Packages

Packages for Avoiding Name Collisions

- Modelica provide a safe and systematic way of avoiding name collisions through the package concept
- A package is simply a container or name space for names of classes, functions, constants and other allowed definitions
### Packages as Abstract Data Type: Data and Operations in the Same Place

**Keywords denoting a package**

- `encapsulated` package

- `ComplexNumber` record

- `Real re;`  
  `Real im;`  

- `ComplexNumbers.Complex` class

**Usage of the `ComplexNumber` package**

- `ComplexNumbers.Complex` class

- `ComplexNumbers.Complex  a(re=1.0, im=2.0);`  
  `ComplexNumbers.Complex  b(re=1.0, im=2.0);`  

- `ComplexNumbers.Complex  z,w;`  

- `ComplexNumbers.Complex  z,w;`  

- `equation`  
  `z = ComplexNumbers.multiply(a,b);`  
  `w = ComplexNumbers.add(a,b);`  

**Accessing Definitions in Packages**

- Access reference by prefixing the package name to definition names

- Shorter access names (e.g. `Complex`, `multiply`) can be used if definitions are first imported from a package (see next page).
Importing Definitions from Packages

The four forms of import are exemplified below assuming that we want to access the addition operation (add) of the package Modelica.Math.ComplexNumbers.

```
import Modelica.Math.ComplexNumbers;  // Access as ComplexNumbers.add
import Modelica.Math.ComplexNumbers.add;  // Access as add
import Modelica.Math.ComplexNumbers.*;  // Access as add
import Co = Modelica.Math.ComplexNumbers;  // Access as Co.add
```

Qualified Import

The qualified import statement

```
import <packagename>;
```

imports all definitions in a package, which subsequently can be referred to by (usually shorter) names

```
simplepackagename.definitionname, where the simple package name is the packagename without its prefix.
```

```
encapsulated package ComplexUser1

  import Modelica.Math.ComplexNumbers;

  class User
    ComplexNumbers.Complex a(x=1.0, y=2.0);
    ComplexNumbers.Complex b(x=1.0, y=2.0);
    ComplexNumbers.Complex z,w;
    equation
      z = ComplexNumbers.multiply(a,b);
      w = ComplexNumbers.add(a,b);
    end User;
  end ComplexUser1;
```

This is the most common form of import that eliminates the risk for name collisions when importing from several packages.
Single Definition Import

The single definition import of the form
import <packagename>.<definitionname>;
allows us to import a single specific definition (a constant or class but not a subpackage) from a package and use that definition referred to by its definitionname without the package prefix.

Unqualified Import

The unqualified import statement of the form
import <packagename>.*;
imports all definitions from the package using their short names without qualification prefixes. Danger: Can give rise to name collisions if imported package is changed.
Renaming Import

The *renaming import* statement of the form:

```plaintext
import <shortpackagename> = <packagename>
```

imports a package and renames it locally to `<shortpackagename>`. One can refer to imported definitions using `<shortpackagename>` as a presumably shorter package prefix.

```plaintext
class ComplexUser

import Co = ComplexNumbers;

Co.Complex a(x=1.0, y=2.0); 
Co.Complex b(x=1.0, y=2.0); 
Co.Complex z,w;

equation 
z = Co.multiply(a,b); 
w = Co.add(a,b);
end ComplexUser;
```

This is as safe as qualified import but gives more concise code.

Package and Library Structuring

A well-designed package structure is one of the most important aspects that influences the complexity, understandability, and maintainability of large software systems. There are many factors to consider when designing a package, e.g.:

- The name of the package.
- Structuring of the package into subpackages.
- Reusability and encapsulation of the package.
- Dependencies on other packages.
Subpackages and Hierarchical Libraries

The main use for Modelica packages and subpackages is to structure hierarchical model libraries, of which the standard Modelica library is a good example.

```modelica
encapsulated package Modelica
  // Modelica
  encapsulated package Mechanics // Modelica.Mechanics
        ...
        end Inertia;
        ...
        end Torque;
        ...
        end Rotational;
        ...
      end Mechanics;
      ...
      end Modelica;
end
```

Encapsulated Packages and Classes

An encapsulated package or class prevents direct reference to public definitions outside itself, but as usual allows access to public subpackages and classes inside itself.

- Dependencies on other packages become explicit
  – more readable and understandable models!
- Used packages from outside must be imported.

```modelica
encapsulated model TorqueUserExample1
  import Modelica.Mechanics.Rotational;  // Import package Rotational
  Rotational.Torque t2;                 // Use Torque, OK!
  Modelica.Mechanics.Rotational.Inertia w2;
  // Error! No direct reference to the top-level Modelica package
  ...
  // to outside an encapsulated class
  end TorqueUserExample1;
end
```
within Declaration for Package Placement

Use short names without dots when declaring the package or class in question, e.g. on a separate file or storage unit. Use within to specify within which package it is to be placed.

```modelica
within Modelica.Mechanics;
    import ...;
    connector Flange_a;
    ...
end Flange_a;
...
end Interfaces;
model Inertia ...
end Inertia;
...
end Rotational;
```

The within declaration states the prefix needed to form the fully qualified name.

The subpackage Rotational declared within Modelica.Mechanics has the fully qualified name Modelica.Mechanics.Rotational, by concatenating the package prefix with the short name of the package.

Mapping a Package Hierarchy into a Directory Hierarchy

A Modelica package hierarchy can be mapped into a corresponding directory hierarchy in the file system.

```
C:\library
\Modelica
    package.mo
\Blocks
    package.mo
    Continuous.mo
    Interfaces.mo
\Examples
    package.mo
    Example1.mo
\Mechanics
    package.mo
    Rotational.mo
    ...
```

Modelica

- Blocks
- Mechanics
- Rotational
- Continuous
- Interfaces
- Examples
- Example1

...
Mapping a Package Hierarchy into a Directory Hierarchy

within:
encapsulated package Modelica
  "Modelica root package";
end Modelica;

It contains an empty Modelica package declaration since all subpackages under Modelica are represented as subdirectories of their own. The empty within statement can be left out if desired.

within Modelica.Blocks;
encapsulated package Examples
  "Examples for Modelica.Blocks";
import ...;
end Examples;

end Modelica;

within Modelica.Blocks.Examples;
model Example1
  "Usage example 1 for Modelica.Blocks";
  ...
end Example1;

within Modelica.Mechanics;
encapsulated package Rotational
  encapsulated package Interfaces
  import ...;
  connector Flange_a;
  ...
end Flange_a;

end Interfaces;
model Inertia
  ...
end Inertia;
end Rotational;

The subpackage Rotational stored as the file Rotational.mo. Note that Rotational contains the subpackage Interfaces, which also is stored in the same file since we chose not to represent Rotational as a directory.