⋆˚₀⋆✧˚ Chapter 4 : Introduction to OOP modeling ˚✧⋆₀˚⋆

In every professional field, modeling serves as a vital bridge between a concept and its creation. Just as an architect drafts blueprints for a home or an engineer simulates an aircraft's aerodynamics, software developers use models to simplify reality. Modeling allows us to strip away unnecessary details and focus on the essential structure of a system before a single line of code is written.

Today, modeling is a cornerstone of successful IT projects. It addresses the reality that complex systems are impossible to understand in their entirety at once. By creating a model, we achieve four critical goals: we **visualize** the system's current or intended state, **specify** its precise structure and behavior, provide a **template** to guide its physical construction, and establish a permanent **documentation** of our design decisions. Ultimately, modeling is the art of managing complexity, ensuring that our software remains scalable, maintainable, and aligned with real-world requirements.

1. Principles of modeling

To build effective software, modeling must follow a set of fundamental principles that ensure the design is both practical and accurate.

- **Influence on Approach:** The choice of which models to create significantly shapes how we perceive a problem and, ultimately, how we formulate its solution. A well-chosen model guides the developer toward the most efficient architectural path.
- **Varying Levels of Precision:** Models are not "one size fits all." They can be expressed at different levels of detail depending on who is using them (a high-level overview for stakeholders or a high-precision blueprint for developers).
- **Real-World Grounding:** The most successful models are those directly connected to reality. A model should accurately reflect the actual requirements and constraints of the physical or business environment it represents.
- **Multiple Viewpoints:** No single model can capture the entire scope of a complex, non-trivial system. Instead, the best approach involves a small set of nearly independent

models that provide different perspectives, ensuring all aspects of the system's behavior and structure are covered.

2. Object oriented modeling

The evolution of software development has shifted from a functional focus to a structural one, prioritizing objects that mirror the real world. Historically, software was built around **procedures and functions**. While effective for small tasks, this approach struggles with scaling. As systems grow and requirements change, algorithmic code often becomes tangled and difficult to maintain.

Today, the fundamental building blocks are **objects and classes** where a **class** is a blueprint or description of a set of similar objects and an **object** is a concrete "thing" derived from the problem or solution.

Every object possesses three critical characteristics:

- **Identity:** A way to distinguish it from other objects (e.g., a unique name).
- **State:** The data or properties associated with it.
- **Behavior:** The actions it can perform or how it communicates with others.

For example, in a billing system, objects appear at every level. The **User Interface** contains buttons and menus; the **Business Layer** handles transactions and rules; the **Database** manages tables representing customers and products.

In the early days of OOP, developers used competing methods (OMT by Rumbaugh, OOSE by Jacobson, and Booch by Grady Booch). These three pioneers eventually joined forces to create the **Unified Modeling Language (UML)**.

UML is not a "method" (a step-by-step process), but a **modeling language** (a standardized vocabulary and set of rules used to represent software plans). Because it is independent of any specific programming language, it has become the global standard for architects and designers.

UML is defined by its ability to handle four key tasks in software development:

- **Visualization:** It provides a graphical representation of the system, making it easier for developers to understand and maintain complex logic.

- **Specification:** It allows for the creation of precise, complete, and unambiguous models that leave no room for guesswork.
- **Implementation:** UML models are designed to be easily "translated" into code for object-oriented languages like **Java, C++, or C#**.
- **Documentation:** It serves as a detailed record of the system's architecture, requirements, and testing procedures.

While primarily intended for software, UML's versatility has made it essential in various complex fields like **Finance** (Banking systems and financial services), **Infrastructure:** (Telecommunications, transportation, and web services), **Critical Systems:** (Aerospace, defense, and medical electronics) and **Research:** (Science and large-scale business information systems).

3. UML overview

To master UML, a conceptual model of the language must be formed. This process involves learning three fundamental pillars: the **basic building blocks**, the **rules** for their arrangement, and the **common mechanisms** applied throughout the system.

A. The Basic Building Blocks of UML

The UML vocabulary is organized into three primary categories: **Things**, **Relationships**, and **Diagrams**.

a. Things: The Core Abstractions

"Things" represent the fundamental elements within a model. These are categorized into four types based on their specific role:

- **Structural Things (The Nouns)**

These represent the static components of a model, describing physical or conceptual parts. Collectively referred to as **Classifiers**, they define the "what" of the system. An example of these things are Class, Interface, Collaboration, Use Case, Active Class, Component, Artifact, and Node.

- **Behavioral Things (The Verbs)**

These constitute the dynamic portions of a UML model. They represent behavior over time and space, acting as the operational "verbs." Interactions (sequences of messages), State Machines (lifecycle transitions), and Activities (flow of tasks) are example of these things.

