Chapter 5

The concept of a protocol

The concept of a *protocol*[1] is central to software engineering. In this chapter we explain what this concept means, especially in relation to object-oriented programming in CLOS.

5.1 Modules, interface, and implementation

In software-engineering, it is widely agreed that in order for a large program to be *maintainable*, it must divided into a collection of weakly-connected *modules*. In order for modules to have weak connections between them, each module consists of two different conceptual parts with very different purposes. The smaller of the two parts is the *interface*, meant to be referred to by *client* modules. The larger one contains the *implementation* of the interface. The client of some module is only allowed to refer to its interface, thus making it possible for the implementation to evolve without taking into account how the module is used by its clients.

The interface of a module must be independent of how the module is implemented. The way this independence is accomplished is that the interface consists of a collection of *operations* and a collection of *named data types*. The operations can be ordinary functions or generic functions. Instances of the named data types can be arguments or return values of the operations.

[1]In the literature, the terms *protocol* and *interface* are often used interchangeably. In this book we use them in slightly different ways.

No information is provided about the named data types other than how their instances can be arguments and return values of the operations. In particular, no information is provided about the *representation* of instances of the named data types.

The interface of a module consists of one or more *protocols*. An operation is a member of exactly one protocol, whereas a named data type can appear in more than one protocol. Operations are divided into protocols based on an informal concept of a *strong relationship* as described in the next section.

5.2 Definition of protocol

A *protocol* P consists of a (possibly empty) set τ of *types* that are *introduced* by P, and a non-empty set ω of *operations*. Each of the types in τ is related to at least one of the operations in ω by the fact that the operation either takes an element of the type as an *argument* or that the operation returns an element of the type as a return value.

The operations of a protocol may take arguments of types other than the ones in τ, and may return values of types other than the ones in τ. These other types are said to *participate* in the protocol, as opposed to being *introduced* by it. Frequently, the set of participating types is omitted from the description of the protocol.

In addition, τ and ω obey the following restrictions:

1. No operation in ω is a member of the set of operations of any other protocol. The types in τ are not subject to any similar restriction.
2. There does not exist partitions P_1 and P_2 of P and τ_1 and τ_2 of τ such that P_1 is a protocol with τ_1 as its set of types and P_2 is a protocol with τ_2 as its set of types.

Notice that the concept of a protocol does not have any linguistic support in the Common Lisp standard; it is related only to the *architecture* of the software containing it.

In terms of CLOS programming, the types in τ are often standard classes (i.e. instances of the class `standard-class`), and some classes in τ may be related by the *subclass* relationship.

5.3 Relation to traditional classes

For readers familiar with object-oriented programming as used in languages with single dispatch, it is useful to think of a protocol as a generalization of a *class* as used in such languages. The set of operations ω then corresponds to the *methods* of the class, and the set τ contains the class itself.

5.4 Examples of protocols

As a very simple example, consider the protocol with $\omega = \{\texttt{cons}, \texttt{car}, \texttt{cdr}\}$ with $\tau = \{\texttt{cons}\}$. The type `t` (and, as a consequence, any Common Lisp type) *participates* in this protocol in that the operation `cons` can take arguments of any type, and the operations `car` and `cdr` can return objects of any type.

For a more complex example, consider a set of operations for drawing geometric figures on some sort of *medium* such as a screen. Such a protocol may contain the operations `draw` and `erase`, and the set of types might contain `figure` and `medium`.

The last restriction in the definition of a protocol means that some sets such as $\omega = \{\texttt{cons}, \texttt{car}, \texttt{cdr}, \texttt{+}\}$ with $\tau = \{\texttt{cons}, \texttt{number}\}$ do not constitute a protocol, simply because it can be divided into $\omega_1 = \{\texttt{cons}, \texttt{car}, \texttt{cdr}\}$ with $\tau_1 = \{\texttt{cons}\}$ and $\omega_2 = \{\texttt{+}\}$ with $\tau_2 = \{\texttt{number}\}$.

5.5 Completeness

If any operation in ω has a parameter such that, no matter what value is given as the corresponding argument when that operation is invoked, that value does not in any way influence the result (i.e. return value or side effect) of that invocation or the invocation of any subsequent operation in the protocol, then we say that the protocol is *incomplete*.

For example, $\omega = \{\texttt{cons}, \texttt{car}\}$ with $\tau = \{\texttt{cons}\}$ being the set of types is an incomplete protocol, because the second argument to the operation `cons` does not influence any subsequent operations.

We will be interested only in complete protocols in this book.

5.6 Minimality

We will be interested in *minimal* protocols, i.e., protocols in which no operation can be implemented in terms of the others.

As an example, consider $\omega = \{\texttt{cons}, \texttt{car}, \texttt{cdr}, \texttt{list*}\}$ with $\tau = \{\texttt{cons}\}$ being the associated set of types. This protocol is not minimal, because the operation `list*` can be implemented in terms of the operation `cons`.

We are nevertheless interested in sets of operations that are not minimal and therefore not considered to be protocols. In this case we call the set an *extended protocol.*

Often, it is desirable to include *macros* provided by a module as part of the interface. Such macros are then considered part of the extended protocol.

5.7 Protocol classes

A *protocol class* is a class that is introduced by some protocol, but it is not meant to be instantiated directly. Only more specific subclasses of the protocol class should be instantiated.

The class named `figure` in the example of Section 5.4 is a protocol class. Subclasses such as `circle` and `rectangle` might be instantiable subclasses of it.

Notice that the concept of a protocol class does not have any linguistic support in the Common Lisp standard. Therefore, typically nothing prevents an application from instantiating a protocol class. It is sometimes worthwhile to include a check that no such attempt is made.

5.8 Description of operations

The description of an operation in a protocol may contain the following information:

- An informal description of the return values and side effects of the operation.

- Some restrictions on the arguments in order for client code to have the right to invoke the operation. Such restrictions are called the *preconditions* of the operation. The operation may or may not check the preconditions and signal a condition if it is not respected, based on the computational cost of the check.

- If the operation takes a *collection* such as a Common Lisp list or a Common Lisp array of objects as an argument, then the description typically includes information as to whether the operation may modify the collection.

- If the operation returns a *collection* such as a Common Lisp list or a Common Lisp array of objects as one of its return values, then the description typically includes information as to whether the client is allowed to modify the collection.

- The description may contain some bounds (upper, lower, or both) on the *computational complexity* of the operation.