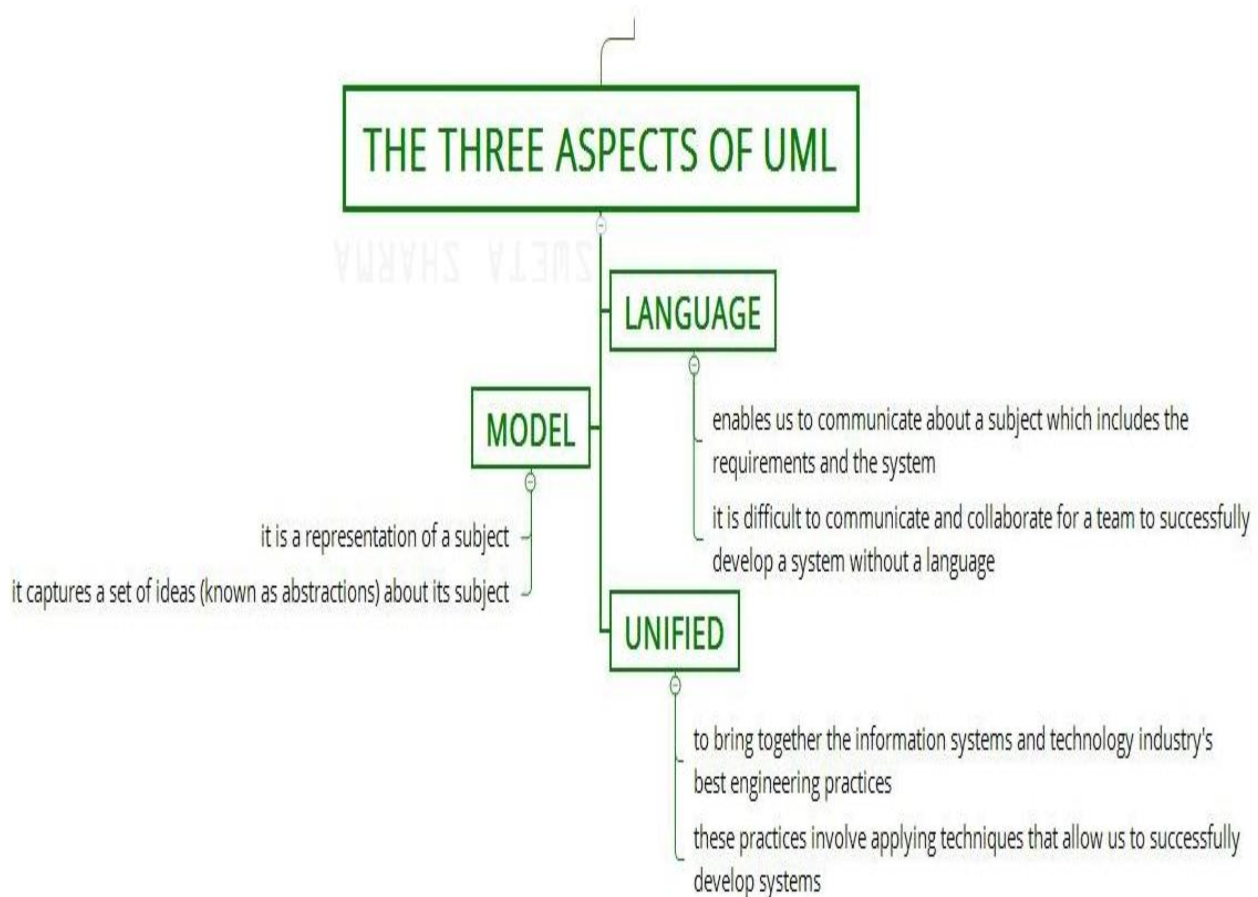
UML BASICS

The Unified Modeling Language (UML) is a standard visual language for describing and modelling software blueprints. The UML is more than just a graphical language. Stated formally, the UML is for: Visualizing, Specifying, Constructing, and Documenting.
The artifacts of a software-intensive system (particularly systems built using the object-oriented style).

