Aspect-Oriented Programming in the Design of Computer Algebra Libraries

Markus A. Hitz
Department of Mathematics and Computer Science
North Georgia College & State University
Dahlonega, GA 30597, USA
Email: mahitz@ngcsu.edu

Abstract

During the past five years, the Java programming language has been steadily gaining ground against C++, and older, structured languages. Its major strengths are portability, strong typing, and built-in support of networking, threading, and distributed computation. Most often, poor performance, lack of operator overloading, and limitations in sub-classing, are cited as weak points of Java. With the introduction of "generics" in the next major release, Java will add its own version of templates. The suitability for symbolic and algebraic computation has been discussed in [4], with respect to general features of the language, and in [7] with a special focus on run-time performance. The "Meditor" project [6] started out as a text editor that could process in-line mathematical expressions, and generate MathML output. Over time, it evolved into a small, yet impressive computer algebra system, entirely implemented in Java. It includes versions for handheld computer architectures. Industry analysts predict that by 2006, 60% of applications for mobile devices will be built in Java.

Aspect-Oriented Programming (AOP) was originally conceived as a "post-" Object-Oriented Programming (OOP) method. However, AOP does not mean to replace OOP, it tries to complement it. As the introduction to [8] notes: "While the tendency in OOP is to find commonality among classes and push it up in the inheritance tree, AOP attempts to realize scattered concerns as first-class elements,

and eject them horizontally from the object structure." Indeed, "separation of concerns," cross-cutting, and weaving are the main themes of AOP. Aspects add a third dimension to Java, where classes and interfaces constitute OOP elements. By defining "join points," aspects can attach to data fields and/or methods, without jeopardizing OOP type checking. The most mature AOP library for Java is AspectJ [2].

We propose to employ AOP at several points in the design of Computer Algebra Systems (CAS). AOP can be used to decrease the amount of duplicated code, to check for special cases in a non-intrusive manner, and might even improve the performance of the system. A simple example would be the enforcement of domain-specific rules (e.g., for "add" methods that reside in disparate sub-classes). Using the additional "degree of freedom," more elaborate type systems (e.g., in the spirit of AXIOM) can be built. In particular, aspects could be used to define algebraic structures over data types that are defined by Java classes, while interfaces would provide for restriction and/or composition. Our samples take ideas from the design of the GiNaC system [3], and the Java code of [6].

Acknowledgement

This work is supported in part by the National Science Foundation under Grant No. CCR-0098175.

References

[1] `http://aosd.net/`. Aspect-Oriented Software Development web site.

[2] `http://eclipse.org/aspectj/`. AspectJ project web site.

[3] Christian Bauer, Alexander Frink, and Richard Kreckel. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *J. Symbolic Computation*, 33:1–12, 2002.

[4] Laurent Bernadin, Bruce Char, and Erich Kaltofen. Symbolic computation in Java: An appraisement. In *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation, ISSAC'99*, pages 237–244, 1999.

[5] Johannes Grabmeier, Erich Kaltofen, and Volker Weispfenning, editors. *Computer Algebra Handbook*. Springer-Verlag, 2003.

[6] Raphael Jolly. `http://jscl-meditor.sourceforge.net/`. jscl-meditor project web site.

[7] Arthur C. Norman. Further evaluation of Java for symbolic computation. In *Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation, ISSAC'00*, pages 258–265, 2000.

[8] Guest Editors Tzilla Elrad, et al. Special issue on aspect-oriented programming. *Communications of the ACM*, 44:28–97, October 2001.