Aspect-Oriented LISP Framework

Miklós Espák

Institute of Informatics, Department of Information Technology,
University of Debrecen,
H-4010 Debrecen, Egyetem tér 1. Hungary
espakm@inf.unideb.hu

The main challenge of software modelling is to maintain the model of the world as compact as possible, as well as to improve the flexibility of the program in terms of later development. As the concepts of the worlds to be modelled are connected in several ways at the same time, the key issue of software engineering is to make the decision: how to break down the system into smaller parts. This decision has to be made in the early stage of the lifecycle, and has far-reaching consequences for the later development. Unfortunately, using the wide-spread OO languages, several concerns of the software cannot be separated, and these remain tangled with the main part of the software, making understanding and further development of it more difficult.

Aspect-oriented programming (AOP) provides a way to separate these concerns into distinct modules, so-called aspects. However, these concerns are written in distinct modules, they are – of course – not independent from each other. Aspects may have to access and modify some values at an execution point of the base program, and – which is mostly a missing feature of mainstream programming languages – they may require some manipulation of the structure or the behaviour of the base.

Therefore, aspect-oriented (AO) systems need special support by the language. Unfortunately, the mainstream OO languages (like C++ and Java) lack this support, so implementing an AO system has to be done using extralingual techniques. Although there are several solutions, they suffer from various problems:

- The technique used determines the capabilities of the AO system: compile-time and run-time dynamic AO systems perform poorly; run-time systems cannot allow structural modifications.
- In many system there is no way to change the behaviour of the system classes.
- The implementation of the AO systems is difficult, which slows down the development of them.
- As the implementation is not accessible from the language, there is no way to change it from the program itself.
- Though, there is reflection in Java, the reflective capabilities of the language are not sufficient and not extensible. Because of this, the Java AO systems implement their own reflective subsystem, enforcing the programmer to learn and use the new one.

There are several languages (mainly among the functional ones), which do not suffer from these problems. To the contrary, they incorporate the above-mentioned features as their elemental part. Let us look at the individual requirements, and how Common Lisp supports them:

As *extensibility* is one of the main characteristics of every Lisp system, implementing an AO framework in Lisp itself is self-evident. The need of the aspects to *access or change* some *values* in the base program is a typical issue of dynamic scoping, which is a basic feature of Lisp systems. Through the reflective capabilities of Lisp and the CLOS Metaobject Protocol, the *structural and behavioral changes* described by the aspects can be done very simply, within the language itself. *No special compiler*, *virtual machine or low-level manipulation* of the program is required. Additionally, the system is *compatible* with the existing programs, and it is working at run-time, providing the highest *flexibility* that is possible.

To improve the uniformity of AO systems, several researchers of this field founded the AOP Alliance. The alliance defined a common abstract framework for AO systems. The main techniques of implementing the CLOS equivalent of this framework is described in the paper. It is showed that, thanks to the extensibility and dynamics of Common Lisp, the implementation is clear and simple.