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

Module Assembly

Martin Fowler

When people talk about modularity, they usually focus on separating the interface and implementation. The whole of point of a module is that it should hide a secret in its implementation that isn't apparent from the interface. A user can then take advantage of what the module does without knowing all the gory details of its implementation.

For example, say we have a program that prints the names and addresses of people who have a birthday this week. To get this information, the program uses a separation address-book module that stores the names and addresses. By making the address book a separate module, we can hide the storage mechanism from the birthday printer.

To take advantage of this, the writer of the print module must program only to the address book's interface without taking advantage of its implementation. Mechanisms that make only the interface visible could enforce this, but even if the implementation is visible, it's just as important to program only to the interface.

Hiding the implementation accomplishes a couple of things. First, it means that if I'm writing the birthday printer module, I don't have to worry my ugly little head about the details of the storage mechanism. Second, I can change the storage mechanism (perhaps switching from a file to a database) without changing the birthday printer—that is, different implementations of the interface can substitute for each other.

Substitutability is important, but it raises another question: How do we select which implementation to talk to? After all, despite the fact that the birthday printer knows only about the interface, it has to talk to an implementation. If several implementations are available, it must talk to the correct one and let us easily change implementations.

Assembly through linkage

One way to sort this out is through a linkage mechanism. A simple C program might handle this by dividing the task into three files. `AddressBook.h` would be a header file that defines the address book interface. `AddressBook1.c` would be an implementation (and there could be others), and `BirthdayPrinter.c` would be the birthday printer program. To compile BirthdayPrinter, I just include `AddressBook.h`. To get a fully running program, I would link it to `AddressBook1` when I link.

Such a scheme performs some separation, but it means I must choose my address book implementation at link time. I can defer this decision, using dynamic link mechanisms that defer the decision of which address book implementation to use until runtime. If I were doing this in Windows, I might package BirthdayPrinter into an exe file and put an address book implementation into a dll. The main program could then load the dll to get the implementation. Different computers can use different address book implementations by installing the correct dll for their environment.

Assembly through linkage works to a point, but it's somewhat limited because my physical packaging depends on my module structure. I must package alternative implementations into separate physical packages, and I also must juggle around with linkage to get the right

0740-7459/04/$20.00 © 2004 IEEE

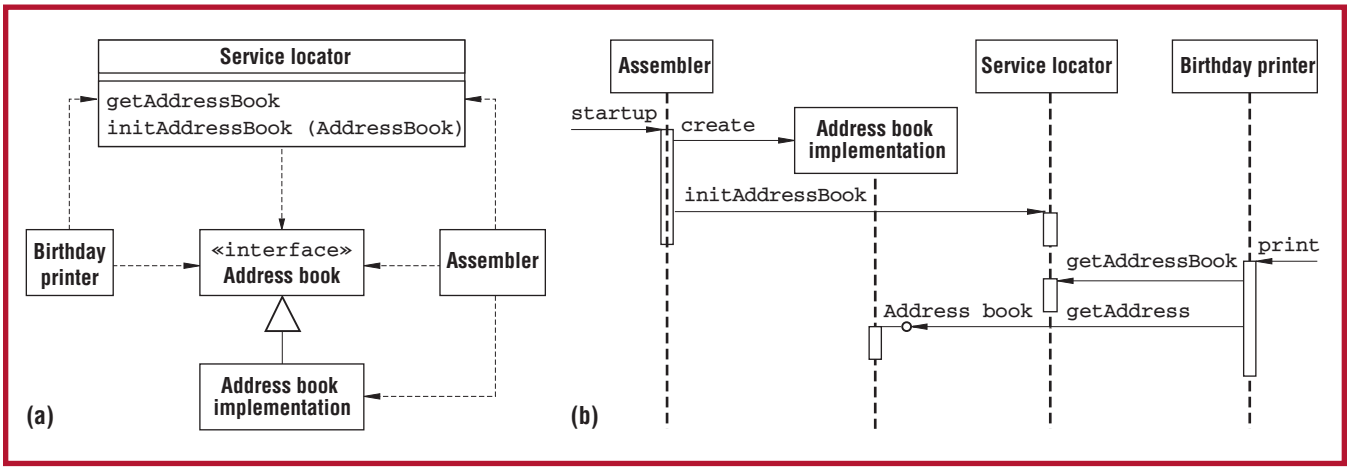

Figure 1. (a) UML class and (b) sequence diagrams for a service locator. With a service locator, an assembler loads implementations into the locator at startup and then clients call the locator to obtain suitable implementations.

implementations in place. Once you go beyond a simple program, such juggling becomes messy.