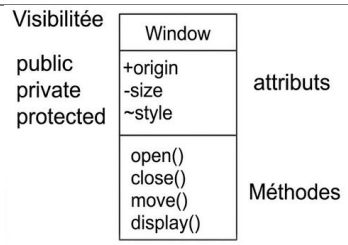
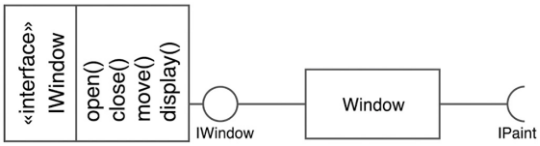
- **Grouping Things (The Organizational Folders)**

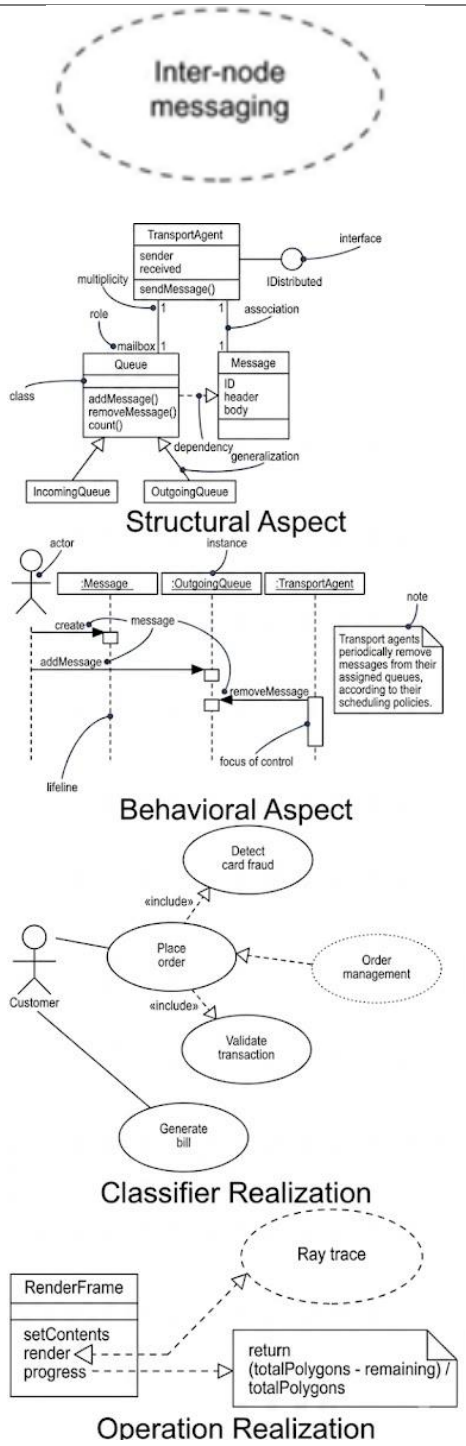
Grouping things serve as the organizational framework for a model. They allow complex systems to be decomposed into smaller, manageable parts. The **Package** is the primary grouping mechanism used to organize other elements into folders or modules.

- **Annotational Things (The Explanatory Notes)**

These are the descriptive parts of the UML model. They provide space for comments, clarifications, or highlights regarding any other element in the design. The **Note** is the only annotational thing. It is represented as a rectangle with a folded corner.

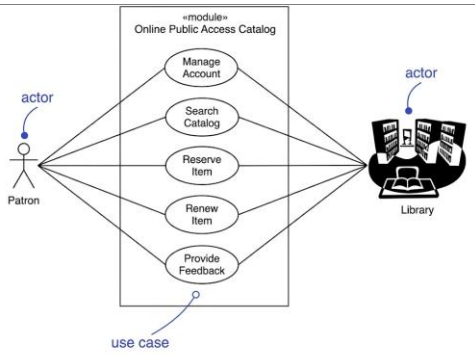
The following table summarizes the symbols with a brief explanation of the four types of things.

Figure (Symbol)	Description
	A class is a set of objects sharing attributes and operations. Represented by a rectangle with three sections (Name, Attributes, Methods).
	An interface is a set of operations that other elements must implement. Represented by a circle (lollipop) or a class with the «interface» keyword.



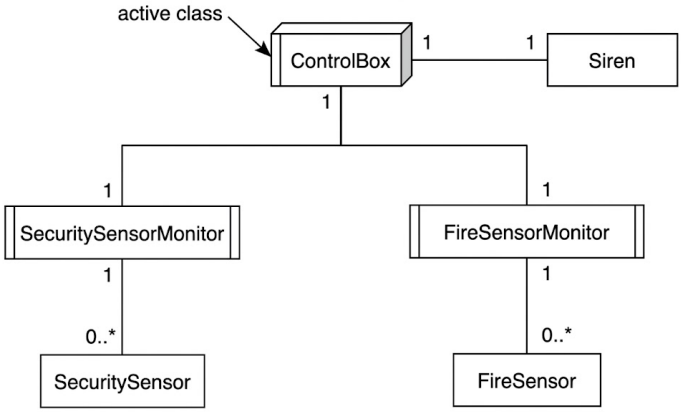
A collaboration is a combination of elements working together to provide cooperative behavior. Represented by a **dotted ellipse**.

