

Dependences and Decoupling

6.170 Lecture 9

Spring 2004

Coupling is the path to the dark side. Coupling leads to complexity. Complexity leads to confusion. Confusion leads to suffering. Once you start down the dark path, forever will it dominate your destiny, consume you it will. Yoda (slightly paraphrased)

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 key idea that we'll introduce today is that of a *specification*. Don't think that specifications are just boring documentation. On the contrary, they are essential to decoupling and thus to high-quality design. And we'll see that in more advanced designs, specifications become design elements in their own right.

Our course text treats the terms *uses* and *depends* as synonyms. In this lecture, we'll distinguish the two, and explain how the notion of *depends* is a more useful one than the older notion of *uses*. You'll need to understand how to construct and analyze dependency diagrams; uses diagrams are explained just as a stepping stone along the way.

1 Decomposition

A program is built from a collection of parts. What parts should there be, and how should they be related? This is the problem of *decomposition*.

1.1 Why Decompose?

What benefits come from dividing a program into smaller parts. Here are some:

- ▷ *Division of labor*. A program doesn't just appear out of thin air: it has to be built gradually. If you divide it into parts, you can get it built more quickly by having different people work on different parts.
- ▷ *Reuse*. Sometimes it's possible to factor out parts that different programs have in common, so they can be produced once and used many times.
- ▷ *Modular Analysis*. Even if a program is built by only one person, there's an advantage to building it in small parts. Each time a part is complete, it can be analyzed for correctness.
- ▷ *Localized Change*. Any useful program will need adaptation and extension over its lifetime. If a change can be localized to a few parts, a much smaller portion of the program as a whole needs to be considered when making and validating the change.

1.2 What Are The Parts?

What are the parts that a program is divided into? We'll use the term 'part' rather than 'module' for now so we can keep away from programming-language specific notions. For now, all we need to note is that the parts in a program are *descriptions*: in fact, software development is really all about producing, analyzing and executing descriptions. We'll soon see that the parts of a program aren't all executable code – it's useful to think of specifications as parts too.

1.3 Top Down Design

Suppose we need some part A and we want to decompose into parts. How do we make the right decomposition? This topic is a large part of what we'll be studying in this course. Suppose we decompose A into B and C . Then, at the very least, it should be possible to build B and C , and putting B and C together should give us A .

In the 1970's, there was a popular approach to software development called *Top-down Design*. The idea is simply to apply the following step recursively:

- ▷ If the part you need to build is already available (for example, as a machine instruction), then you're done;
- ▷ Otherwise, split it into subparts, develop them, and combine them together.

The idea was appealing, and there are still people who talk about it with approval. But it fails miserably, and here's why. The very first decomposition is the most vital one, and yet you don't discover whether it was good until you reach the leaves of the decomposition tree. You can't do much evaluation along the way; you can't test a decomposition into two parts that haven't themselves been implemented. Once you get to the bottom, it's too late to do anything about the decompositions you made at the top. So from the point of view of risk – making decisions when you have the information you need, and minimizing the chance and cost of mistakes – it's a very bad strategy.

This isn't to say, of course, that viewing a system hierarchically is a bad idea. It's just not possible to develop it that way.

1.4 A Better Strategy

A much better strategy is to develop a system structure considering of multiple parts at a roughly equal level of abstraction. You refine the description of every part at once, and analyze whether the parts will fit together and achieve the desired function before starting to implement any of them. It also turns out that it is much better to organize a system around data than around functions.

Perhaps the most important consideration in evaluating the decomposition into parts is how the parts are coupled to one another. We want to minimize coupling – to *decouple* the parts – so that we can work on each part independently of the others. This is the topic of our lecture today; later in the course, we'll see how we can express properties of the parts and the details of how they interact with one another.

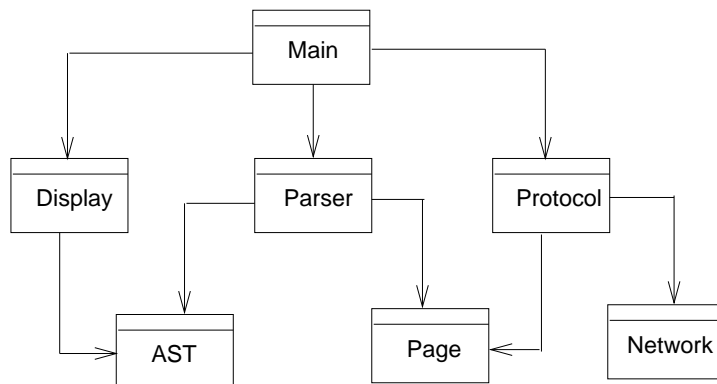
2 Dependence Relationships

2.1 Uses Diagram

The most basic notion relationship between parts is the *uses* relationship. We say that a part A uses a part B if A refers to B in such a way that the meaning of A depends on the meaning of B .

