Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

Reducing Coupling

Martin Fowler

One of the earliest indicators of design quality was coupling. It appeared, together with cohesion, in the earliest works in structured design, and it has never gone away. I still always think of it when considering a software design.

There are several ways to describe coupling, but it boils down to this: If changing one module in a program requires changing another module, then coupling exists. It might be that the two modules do similar things at one point, so the code in one module is effectively a duplicate of the code in the other. This is an example of the primary and obvious sin of duplication. Duplication always implies coupling, because changing one piece of duplicate code implies changing the other. It can also be hard to spot, because there might not be an obvious relationship between the two pieces of code.

Coupling also occurs when code in one module uses code from another, perhaps by calling a function or accessing some data. At this point, it becomes clear that, unlike duplication, you can't treat coupling as something to always avoid. You can break a program into modules, but these modules will need to communicate in some way—otherwise, you'd just have multiple programs. Coupling is desirable, because if you ban coupling between modules, you have to put everything in one big module. Then, there would be lots of coupling—just all hidden under the rug.

So coupling is something we need to control, but how? Do we worry about coupling everywhere, or is it more important in some places than others? Which factors make coupling bad, and which are permissible?

Look at dependencies

I concern myself most with coupling at the highest-level modules. If we divide a system into a dozen (or fewer) large-scale pieces, how are these pieces coupled? I focus on the coarser-grained modules, because worrying about coupling everywhere is overwhelming. The biggest problems come from uncontrolled coupling at the upper levels. I don't worry about the number of modules coupled together, but I look at the pattern of dependency relationship between the modules. I also find a diagram very helpful.

When I use the term *dependency*, I use it as defined in the Unified Modeling Language (UML). So, the UI module depends on the domain module if any code in the UI module references any code in the domain model—by

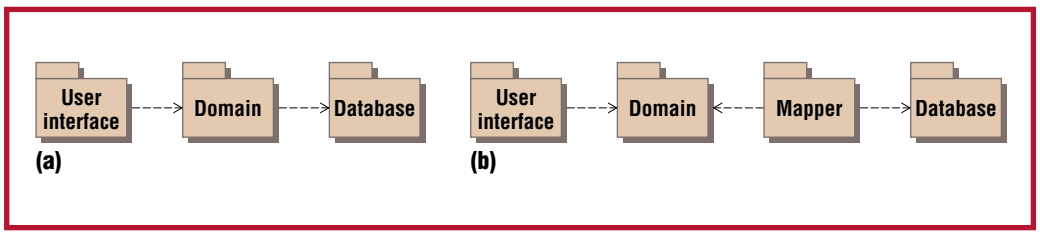


Figure 1. (a) A simple package diagram and (b) a mapper package that insulates the domain and database packages from each other.

0740-7459/01/$10.00 © 2001 IEEE

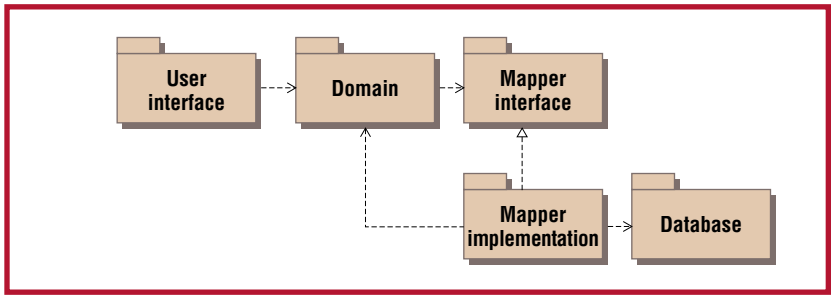


Figure 2. Introducing an interface implementation split.

calling a function, using some data, or using types defined in the domain module. If someone changes the domain, there is a chance the UI model will also need to change. A dependency is unidirectional: The UI module usually depends on the domain module, but not the other way around. We would have a second dependency if the domain module also depended on the UI module.

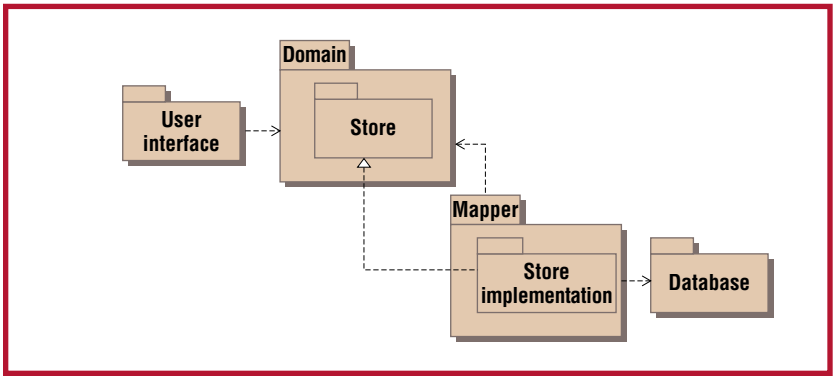
UML dependencies are also nontransitive. If the UI module depends on the domain module, and the domain module depends on the database module, we can't assume that the UI module depends on the database module. If it does, we must describe this with an extra dependency directly between UI and database modules. This nontransitivity is important because it lets us show that the domain model insulates the UI from changes in the database. Thus, if the database's interface changes, we don't immediately have to worry about a change in the UI. The UI will only change if the change in the database causes a big enough change in the domain that the domain's interface also changes.