Three Aspects of UML:



Language, Model, and Unified are the important aspect of UML as described in the map above.

1. **Language:**

It enables us to communicate about a subject which includes the requirements and the system.
It is difficult to communicate and collaborate for a team to successfully develop a system without a language.

2. **Model:**

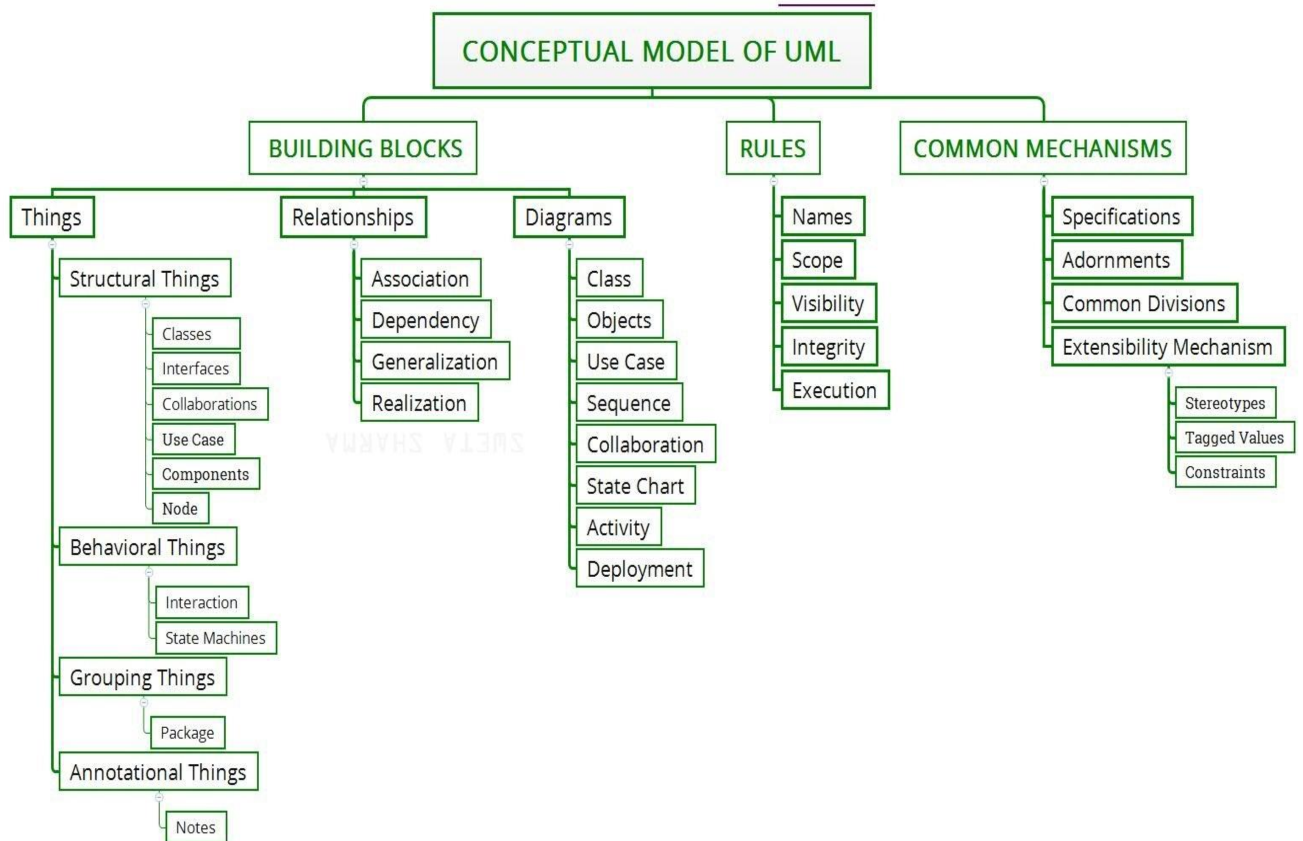
- It is a representation of a subject.
- It captures a set of ideas *(known as abstractions)* about its subject.

