Module Dependence Diagrams

6.170 Lecture 10

Fall 2005

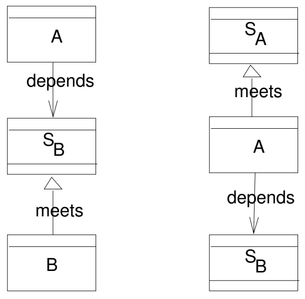
A central issue in designing software is how to decompose a program into parts. In this lecture, we'll introduce some fundamental notions for talking about parts and how they relate to one another. Our focus will be on identifying the problem of *coupling* between parts, and showing how coupling can be reduced.

A good programming language allows you to express the dependences between parts, and control them – preventing unintended dependences from arising. In this lecture, we'll show the features of Java can be used to express and tame dependences through a simple coding exercise, that illustrates in particular the role of interfaces.

1 Module Dependency Diagrams

Let's start with introducing the module dependency diagram (MDD). An MDD shows two kinds of program parts: implementation parts (classes in Java) shown as boxes with a single extra stripe at the top, and specification parts shown as boxes with a stripe at the top and bottom. Organizations of parts into groupings (such as packages in Java) can be shown as contours enclosing parts in Venn-diagram-style.

A plain arrow with an open head connects an implementation part `A` to a specification part `S`, and says that the meaning of `A` depends on the meaning of `S`. Since the specification `S` cannot itself have a meaning dependent on other parts, this ensures that a part's meaning can be determined from the part itself and the specifications it depends on, and nothing else. An arrow from an implementation part `A` to a specification part `S` with a closed head says that `A` meets `S`: its meaning conforms to that of `S`. By introducing a specification part `S`, we can say that `A` depends on `S` and `B` meets `S`. The diagrams below illustrate this; note the use of two double lines to distinguish specification parts from implementation parts.

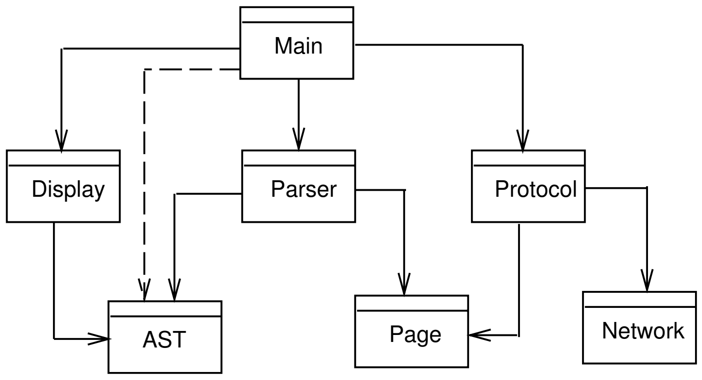


Each arc incurs an obligation. The writer of `A` must check that it will work if it is assembled with a part that satisfies the specification `S`. And 'works' is now defined by explicitly meeting specifications: `B` will be usable in `A` if it works according to the specification `S`, and `A` will be deemed to work if it meets whatever specification is given for its intended uses – `T` say. The diagram on the right shows this. It's the same depends-meet chain centered on an implementation part rather than a specification part.

Because specifications are so essential, we will always assume they are present. Most of the time, we will not draw specification parts explicitly, and so a dependence arrow between two implementation parts `A` and `B` should be interpreted as short for a dependence from `A` to the specification of `B`, and a meets arrow from `B` to its specification. We will show Java interfaces as specification parts explicitly.

A dotted arrow from `A` to `B` is a weak dependence; it says that `A` depends only the existence of a part satisfying the specification of `B`, but actually has no dependence on any details of `B`. In other words, `A` references the name of `B`, but not the name of any of its members. `A` knows that the class or interface `B` exists, and refers to variables of that type, but calls no methods of `B`, and accesses no fields of `B`.

The MDD for a browser is shown below.



The `Main` part uses the `Protocol` part to engage in the HTTP protocol, the `Parse` part to parse the HTML page received, and the `Display` part to display it on the screen. These parts in turn use other parts. `Protocol` uses `Network` to make the network connection and to handle the low-level communication, and `Page` to store the HTML page received. `Parser` uses the part `AST` to create an abstract syntax tree from the HTML page – a data structure that represents the page as a logical structure rather than as a sequence of characters. `Parser` also uses `Page` since it must be able to access the raw HTML character sequence.

In our browser, for example, the abstract syntax tree in AST may be accessible as a global name (using the Singleton pattern, which we'll see later). But for various reasons – we might for example later decide that we need two syntax trees – it's not wise to use global names in this way. An alternative is for the `Main` part to pass the `AST` part from the `Parse` part to the `Display` part. This would induce a weak dependence of `Main` on `AST`. The same reasoning would give a weak dependence of `Main` on `Page` (not shown in figure).

