Lecture 3: Decoupling II

In the last lecture, we talked about the importance of dependences in the design of a program. 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 how the features of Java can be used to express and tame dependences. We'll also study a variety of solutions to a simple coding problem, illustrating in particular the role of interfaces.

3.1 Review: Module Dependency Diagrams

Let's start with a brief review of the module dependency diagram (MDD) from the last lecture. 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. A dotted arrow from A to S is a weak dependence; it says that A depends only the existence of a part satisfying the specification S, but actually has no dependence on any details of S. 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.

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.

3.2 Java Namespace

Like any large written work, a program benefits from being organized into a hierarchical structure. When trying to understand a large structure, it's often helpful to view

it top-down, starting with the grossest levels of structure and proceeding to finer and finer details. Java's naming system supports this hierarchical structure. It also brings another important benefit. Different components can use the same names for their subcomponents, with different local meanings. In the context of the system as a whole, the subcomponents will have names that are qualified by the components they belong to, so there will be no confusion. This is vital, because it allows developers to work independently without worrying about name clashes.

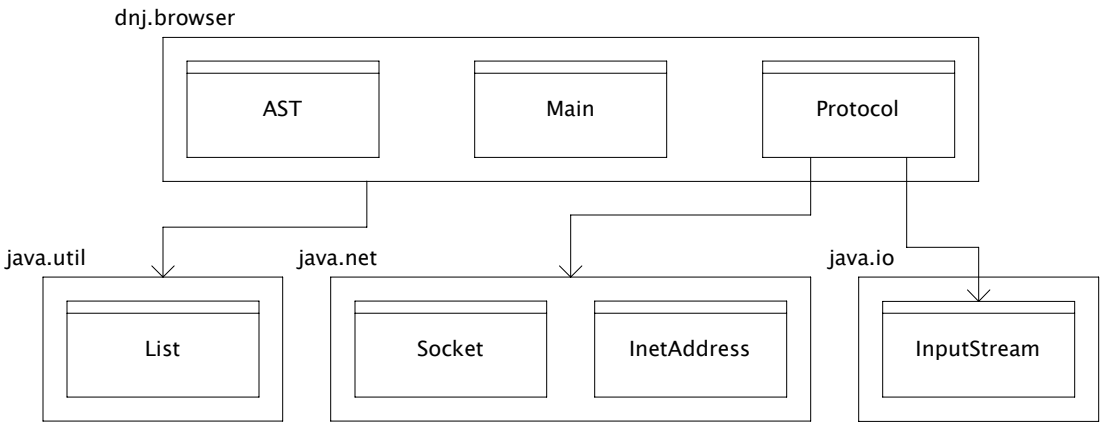
Here's how the Java naming system works. The key named components are classes and interfaces, and they have named methods and named fields. Local variables (within methods) and method arguments are also named. Each name in a Java program has a scope: a portion of the program text over which the name is valid and bound to the component. Method arguments, for example, have the scope of the method; fields have the scope of the class, and sometimes beyond. The same name can be used to refer to different things when there is no ambiguity. For example, it's possible to use the same name for a field, a method and a class; see the Java language spec for examples.

A Java program is organized into packages. Each class or interface has its own file (ignoring inner classes, which we won't discuss). Packages are mirrored in the directory structure. Just like directories, packages can be nested arbitrarily deeply. To organize your code into packages, you do two things: you indicate at the top of each file which package its class or interface belongs to, and you organize the files physically into a directory structure to match the package structure. For example, the class *dnj.browser.Protocol* would be in a file called *Protocol.java* in the directory *dnj/browser*.

We can show this structure in our dependence diagram. The classes and interfaces form the parts between which dependences are shown. Packages are shown as contours enclosing them. It's convenient sometimes to hide the exact dependences between parts in different packages and just show a dependence arc at the package level. A dependence from a package means that some class or interface (or maybe several) in that package has a dependence; a dependence on a package means a dependence on some class or interface (or maybe several) in that package.

3.3 Access Control

Java's access control mechanisms allow you to control dependences. In the text of a class, you can indicate which other classes can have dependences on it, and to some extent you can control the nature of the dependences.



A class declared as *public* can be referred to by any other class; otherwise, it can be referred to only by classes in the same package. So by dropping this modifier, we can prevent dependences on the class from any class outside the package.

Members of a class – that is, its fields and methods – may be marked *public*, *private* or *protected*. A *public* member can be accessed from anywhere. A *private* member can be accessed only from within the class in which the field or method is declared. A *protected* member can be accessed within the package, or from outside the package by a subclass of the class in which the member is declared – thus creating the very odd effect that marking a member as *protected* makes it more, not less, accessible.