Module assemblers

A more sophisticated approach lets me package multiple executables into a single physical package, and then I can choose which ones to use through a separate assembler module. The assembler typically runs at application startup and chooses which module implementations to use.

Because the whole point of this module assembly is to let people vary their choice of modules with each deployment, the common approach is to have a configuration file that lets a deployer indicate which implementations should be used. You then have a single module assembler that reads the configuration file and prepares the assembly to suit. These configuration files are commonly done as text files. The current fashion these days is to have them as XML files. XML is good at handling hierarchical data, so it's often a good choice.

But another way that's often overlooked is to write the assembly description using a programming language. Most programming languages, especially scripting languages, can be more readable than XML. More importantly, a programming language can take an active role, probing the environment and making choices, similar to how modern hardware installers work.

People often object to using a programming language because they want to avoid compilation or think that nonprogrammers should be able to perform assembly description. But neither view holds much water.

Compilation is an issue in some environments but not all. In some cases you can compile the assembler separately and use dynamic link assembly to bind the assembler into the main program. If such a separate compilation isn't an option, you can consider a scripting language. Many development environments let you combine scripting languages with an application in another language—and scripting languages can make very readable and powerful assemblers.

Although nonprogrammers can make some simple assembly choices, most of the issues in a complicated assembly must be done by technical people who shouldn't be scared by a programming language. I've seen complex configuration files that are pretty much a programming language themselves, in which case you might as well use a full language.

I'm not saying that you should never use simple configuration files—after all, if your configuration is simple, then you don't need a programming language. However, a complex configuration file that has a bad smell might imply deeper problems with the application's configurability.

But the important point about assembly descriptions is not whether they are written in some nonprogramming environment—it's that they are defined separately from module usage.

Applications include a separate area for module assembly that decides which implementations to use. This assembler is packaged separate from the application and its components so that we can easily change it with each deployment and so that changes to the module assembly do not change the core application.

Patterns for finding an implementation

There are a couple of common styles for how modules find the appropriate implementation. One is the Service Locator, in which one module essentially takes responsibility for being the clearinghouse for finding module implementations (see Figure 1). The assembler's job is to plug the right implementations into the locator, and, once completed, any code that needs a variable module can ask the locator for an implementation.

Another approach is what I call *dependency injection*. With dependency injection, each module has fields for every other module implementation on which it depends (see Figure 2). It then declares its dependencies in some well-known way—using, for example, constructor arguments, setting methods, or an injection interface. The assembler then loads implementations into those fields during module startup. (I have a

