

More Relationships



Relationships can be put in place between classes, which describe what sort of link or connection their objects have. The concept of relationships and their multiplicities, navigabilities and types are dealt with in this section.

Relationships

Objects are often related to other objects. This connection is called a **relationship**. Relationships are modelled as lines connecting the two classes whose instances (objects) are involved in the relationship. When you model relationships in UML class diagrams, you show them as a thin line connecting two classes.

Each relationship link has two ends which are called **roles**. Each role has a **name**, a **multiplicity**, a **navigability** and a **type**.

The following diagram outlines the notation for relationships.

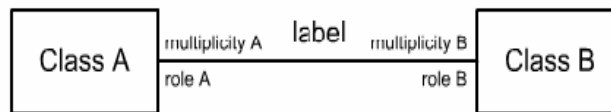


Figure 1: UML notation for relationships

The four parts of relationships are covered in the sequel of this section.

Relationship Name

The **name** is represented by a label indicating what the relationship does, facilitating model understanding. The name of the relationship appears in the middle of the line that symbolizes the association. In general it can be on, above or below the line.

A general rule is that classes and properties are **nouns** while methods and relationships are **verbs**. Without making it a systematic rule, experience recommends naming relationships using a verbal form – either active, like 'works for', or passive, like 'is employed by'. Examples in the Sloopy world are *is friendly with* (between two Sloopies), *rides* (*Sloopy* and *Skateboard*) and *uses* (*Sloopy* and *Wings*).

The two classes *Sloopy* and *SloopyLand* are connected by the relationship *lives in*.

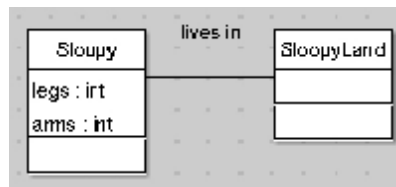


Figure 2: Sloopy *lives in* Association

Relationships may be more complex than just one class connected to another. Several classes can connect to one class.

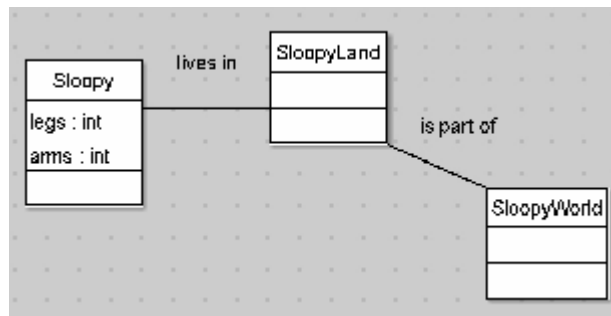


Figure 3: *lives in* and *is part of* Relationships

Relationship Multiplicity

The **Multiplicity** of a role shows the number of objects from one class that relate with a number of objects in a related class. It indicates how many objects can participate in the relationship. One class can be related to another class in the following relationships:

- **one-to-one**, indicated by 1..1 (or left blank)
- **one-to-many**, indicated by 1..*

Place multiplicity notations near the ends of a relationship. These symbols indicate the number of instances of one class linked to one instance of the other class. Below is a class diagram which contains both relationship types. One *Sloopy* lives in one *SloopyLand* (one-to-one), while one *SloopyLand* accommodates many *Sloopies* (one-to-many). One *Sloopy* rides one *Skateboard* and one *Skateboard* is ridden by one *Sloopy*. These are both one-to-one relationships.

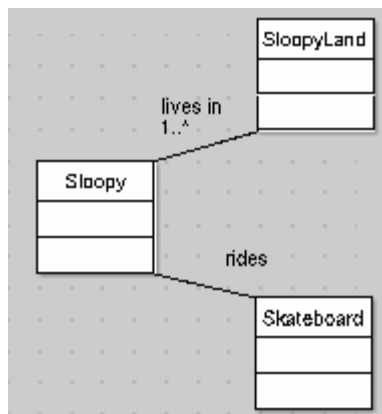


Figure 4: One-to-many and one-to-one Relationship

Sometimes one has to model a **many-to-many** relationship. However, UML does not support such a relationship, since it can be modelled using two one-to-many relationships. In the example below, instead of modelling a direct many-to-many relationship between *SloopyBoy* and *SloopyGirl*, two individual one-to-many relationships are modelled.

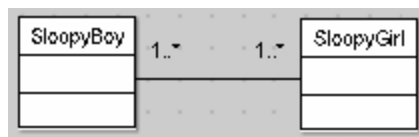


Figure 5: Many-to-many Relationship Replacement

Only for completeness:

UML provides some more detailed multiplicities, which distinguish between **one-to-many** (1..*) and **zero-to-many** (0..*) relationships (as well as **one-to-one** and **zero-to-one**). The one-to-one relationship in the *Skateboard* example above is technically a zero-to-one relationship, since one *Sloopy* can either ride one or no skateboard. However, for the purpose of this course, this distinction can be ignored.

There exists a concept called reflexive or **recursive relationship**, where a class is in relationship with itself. This can happen when a class has objects that can play a variety of roles. An example is a *plays with* relationship in the *Sloopy* family, where a *Sloopy plays with* another *Sloopy*. It is usually recommended to avoid reflexive relationships.

Relationship Navigability

The **navigability** of a role shows which class has the responsibility for maintaining the relationship between the classes. It indicates which end of a relationship is aware of the other end.

For example, Eddy recognizes Nina. This does not state whether Nina does or does not recognize Eddy. You could refine this idea by creating a directed relationship, indicating that the relationship goes in only one direction.

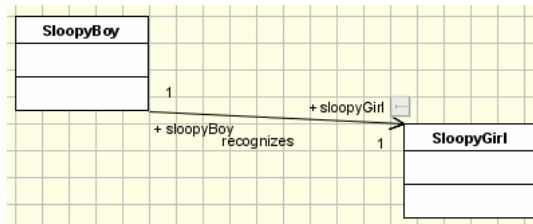


Figure 6: Directional *recognizes* Relationship

Other non Sloopy examples are listed below:

- An *Item* in a *List*; a *List* needs to know all of the items it contains, but an *Item* does not need to know in which lists it appears.
- A *Company* references *Clients*, but the *Client* class does not have any knowledge of the *Company*.
- A school *Class* has a responsibility for knowing which *Students* attend. Therefore, you don't have to explicitly define a *getStudents* method in the *School* class (although you may) since it is already implicitly defined by the navigability of the relationship.

Relationships can have **bidirectional** navigability (each end is aware of the other) or **single directional** navigability (one end is aware of the other, but not vice versa). Direction in UML is represented by arrows at the end of a relationship. You can set the navigability on either, neither or both ends of your associations. If there is no navigability shown then the navigability is unspecified.

Relationship Types

A role can have one of three types: **association**, **composition** or **aggregation**.

An **association** indicates that the two classes have a relationship. Composition and aggregation indicate additional constraints about the relationship:

A **composition** indicates that one class belongs to the other. A polygon is made up of several points. If the polygon is destroyed, so are the points.

An **aggregation** is similar to composition, but is a less rigorous way of grouping things. An order is made up of several products, but a product continues to exist even if the order is destroyed.

The three types of relationships, how they are modelled in UML and how they are handled in Poseidon are discussed in the next 3 sections.



The powerful concept of relationships has been introduced and their roles represented by multiplicities, navigabilities and types have been dealt with.