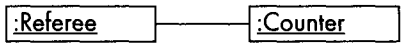

Figure 4.6 *A Referee object uses the services of a Counter object*

natural number, all the modules that knew that score was an integer would have to be changed. However, if modules communicate with each other by message passing via a public interface, as objects do, then changes to the code that implements the operations do not affect client modules; as long as the interface is not changed, they are not affected. This is true for the actual body of code as well as for data structures involved. For example, if we decide to change the code implementing an operation in order to make it execute faster, so long as we don't change the interface, no client object will be affected. The use of a public interface not only hides the data, it also hides the code.

Dependencies can also complicate the reuse of modules. For example, let us suppose the Counter object is used in a simulation of a game of rugby where a Referee object uses the Counter object to keep the score up to date as in Figure 4.6. The Referee object uses the services of, i.e. is a client of, the Counter object.

This would mean that, if we wanted to reuse the Referee object in another program, we would also have to use the Counter object. In this case there is not really a problem, because we know about the dependency, it is clearly declared. If we want to store away Referee for possible reuse, we can store it bundled together with Counter in a package. You can read about packages in Chapter 5.

Problems arise if two modules are dependent on each other in undeclared ways. In programs written in early programming languages it was possible to jump without restriction to another part of the program or to another module, execute a few lines of code or access some data when you got there, then jump back. Such jumps made it very difficult to read through the code and understand what was happening. If a maintaining programmer had to change one part of the code it was hard to work out how the rest of the program would be affected. Unless he read right through all the code, he could not see which parts of the program might use the section he had just changed: this was the problem of side-effects. Declaring a public interface and using a programming language that enforces it (such as Java or C++) therefore has two big advantages for maintenance.

- It makes it much more obvious to someone reading the code that a client–server relationship exists. That makes the job of changing the code quicker, both because it is easier to