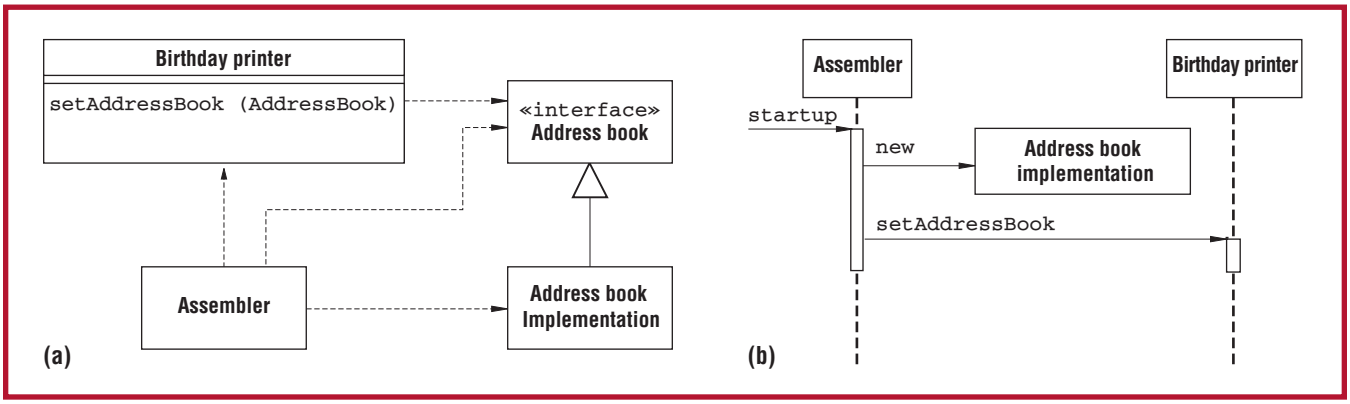

Figure 2. (a) UML class and (b) sequence diagramas for dependency injection. With a dependency injector, the assembler puts implementations directly in properties of the client classes.

longer article on these strategies at http://martinfowler.com/articles/injection.html).

Internal assembly

So far I've addressed this issue from the perspective of coarse-grained assembly of components, typically where an application combines components from different teams or building systems that need different modules in different deployments. But the principle of separating assembly from usage applies even in small-scale situations; indeed, it's one of the essential (and oft missed) attributes of object-oriented development.

Consider a stereotypical example of inheritance and polymorphism (see Figure 3). The reason this works well is that `Order` is connected to the correct `Customer` subsclass once at the beginning. However, all the logic that varies with the `Customer` type is then correctly selected by polymorphism when it delegates to the customer. This is the same principle of separate modules that I talked about earlier but at a much finer level of granularity. You can do this comfortably in an OO language because inheritance and polymorphism provide a very straightforward mechanism for substitutability.

This is even more apparent in a dynamically typed language. In a static-typed language, you can only substitute for a defined interface (or superclass) using inheritance. With a dynamically typed language, you look at the client to see what operations the client uses and can substitute any object that implements that set of operations. As soon as you connect to the substitute class, you are making a single configuration decision that's separate from the uses of that class.

One of OO's strengths—indeed, probably the biggest strength—is that it encourages these modular principles right through the language. Although inheritance and polymorphism aren't perfect for every case, they do handle many cases in a handy way. The pity is that although OO languages are becoming more mainstream, they are usually used in a way that doesn't support this kind of polymorphic assembly. I suspect that although OO languages are quite mainstream, it will still be a while before OO programming is widely understood.

When you're using substitutable modules at this fine-grained level, it isn't always as important to separate the configuration and use as widely. It's quite common to see code such as

```
class Company …
  private List movies =
    new ArrayList()
```

In this case, the using class itself chooses the implementation, so there isn't a separate assembler in play. Often this more minimal separation is fine. However, notice that the field type is set by the interface, not by the implementation. The principle that you should program to interfaces still holds. Generally, you should always use the most abstract interface or class possible in type declarations.

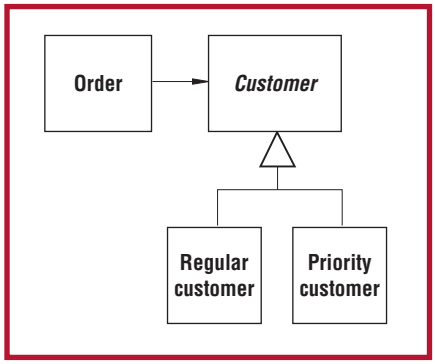

Figure 3. Instances of `Order` are connected to `Customer` subclasses, but an instance of order doesn't know which subclass it's connected to.

How to modularize a program is one of the most general and important issues in software design. Approaches such as object orientation, aspect orientation, components, and services are all different twists to modularization. Whatever route you take, separating the interface from the implementation and separating configuration from use are two vital principles in a good modularization scheme.

Acknowledgments

My thanks to my colleagues Bill Caputo, Paul Hammant, Dave Pattinson, Jack Bolles, and Mike Roberts for comments on drafts of this column.

Martin Fowler is the chief scientist for ThoughtWorks, a systems delivery and consulting company. Contact him at fowler@acm.org.