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

Lecture 8 : Modularity: Coupling, Cohesion, and Information Hiding

Software within subsystems must be organized to minimise the risk and expense of expected change, and to allow individual developers to understand and be responsible for their areas of concern. The term module refers a unit of software organization at this level. We review the various structuring concerns which apply when drawing module boundaries.

Software Units
There are numerous words in our vocabulary which designate units of software: subsystem, process, module, library, component, class, procedure, package, program, application, tool, plug-in, add-on, framework; and some less well-defined words which designate cross-cutting concerns: feature, facility, capability, extension.

Many such words are needed because of the many issues in software development, not to mention sales and marketing, which must be managed, including

- maintenance pressure: correct, enhance, improve
- evolution pressure: reuse, migrate
- cognitive pressures from users and developers: understand, predict, explain, internalize
- personnel and task management pressures: assign, track,

Module
Our concern in this lecture is the *module*, which we take to be that unit of software which is

- the smallest subject of work assignment (*Jones is working on the file caching module today.*)
- the smallest unit of information hiding (*The parser uses a finite-state machine table in an array.*)
- the smallest unit of software change (*We could rewrite the output module.*)
- the structural component of subsystems (*The compiler consists of five modules: the parser, the semantic analyser, the error handler, the optimiser, and the code generator.*)

Much of what is said will apply to other kinds of software unit as well, though. Historically the *module* was introduced (by D. Parnas) with the information hiding and work assignment definitions, in order to describe and prescribe software structuring norms. It was perhaps meant to be the most colourless structuring term. As software theory and practice has developed, the module has simply taken its place within the larger domain of structuring ideas. In these lectures I prefer *unit* for the colourless term.

One might add to the above definition of *module* that many people use it in a rather static sense: a module might be loaded dynamically, but probably wouldn't be self-describing or provide unspecified or third-party functionality. Terms like *plug-in* or *component* or *extension* are more often used for this.

Dependence and Containment
Whenever we consider a collection of software units which are interrelated (according to the many kinds of dependency we have seen: protocol-base conversation, message exchange, subroutine call, event trigger, shared memory) we can draw a graph in which the units are nodes and the relationships are edges.

Systems are organizations of subsystems, and we might say that a system *contains* the subsystems in it: although this containment is not very static. The subsystems in a system might be changed tomorrow, by removing or adding subsystems to enhance or change the features, adapt the architecture, *etc.*

We tend to use *contains* when the dependence is essential, that is,

- installing the container entails all its contents
- compiling the container entails (in general) compiling its contents
- a fault in a content is a fault in the container

- A failure in a content is a failure of the container

[Diagram: lifting relationships]

A module may be taken as *containing* lower-level definitions of software entities, such as classes (with their methods and attributes), procedures (with their statements and local variables), and global static data. Modules don't contain subsystems or other modules.

Modules in Layering and Subsystems
Module boundaries don't cross subsystem boundaries or layer boundaries.

Diagram: subsystems, layers, and modules in the cross-cut. So for the remainder of this lecture, modules are the *structural units of software which inhabit layers and are aggregated into subsystems.*

Modularity Properties
A good modular design exhibits
information hiding
high cohesion
low coupling, which last two together are *functional independence*
Design decisions must from time to time be changed. We want to be able to do this as cheaply and freely as possible: that's just good engineering. So we plan to
distinguish decisions which are likely to change or ought to be easy to change
cluster (keep close together) decisions which tend to change together
control the ripple or propagation effects of change

Coupling
The term *coupling* refers vaguely to the amount and measure of connexion between modules. There are some standard terms for different kinds of coupling (from less to more desirable):
- *content coupling*: modules control each other by direct access to internal data areas and methods
- *common coupling*: modules share use of external or global ("common") data and devices
- *control coupling*: exchange of control flags etc. as parameters: early binding
- *stamp coupling*: exchange of concrete data representations
- *data coupling*: abstract data representation by parameter lists
- *message coupling*: exchange of messages in an (open) protocol

It's important to note that coupling can't be avoided, and that within a module tighter coupling is more appropriate than between modules (or between subsystems).

Coupling can be *direct* as when the design of A contains direct references to parts of B; or *indirect* as when there is an interface module or protocol converter or broker between them.

Change Propagation and Flexibility
It's a key fact that coupling is a *directed* relationship: A is coupled to B doesn't mean that B is coupled to A, or if it is, that the coupling is the same. For example, A may refer to a private data stucture of B (content coupling) but B may be maintained with no reference to A's design. In general, though, changes to B will affect A's design, possibly requiring changes to A.

Change to a module we can identify with *any change to the source code of a module*. (This might ignore the issue of change to the configuration of a system, unless we're careful to include configuration issues as an aspect of the module structure.) The presence of coupling between modules causes the phenomenon of *change propagation*: when a module A is coupled to a module B, then any change to B might entail a corresponding change to A. Another way of looking at this is inflexibility propagation: when a module A is coupled to a module B, then changes to B become harder to manage in the context of the system development as a whole.

In short, when A is coupled to B, B becomes *less flexible* and A becomes *more volatile*. But coupling is unavoidable, so we avoid those kinds of (tight) coupling which are known to accelerate or worsen this phenomenon, and prefer those kinds of (loose) coupling which weaken its effects.

Cohesion

The term *cohesion* refers vaguely to the amount of functional independence exhibited by the module by itself. A cohesive module performs a single task. The more cohesive a module is, the less its maintenance will introduce undesirable or risky coupling with other modules. Yourdon and others identify the following cohesion characteristics (from less to more desirable):

- *coincidental* – loose ad hoc relationship (e.g. every 75 lines of a long program)
- *logical* – similar function (output module)
- *temporal* – used or invoked at a similar time (login module, exit module)
- *procedural* – a collection of procedures which must be executed in a particular sequence
- *communication* – relating to a region of a data structure (relating to an aspect of a class of the conceptual domain)
- *functional* – relating to a single abstract function (relating to a method of the conceptual domain)
- *object* – relating to a single object or class in the conceptual domain and exhibiting "good" object-oriented design.

The cohesion heuristics are more useful for identifying bad design than good design, apparently. To a certain extent good cohesion has been subsumed into the more modern notion of good object-oriented design, which we'll discuss in more detail later.

Here our goal is to increase the likelihood that a change (be it bugfix, improvement, or adaptation to new requirements) should be localized within a module. It's the changes which hit everything that cause the most trouble.

Information Hiding

General goal: distinguish internal features of a module from external ones. When a module is a single class, *private* and *public* can be used to distinguish. The purpose of information hiding is to classify design decisions with respect to *scope of change*. In other words, change propagation is controlled with respect to those decisions which are likely change (rather than, as in coupling, with respect to the easy of changing them).

Some Modularity Heuristics

1. explode or implode modules from the first iteration, to improve c&c.
2. minimize direct fan-out; increase fan-in by depth; "oval"
3. keep scope of effect within scope of control (s of e: those affected by decision; s of c: those receiving control from)
4. reduce complexity and redundancy; increase consistency, at module interface
5. predictable but not restrictive
6. avoid tight coupling more rigorously the larger and more complex modules become

References:

Pressman p 361ff
D Parnas. "On A Buzzword: Hierarchical Decomposition"