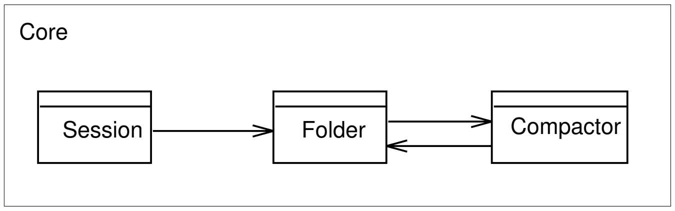
2 Example: Instrumenting a Program

For the remainder of the lecture, we'll study some decoupling mechanisms in the context of an example that's tiny but representative of an important class of problems.

Suppose we want to report incremental steps of a program as it executes by displaying progress line by line. For example, in a compiler with several phases, we might want to display a message when each phase starts and ends. In an email client, we might display each step involved in downloading email from a server. This kind of reporting facility is useful when the individual steps might take a long time or are prone to failure (so that the user might choose to cancel the command that brought them about). Progress bars are often used in this context, but they introduce further complications (marking the start and end of an activity, and calculating proportional progress) which we won't worry about.

As a concrete example, consider an email client that has a package core that contains a class `Session` that has code for setting up a communication session with a server and downloading messages, a class `Folder` for the objects that models folders and their contents, and a class `Compactor` that contains the code for compacting the representation of folders on disk. Assume there are calls from `Session` to `Folder` and from `Folder` to `Compactor`, but that the resource intensive activities that we want to instrument occur only in `Session` and `Compactor`, and not in `Folder`.

The module dependency diagram shows that `Session` depends on `Folder`, which has a mutual dependence on `Compactor`.



We'll look at a variety of ways to implement our instrumentation facility, and we'll study the advantages and disadvantages of each. Starting with the simplest, most naive design possible, we might intersperse statements such as

```
System.out.println ("Starting download");
```

throughout the program.

2.1 Abstraction by Parameterization

The problem with this scheme is obvious. When we run the program in batch mode, we might redirect standard out to a file. Then we realize it would be helpful to timestamp all the messages so we can see later, when reading the file, how long the various steps took. We'd like our statement to be

```
System.out.println ("Starting download at: " + new Date ());
```

instead. This should be easy, but it's not. We have to find all these statements in our code (and distinguish from other calls to `System.out.println` that are for different purposes), and alter each separately.

Of course, what we should have done is to define a procedure to encapsulate this functionality. In Java, this would be a static method:

```
public class StandardOutReporter {
     public static void report (String msg) {
         System.out.println (msg);
     }
}
```

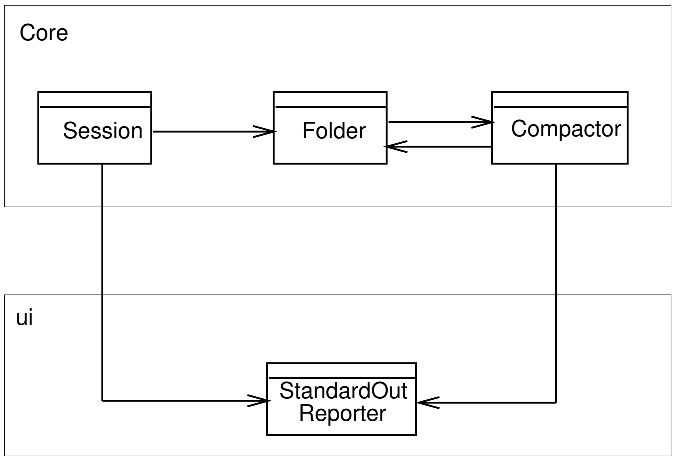
Now the change can be made at a single point in the code. We just modify the procedure:

```
public class StandardOutReporter {
     public static void report (String msg) {
          System.out.println (msg + " at: " + new Date ());
     }
}
```

The mechanism in this case is one you're familiar with: what 6001 called abstraction by parameterization, because each call to the procedure, such as

```
StandardOutReporter.report ("Starting download");
```

is an instantiation of the generic description, with the parameter `msg` bound to a particular value. We can illustrate the single point of control in a module dependence diagram. We've introduced a single class on which the classes that use the instrumentation facility depend: `StandardOutReporter`. Note that there is no dependence from `Folder` to `StandardOutReporter`, since the code of `Folder` makes no calls to it.



