Using Aspect-Oriented Programming to extend Protégé

Henrik Eriksson

Dept. of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

her@ida.liu.se

Introduction

The plug-in architecture of Protégé [3] is extremely successful in that it allows developers to custom-tailor Protégé for new purposes and to extend significantly the functionality the Protégé environment. Until now, developers have mainly extended Protégé by adding new tab plug-ins. For example, there are tab extensions that support visualization, validation, and reasoning, as well as domain-specific additions for individual projects.

Extensions at the tab-level have proved to be the most popular grain size for new functionality in Protégé.The tab plug-in mechanism allows developers to create new front-ends to Protégé for their own projects (i.e., by creating a custom-tailored tab and disabling unneeded standard tabs). In addition to tab plus-ins, you can extend Protégé with slot widgets and storage back-ends. Slot widgets add new editing functionality for slots in Protégé-generated forms. Storage back-ends are responsible for the persistent storage of Protégé projects, and they define the file or database format used.

Because the Protégé architecture is a modular Java application, it is possible to add more functionality at other growth points. At this level, there is no documented plug-in interface for extensions; object orientation provides the functionality for extending Protégé. Here, a Java developer extends Protégé through object-oriented programming by implementing interfaces, subclassing standard classes, and substituting object factories. An example of such a Java extension is the OWL plug-in, which essentially replaces the default frame system in Protégé with its own version to provide the functionality required for OWL. Furthermore, because the Protégé source code is available, it is possible for developers to extend Protégé by modifying the source code, and, thus, creating their own development branch.

Although Protégé has an expandable architecture, there are still areas where additional flexibility is required. Modifying the Protégé source code directly and spawning new development branches is often undesirable, because it is difficult to keep up with the rapid release of new official Protégé versions. Unfortunately, it is problematical to develop Protégé extensions in the middle ground between the documented plug-in interfaces and the direct modification of source code. We cannot easily modify the behaviour of Protégé where there is no growth point. For example, it is not possible to modify the behaviour of the Protégé frame inheritance model without replacing the entire internal knowledge-base representation in Protégé. Aspect-oriented programming is a technique that can assist developers in overcoming the problem of extending Protégé where there is otherwise no appropriate growth point for standard object-oriented programming.

Aspect-Oriented Programming

Aspect-oriented programming is an extension of object-oriented paradigm that allows developers to isolate important aspects of the program functionality that cross cut the entire program and to implement them separately as *aspects*. Moreover, aspect-oriented programming supports the modification of existing programs by allowing aspect introduction to pre-existing programs. The process of extending programs by introducing aspects is called *weaving*. It is analogous to compilation of regular programs.

AspectJ [4,5] is a popular aspect-oriented extension to Java. In AspectJ, it is possible to define interesting points in the program in a generic manner; that is, by writing expressions that define these locations or *point-cuts*. Once you have defined the point-cuts, you can specify the code to invoke when the program reaches these points. It is possible to view AspectJ as a high-level language for patching Java programs, because AspectJ can weave your aspects into a pre-existing program. AspectJ can take both Java source files and compiled files as input to the weaving process. This feature means that you can modify and extend the functionality of Protégé by writing aspects, which the AspectJ weaver combines with the compiled Protégé files (i.e., the `protege.jar` file).

Protégé Weaving

Let us examine how we can use AspectJ to modify Protégé without changing its source code. Suppose we want to be able to modify the method for accessing Protégé slots values in the default knowledge base. The following aspect overrides the `getOwnSlotValues()` method of the `DefaultKnowledgeBase` class and replaces it with code that invokes the Jess function `slot-value-using-class`, if it exists.

```
aspect SlotValueOverride {

   pointcut getOwnSlotValues(Frame frame, Slot slot) : args(frame, slot) &&
     execution(public Collection DefaultKnowledgeBase.getOwnSlotValues(Frame, Slot)) &&
     !cflow(execution(private Value SlotValueOverride.slotValueUsingClass(Cls, Instance, Slot)));

   Collection around(Frame frame, Slot slot) : getOwnSlotValues(frame, slot) {
        Userfunction uf = JessTab.getEngine().findUserfunction("slot-value-using-class");
        if (uf instanceof Userfunction && frame instanceof Instance) {
          Instance inst = (Instance)frame;
          return slotValueUsingClass(inst.getDirectType(), inst, slot);
        } else return proceed(frame, slot);
   }

   private Value slotValueUsingClass(Cls cls, Instance inst, Slot slot) throws JessException {

      // Call slot-value-using-class Jess function and return the results

   }
}
```

The pointcut definition specifies the location in the program in which the method `getOwnSlotValues()` is called. To avoid infinite loops, the pointcut definition also excludes recursive calls. The advice declaration defines what should occur at the pointcut. Here, we use an `around` advice to add the code for testing if the Jess function `slot-value-using-class` exists and calling it if the frame is an instance. Alternatively, if the Jess function is not callable in this manner, the call to `proceed()` continues the execution after the pointcut (i.e., the standard method `getOwnSlotValues()` provides the return value).

Using the above design pattern, we have experimented with extending Jess [2] and JessTab [1] with additional functionality similar to the CommonLisp Object System (CLOS) meta-object protocol (MOP) [5]. The results are encouraging in that it is possible to implement several important MOP functions for Jess that could not otherwise be realized (i.e., without modifying the Protégé source code directly).

Discussion

It is important to have adequate means of distributing and installing extension packages to Protégé. One of the major advantages of tab widgets, slot widgets, and storage back-ends is that they are loaded as modules into Protégé. It is possible to distribute compiled plug-ins, which a standard Protégé distribution can use directly. Such extensions usually work across several Protégé versions, but occasionally it is necessary to update them to the evolving Protégé application programmer interface. We have found that aspect-oriented programming is an interesting approach to extending Protégé in situations where source-code modification would otherwise have been required. The current version of AspectJ accomplishes this task in general, but supple weaving during Java class load-time will be needed to avoid the annoyance and extra work for the user of running the weaver on the compiled Protégé jar file.

Summary and Conclusions

Aspect-oriented programming is a powerful approach to extending the Protégé functionality when the regular plug-in mechanisms are insufficient. Currently, it is possible to use AspectJ to weave new aspects together with the existing Protégé application. To be truly useful, however, this approach should support load-time weaving to relieve the users from the task of running the AspectJ weaver on their Protégé installation. The AspectJ developers will add support for this functionality in a forthcoming release. Hence, we believe that developers soon could use AspectJ routinely to create powerful Protégé extensions.

Acknowledgements

This work was supported by Vinnova (grant no. 2002-00907) and by The Swedish Research Council (grant no. 621-2003-2991).

References

[1] Henrik Eriksson. Using JessTab to integrate Protégé and Jess. *IEEE Intelligent Systems*, 18(2):43–50, 2003.

[2] Ernest Friedman-Hill. *Jess in Action: Java Rule-based Systems*, Manning, 2003.

[3] John H. Gennari, et al. The evolution of Protégé: An environment for knowledge-based systems development. *International Journal of Human–Computer Studies*, 58(1):89–123, 2003.

[4] Joseph D. Gradecki and Nicholas Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*, Wiley, 2003.

[5] Gregor Kiczales, Jim des Rivières, Daniel G. Bobrow. *The Art of the Metaobject Protocol*, MIT Press, 1991.

[6] Ramnivas Laddad. *AspectJ in Action*, Manning, 2003.

URLs

Protégé: http://protégé.stanford.edu/

AspectJ: http://eclipse.org/aspectj/