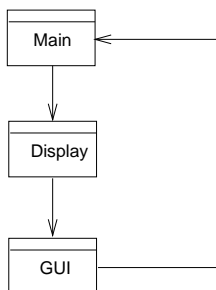
When A and B are executable code, the meaning of A is its behavior when executed, so A uses B when the behavior of A depends on the behavior of B .

Suppose, for example, we are designing a web browser. The diagram shows a putative decomposition into parts:



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.

A uses graph is usually not a tree because reuse implies that a part may have multiple users. A uses graph may be layered. Finally, there may be cycles in the uses graph.



Suppose we have a *GUI* part that provides functions for writing to a display, and handles input by making calls (when buttons are pressed, etc) to functions in other parts. Then *Display* may use *GUI* for output, and *GUI* may use *Main* for input.

What can we do with the uses-diagram?

- ▷ *Reasoning*. Suppose we want to determine whether a part P is correct. Aside from P itself, which parts do we need to examine? The answer is: the parts P uses, the parts they use, and so on – in other words all parts reachable from P . Conversely, if we make a change to P , which parts might be affected? The answer is all parts that use P , the parts that use them, and so on.
- ▷ *Reuse*. To identify a subsystem – a collection of parts – that can be reused, we have to check that none of its parts use any other parts not in the subsystem. The same determination tells us how to find a minimal subsystem for initial implementation. For example, the parts *Display* and *AST* form a collection without dependences on other parts, and could be reused as a unit.

- ▷ *Construction Order.* The uses diagram helps determine what order to build the parts in. We might assign two sets of parts to two different groups and let them work in parallel. By ensuring that no part in one set uses a part in another set, we can be sure that neither group will be stalled waiting for the other.

Thinking about these considerations can shed light on the quality of a design. The cycle we mentioned above, (Main–Display–GUI–Main), for example, makes it impossible to reuse the *Display* part without also reusing *Main*.

There’s a problem with the uses diagram though. Most of the analyses we’ve just discussed involve finding all parts reachable or reaching a part. In a large system, this may be a high proportion of the parts in a system. And worse, as the system grows, the problem gets worse, even for existing parts which refer directly to no more parts than they did before. To put it differently, the fundamental relationship that underlies *uses* is transitive: if *A* is affected by *B* and *B* is affected by *C*, then *A* is affected by *C*. It would be much better if reasoning about a part, for example, required looking at only at the parts it refers to.

2.2 Dependences and Specifications

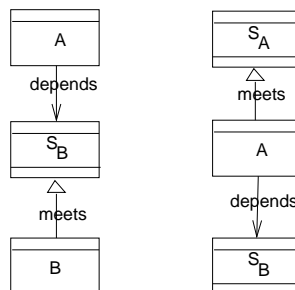
The solution to this problem is to have instead a notion of dependence that stops *after one step*. To reason about some part *A*, we will need to consider only the parts it depends on. To make this possible, it will be necessary for every part that *A* depends on to be complete, in the sense that its description completely characterizes its behavior. It cannot itself depend on other parts. Such a description is called a *specification*.

A specification cannot be executed, so we’ll need for each specification part at least one implementation part that behaves according to the specification. Our diagram, the *dependency diagram*, therefore has two kinds of arcs. An implementation part may *depend* on a specification part, and it may *fulfill* or *meet* a specification part.

In comparison to what we had before, we have broken the *uses* relationship between two parts *A* and *B* into two separate relationships. By introducing a specification part *S*, we can say that *A* depends on *S* and *B* meets *S*. The diagram on the left illustrates 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 by 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.

This is a much more useful and powerful framework than *uses*. The introduction of specifications brings many advantages:



- ▷ *Weakened Assumptions.* When *A* uses *B*, it is unlikely to rely on every aspect of *B*. Specifications allow us to say explicitly which aspects matter. By making specifications much smaller and simpler than implementations, we can make it much easier to reason about correctness of parts. And a weak specification gives more opportunities for performance improvements.
- ▷ *Evaluating Changes.* The specification *S* helps limit the scope of a change. Suppose we want to change *B*. Must *A* change as well? Now this question doesn't require looking at *A*. We start by looking at *S*, the specification *A* requires of the part it uses. If the new *B* will still meet *S*, then no change to *A* will be needed at all.
- ▷ *Communication.* If *A* and *B* are to be built by different people, they only need to agree upon *S* in advance. *A* can ignore the details of the services *B* provides, and *B* can ignore the details of the needs of *A*.
- ▷ *Multiple Implementations.* There can be many different implementation parts that meet a given specification part. This makes a market in interchangeable parts possible. Parts are marketed in a catalog by the specifications they meet, and a customer can pick any part that meets the required specification. A single system can provide multiple implementations of a part.