A collaboration is also the specification of how an element, such as a classifier (including a class, interface, component, node, or use case) or an operation, is realized by a set of classifiers and associations. A collaboration has a structural and a behavioral aspect. It is represented by a dotted ellipse. The classifier implementation (realisation d'un classeur) figure models a set of use cases for a credit card validation system. The operation implementation figure models the operations of an active class.

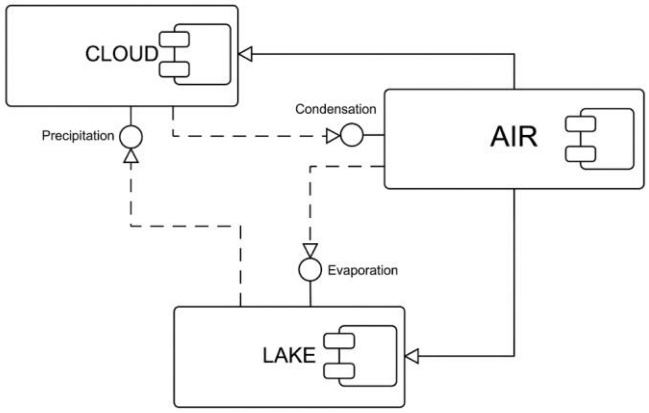


Use Case Diagram of an Online Library.

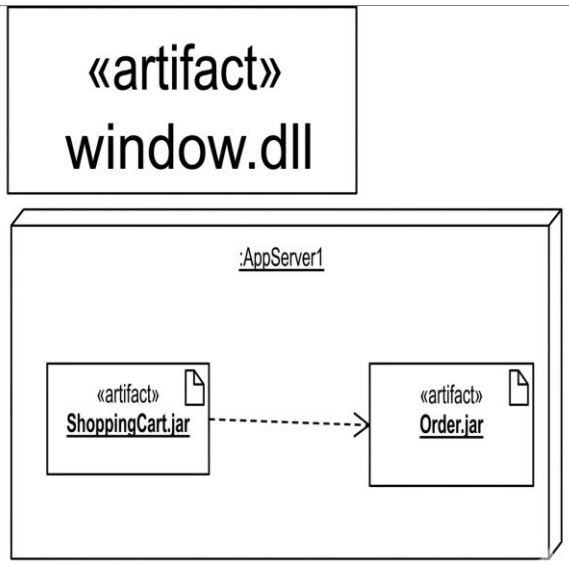
A use case is a sequence of actions a system performs for an actor to yield a result of value. Represented by an **ellipse**.



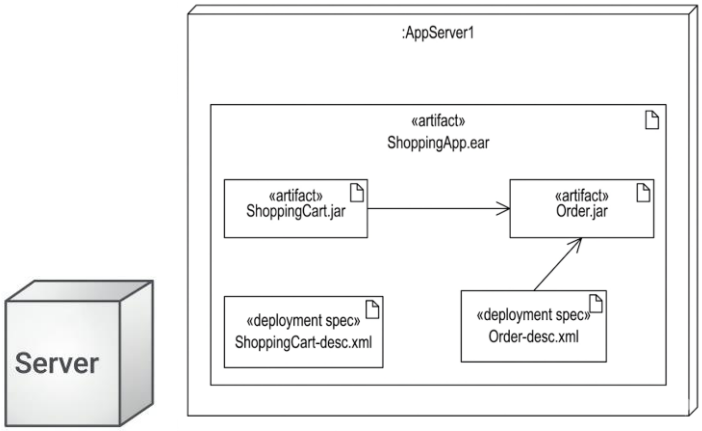
An active class is a class whose objects own a process or thread. Represented as a rectangle with **double vertical lines** on the sides.



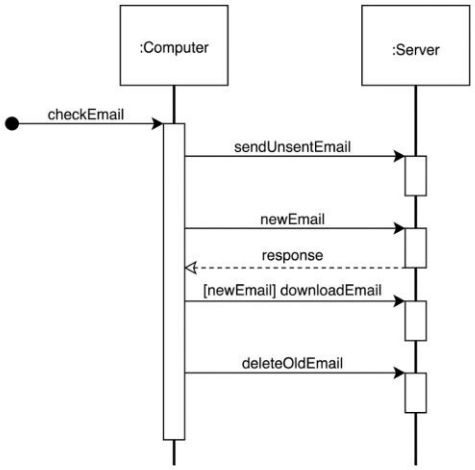
A component is a modular part of a system for which UML diagrams are created. Shown as a rectangle with a **special icon** in the corner.



