Class Diagram

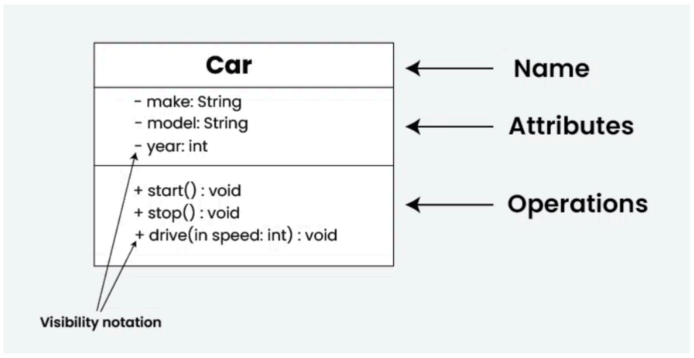
A UML class diagram is made up of:

- A set of classes and
- A set of relationships between classes

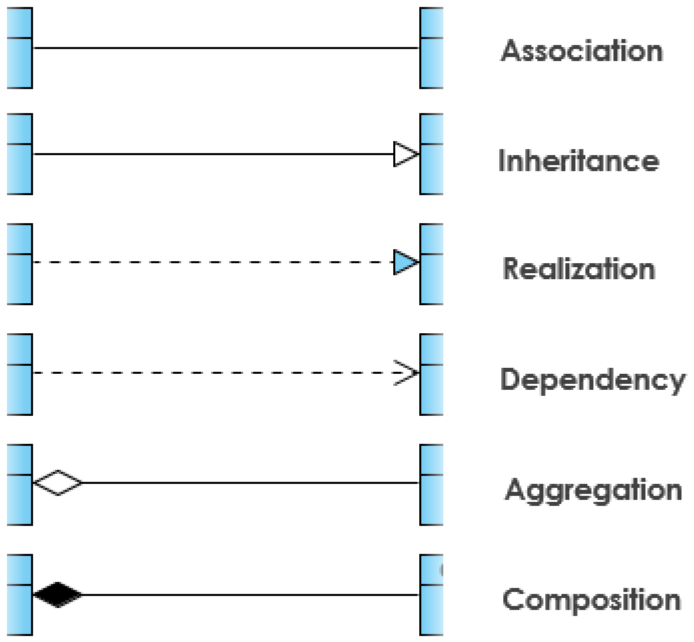
Class Notation

A class notation consists of three parts:

1. **Class Name**
    - The name of the class appears in the first partition.
2. **Class Attributes**
    - Attributes are shown in the second partition.
    - The attribute type is shown after the colon.
    - Attributes map onto member variables (data members) in code.
3. **Class Operations** (Methods)
    - Operations are shown in the third partition. They are services the class provides.
    - The return type of a method is shown after the colon at the end of the method signature.
    - The return type of method parameters is shown after the colon following the parameter name.
    - Operations map onto class methods in code.



Class Diagram Relationships



Generalization (Inheritance)

Inheritance represents an **"is-a"** relationship where a **subclass (child)** inherits properties and behaviors from a **superclass (parent)**. It is depicted by a **solid line with a hollow arrowhead** pointing from the subclass to the superclass.

Generalization defines a **hierarchical relationship**, where a specific class extends a more general class. For example, **BankAccount** is a general class, while **CurrentAccount, SavingsAccount, and CreditAccount** are specialized subclasses.

- Represents an **"is-a"** relationship.
- Subclasses **inherit and extend** superclass functionality.
- Abstract classes are shown in *italics* in UML diagrams.
- Shown as a **solid line with a hollow arrowhead** in class diagrams.

This structure promotes **code reuse and polymorphism**, ensuring flexibility in object-oriented design.

Association

An **association** represents a **structural link** between two peer classes, indicating a **bi-directional relationship**. It is depicted as a **solid line** connecting the two classes in a class diagram.