Recall that a dependence of A on B really means a dependence of A on the specification of B. Modifiers on members of B allow us to control the nature of the dependence by changing which members belong to B's specification. Controlling access to the fields of B helps give representation independence, but it does not always ensure it (as we'll see later in the course).

3.4 Safe Languages

A key property of a program is that one part should only depend on another if it names it. This seems obvious, but in fact it's a property that only holds for programs written in so-called 'safe languages'. In an unsafe language, the text in one part can affect the behaviour of another without any names being shared. This leads to insidious bugs that are very hard to track down, and which can have disastrous and unpredictable effects.

Here's how it happens. Consider a program written in C in which one module (in C,

just a file) updates an array. An attempt to set the value of an array element beyond the bounds of the array will sometimes fail, because it causes a memory fault, going beyond the memory area assigned to the process. But, unfortunately, more often it will succeed, and the result will be to overwrite an arbitrary piece of memory – arbitrary because the programmer does not know how the compiler laid out the program's memory, and cannot predict what other data structure has been affected. As a result, an update of the array *a* can affect the value of a data structure with the name *d* that is declared in a different module and doesn't even have a type in common with *a*.

Safe languages rule this out by combining several techniques. Dynamic checking of array bounds prevents the kind of updating we just mentioned from occurring; in Java, an exception would be thrown. Automatic memory management ensures that memory is not reclaimed and then mistakenly reused. Both of these rely on the fundamental idea of *strong typing*, which ensures that an access that is declared to be to a value of type *t* in the program text will always be an access to a value of type *t* at runtime. There is no risk that code designed for an array will be mistakenly applied to a string or an integer.

Safe languages have been around since 1960. Famous safe languages include Algol-60, Pascal, Modula, LISP, CLU, Ada, ML, and now Java. It's interesting that for many years industry claimed that the costs of safety were too high, and that it was infeasible to switch from unsafe languages (like C++) to safe languages (like Java). Java benefited from a lot of early hype about applets, and now that it's widely used, and lots of libraries are available, and there are lots of programmers who know Java, many companies have taken the plunge and are recognizing the benefits of a safe language.

Some safe languages guarantee type correctness at compile time – by 'static typing'. Others, such as Scheme and LISP, do their type checking at runtime, and their type systems only distinguish primitive types from one another. We'll see shortly how a more expressive type system can also help control dependences.

If reliability matters, it's wise to use a safe language. In lecture, I told a story here about use of unsafe language features in a medical accelerator.

3.5 Interfaces

In languages with static typing, one can control dependences by choice of types. Roughly speaking, a class that mentions only objects of type *T* cannot have a dependence on a class that provides objects of a different type *T'*. In other words, one can tell from the types mentioned in a class which other classes it depends on.

However, in languages with subtyping, something interesting is possible. Suppose class *A* mentions only the class *B*. This does *not* mean that it can only call methods on objects created by class *B*. In Java, the objects created by a subclass *C* of *B* are regarded as also having the type *B*, so even though *A* can't create objects of class *C* directly, it can be passed them by another class. The type *C* is said to be a *subtype* of the type *B*, since a *C* object can be used when a *B* object is expected. This is called 'substitutability'.

Subclassing actually conflates two distinct issues. One is subtyping: that objects of class *C* are to be regarded as having types compatible with *B*, for example. The other is inheritance: that the code of class *C* can reuse code from *B*. Later in the course we'll discuss some of the unfortunate consequences of conflating these two issues, and we'll see how substitutability doesn't always work as you might expect.

For now, we'll focus on the subtyping mechanism alone, since it's what's relevant to our discussion. Java provides a notion of *interfaces* which give more flexibility in subtyping than subclasses. A Java interface is, in our terminology, a pure specification part. It contains no executable code, and is used only to aid decoupling.

Here's how it works. Instead of having a class *A* depend on a class *B*, we introduce an interface *I*. *A* now mentions *I* instead of *B*, and *B* is required to meet the specification of *I*. Of course the Java compiler doesn't deal with behavioural specifications: it just checks that the types of the methods of *B* are compatible with the types declared in *I*. At runtime, whenever *A* expects an object of type *I*, an object of type *B* is acceptable.

For example, in the Java library, there is a class *java.util.LinkedList* that implements linked lists. If you're writing some code that only requires that an object be a list, and not necessarily that it be a linked list, you should use the type *java.util.List* in your code, which is an interface implemented by *java.util.LinkedList*. There are other classes, such as *ArrayList* and *Vector* that implement this interface. So long as your code refers only to the interface, it will work with any of these implementation classes.