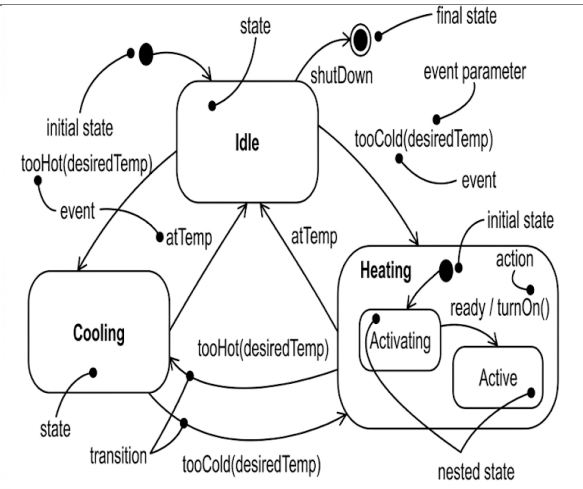
An artifact is a physical and replaceable part of a system (bits, scripts, executables). Represented as a rectangle with the «artifact» keyword.



A node is a physical runtime resource (like a server) with memory and processing power. Represented as a **3D cube**.



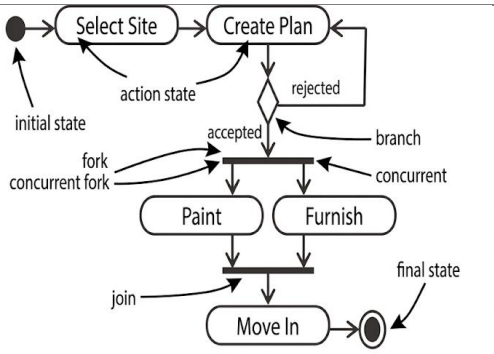
An interaction is a behavior comprising messages exchanged between objects to achieve a goal. Represented by a **directed line** (arrow).



State: rounded rectangle.

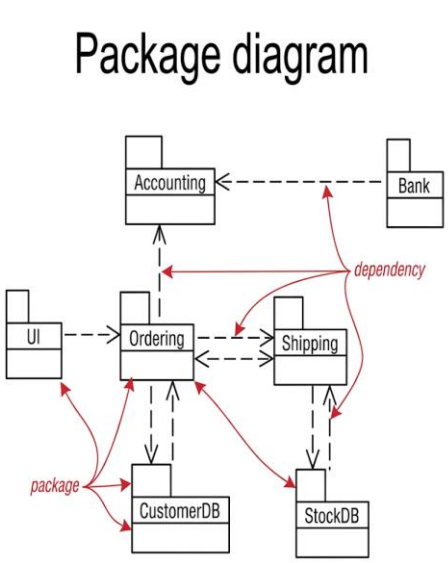
Event: text.

A stat machine specifies the sequences of states an object follows during its existence. States are shown as **rounded rectangles**.

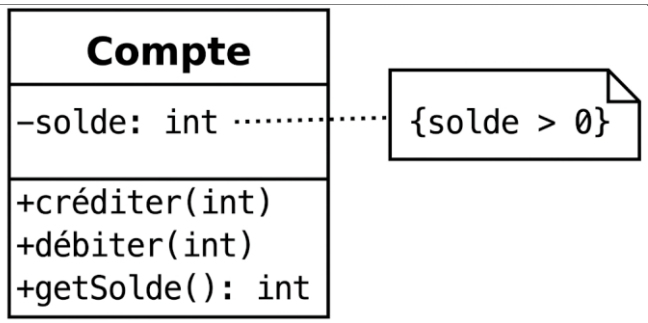


Activity : rounded rectangle.

An activity specifies a sequence of steps in a computational process. Steps are called actions and shown as **rounded rectangles**.



A package is a mechanism for organizing elements into groups. Represented as a **tabbed folder**.



A note is a symbol to express constraints or comments attached to elements. Represented as a rectangle with a **dog-eared corner**.

b. Relationships

Understanding the connections between model elements is essential for building a coherent system. UML offers four primary relationships :

- **Dependency**

A **dependency** is a "using" relationship. It indicates that one element (the client) requires another element (the supplier) for its implementation or specification. If the supplier changes, the client may be affected. A dependency is represented using **dashed line** with an open arrowhead pointing toward the supplier.

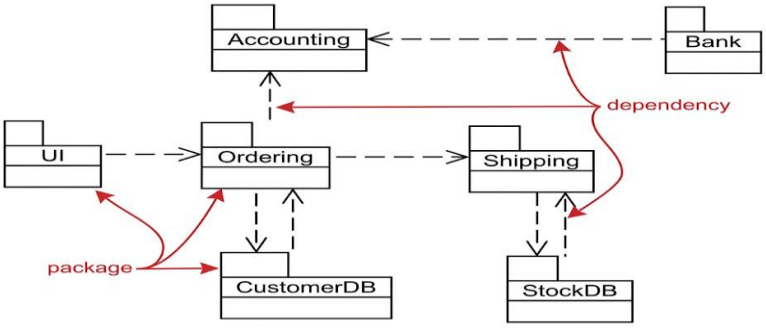


Figure 18. Example of dependency relationships

- **Association**

An association is a structural relationship between classes that describes a set of links, a link being a connection between objects that are instances of the classes. An association is represented as a solid, possibly directed, line, sometimes including a label and often containing other adornments, such as **multiplicity** and **role names**.