Figure 1a shows how I'd diagram this using UML notation. The UML is designed for an OO system, but the basic notion of modules and dependencies applies to most styles of software. The UML name for this kind of high-level module is *package*, so I'll use that term from now on (so the UML police won't arrest me!). Because these are packages, I call this kind of diagram a *package diagram* (although strictly in UML, it's a *class diagram*).

What I'm describing here is a layered architecture, which should be familiar to anyone who works in information systems. The layers in an information system make good fodder for describing things we must consider when thinking about dependencies.

A common piece of advice regarding dependency structures is to avoid cycles. Cycles are problematic, because they indicate that you can get in a situation in which every change breeds other changes that come back to the original package. Such systems are harder to understand because you have to go around the cycle many times. I don't view the need to avoid cycles

Figure 3. Defining an interface in one package that's implemented by another.



between packages as a strict rule—I'll tolerate them if they're localized. A cycle between two packages in the same layer of an application is less of a problem.

A mapper package

In Figure 1a, all the dependencies run in a single direction. This is a sign—but not requirement—of a well-controlled set of dependencies. Figure 1b shows another common feature of information systems, when a mapper package separates the domain from the database. (A *mapper* is a package that provides insulation in both directions.) The mapper package provides insulation in both directions, which lets the domain and database change independently of each other. As a result, you often find this style in more complex OO models.

Of course, if you think of what happens when you load and save data, you realize that this picture isn't quite right. If a module in the domain needs some data from the database, how does it ask for it? It can't ask the mapper, because if it could, it would introduce a dependency from the domain to the mapper, which would be a cycle. To get around this problem, I need a different kind of dependency.

So far, I've talked about dependencies in terms of code using other parts of code. But there's another kind—the relationship between an interface and its implementation. An implementation depends on its interface but not vice versa. In particular, any caller of an interface depends only on the interface, even if a separate module implements it.

Figure 2 illustrates this idea. The domain depends on the interface but not the implementation. The domain won't work without some mapper implementation, but only changes in the interface would cause the domain to change.

In this situation, there are separate packages, but this isn't necessary. Figure 3 shows a store package contained within the domain, implemented by a store implementation contained within the mapper. In this case, the domain defines the interface for the mapper. It's essentially saying that the domain package will work with any mapper that chooses to implement the store interface.

Defining an interface in a module that a separate module intends to implement is a fundamental way to break dependencies and reduce coupling. This approach appears in many forms, the most primitive of which is the *call back*. In this form, the caller is asked to supply a reference to a function with a certain signature, which is called later. A common example in the Java world is a listener. Because listeners are classes, they are more explicit, which clarifies things.

Another example is a module defining an event that it passes out, to which others can react. You can think of an event as defining an interface to which the listening module conforms. The caller of the call back function, the definer of the listener interface, and the producer of the event know nothing about the module that actually is called, so there's no dependency.

I'm left feeling a lack of closure, because much of what I've said involves weasel words such as "well-controlled dependencies." It's difficult to offer hard pieces of guidance when trying to define a well-controlled set of dependencies. Certainly, it's about reducing the amount of dependencies, but that's not the whole issue. The direction of the dependencies and the way they flow, such as to avoid big cycles, is also important. In addition, I treat all dependencies the same, without considering the interface's width. It seems that worrying too much about what you depend on is less important than the fact that there is a dependency there.

The basic rule that I follow is to visualize my high-level dependencies and then rationalize them, separating the interface and implementation to break dependencies I don't want. Like so many design heuristics, this seems awfully incomplete. Yet, I have found it useful—and in the end, that is what counts.

Martin Fowler is the chief scientist for ThoughtWorks, an Internet systems delivery and consulting company. Contact him at fowler@acm.org; http://martinfowler.com.

IEEE Software

How to Reach Us

Writers
For detailed information on submitting articles, write for our Editorial Guidelines (software@computer.org), or access computer.org/software/author.htm.

Letters to the Editor
Send letters to

Letters Editor
IEEE Software
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
software@computer.org

Please provide an email address or daytime phone number with your letter.

On the Web
Access computer.org/software for information about *IEEE Software*.

Subscription Change of Address
Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software*.

Membership Change of Address
Send change-of-address requests for the membership directory to help@computer.org.

Missing or Damaged Copies
If you are missing an issue or you received a damaged copy, contact help@computer.org.

Reprints of Articles
For price information or to order reprints, send email to software@computer.org or fax +1 714 821 4010.

Reprint Permission
To obtain permission to reprint an article, contact William Hagen, IEEE Copyrights and Trademarks Manager, at whagen@ieee.org.