3. **Unified:**

- It is to bring together the information systems and technology industry's best engineering practices.
- These practices involve applying techniques that allow us to successfully develop systems.

A Conceptual Model:
A conceptual model of the language underlines the three major elements:

- The Building Blocks
- The Rules
- Some Common Mechanisms



Building Blocks:

The vocabulary of the UML encompasses three kinds of building blocks:

1. **Things:**

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

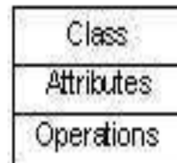
   There are 4 kinds of things in the UML:

```
1. Structural things
2. Behavioral things
3. Grouping things
4. Annotational things
```

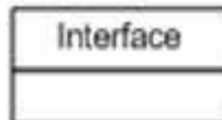
Structural Things

Structural things define the static part of the model. They represent the physical and conceptual elements. Following are the brief descriptions of the structural things.

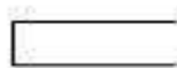
Class − Class represents a set of objects having similar responsibilities.



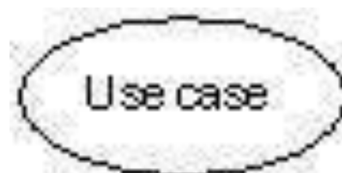
Interface − Interface defines a set of operations, which specify the responsibility of a class.



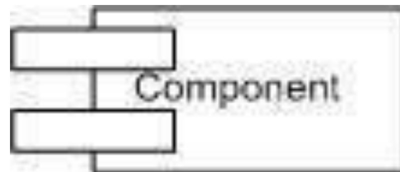
Collaboration −Collaboration defines an interaction between elements.



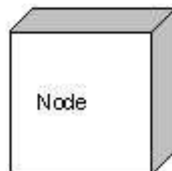
Use case −Use case represents a set of actions performed by a system for a specific goal.



Component −Component describes the physical part of a system.



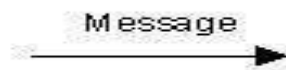
Node − A node can be defined as a physical element that exists at run time.



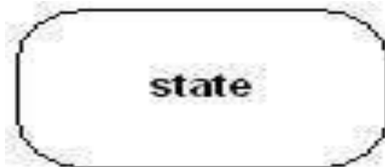
Behavioral Things

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



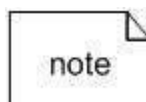
Grouping Things

Grouping things can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available −

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

Annotational Things

Annotational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements. **Note** - It is the only one Annotational thing available. A note is used to render comments, constraints, etc. of an UML element.



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

Relationships:

There are 4 kinds of relationships in the UML:

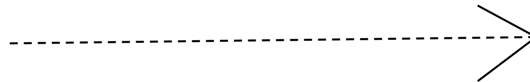
1. Dependency
2. Association
3. Generalization
4. Realization

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

There are four kinds of relationships available.

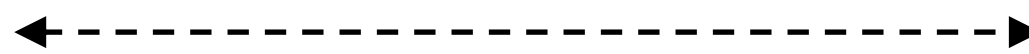
Dependency

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



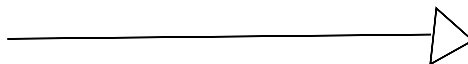
Association

Association is basically a set of links that connects the elements of a UML model. It also describes how many objects are taking part in that relationship.



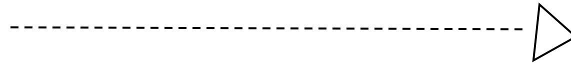
Generalization

Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes the inheritance relationship in the world of objects.



Realization

Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility, which is not implemented and the other one implements them. This relationship exists in case of interfaces.



These relationships are the basic relational building blocks of the UML.

Diagrams:
It is the graphical presentation of a set of elements. It is rendered as a connected graph of vertices (things) and arcs (relationships).