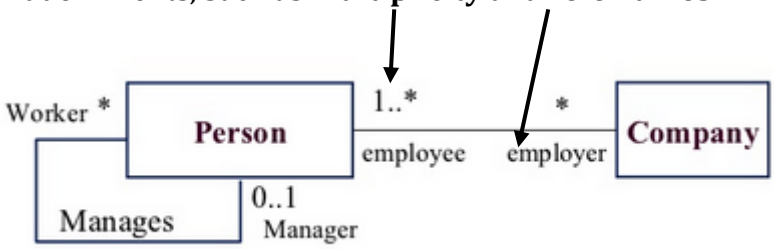


Figure 19. Example of association relationships

Multiplicity is used in UML associations to specify how many instances of one class can be linked to a single instance of another class.

Adornment	**Semantics (Meaning)**
0..1	Zero or 1
1	Exactly 1
0..* or *	Zero or more
1..*	1 or more
1..6	1 to 6
1..3, 7..10, 15, 19..*	1 to 3 or 7 to 10 or 15 exactly or 19 to many

In UML, we often need to model a **"whole/part"** relationship where one class (the whole) is made up of smaller entities (the parts). This is called **aggregation**. It represents a an association relationship where the part can exist independently of the whole. It is drawn as a solid association line with an **unfilled diamond** at the "whole" end.

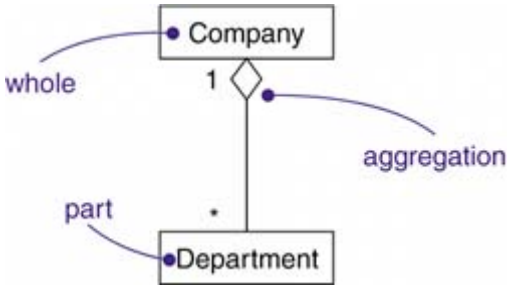


Figure 20. Example of an aggregation relationships

Composition is a strong form of the **whole/part** relationship, often referred to as the **"death relationship."** Unlike aggregation, the parts in a composition are strictly bound to the lifecycle of the whole. If the "whole" object is deleted, all its associated "part" objects are destroyed simultaneously. A part is owned exclusively by one whole at a time and cannot be shared between multiple instances. A composition is represented by a solid association line with a **filled (black) diamond** at the "whole" end.

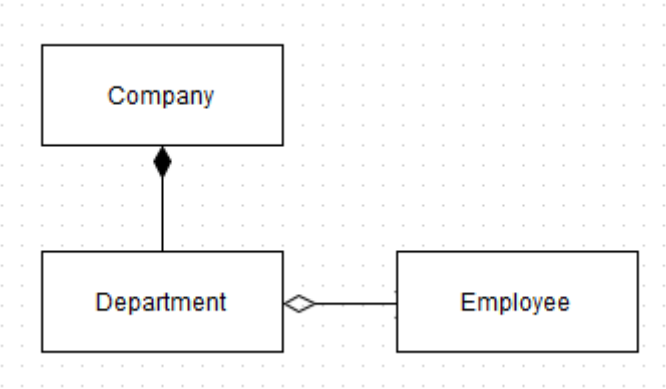


Figure 21. Example of composition relationships

Hierarchically, these relationships are nested: **composition** is a specialized form of **aggregation**, and **aggregation** is itself a specialized form of **association**.

- **Generalization**

Generalization is a relationship between a general element (parent) and a more specific element (child). This is the standard UML term for **inheritance**. It is represented using a

solid line with a hollow arrowhead pointing toward the parent. The child inherits all attributes and behaviors of the parent while having the ability to add its own unique features.

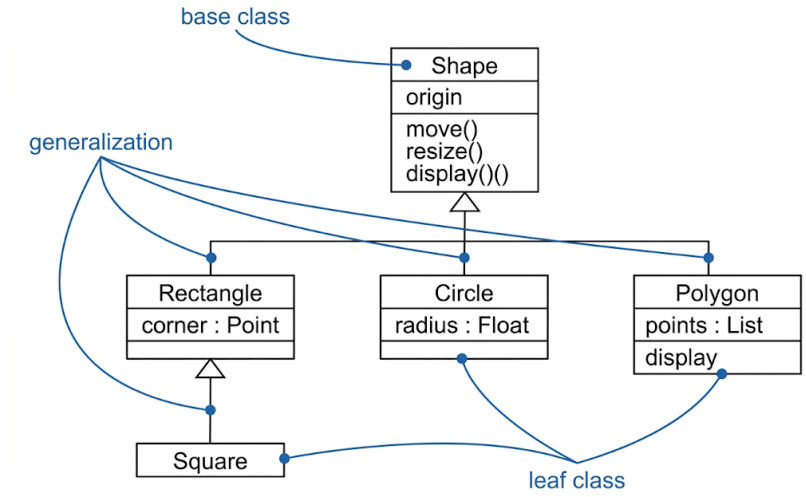


Figure 22. Example of generalization relationships

- **Realization**

A realization is a semantic relationship where one element (the implementation) carries out a "contract" defined by another element (the specification). It is represented using a **dashed line with a hollow arrowhead**. We can found a realization between an **Interface** and the **Class** that implements its operations or between a **Use Case** and the **collaboration** that describes how that use case is actually performed.

