Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

The Tipping Point: Stability and Instability in OO Design

Sometimes it's all about balance. Values like reusability and testability depend on designs that support independent deployabilitya goal obtained only by carefully monitoring your software's module stability.

March 01, 2005
URL:http://www.drdobbs.com/the-tipping-point-stability-and-instabil/184415285

Software is designed on many levels. On a small scale, every class and method has a design. Indeed, every line of code makes its own small contribution to overall design. However, on a larger scale, the decomposition of the system into modules represents significant design decisions that affect the reusability and testability of your software and its components.

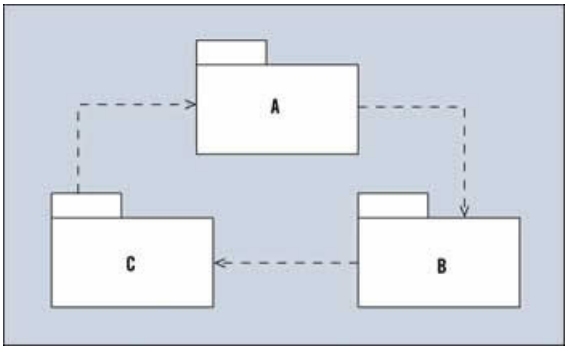
In Java, large-scale modules are typically placed in JAR files; in C#, they're placed in DLL files; and in C++, they may be placed in DLL or SO files. In each case, these files are binary components that are deployed to users. For my purposes here, I'll call these components *modules*.

Modules and packages are different things. Packages in Java and C#, and namespaces in C++, are scopes in which to declare names. They're not binary modules like JAR files; however, in well-designed software, each binary module is represented by a top-level package containing lower-level packages that comprise the code to be compiled into the binary module. Thus, we ought to see a package hierarchy in which top-level packages represent the binary modules (JAR files) and lower-level packages are wholly contained by those top-level packages.

Not all systems are decomposed into independently deployed binary modules. Some Java systems are shipped in a single JAR file, some C# systems are shipped in a single DLL file, and some C++ systems are shipped as a single EXE. However, all systems should be designed so that they *could* be broken down into independently deployable modules. The easier and more natural it is to subdivide the system into independently deployable modules, the better the system's design. Indeed, it's the lack of such independence that results in systems whose components aren't reusable or testable. Reusability and testability depend on designs that support independent deployability.

What makes a module independently deployable? In a word: *dependencies*. Consider "System A."

This system isn't made up of independently deployable modules: Module A depends on module B, which depends upon module C, which depends back on module A. This cycle of dependencies ties all the modules into a knot of codependence, and none of the individual modules can be broken out of the system and reused elsewhere.

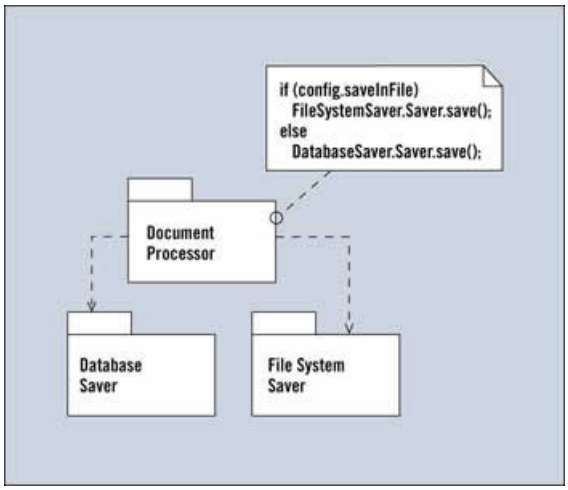


[click for larger image]
System A

For a host of reasons, dependency cycles among modules are evil. They make build order ambiguous, cause huge transitive dependency problems, and prevent independent reuse of modules. This is why good object-oriented designs follow the Acyclic Dependencies Principle (ADP), which states that the dependencies among modules must be arranged in a Directed Acyclic Graph (DAG). There can be no cycles of dependencies among modules. (See *Agile Software Development: Principles, Patterns and Practices* [Prentice Hall, 2002] for more information.)