2.2 Decoupling with Interfaces

This scheme is far from perfect though. Factoring out the functionality into a single class was a good idea, but the code still has a dependence on the notion of writing to standard out. If we wanted to create a new version of our system with a graphical user interface, we'd need to replace this class with one containing the appropriate GUI code. That would mean changing all the references in the core package to refer to a different class, or changing the code of the class itself, and now having to handle two incompatible versions of the class with the same name. Neither of these is an attractive option.

In fact, the problem is even worse than that. In a program that uses a GUI, one writes to the GUI by calling a method on an object that represents part of the GUI: a text pane, or a message field. In Swing, Java's user interface toolkit, the subclasses of `JTextComponent` have a `setText` method. Given some component named by the variable `outputArea`, for example, the display statement might be:

```
outputArea.setText (msg)
```

How are we going to pass the reference to the component down to the call site? And how are we going to do it without now introducing Swing-specific code into the reporter class?

Java interfaces provide a solution. We create an interface with a single method `report` that will be called to display results.

```
public interface Reporter {
     void report (String msg);
}
```

Now we add to each method in our system an argument of this type. The `Session` class, for example, may have a method `download`:

```
void download (Reporter r, ...) {
     r.report ("Starting downloading" );
          ...
}
```

Now we define a class that will actually implement the reporting behavior. Let's use standard out as our example as it's simpler:

```
public class StandardOutReporter implements Reporter {
     public void report (String msg) {
          System.out.println (msg + " at: " + new Date ());
     }
}
```

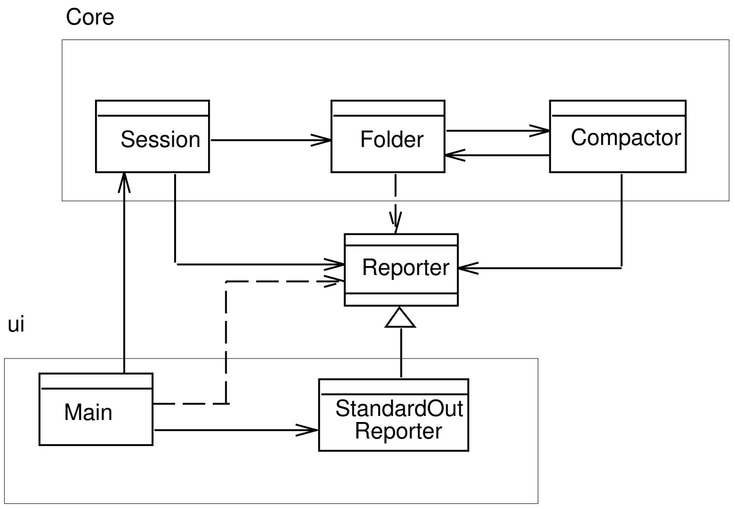
This class is not the same as the previous one with this name. The method is no longer static, so we can create an object of the class and call the method on it. Also, we've indicated that this class is an implementation of the `Reporter` interface. Of course, for standard out this looks pretty lame and the creation of the object seems to be gratuitious. But for the GUI case, we'll do something more elaborate and create an object that's bound to the particular widget:

```
public class JTextComponentReporter implements Reporter {
     JTextComponent comp;
     public JTextComponentReporter (JTextComponent c) {comp = c;}
     public void report (String msg) {
          comp.setText (msg + " at: " + new Date ());
     }
}
```

At the top of the program, we'll create an object `port` and pass it in to a `Session` object's `download` method:

```
Reporter port = new StandardOutReporter ();       // in Main
session.download (port, ...);                     // in Session
```

Now we've achieved something interesting. The call to `report` now executes, at runtime, code that involves `System.out`. But methods like download only depend on the interface `Reporter`, which makes no mention of any specific output mechanism. We've successfully decoupled the output mechanism from the program, breaking the dependence of the core of the program on its I/O.



Look at the module dependency diagram. Recall that an arrow with a closed head from `A` to `B` is read `A` meets `B`. `B` might be a class or an interface; the relationship in Java may be implements or extends. Here, the class `StandardOutReporter` meets the interface `Reporter`. Similarly, the class `JTextComponentReporter` will also meet `Reporter`, and there will be a dependence on `Main` on `JTextComponentReporter` (not shown in figure).

The key property of this scheme is that there is no longer a dependence of any class of the `core` package on a class in the `ui` package. All the dependences point downwards (at least logically!) from `ui` to `core`. To change the output from standard output to a GUI widget, we would simply replace the class `StandardOutReporter` by the class `JTextComponentReporter`, and modify the code in the main class of the `ui` package to call its constructor on the classes that actually contain concrete I/O code. We just modify the code as:

```
Reporter port = new JTextComponentReporter(outputArea);         // in Main
```

and pass the object to `Session` as before. This idiom is perhaps the most popular use of interfaces, and is well worth mastering.