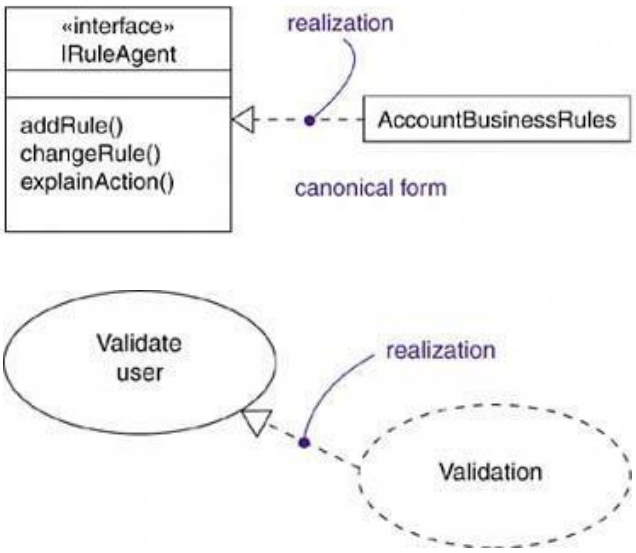


Figure 23. Example of realization relationships

c. Diagrams

A **diagram** is a graphical representation of system elements, typically structured as a connected graph of **nodes** (representing things) and **edges** (representing relationships). In UML, diagrams serve as specific projections or "views" of a system's model.

UML consists of **fourteen** unique diagram types, which are formally organized into three primary categories: **Structural**, **Behavioral**, and **Model Management**.

Diagram Type	Category	Main Usage
Class Diagram	Structure	Models the static structure of the system (classes, attributes, operations)
Object Diagram	Structure	Shows snapshots of objects and their relationships at a specific moment
Component Diagram	Structure	Describes the organization and dependencies among software components
Composite Structure Diagram	Structure	Shows internal structure of a class and interactions between its parts
Package Diagram	Management	Organizes elements into packages; shows dependencies between them
Deployment Diagram	Structure	Models the physical deployment of software on hardware nodes
Profile Diagram	Structure	Customizes UML models using stereotypes and profiles
Use Case Diagram	Behavior	Represents functional requirements and interactions between users and system
Activity Diagram	Behavior	Describes workflows and sequence of activities (flowchart-like)
State Machine Diagram	Behavior	Models state changes of an object in response to events
Sequence Diagram	Behavior	Shows time-ordered interaction between objects or components
Communication Diagram	Behavior	Displays interactions focusing on object relationships
Interaction Overview Diagram	Behavior	Combines activity and sequence diagrams to show control flow
Timing Diagram	Behavior	Visualizes object behavior over time with timing constraints

B. Rules

Just as any natural language requires grammar to be understood, UML relies on a set of **Rules** to ensure that models are "well-formed"—meaning they are semantically consistent, harmonious, and logically sound. UML dispose of five essential rules for maintaining coherence in UML models:

- **Names**

Every entity in a UML model must have a name for identification and referencing. This includes elements (classes, objects), relationships, and the diagrams themselves. A name must refer to the same element throughout its context. If a class is named Customer, every reference to Customer must point to that specific definition. Names should be meaningful and follow standard conventions to prevent ambiguity.

- **Scope**

Scope defines the specific context or "boundary" within which a name has a particular meaning. The same name can be used in different parts of a system to mean different things, provided their scopes do not overlap. For this, UML requires clear boundaries (such as within a Class or a Package) so that a variable like status in one module isn't confused with a status variable in another.

- **Visibility**

Visibility determines the "reach" or access level of an element (like an attribute or operation) from other parts of the system. It mirrors access modifiers in programming. There are four standard visibility levels in UML:

- **Public (+)**: The element is accessible from any other class.
- **Private (-)**: The element is only accessible within the class itself.
- **Protected (#)**: The element is accessible within the class and its subclasses.
- **Package (~)**: The element is accessible to all classes in the same package.

- **Integrity**

Integrity ensures that relationships between elements remain logically and semantically correct. It ensures that associations, dependencies, and constraints (like multiplicity) are respected. For example, if a rule states a Customer can have 1..* (one-to-many) Orders, the integrity rule ensures this relationship is maintained across all design and execution phases.

- **Execution**

Execution involves simulating or verifying the dynamic behavior defined in the model. This applies to behavioral diagrams like **Sequence**, **State Machine**, and **Activity**

diagrams. It checks how the system behaves over time (ensuring that when a user triggers an action, the objects interact, change states, and execute operations exactly as intended).