Several classes may implement the same interface, and a class may implement several interfaces. In contrast, a class may only subclass at most one other class. Because of this, some people use the term 'multiple specification inheritance' to describe the interface feature of Java, in contrast to true multiple inheritance in which can reuse code from multiple superclasses.

Interfaces bring primarily two benefits. First, they let you express pure specification parts in code, so you can ensure that the use of a class *B* by a class *A* involves only a dependence of *A* on a specification *S*, and not on other details of *B*. Second, interfaces let you provide several implementation parts that meet a single specification, with the

selection being made at compile time or at runtime.

3.6 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*.

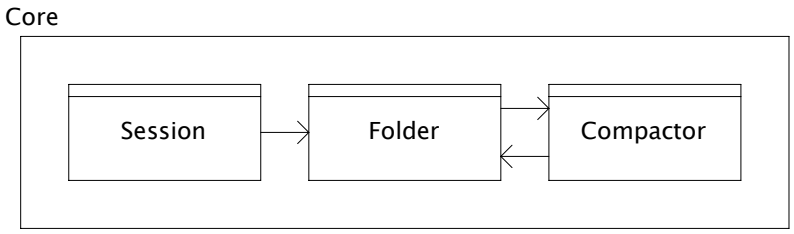
We'll look at a variety of ways to implement our instrumentation facility, and we'll study the advantages and disadvantages of each. Starting with the simplest, most naive design possible, we might intersperse statements such as

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

throughout the program.

3.6.1 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);
        }
    }
```

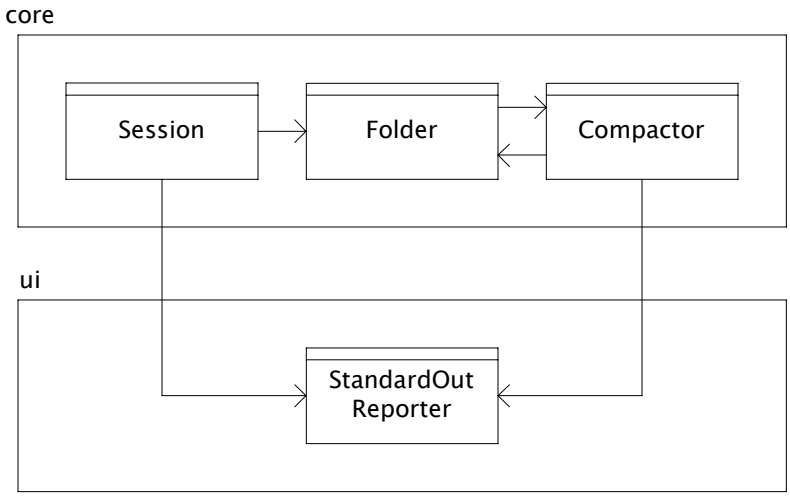
Now the change can be made at a single point in the code. We just modify the procedure:

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

Matthias Felleisen calls this the 'single point of control' principle. 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 instrumenta-



tion facility depend: *StandardOutReporter*. Note that there is no dependence from *Folder* to StandardOutReporter, since the code of *Folder* makes no calls to it.

3.6.2 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's 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" );
    ...
    }
```