Typically, eliminating dependency cycles among modules requires tool support. Java programmers can use JDepend to detect cycles in their module structures, and C++ users can use HeadWay. By integrating tools like these into your build process, and setting them up so that they fail when a cycle is detected, you can prevent cycles from forming (see my blog for more details.)

Eliminating cycles is the first step in creating modular designs. But creating independently deployable modules requires much more. Consider "System B."



[click for larger image]

System B

Once again, these modules aren't independently deployable. The DocumentProcessor module depends on both kinds of savers, and uses an "if" statement to choose between the two.

A better design would eliminate the "if" statement and use polymorphism, as illustrated in "System C."

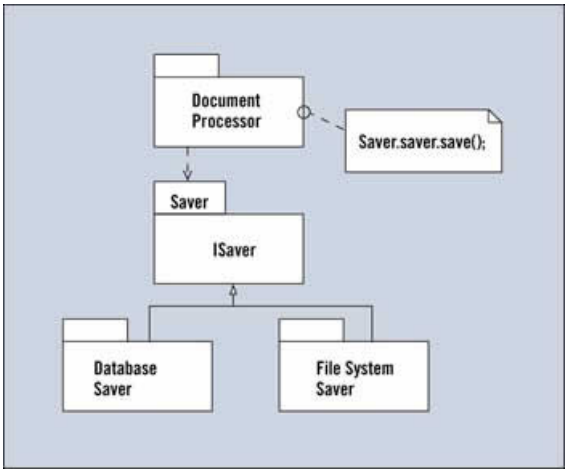
Both the DatabaseSaver and FileSystemSaver modules contain classes that implement the ISaver interface in the Saver module. The DocumentProcessor module uses the ISaver interface. Thus, DocumentProcessor has no direct dependence on DatabaseSaver or FileSystemSaver.

This design separates the modules based on their different volatilities. For example, the DatabaseSaver and FileSystemSaver represent a very volatile part of the design, and there will be many more kinds of Savers in the future. Moreover, the operation of these savers can change without affecting any other part of the system.

If the DocumentProcessor is volatile, it changes for very different reasons than do the Savers. So this design separates two different *kinds* of volatility.

The Saver module is much more stable than any of the others, and this relative lack of volatility allows the others to depend on it. They're safe from each others' changes because Saver is interposed between them.

This leads us to a simple rule, the Common Closure Principle (CCP): First, code with different kinds of volatility should be placed in different modules. Second, modules should not depend on other modules that are more volatile than they are. Rather, they should depend on less volatile ones.



[click for larger image]

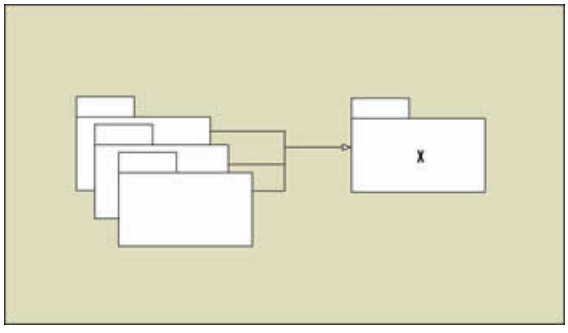
System C

Volatility is difficult to measure. However, we can quantify something that's closely related—I call it *stability*. Stability is the measure of how *difficult* a module is to change. It's a good bet that modules that are difficult to change are likely to be involatile, and vice versa. Indeed, good design ensures that volatile code goes into instable modules and involatile code goes into stable modules.

Consider "Module X." Is it stable or instable?

It has many other modules depending upon it, so if it's changed, all those other modules may need to be changed, as well. In other words, the more incoming dependencies a module has, the harder that module is to change, and thus, the more stable it is. Now consider "Module Y."

This module is easy to change because no other module depends upon it. It is instable. Moreover, module Y has many reasons to change because it depends on many other modules.