C. Common mechanisms

Just as architectural styles define the structure and appearance of a building, UML uses **mechanisms** to define how models are structured, extended, and customized. These mechanisms provide the necessary depth and flexibility to adapt UML to various domains.

There are four fundamental mechanisms in UML:

- **Specification**

UML is not just a collection of drawings. Every graphical element is backed by a **specification** (a detailed textual definition of its syntax and semantics). For example, behind a simple class rectangle lies a specification that defines its full set of attributes, operations, and behaviors in a way that tools can interpret.

- **Adornments**

Adornments are graphical or textual markers added to a basic element to provide more detail. They prevent diagrams from becoming overly complex by keeping basic symbols simple while allowing for "layers" of detail. For example, adding a + or - to an attribute to show visibility, or using *italics* to indicate an abstract class.

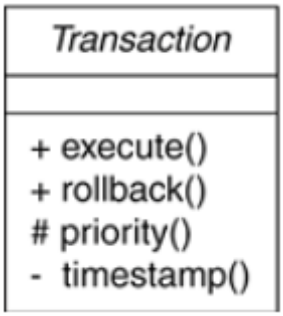


Figure 24. Example of adornment by showing visibility

- **Common Divisions**

UML encourages partitioning a system into manageable parts to distinguish between different logical layers:

- **Classes vs. Objects:** Separates the **blueprint** (Design-time) from the **actual instance** (Runtime). For example, Car is the class, and myCar:Car is the specific object.

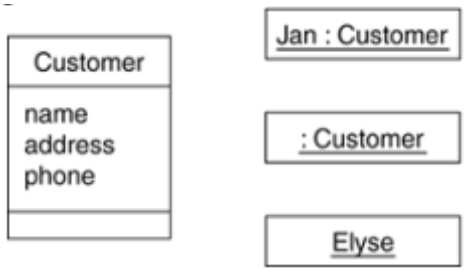


Figure 24. Example of division by separating classes from objects

- **Interface vs. Implementation:** Distinguishes **what** an element does (its contract) from **how** it does it (the logic). This promotes encapsulation and interchangeability.

Figure 25. Example of division by separating interface from implementation

- **Role vs. Type:** Separates the **definition** of an object from the **function** it performs in a specific scenario. An object of the type Person might play the role of Customer in one context and Employee in another.

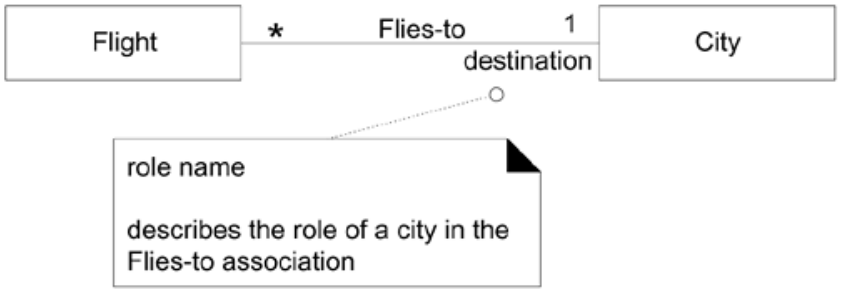


Figure 26. Example of division by separating role from type

- **Extensibility Mechanisms**

UML is designed as an "open" language. Because a single, fixed vocabulary cannot capture the unique requirements of every industry (such as aerospace, healthcare, or web development). UML provides four **Extensibility Mechanisms**. These allow you to adapt and broaden the language's capabilities in a controlled and standardized way.

- **Stereotypes**

Stereotypes are used to expand the UML vocabulary by creating new types of building blocks based on existing ones. They are typically denoted by double guillemets (e.g., «interface»). They allow you to categorize elements specifically for your problem domain. For example, you might mark a standard class as a «sensor» or «database» to clarify its specific role in the system.

- **Tagged Values**

Tagged values allow you to add custom properties to a model element's specification. They are usually written as a key-value pair within curly braces, such as {version = 2.1}. While stereotypes define a new "kind" of element, tagged values define new data for that element. This is useful for tracking metadata like author names, deadlines, or hardware requirements that aren't part of the standard UML definition.

- **Constraints**

Constraints are used to extend the semantics (meaning) of a base element by adding new rules or modifying existing ones. They are written as text strings within curly braces. They define logic that the model must follow. For instance, you could add a constraint to an attribute to ensure {value > 0} or to a relationship to specify that it is {ordered}.