Cardinality in Associations:

Cardinality defines how many instances of one class relate to instances of another:

- **One-to-One** (1….1)
- **One-to-Many** (1….*)
- **Many-to-Many** (*….. *)
- **Zero-to-One** (0…..1)
- **Zero-to-Many** (0…..*)

Example:

In a library system, a **Library** contains multiple **Books**, and each **Book** belongs to a specific **Library**. This **Library-Book** relationship represents an **association**, where the **Library** class references multiple instances of the **Book** class, establishing a **one-to-many** relationship.

Associations help define **logical connections** between objects, improving **structural clarity** in object-oriented design.

Aggregation

Aggregation is a **special type of association** that represents a **"whole-part"** relationship. It indicates that one class (**the whole**) contains or is composed of another class (**the part**) while maintaining **separate lifetimes**—the part can exist independently of the whole.

- Represents a **“part of”** relationship.
- The **whole class** contains multiple instances of the **part class** (*).
- Objects have **independent lifetimes**—the part can exist without the whole.
- **Depicted as** a **solid line** with an **unfilled diamond** at the whole class's end.

Example:

A **Company** (whole) has multiple **Employees** (parts). While employees belong to a company, they **can still exist** if the company ceases to operate.

Aggregation helps structure **complex relationships** in object-oriented design while preserving **object independence**.

Composition

Composition is a **special type of aggregation** that represents a **strong ownership** relationship, where the **part cannot exist without the whole**. If the **whole** is destroyed, its **parts are also destroyed**.

- Represents a **"strong part-of"** relationship.
- The **part** (contained object) **lives and dies** with the **whole**.
- The **part cannot exist independently** of the whole.
- **Depicted as** a **solid line** with a **filled diamond** at the whole class's end.

Example:

In a **house**, the **House** (whole) consists of multiple **Rooms** (parts). If the **House is demolished**, all its **Rooms** are also destroyed, as they cannot exist independently.

Human Body and Organs
In a **human body**, the **Body** (whole) contains various **Organs** (parts) such as the heart, lungs, and liver. If the **Body dies**, all its **Organs** also stop functioning, as they cannot exist separately.

In a **tree**, the **Tree** (whole) has multiple **Leaves** (parts). If the **Tree dies**, all its **Leaves** wither and fall, as they depend entirely on the tree for survival.

Composition enforces **strict dependency**, ensuring that **parts cannot outlive their whole**, making it useful for **strongly coupled relationships** in object-oriented design.

Dependency

A **dependency** is a **loose coupling** relationship between two classes where one class **relies on** another, but the relationship is not as strong as **association** or **inheritance**. A class depends on another if changes to the definition of one may cause changes to the other, but the reverse is not true. This type of relationship is often depicted as a **dashed line with an open arrow**.

- Represents a **loose coupling** between classes.
- One class **depends** on another but doesn’t **own** or **control** it.

- The **dependent class** uses the services or methods of the **dependency** class, but it does not store an object of the dependent class.
- Changes in the dependent class might affect the dependent class, but the opposite is not true.

Example:

In a scenario where a **Person** class depends on a **Book** class:

- The **Person** class (dependent) might have a method hasRead(Book book) to check if the person has read a specific book (perhaps by querying a database).
- The **Book** class (dependency) represents the book being read. It is independent and can exist without the **Person** class.

Here, the **Person** class relies on the **Book** class to check if the person has read it. However, the **Book** class doesn't rely on the **Person** class, demonstrating a **one-way dependency**.

A **dependency** indicates a **loosely coupled relationship**, where one class uses another temporarily but does not have ownership or lasting control over it.

Realization (Interface Implementation)

Realization indicates that a class implements the features of an interface. It is shown as a **dashed line with an open arrowhead** pointing from the implementing class to the interface.

- Represents a class **implementing an interface**.
- The class provides concrete implementations for the methods defined in the interface.
- **Depicted by a dashed line** with an **open arrowhead**.

Example:

- The **Owner interface** defines methods like acquire(property) and dispose(property).
- The **Person** and **Corporation** classes **realize** this interface by providing their own implementations of these methods (e.g., acquiring houses or real estate).

Realization connects an interface with the class that implements its methods, allowing different classes to provide specific behavior for the same interface.