1. Class Diagram
2. Object Diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Statechart diagram
7. Activity diagram
8. Component diagram
9. Deployment diagram

Rules:
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:

1. **Names –** What you can call things, relationships, and diagrams.
2. **Scope –** The context that gives specific meaning to a name.
3. **Visibility –** How those names can be seen and used by others.
4. **Integrity –** How things properly and consistently relate to one another.
5. **Execution –** What it means to run or simulate a dynamic model.

Common Mechanisms:
The UML is made simpler by the four common mechanisms. They are as follows:

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

SPECIFICATIONS

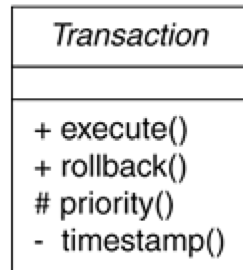
The UML is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and semantics of that building block. For example, behind a class icon is a specification that provides the full set of attributes, operations (including their full signatures), and behaviors that the class embodies; visually, that class icon might only show a small part of this specification. Furthermore, there might be another view of that class that presents a completely different set of parts yet is still consistent with the class's underlying specification. We use the UML's graphical notation to visualize a system; We use the UML's specification to state the system's details. Given this split, it's possible to build up a model incrementally by drawing diagrams and then adding semantics to the model's specifications, or directly by creating a specification, perhaps by reverse engineering an existing system, and then creating diagrams that are projections into those specifications.

The UML's specifications provide a semantic backplane that contains all the parts of all the models of a system, each part related to one another in a consistent fashion. The UML's diagrams are thus simply visual projections into that backplane, each diagram revealing a specific interesting aspect of the system.

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. For example, the notation for a class is intentionally designed to be easy to draw, because classes are the most common element found in modeling object-oriented systems. The class notation also exposes the most important aspects of a class, namely its name, attributes, and operations.

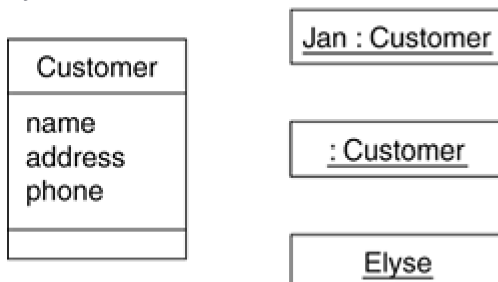
A class's specification may include other details, such as whether it is abstract or the visibility of its attributes and operations. Many of these details can be rendered as graphical or textual adornments to the class's basic rectangular notation. For example, here in a class, adorned to indicate that it is an abstract class with two public, one protected, and one private operation.



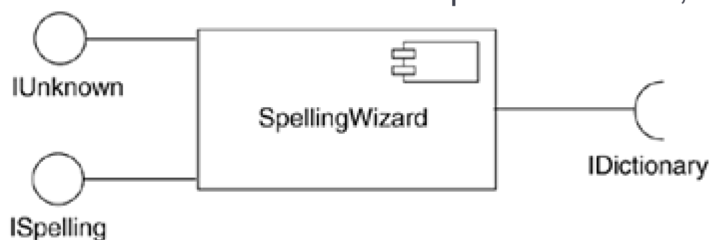
Every element in the UML's notation starts with a basic symbol, to which can be added a variety of adornments specific to that symbol.

COMMON DIVISIONS

In modeling object-oriented systems, the world often gets divided in several ways.
First, there is the division of class and object. A class is an abstraction; an object is one concrete manifestation of that abstraction. In the UML, we can model classes as well as objects, Graphically, the UML distinguishes an object by using the same symbol as its class and then simply underlying the object's name.

