Software Design and Architectures
CS446 / ECE452 / CS646 / SE-2

Lecture 9 : The Development View of Software Modules

Modules form the bridge between the logical (domain) view of the design and the development view. In this lecture we consider what modules have to 'look like' from a development point of view.

Designing the Module Structure
The basic module structure appears within the crosscut of a subsystem's responsibilities and the general architectural layering of the system. Coupling between modules in different subsystems can exist (makes sense) at the same layer (not at different layers because they can have no responsibilities or interests in common). Coupling between modules in different layers respects the layering exactly when an upper module couples to an immediately lower one ("morganatically"?) By accepting the layering we accept that any information or control flow 'back up' must be indirect, dynamically bound, called back, *etc.*

Within a layer/subsystem cross-cut, the assigned or perceived responsibilities may still be too many for one developer (or pair) to handle. In that case, we must split the module up into submodules. These modules (within the same subsystem *a priori*) can depend on each other, even in a circular way.

(Note that already-designed subsystems can only be "reused" when their layering assumptions are the same as those of the system being developed. Otherwise, any interaction between the system under development and the to-be-reused subsystem must be "at arms length", in other words, through a (new) proxy within the system, and low-level communication.)

Representing the Content of Modules
A module contains software units which embody the design decisions clustered with that module: or equivalently, which meet the responsibilities assigned to that module. They are called the *implementation* or the *source* of the module. Information hiding (see below) entails that a subset of these decisions be "visible externally", for other modules to couple to this module. That subset may be called the *public interface*.

A module must be tracked by and assigned to developers. So it must be easily subject to developers' operations. Historically this has meant that a module is a single source file or source management unit.

Typically coupling is specified by *built-in operations* on *names* and resolved by some *linking* mechanism (at build time) or (dynamic) *loading* mechanism (at run time).

The simplest interface is thus simply a set of names defined by one module and used by another'

It seems to me essential that the interface between two modules should be something discoverable *statically*, that is, determinable from fairly straightforward analysis of the code. The interface is not the protocol, dynamic usage, or collaboration history of two modules at run time.

The interface might be stored in a separate source file, both because its usage is different (clients have to see it) and because its change rules are different (it should change less often).

Representing and Enforcing Coupling
By *import* a module defines which other modules it can couple to. This helps to enforce coupling at design time, because you have to make a declaration first. Typically, one writes *import Module;* or some similar syntax, and then one makes references to that module are allowed within the code of the design. If "public" module interfaces are defined in the language, then *import* declares access to those (only); internal attributes of the module remain inaccessible.

Many languages, especially older ones, have no formal import facility, but designers have adopted conventional patterns ("coding styles") which serve a similar informal purpose. The goal, after all, is for module coupling to be visible.

A compiler or linker or other tool may be able to process *signatures* of the interface, in order to validate usage beyond simple name matching. In some languages it is also possible to mark imports or exports *read-only*, *read-write*, or even *write-only*, to further limit access.

In C, the "public" interface of a module is conventionally declared in a "header file" for the module; including this file is the moral equivalent of importing the module. Of course the linker doesn't care and will cheerfully couple external identifiers whether you declared them in C or not. (In C, missing external declarations are implicitly *int (*)()*.)

In C++, all identifiers must be declared, and so one must import modules somehow. Hence the *include* facility almost becomes a true *import* facility: except that one is ultimately declaring individual attributes (procedures and classes) of a module, and using those. There is no facility of importing a whole module's interface.

In Java, the *import* facility doesn't really mean *import*, it means "using", that is, the intent to use certain (visible) names from a package. One need not write *import*, but doing so allows the visible name to be used without further qualification. This helps when the name fully qualified is **javax.swing.JFace** for example. With import one just writes **JFace**. There is no way in Java to prevent coupling with some module, though.

Representing and Enforcing Cohesion
This is tricky, and ultimately impossible. One wants to choose a module representation which tends to keep in one module those design decisions that change together. Or makes it easy to reassign design responsibilities when they are seen to change together, going forward.

