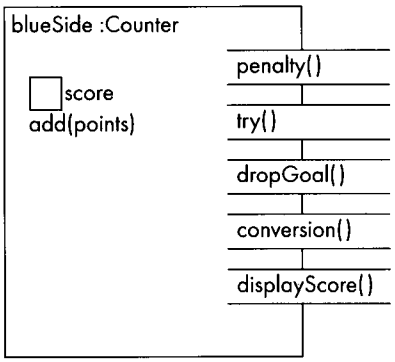

Figure 4.5 *A Counter object showing its public interface and a private operation, add(points)*

must do it by sending a *message* to the object blueSide, requesting that it update the score. The message must correctly identify the object to which it is sent and the operation's signature. It must be in a specific format: starting with the name of the object, followed by a full stop, followed by the signature of the operation to be invoked, for example blueSide.try(). For the message to work, it must correctly address the object by name and the operation it specifies must be part of that object's public interface.

Counter may have other operations designed for its own internal use, for example it might have a operation add(points), used by most of the operations in the public interface, to add a specified number of points to the attribute score (see Figure 4.5). Such internal operations are known as private operations and are not part of the services that a Counter object makes available to other objects; they form a private interface for an object to send messages to itself.

Dependencies. As we have seen, one advantage of using a clearly defined public interface is that it encapsulates data; another advantage is that it clarifies *dependencies* between software constructs and limits the damage that can be done by dependencies. Two modules are said to be dependent on each other if one uses the services of the other. This is known as a *client–server* relationship.

Dependencies between modules can cause problems with maintenance if they are not carefully controlled. As soon as a dependency is established it becomes possible, at least in theory, that changes to one module may affect the other. For example, if we had designed the counter example above in such a way that score was visible to other modules and could be directly manipulated by processes in other modules, we would have to be very careful if we changed score. If we decided to implement score as a real or a