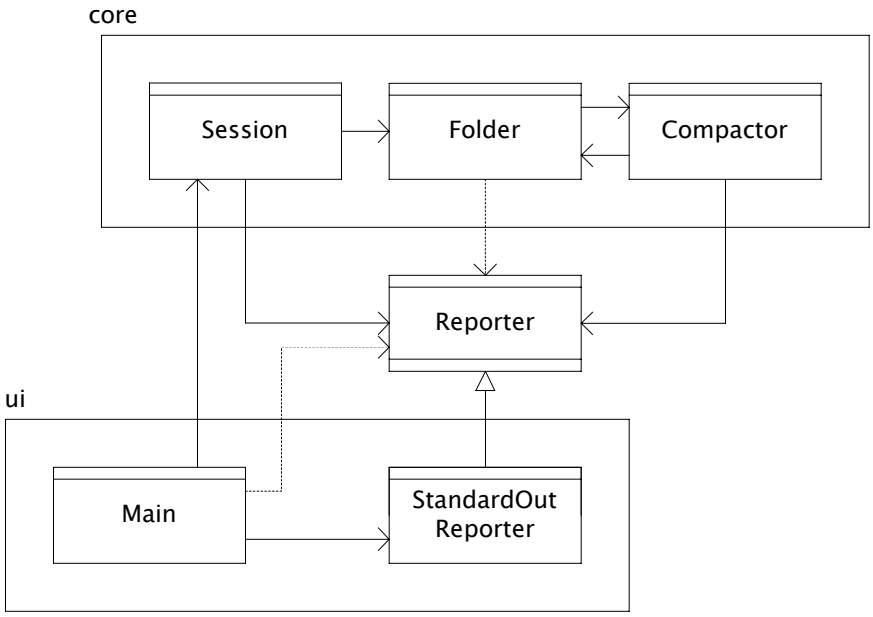
Now we define a class that will actually implement the reporting behaviour. Let's use standard out as our example as it's simpler:

```
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 and pass it in:



s.download (new StandardOutReporter (), ...);

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*.

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 *gui* package. All the dependences point downwards (at least logically!) from *gui* 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 *gui* package to call its constructor on the classes that actually contain concrete I/O code. 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.

3.6.3 Interfaces vs. Abstract Classes

You may wonder whether we might have used a class instead of an interface. An abstract class is one that is not completely implemented; it cannot be instantiated, but must be extended by a subclass that completes it. Abstract classes are useful when you want to factor out some common code from several classes. Suppose we wanted to display a message saying how long each step had taken. We might implement a *Reporter* class whose objects retain in their state the time of the last call to report, and then take the difference between this and the current time for the output. By making this class an abstract class, we could reuse the code in each of the concrete subclasses *StandardOutReporter*, *JTextComponentReporter*, etc.

Why not pass make the argument of download have this abstract class as its type, instead of an interface? There are two related reasons. The first is that we want the dependence on the reporter code to be as weak as possible. The interface has no code at all; it expresses the minimal specification of what's needed. The second is that there in no multiple inheritance in Java: a class can only extend at most one other class. So when you're designing the core program, you don't want to use the opportunity to subclass prematurely. A class can implement any number of interfaces, so by choosing an interface, you leave it open to the designer of the reporter classes how they will be implemented.

3.6.4 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 method 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:

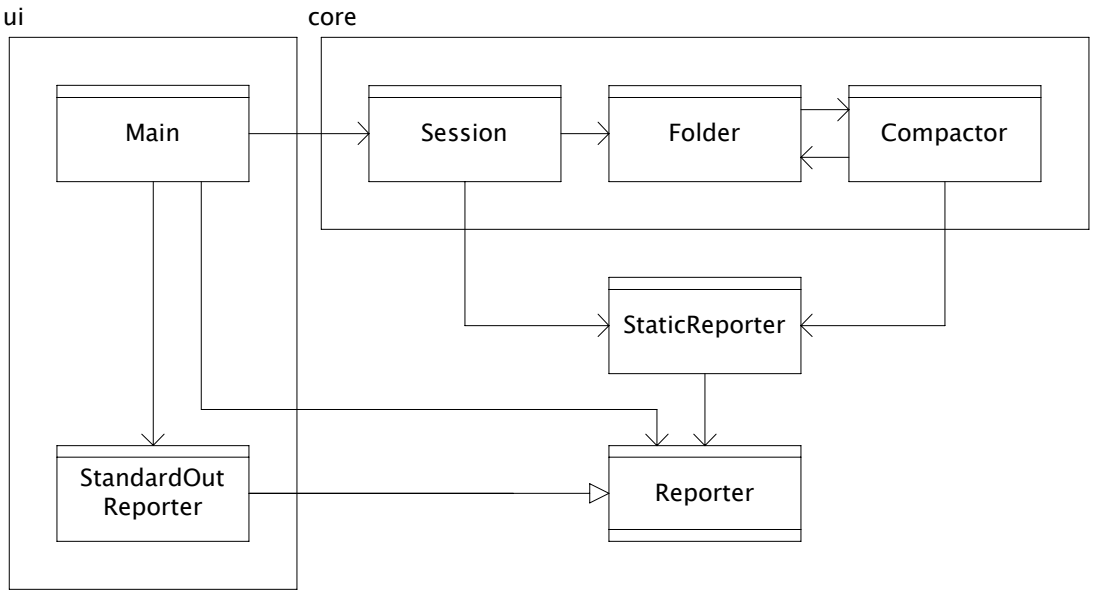
```
void download (...) {
    StaticReporter.report ("Starting downloading" );
    ...
    }
```

In the module dependency diagram, 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 behaviour 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's 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 com-



ponents 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 wouldn't 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.