Specifications are so useful that we'll assume that there is a specification part corresponding to every implementation part in our system, and we'll conflate them, drawing dependences directly from implementations to implementations. In other words, a dependence arc from *A* to *B* means that *A* depends on the specification of *B*.

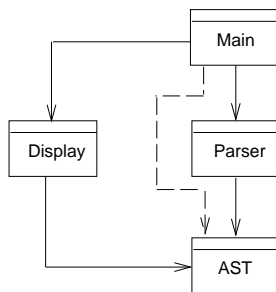
So whenever we draw a diagram like the one of our browser above, we'll interpret it as a dependence diagram and not as a uses diagram. For example, it will be possible to have teams build *Parser* and *Protocol* in parallel as soon as the *specification* of *Page* is complete; its implementation can wait.

Sometimes, though, specifications are design elements in their own right and we'll want to make explicit their presence. Java provides some useful mechanisms for expressing decoupling with specifications, and we'll want to show these. Design patterns, which will be studying later in the term, make extensive use of specifications in this way.

2.3 Weak Dependences

Sometimes a part is just a conduit. It refers to another part by name, but doesn't make use of any service it provides. The specification it depends on requires only that the part exist. In this case, the dependence is called a *weak dependence*, as is drawn as a dotted or a dashed arc.

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*.



For example, the *Display* part of our browser may use a part *UI* for its output, but need not know whether the *UI* is graphical or text-based. This part can be a specification part, met by an implementation part *GUI* which *Main* depends on (since it creates the actual GUI object). In this case, *Main*, because it passes an object whose type is described as *UI* to *Display*, must also have a weak dependence on the specification part *UI*.

3 Techniques for Decoupling

So far, we've discussed how to represent dependences between program parts. We've also talked about some of the effects of dependences on various development activities. In every case, a dependence is a *liability*: it expands the scope of what needs to be considered. So a major part of design is trying to minimize dependences: to *decouple* parts from one another.

The most effective way to reduce coupling is to design the parts so that they are simple and well defined, and bring together aspects of the system that belong together and separate aspects that don't. There are also some tactics that can be applied when you already have a candidate decomposition: they involve introducing new parts and altering specifications. We'll see many of these throughout the term. For now, we'll just mention some briefly to give you an idea of what's possible. They are:

- ▷ Hiding representation: By eliminating the dependence of the using part *A* on the representation of data in the used part *B*, it is easier to understand the role that *B* plays in *A*.
- ▷ Facade design pattern: Interposing a new implementation part between two sets of parts can help decouple one layer from another in layered system.
- ▷ Polymorphism: A program part *C* that provides container objects has a dependence on the program part *E* that provides the elements of the container. The specification that connects *C* to *E* is weakened using polymorphism. Rather than *C* depending on the specification of *E* in the monomorphic case, *C* depends on a specification *S* that says only that the part must provide objects with, say, an equality test, in the polymorphic case.
- ▷ Callbacks: A part *A* might pass another part *B* at runtime a reference to one of its procedures. When this procedure is called by the *B*, it has the same effect it would have had if the procedure had been named in the text of *B*. But since the association is only made at runtime, there is no dependence of *B* on *A*. This arrangement is a *callback*, since *B* 'calls back' to *A* against the usual direction of procedure call.

4 Coupling Due to Shared Constraints

There's a different kind of coupling which isn't shown in a module dependency diagram. Two parts may have no explicit dependence between them, but they may nevertheless be coupled because they are required to satisfy a constraint together.

For example, suppose we have two parts, *Read*, which reads files, and *Write*, which writes files. If the files read by *Read* are the same files written by *Write*, there will be a constraint that the two parts agree on the file format. If the file format is changed, both parts will need to change.

To avoid this kind of coupling, you have to try to localize functionality associated with any constraint in a single part. This is what Matthias Felleisen calls 'single point of control' in his novel introduction to programming in Scheme (*How to Design Programs, An Introduction to Programming and Computing*, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi, MIT Press, 2001).

David Parnas suggested that this idea should form the basic of the selection of parts. You start by listing the key design decisions (such as choice of file format), and then assign each to a part that keeps that decision 'secret'. This is explained in detail with a nice example in his seminal paper *On the Criteria To Be Used in Decomposing Systems into Modules*, Communications of the ACM, Vol. 15, No. 12, December 1972 pp. 1053–1058.