Recall that the dotted arrows are weak dependences. A weak dependence from `A` to `B` means that `A` references the name of `B`, but not the name of any of its members. In other words, `A` knows that the class or interface `B` exists, and refers to variables of that type, but calls no methods of `B`, and accesses no fields of `B`.

The weak dependence of Main on `Reporter` simply indicates that the Main class may include code that handles a generic reporter; it's not a problem. The weak dependence of `Folder` on `Reporter` is a problem though. It's there because the `Reporter` object has to be passed via methods of `Folder` to methods of `Compactor`. Every method in the call chain that reaches a method that is instrumented must take a `Reporter` as an argument. This is a nuisance, and makes retrofitting this scheme painful.

2.3 Static Fields

The clear disadvantage of the scheme just discussed is that the reporter object has to be threaded through the entire core program. If all the output is displayed in a single text component, it seems annoying to have to pass a reference to it around. In dependency terms, every module has at least a weak dependence on the interface `Reporter`.

Global variables, or in Java static fields, provide a solution to this problem. To eliminate many of these dependences, we can hold the reporter object as a static field of a class:

```
public class StaticReporter {
     static Reporter r;
     static void setReporter (Reporter r) {
          this.r = r;
     }
     static void report (String msg) {
          r.report (msg);
     }
}
```

Now all we have to do is set up the static reporter at the start:

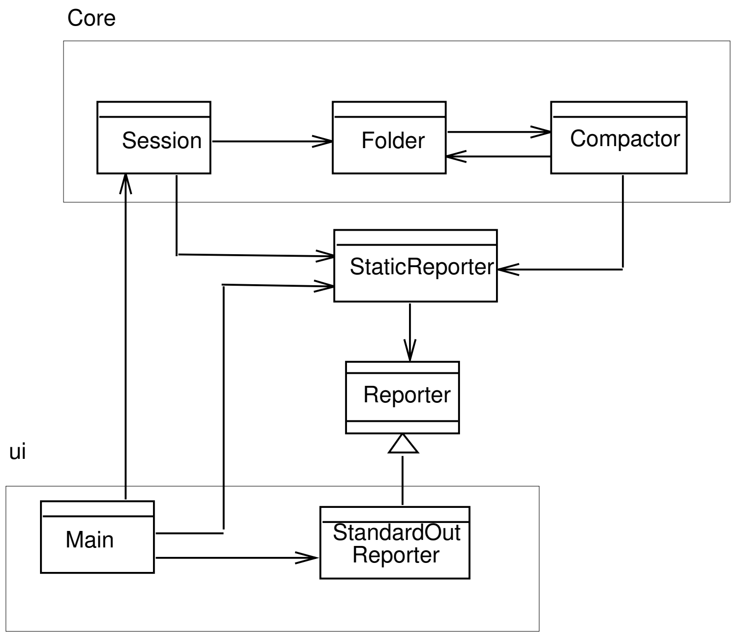
```
StaticReporter.setReporter (new StandardOutReporter ());
```

and we can issue calls to it without needing a reference to an object:

```
void download (...) {
     StaticReporter.report ("Starting downloading" );
          ...
}
```

In the module dependency diagram (see below), the effect of this change is that now only the classes that actually use the reporter are dependent on it.

Notice how the weak dependence of `Folder` has gone. We've seen this global notion before, of course, in our second scheme whose StandardOutReporter had a static method. This scheme combines that static aspect with the decoupling provided by interfaces.



Global references are handy, because they allow you to change the behavior of methods low down in the call hierarchy without making any changes to their callers. But global variables are dangerous. They can make the code fiendishly difficult to understand. To determine the effect of a call to `StaticReporter.report`, for example, you need to know what the static field `r` is set to. There might be a call to `setReporter` anywhere in the code, and to see what effect it has, you'd have to trace executions to figure out when it is executed relative to the code of interest.

Another problem with global variables is that they only work well when there is really one object that has some persistent significance. Standard out is like this. But text components in a GUI are not. We might well want different parts of the program to report their progress to different panes in our GUI. With the scheme in which reporter objects are passed around, we can create different objects and pass them to different parts of the code. In the static version, we'll need to create different methods, and it starts to get ugly very quickly.

Concurrency also casts doubt on the idea of having a single object. Suppose we upgrade our email client to download messages from several servers concurrently. We would not want the progress messages from all the downloading sessions to be interleaved in a single output.

A good rule of thumb is to be wary of global variables. Ask yourself if you really can make do with a single object. Usually you'll find ample reason to have more than one object around. This scheme goes by the term *Singleton* in the design patterns literature, because the class contains only a single object.