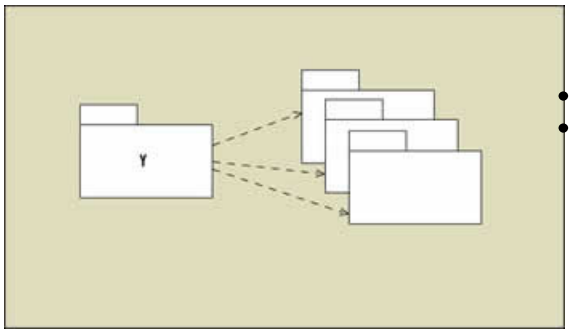
[click for larger image]

Module X

Good design tells us that we should put our volatile code into modules like Y and our involatile code into modules like X. Good design also demands that modules like X should never depend upon modules like Y. Consider what would happen if the author of X mentioned the name of some class in module Y! (See "Module X and Y.")

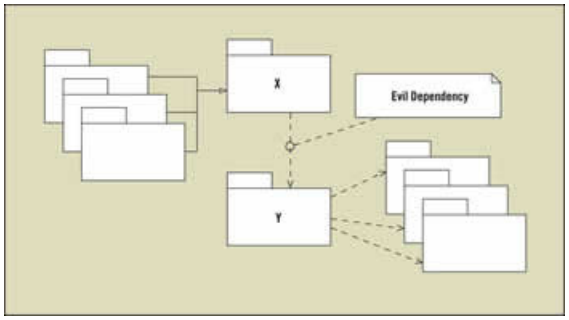
It is the perversity of software that the maintenance characteristics of a module can be ruined simply by mentioning its name. No code in module Y has changed, yet Y has suddenly been made difficult to change—a tragic event, since the designers had intentionally loaded Y with code that was *volatile!* Suddenly, all that volatile code is stuck in a module that's stable (that is, hard to change).

We can prevent this by measuring each module's stability and enforcing a simple rule—the Stable Dependencies Principle (SDP)—that decrees that



[click for larger image]

Module Y



[click for larger image]

Module X and Y

no module should depend on a module that is less stable than it is.

We measure the stability of a module by counting dependencies.

- *Ce* (fan-out) the number of modules that this module depends upon.
- *Ca* (fan-in) the number of modules that depend upon this module.

Given these two metrics, we can calculate instability as follows: *I* (Instability) = *Ce* / (*Ca* + *Ce*)

I is a metric in the range [0,1]. If *I* is 0, the module is very stable. If *I* is 1, the module is instable. The values between 1 and 0 represent varying degrees of stability.

Given this metric, we can enforce the rule by making sure that every dependency between modules points in the direction of *decreasing I*—that is, each module depends upon modules that are more stable than it is.

Again, tool support is helpful in maintaining this rule. And again, tools like JDepend or HeadWay are good places to start. They could be used during the build process to ensure that every dependency between modules points in the direction of decreasing instability.

By separating volatility in this way, we vastly increase our modules' ability to be independently deployable. However, we still have a problem. Applying this rule *guarantees* that some modules will be highly stable, and, by definition, hard to change, leaving us with inflexible modules in our designs.

Fortunately, modules that are stable aren't necessarily inflexible. The Open Closed Principle (OCP) offers an escape hatch. In short, it states that modules that are abstract can be extended without alteration. Thus, an abstract module can be flexible, even though it's hard to change.

This leads us to the most important concept: The Dependency Inversion Principle (DIP), which states that dependencies should point in the direction of abstraction. To illustrate this, how can we solve the dilemma posed by the dependency between module X and Y (See "Module X and Y")? Clearly, module Y contains some class that module X wants to use. Just as clearly, that class belongs in module Y because it shares volatility with the other classes in Y. How can we get X to use Y and not violate the SDP?

We create a new module named I, which contains an abstract class (or interface) named AC. Module X uses this class, and module Y implements it, thus resolving all the issues. X no longer depends upon Y. The volatile code in Y stays in Y. And both X and Y depend on a module (I) that is more stable than either of them. Moreover, both X and Y "depend" in the direction of abstraction. (See "Solution: Module X and Y.")