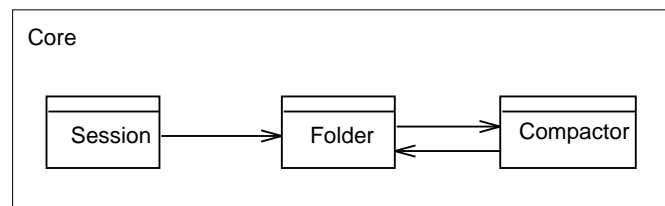
5 Example: Instrumenting a Program

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.

5.1 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);
    }
}
```

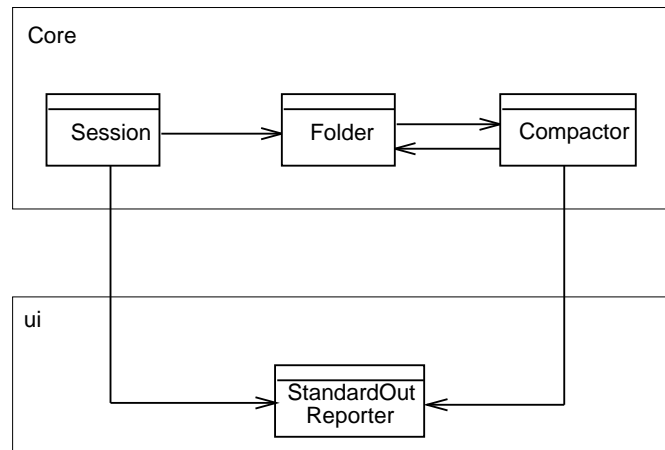
Now the change can be made at a single point in the code. We just modify the procedure:

```
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.



5.2 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 gratuitous. 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 ());
    }
}

```

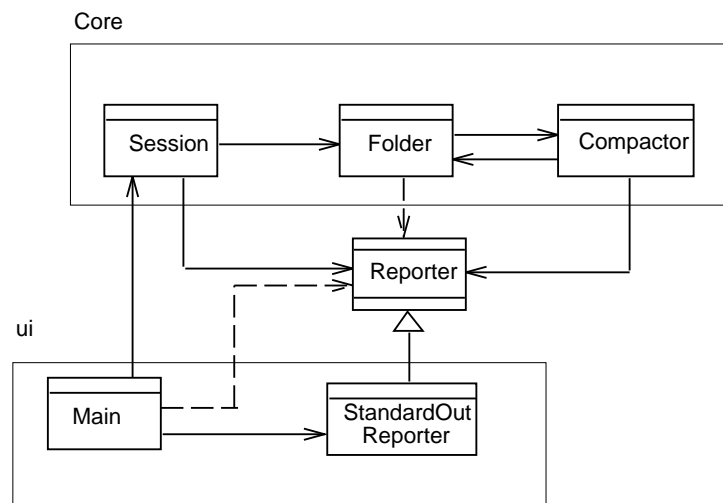
At the top of the program, we'll create an object `port` and pass it in to a `Session` object's `download` method:

```

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.

5.3 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:

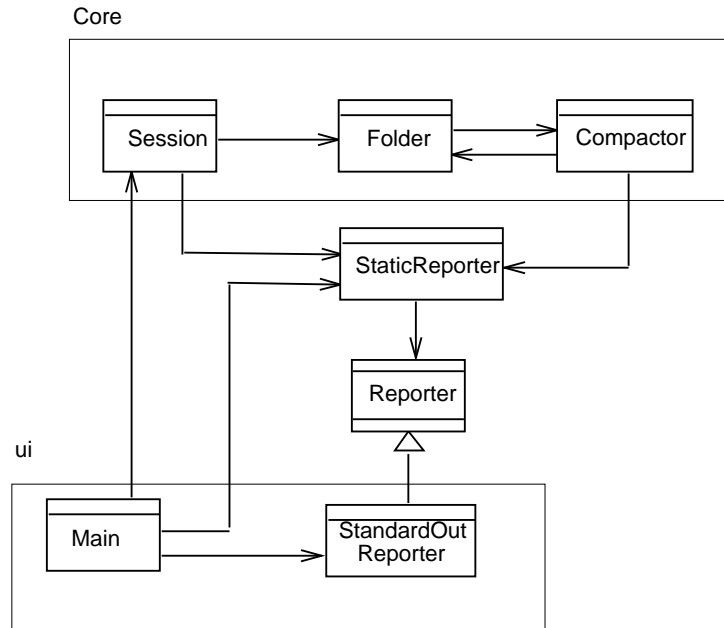
```
StaticReporter.setReporter (new StandardOutReporter ());
```

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.