

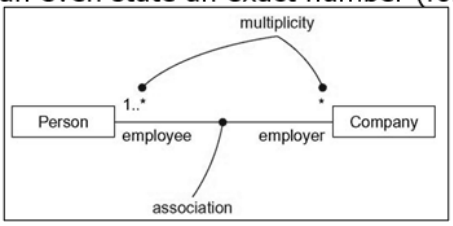
- When a class participates in an association, it has a specific role that it plays in that relationship;
- The same class can play the same or different roles in other associations.
- An instance of an association is called a link



Role Names

Multiplicity

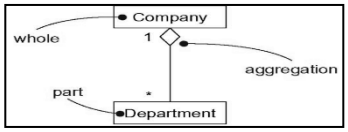
- In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association
- This "how many" is called the multiplicity of an association's role
- You can show a multiplicity of exactly one (1), zero or one (0..1), many (0..*), or one or more (1..*). You can even state an exact number (for example, 3).



Multiplicity

Aggregation

- Sometimes, you will want to model a "whole/part" relationship, in which one class represents a larger thing (the "whole"), which consists of smaller things (the "parts").
- This kind of relationship is called aggregation, which represents a "has-a" relationship, meaning that an object of the whole has objects of the part
- Aggregation is really just a special kind of association and is specified by adorning a plain association with an open diamond at the whole end



Aggregation

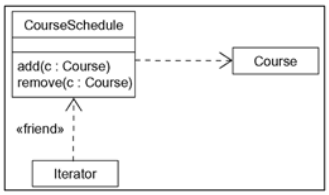
Common Modeling Techniques

Modeling Simple Dependencies

The most common kind of dependency relationship is the connection between a class that only uses another class as a parameter to an operation.

To model this using relationship

Create a dependency pointing from the class with the operation to the class used as a parameter in the operation.

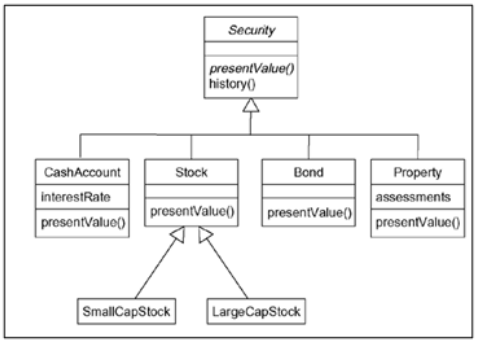


Dependency Relationships

Modeling Single Inheritance

To model inheritance relationships

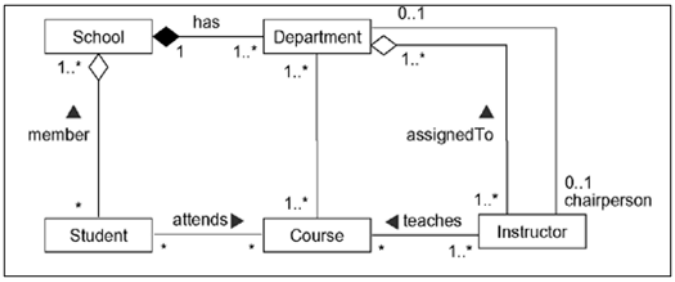
- Given a set of classes, look for responsibilities, attributes, and operations that are common to two or more classes.
- Elevate these common responsibilities, attributes, and operations to a more general class. If necessary, create a new class to which you can assign these
- Specify that the more-specific classes inherit from the more-general class by placing a generalization relationship that is drawn from each specialized class to its more-general parent.



Inheritance Relationships

Modeling Structural Relationships

- When you model with dependencies or generalization relationships, you are modeling classes that represent different levels of importance or different levels of abstraction
- Given a generalization relationship between two classes, the child inherits from its parent but the parent has no specific knowledge of its children.
- Dependency and generalization relationships are one-sided.
- Associations are, by default, bidirectional; you can limit their direction
- Given an association between two classes, both rely on the other in some way, and you can navigate in either direction
- An association specifies a structural path across which objects of the classes interact.



Structural Relationships

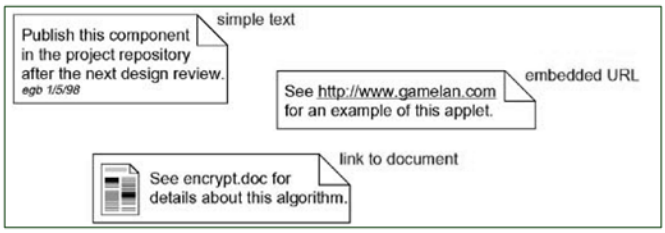
Common Mechanisms

Note

A note is a graphical symbol for rendering constraints or comments attached to an element or a collection of elements

Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment.

