Software Design and Architectures
SE-2 / SE426 / CS446 / ECE426
Fall 2003

Assignment 6 : Modularity in Java or C++

Due in class (1pm RCH 205) on Monday November 10.

Choose to do this for Java or for C++.

In practice modules have to be represented in code and as source files. Propose a conventional representation for modules in your chosen language, to be used in a project such as your telephony project. Define the representation with attenton to the following:

coding pattern :	what syntax pattern(s) are proposed to represent a module
source filing :	what source artefacts (files, directories) and organization is proposed
export :	propose a convention for a module to declare its exports
import :	propose a convention for a module to declare its imports

Keep your proposal simple and usable. You may find that suitable mechanisms already exist, in which case you can choose to adopt them for your convention.

The proposal for export and import should be (if possible) such that the compiler or linker can be made to check conformance. (For example, the **public** attribute of Java classes, or the declare-before-use requirement of C++ **extern**s, are useful to enforce export declarations.) Your import declarations can specify a whole module or only the parts of interest. (Your export declarations must ultimately specify the parts being exported, of course.)

Keep in mind what we say a module is: it must be the right size to be an independent assignment of work to a developer, and suitable for being the boundary of information hiding and good coupling and cohesion practices. It ought to be possble to partition the code of a system into modules: that is, for every line of code in the source we ought to be able to say what module it belongs to.

Solution

For Java, I'd choose to represent a module as a package in the package hierarchy. The Java platform architecture dictates that source packages occupy a particular directory in the package directory hierarchy, with each (public or package-visible) class occupying a single source file there.

Since exactly those classes and interfaces declared **public** are visible outside the package, exports are declared by writing **public** on the exported units. The compiler checks this. There is no place where the exports of a module are listed.

The imports of a module can be specified by means of careful use of **import** in Java. Recall that **import** doesn't mean import as we defined it, it just allows out-of-package classes to be referenced without their package qualifiers. To enforce the import declaration, I would use the -classpath command option when compiling, to ensure that the compiler only had imported modules visible during compilation of the module. This could be automated somewhat by means of makefiles, where the import lists would be gathered together.

For C++, I'd probably choose to represent a module as a source-file scope as in C. In other words, a single source file of type .cc and another of type .h, and exploit the visibility rules of C/C++ for handling import and export.

A module is therefore a sequence of top-level definitions in a single file.

The exports of the module are also declared in the header file. All other definitions of the module are marked **static** to enforce export visibility.

The imports of the module are represented by **#include** statements, which together with the C++ declare-before-use law, plus the actions of the linker, ensure that only units exported from imported modules are usable.