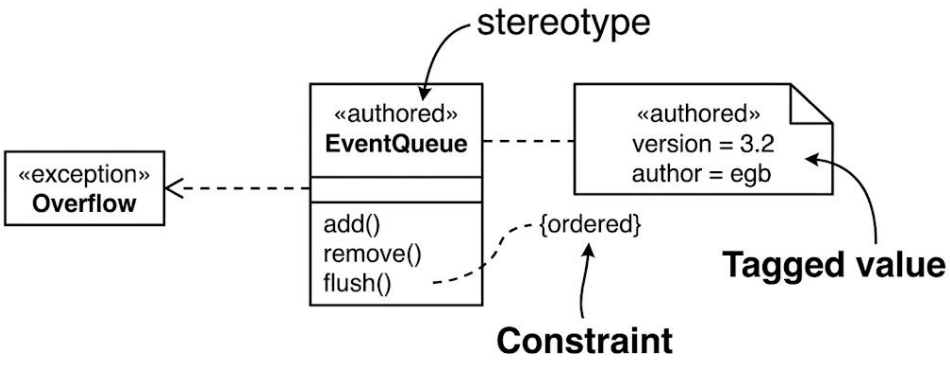


Figure 27. Example of stereotype, constraint and tagged value

- **Profiles**

A Profile is a high-level organization mechanism used to bundle stereotypes, tagged values, and constraints into a single package tailored for a specific domain or platform. Instead of applying individual extensions one by one, a Profile allows you to apply a whole "dialect" of UML at once.

The following schema resume the UML concepts:

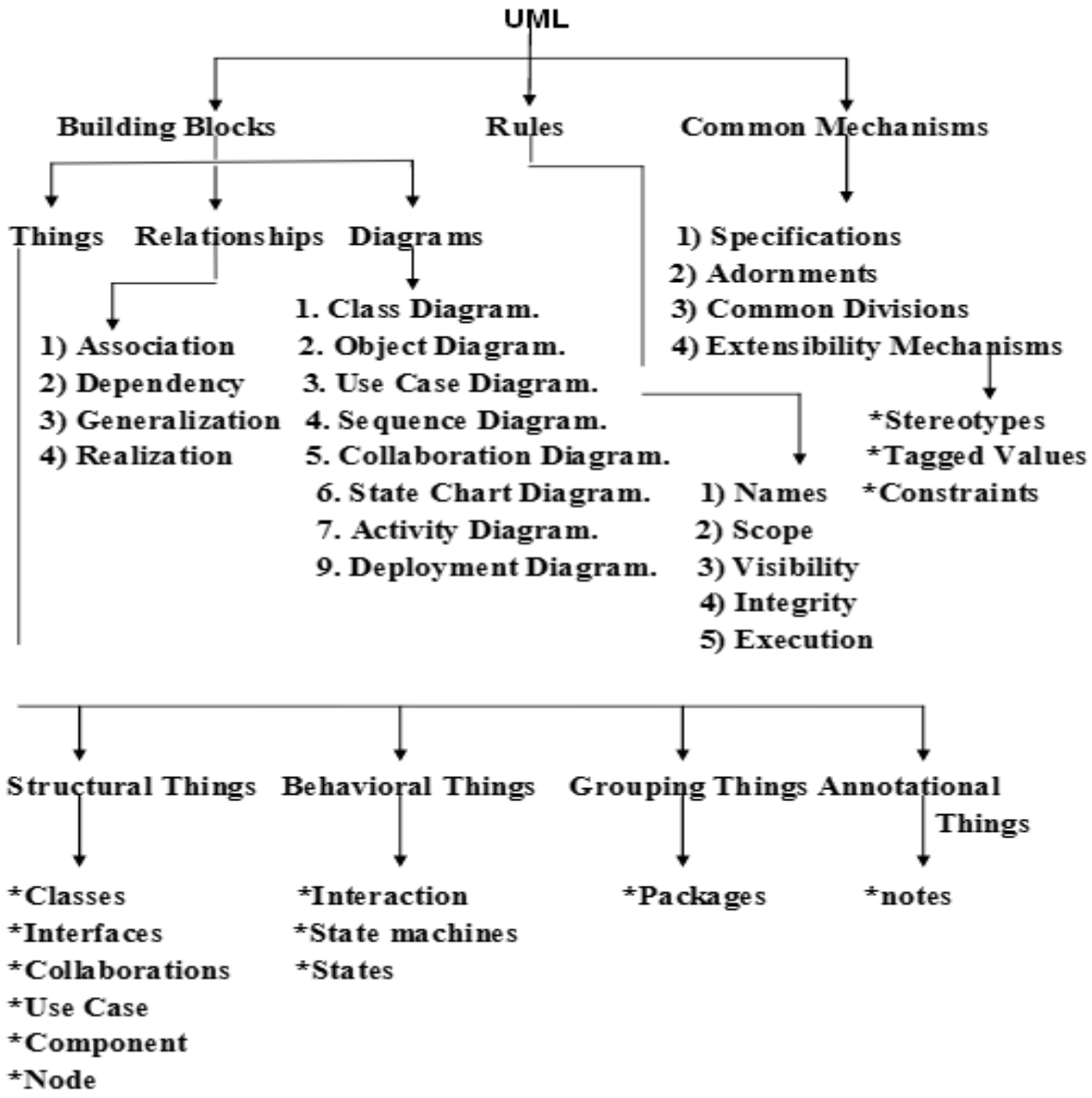


Figure 28. General schema of UML