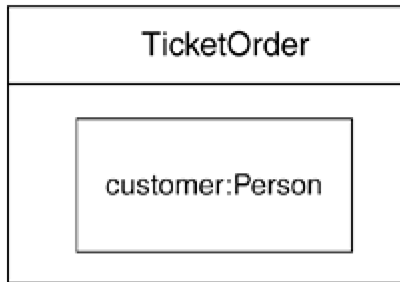


In this figure, there is one class, named Customer, together with three objects: Jan (which is marked explicitly as being
a Customer object), :Customer (an anonymous Customer object), and Elyse (which in its specification is marked as being a kind
of Customer object, although it's not shown explicitly here).
Almost every building block in the UML has this same kind of class/object dichotomy. For example we can have use cases and use case executions, components and component instances, nodes and node instances, and so on.
Second, there is the separation of interface and implementation. An interface declares a contract, and an implementation represents one concrete realization of that contract, responsible for faithfully carrying out the interface's complete semantics. In the UML, we can model both interfaces and their implementations,



Here, there is one component named SpellingWizard.dll that provides (implements) two interfaces, IUnknown and ISpelling. It also requires an interface, IDictionary, that must be provided by another component.
Almost every building block in the UML has this same kind of interface/implementation dichotomy. For example, we can have use cases and the collaborations that realize them, as well as operations and the methods that implement them.

Third, there is the separation of type and role. The type declares the class of an entity, such as an object, an attribute, or a parameter. A role describes the meaning of an entity within its context, such as a class, component, or collaboration. Any entity that forms part of the structure of another entity, such as an attribute, has both characteristics: It derives some of its meaning from its inherent type and some of its meaning from its role within its context



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. For this reason, the UML is opened-ended, making it possible for us to extend the language in controlled ways. The UML's extensibility mechanisms include

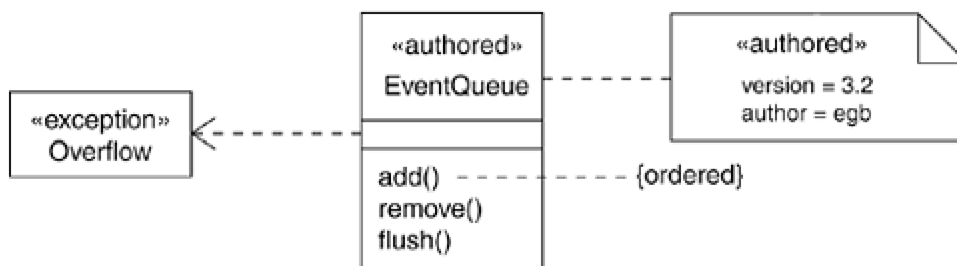
- Stereotypes
- Tagged values
- Constraints

A ***stereotype*** extends the vocabulary of the UML, allowing us to create new kinds of building blocks that are derived from existing ones but that are specific to the problem. For example, if we are working in a programming language, such as Java or C++, we will often want to model exceptions. In these languages, exceptions are just classes, although they are treated in very special ways. Typically, we only want to allow them to be thrown and caught, nothing else. we can make exceptions first-class citizens in our models—meaning that they are treated like basic building blocks—by marking them with an appropriate stereotype

A ***tagged value*** extends the properties of a UML stereotype, allowing us to create new information in the stereotype's specification. For example, if we are working on a shrink-wrapped product that undergoes many releases over time, we often want to track the version and author of certain critical abstractions. Version and author are not primitive UML concepts. They can be added to any building block, such as a class, by introducing new tagged values to that building block.

for example, the class EventQueue is extended by marking its version and author explicitly

A *constraint* extends the semantics of a UML building block, allowing us to add new rules or modify existing ones. For example, we might want to constrain the EventQueue class so that all additions are done in order. we can add a constraint that explicitly marks these for the operation add.



Collectively, these three extensibility mechanisms allow us to shape and grow the UML to our project's needs. These mechanisms also let the UML adapt to new software technology, such as the likely emergence of more powerful distributed programming languages. we can add new building blocks, modify the specification of existing ones, and even change their semantics. Naturally, it's important that we do so in controlled ways so that through these extensions, we remain true to the UML's purpose—the communication of information.