

Packages



Packaging is a mechanism to group related classes to more manageable entities. This chapter introduces packages and the related concept of namespaces.

The Need for Packaging

When working on larger projects, in teams of modellers and programmers or in environments where parts of a program have to be transferred from one department to another, it is practical to package parts that belong to each other. In other words, a package, like a file directory, is a general way to put related classes together to provide better organization.

There exist five criteria for what a software package has to be in order to fulfil the definition:

- Multiple-use
- Non-context-specific
- Composable with other packages
- Encapsulated i.e. non-investigable through its interfaces
- A unit of independent deployment

The advantages of components are manifold:

- Reusability of software, which reduces cost
- Encapsulation of packages, that is the code (which are a company's crown jewels) are hidden from the outside world.
- Extensibility, which allows other modellers/programmers to add (overload and override) to the existing structure.
- Maintainability, because the interface can remain static while the implementation is changed.
- Less errors, since interfaces are clearly defined and messages can only be sent to objects that can 'understand' them.

Packages divide and organize models in much the same way that directories organize file systems.

Packaging in UML

Packages are UML constructs that enable you to organize model elements into groups, making your UML diagrams simpler and easier to understand. Packages are depicted as file folders and can be used in any of the UML diagrams, although they are most common in class diagrams because these models have a tendency to grow.

The symbol that denotes a package in UML is shown in the figure below, representing a folder. A package has a (short and descriptive) **name** and **content** (classes and relationships among them) which is inside the package.

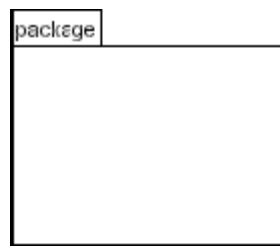


Figure 1: UML Package Symbol

When it comes to class packages there are three rules of thumb:

1. Classes in the **same inheritance hierarchy** typically belong in the same package.
2. Classes **related to one another via composition** often belong in the same package.
3. Classes that **collaborate** with each other a lot often belong in the same package.

Remember the hierarchy of Sloopies in the previous section as shown below? It could easily be wrapped up to a single package.

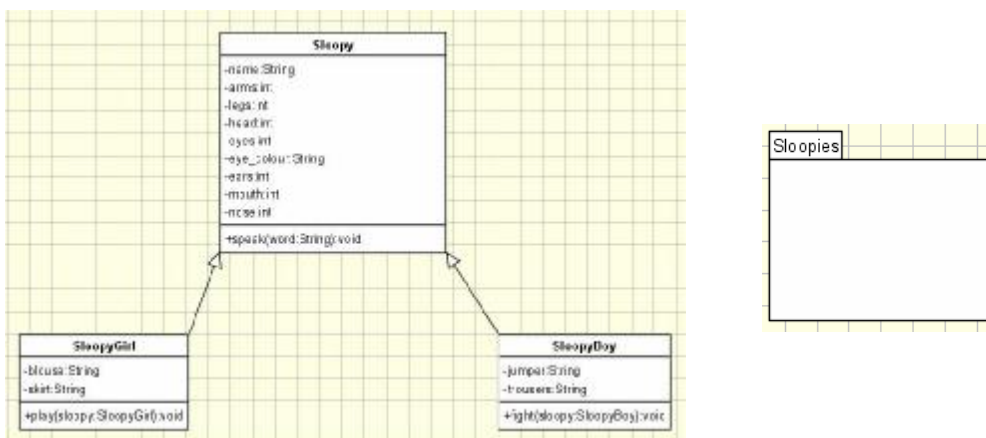


Figure 2: Sloopies Package

Dependencies between Packages

Classes can be connected via different types of relationships (inheritance, association, composition or aggregation). Packages are connected via dependencies. A **dependency** exists between elements and expresses that elements within one package use elements from the package on which it depends, implying that a change in one element may affect or supply information needed by the other element.

UML represents dependencies between two packages as a dotted line with an arrow on the depending site: ----->. Each dependency has an optional name (label), describing the nature of the dependency.

For example, the mini model below contains two packages. *Sloopies* includes all the classes shown in Figure 2 and *Toys* includes all the toy-related classes, such as *Skateboard*. Given that Sloopies can only perform their *Hobbies* (like *skating*), there exists a dependency between the two packages.

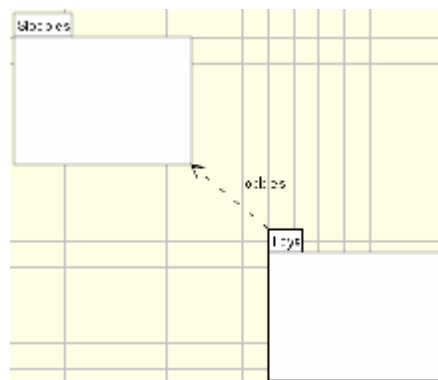


Figure 3: Dependency *Hobbies* between *Sloopies* and *Toys* packages

Namespace

According to the OMG Unified Modeling Language Specification, “a package is a **namespace** for its members, and may contain other packages. Only packageable elements can be owned members of a package. By virtue of being a namespace, a package can import either individual members of other packages, or all the members of other packages.”

This means that a package defines a namespace, so that two distinct elements contained in two distinct packages may have the same name. This is to avoid any conflict of names of classes, properties, methods or relationships when they appear in two different contexts.

For instance, take the *Toys* package in the previous figure. It has been applied in the world (or application domain) of *Sloopies*. We can give this world a name, say *SloopyDomain*, which is also known as namespace. Now, in another application that deals with retail (namespace *Retail*), we might also require a *Toys* package. However, the classes inside the package are likely to represent trade-related information, such as suppliers, stock, price, etc.

Without the concept of namespaces, a conflict would occur. It is possible to transfer information from one namespace to another via a provided import mechanism. Namespaces are critical in large applications and when packages are distributed.

The connection between packages and namespaces can also be demonstrated using the dot notation introduced earlier. For instance, let us assume we have a *Sloopies* package, a *SloopySchool* package and a *SloopyBank* package. The following three attributes have all the same name, but given the dot notation – in effect separating namespaces – they all represent different names.

```
Sloopies.SloopyName
SloopySchool.SloopyName
SloopyBank.SloopyName
```

The first *sloopyName* represents a *Sloopy* class in the *Sloopies* world, the second a registry for a student and the third a customer in a bank.



Packaging is a useful mechanism to cluster related classes into more manageable entities. This chapter dealt with packages, dependencies between packages and the related concept of namespaces.