A note may contain any combination of text or graphics

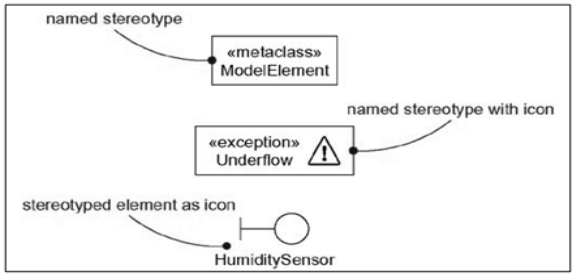


Notes

Stereotypes

A stereotype is an extension of the vocabulary of the UML, allowing you to create new kinds of building blocks similar to existing ones but specific to your problem.

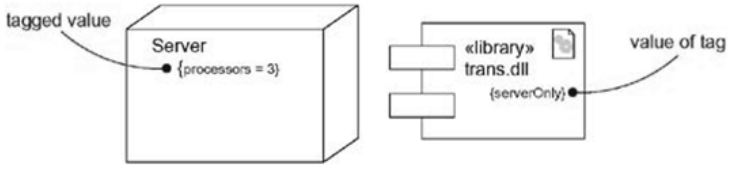
Graphically, a stereotype is rendered as a name enclosed by guillemets and placed above the name of another element



Stereotypes

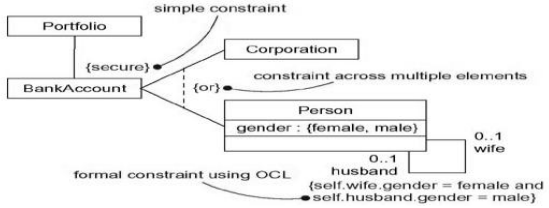
Tagged Values

- Every thing in the UML has its own set of properties: classes have names, attributes, and operations; associations have names and two or more ends (each with its own properties); and so on.
- With stereotypes, you can add new things to the UML; with tagged values, you can add new properties.
- A tagged value is not the same as a class attribute. Rather, you can think of a tagged value as metadata because its value applies to the element itself, not its instances.
- A tagged value is an extension of the properties of a UML element, allowing you to create new information in that element's specification.
- Graphically, a tagged value is rendered as a string enclosed by brackets and placed below the name of another element.
- In its simplest form, a tagged value is rendered as a string enclosed by brackets and placed below the name of another element.
- That string includes a name (the tag), a separator (the symbol =), and a value (of the tag).



Constraints

A constraint specifies conditions that must be held true for the model to be well-formed.
A constraint is rendered as a string enclosed by brackets and placed near the associated element
Graphically, a constraint is rendered as a string enclosed by brackets and placed near the associated element or connected to that element or elements by dependency relationships.



Diagram

- A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).
- A diagram is just a graphical projection into the elements that make up a system
- Each diagram provides a view into the elements that make up the system
- Typically, you'll view the static parts of a system using one of the four following diagrams.
  - Class diagram
  - Object diagram
  - Component diagram
  - Deployment diagram

You'll often use five additional diagrams to view the dynamic parts of a system.

- Use case diagram
- Sequence diagram
- Collaboration diagram
- Statechart diagram
- Activity diagram

Structural Diagrams

- The UML's four structural diagrams exist to visualize, specify, construct, and document the static aspects of a system.
- The UML's structural diagrams are roughly organized around the major groups of things you'll find when modeling a system.
  - Class diagram : Classes, interfaces, and collaborations
  - Object diagram : Objects
  - Component diagram : : Components
  - Deployment diagram : Nodes

Class Diagram

- We use class diagrams to illustrate the static design view of a system.
- Class diagrams are the most common diagram found in modeling object-oriented systems.
- A class diagram shows a set of classes, interfaces, and collaborations and their relationships.
- Class diagrams that include active classes are used to address the static process view of a system.

Object Diagram

- Object diagrams address the static design view or static process view of a system just as do class diagrams, but from the perspective of real or prototypical cases.
- An object diagram shows a set of objects and their relationships.
- You use object diagrams to illustrate data structures, the static snapshots of instances of the things found in class diagrams.

Component Diagram

- We use component diagrams to illustrate the static implementation view of a system.
- A component diagram shows a set of components and their relationships.
- Component diagrams are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

Deployment Diagram

- We use deployment diagrams to illustrate the static deployment view of an architecture.
- A deployment diagram shows a set of nodes and their relationships.
- Deployment diagrams are related to component diagrams in that a node typically encloses one or more components.