In keeping with our observation that bad cohesion is more easily identified and controlled than good cohesion, we should at least identify those language features and design conventions which avoid poor cohesion. Referring to the list of poor cohesions (from the previous lecture) we'll see that

- When a preprocessor or code-generation facility is generally available, then module design and interface can be "corrected" or "adjusted" during software construction, making ad-hoc cohesion tempting.
- When modules to be simple collections of procedures then mere temporal and logical cohesion are tempting
- When data substructures can be exported then mere communication cohesion is tempting
- When individual procedures can be values, arguments, or exports, then procedures tend to be modules (which is just functional cohesion)

All in all, for today, choosing an object-oriented language and making intelligent use of its features, with a minimum of cheating, is the best way to represent and enforce cohesion. For example, when a module defines a set of *types* whose values coupling modules use in the same way as they use built-in data (this is especially true in C++) then it's easier to imagine treating the module interface as a collection of cooperating operations.

Representing and Enforcing Information Hiding
By *export* a module controls which of its components (design decisions) another module may depend on. Some languages provide a facility to export *with protection* when the scope of coupling is meant to be variable.

Whereas import rules are rarely enforced in programming languages, export rules pretty nearly always are. That is: if a language provides an export facility, the linker typically adheres to it.

The exports should change less often than the code, and might well be under the control of a different work flow, or different personnel. For this reason it might make sense to

- put the exports in a separate source file
- use a module definition language or architectural definition language which enfvorce

In statically linked languages (*i.e.* C and C++) the ultimate test of visibility is linking: if a component isn't exported to the link module you can be sure it is "hidden". In dynamically linked languages, especially Java, you rely on the language's own facilities to mark exports.

There are a few different flavours of *export*, depending on what part of the architecture the export is intended for, and whether an *import* facility exists.

Information hiding isn't the same as secrecy.

Examples of Modularity Techniques In Programming Languages
In this section we look at some samples.

COBOL: single source file = single program, usually common-coupled (file structure) or stamp-coupled (unstructured parameter, especially in CICS) or (structured) parameters (CALL USING). In practice this means that functional cohesion (which isn't bad...) is about the best we can hope for.

FORTRAN: single source file = external procedure, with internal procedures and data, common-coupled by named (but unstructured) regions of shared storage.

C: N+1 source files. Sometimes a directory is a module (which would make sense) but more often several modules are collected together in a "source" directory and their headers in a public "header" directory. This means that the module interfaces are under more control, or at least are somewhat harder to change, because they're in a different place from the implementations.

C++: as for C.

Another C++ technique (described in Stroustrup chap. 8) identifies *module* with *namespace*. Since namespaces are "open" (that is, successive declarations of the form *namespace X { ... }* all contribute to namespace X) one may

a) define the module wholly within its own namespace
b) divide the source into two or more source files
c) designate one source file as a "module header" for users to include

This technique is really an extension of the C technique, in which the namespace facility is used to control visibility more finely.

Another C++ technique? each class is a module. **No!** Although classes have interfaces (public and private boundaries) they are too small, too detailed, and have too much nonmodular stuff (dynamic allocation, typing, local scope, etc) to be suitable as modules. It might be worthwhile to require conventionally that each module in a design have a designated *interface class*, for concreteness. (This is an application of the *Façade* design pattern.)

Java: If anything is a module in Java, then it must be *package*. In practice though *package* is more used for delivery and release management than for modularity properly speaking. However, *package* has the right features: large enough to contain cooperating classes and subscopes; designation of public and internal ("package-scope") material; naming.

Shell: In scripting languages typically a source file is a module, and the resulting software unit is equivalent to a command program. The interface is therefore determined by the command line, the return

value, and the environment variables consulted and set by the script. But shell scripts are well able to be much too small to be modules, so a project which made modular use of scripts should adopt a convention for representing them.

Web Scripts:

Are any of the following potentially *modules*?: a web page, a web site, a shell script?

References

B Stroustrup. *The C++ Programming Language / Third Edition*. Addison-Wesley, 1997.