Thus, the more stable a module is, the more abstract it should be, otherwise known as the Stable Abstractions Principle (SAP). (See "SAP Visualized" for a graphical representation of the tenet.)

If we plot the instability of a module against its abstractness, the module should fall close to the line named "The Main Sequence." Modules on that line are as abstract as they are stable, or are as concrete as they are instable.

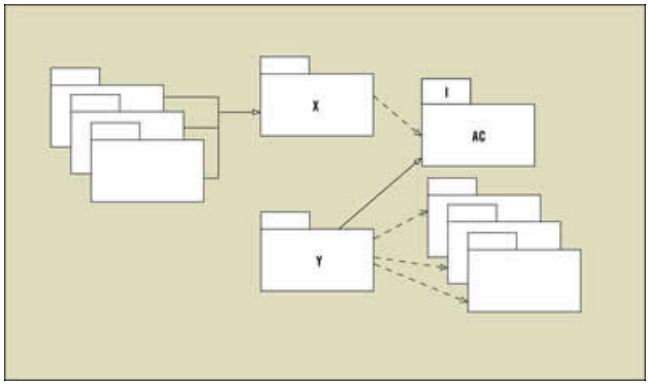
Abstractness is a simple metric to measure.

A (Abstractness) = *Na* / *Nc*, where *Na* is the number of abstract classes (or interfaces) in the module, and Nc is the number of classes in the module.

This metric falls in the range of [0,1], just as the graph requires. Again, tools like JDepend and HeadWay can calculate this metric.

This leaves us with one final metric to calculate: Given a module, how far from the main sequence is it? This metric, called *D*, can be calculated as follows: *D* (distance) = |*A* + *I* - 1|

This metric also falls in the range of [0,1], where 0 means that the

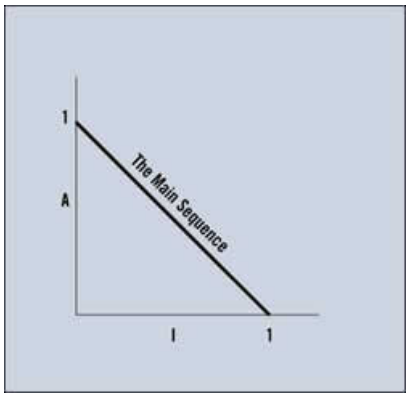


[click for larger image]

Solution: Module X and Y

module is *on* the main sequence, and 1 means that it's as far from the main sequence as possible.

Once again, tools like JDepend and HeadWay can calculate these metrics. Moreover, we can integrate those calculations into the build process with tools like FitNesse coupled with the ModuleMetrics fixture. A system that's well designed for independent deployability, testability and reuse will generally have its modules clustered around the main sequence, with *D* metrics that are close to zero. That said, *these metrics should never be considered laws or rules*; they are, at best, crude indicators of structure. Some modules will have high *D* metrics, yet be harmless to the system structure. And other modules with *D* metrics very close to zero don't help the system's structure in any way. Informed interpretation of these metrics is inevitably essential. No manager should ever mandate *D* = 0 for all modules.



[click for larger image]

SAP Visualized

The Golden Rules

Now let's put this all together. The module structure of a well-designed system should abide by the following tenets:

1. It contains no dependency cycles. (ADP)
2. Every dependency between modules should terminate on a module whose *I* metric is less than or equal to the depending module's *I* metric. (SDP)
3. Every dependency between modules should terminate on a module whose *A* metric is greater than or equal to the depending module's *A* metric. (SAP)

Any module with a *D* metric that is very close to zero (less than .05 or so) will conform closely to points two and three.

Conforming to these three rules will help create systems that comprise independently deployable modules—systems with a high degree of reusability and testability.

Furthermore, maintaining these three rules by using automated tools that are invoked as part of the build process will help ensure that the system's design doesn't slowly degrade over time.

Robert C. Martin *is CEO, president and founder of Object Mentor Inc., in Vernon Hills, Ill., and a frequent speaker at SD conferences.*