Behavioral Diagrams

- The UML's five behavioral diagrams are used to visualize, specify, construct, and document the dynamic aspects of a system.
- The UML's behavioral diagrams are roughly organized around the major ways you can model the dynamics of a system.
  - Use case diagram : Organizes the behaviors of the system
  - Sequence diagram : Focused on the time ordering of messages
  - Collaboration diagram : Focused on the structural organization of objects that send and receive messages
  - Statechart diagram : Focused on the changing state of a system driven by events
  - Activity diagram : Focused on the flow of control from activity to activity

Use Case Diagram

- A use case diagram shows a set of use cases and actors and their relationships.
- We apply use case diagrams to illustrate the static use case view of a system.
- Use case diagrams are especially important in organizing and modeling the behaviors of a system.

Sequence Diagram

- We use sequence diagrams to illustrate the dynamic view of a system.
- A sequence diagram is an interaction diagram that emphasizes the time ordering of messages.

Collaboration Diagram

- We use collaboration diagrams to illustrate the dynamic view of a system.
- A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.
- A collaboration diagram shows a set of objects, links among those objects, and messages sent and received by those objects.
- The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes.

Statechart Diagram

We use statechart diagrams to illustrate the dynamic view of a system.

They are especially important in modeling the behavior of an interface, class, or collaboration.

A statechart diagram shows a state machine, consisting of states, transitions, events, and activities.

Statechart diagrams emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

Activity Diagram

We use activity diagrams to illustrate the dynamic view of a system.

Activity diagrams are especially important in modeling the function of a system.

Activity diagrams emphasize the flow of control among objects.

An activity diagram shows the flow from activity to activity within a system.

An activity shows a set of activities, the sequential or branching flow from activity to activity, and objects that act and are acted upon.

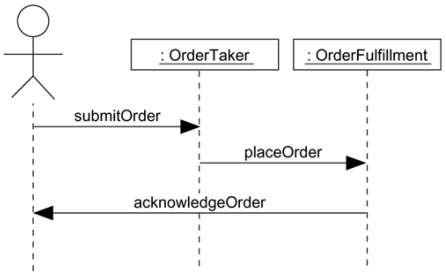
Common Modeling Techniques

Modeling Different Views of a System

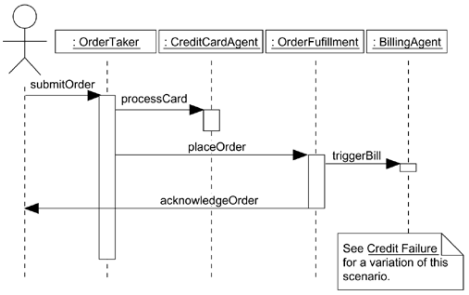
- When you model a system from different views, you are in effect constructing your system simultaneously from multiple dimensions
- To model a system from different views,
    - Decide which views you need to best express the architecture of your system and to expose the technical risks to your project
    - For each of these views, decide which artifacts you need to create to capture the essential details of that view.
    - As part of your process planning, decide which of these diagrams you'll want to put under some sort of formal or semi-formal control.
    - For example, if you are modeling a simple monolithic application that runs on a single machine, you might need only the following handful of diagrams

Modeling Different Levels of Abstraction

- To model a system at different levels of abstraction by presenting diagrams with different levels of detail,
    - Consider the needs of your readers, and start with a given model
    - If your reader is using the model to construct an implementation, she'll need diagrams that are at a lower level of abstraction which means that they'll need to reveal a lot of detail
    - If she is using the model to present a conceptual model to an end user, she'll need diagrams that are at a higher level of abstraction which means that they'll hide a lot of detail
    - Depending on where you land in this spectrum of low-to-high levels of abstraction, create a diagram at the right level of abstraction by hiding or revealing the following four categories of things from your model:



Interaction Diagram at a High Level of Abstraction



Interaction at a Low Level of Abstraction

Modeling Complex Views

- To model complex views,
  - First, convince yourself there's no meaningful way to present this information at a higher level of abstraction, perhaps eliding some parts of the diagram and retaining the detail in other parts.

  - If your diagram is still complex, print it in its entirety and hang it on a convenient large wall. You lose the interactivity an online version of the diagram brings, but you can step back from the diagram and study it for common patterns.

Module-2

Advanced Structural Modeling

Advanced Classes

Advanced classes in UML (Unified Modeling Language) encompass more complex features and relationships than the basic class diagrams. They help in modeling sophisticated systems with intricate structures and interactions. Here's an overview of advanced class modeling concepts in UML:

1. Generalization and Specialization

Generalization:

- **Definition**: Represents an inheritance relationship where a subclass inherits attributes and behaviors from a superclass.

- **Notation**: An arrow with a hollow triangle pointing from the subclass to the superclass.

- **Example**: A Vehicle class might be generalized into Car and Truck subclasses.