JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2002**

Vol. 1, no. 4, September-October 2002

Reflective Software Engineering - From MOPS to AOSD

Dave Thomas, Bedarra Corporation, Carleton University and University of Queensland

Reflective Programming has long been viewed as an elegant but academic subject that is of interest only to educators and researchers. The seminal work on Procedural Reflection by Brian Smith clearly articulated the benefits of allowing an executing program to have access to the underlying data structures and algorithms that govern its own computation [1].

The first implementation of the reflective tower was in 3Lisp. This was followed by work in the Lisp [2, 3]) and Smalltalk communities [4]. The research work on reflection has most frequently appeared in OOPSLA [5,6] and ECOOP conference proceedings as well as Reflection conferences dedicated to the subject [7]

To many outsiders, the phrase "going meta" conjures up visions of taking a trip to Nepal as opposed to a way of thinking about software development. However, those who have experienced the "engine room" via a Scheme meta-circular interpreter (see www-mitpress.mit.edu/sicp), or Smalltalk or CLOS meta-class programming, have a fundamentally deeper perspective on computation.

1 EVERY COMPUTER SCIENTIST SHOULD GO META AT LEAST ONCE IN THEIR LIFE

While other software professionals and researchers are confounded or left obsolete by the frequent changes in language and computational infrastructures, the software professionals who have reflective experience are much more resilient to these same changes. For many years students have remarked on the "aha" impact of seeing the "elementary particles of computation". Actually seeing these variables, environments, expressions, closures, and continuations miraculously opens up a whole new perspective on the simple *read-eval-print* loop.

Unfortunately far too many CS/SE faculty members fail to teach or even appreciate the reflective view of computation. Providing and understanding the meta view is, for computer scientists, the analog of understanding Laplace transform theory for differential equations in mathematics. In the transformation space, the original problem is replaced by a much simpler one that can be easily understood and solved then mapped back or *reflected* into the original problem space.

Many who lack this perspective puzzle over the semantic account of a new language. Those with a deeper understanding of reflective mechanisms can quickly toss off the syntactic baggage of a new programming language and easily identify and focus on its unique features and anomalies. In addition, those languages designed with a clear semantic account actually seem to have fewer anomalies and so do not require behavior experiments or huge amounts of source code debugging in order for the developer to understand the compile and runtime features of the language.

2 PROGRAMS AND DATA NEED SELF DESCRIPTION

A language without reflection is much like a database without a schema. Serialization of complex object structures in Smalltalk and Java is only possible because the languages have class information present at runtime. A more recent example is found in Web Services where the primary advantage of SOAP over CORBA and RMI is the ability of messages to be more self-descriptive. It is a sad commentary on our industry that it has taken so very many years to appreciate this simple and powerful idea. Now that we have rediscovered it let's hope we remember it.

While it can be argued that some programmers don't need meta-programming facilities, those who build development IDEs and runtime support such as debuggers definitely do! The lack of standardized, easily used meta information for C++ was a major inhibitor for C++ tool developers. Significant efforts were required to rebuild the meta model for compiler artifacts (see Lucid Energize, VisualWorks for C++, IBM Research Montana)

Java's weak reflective model was extended and is essential for the tools that manipulate programs such as GUI Builders, code generators, and debuggers. The MS.NET facilities for meta-programming are widely recognized as providing an elegant facility for developers to annotate programs and provide both descriptive and site specific optimization information to JITs, and underlying runtimes. For example, it is possible to annotate a specific C# call at the use site to have code generated for a special legacy calling sequence or to generate a SOAP message without changing the C# code.

3 EVERY MODEL NEEDS A META MODEL

For many software engineers, the whole idea of reflective computation seems impractical at best and silly at worst. Meta Models have long been used and appreciated in the modeling community. However for many years Meta Modelling was something that some architects did and then used the results to generate the concrete descriptions and code for developers. Meta Models are also well accepted by tool vendors. For many years vendors used their own efforts to build intelligent Case (iCase) Meta Models such as IBM ADCycle that would support automated code generation (MDA circa 1980?).

Meta modelling especially without the right tool support also requires a great deal of experience and discipline. Hence many early meta models contain numerous defects or leak at the seam from one meta level into the other. There is of course a danger that when one goes meta several levels, all interesting problems become easy, because the higher-level abstractions have removed essential and important details. These challenges lead many to view meta models with considerable scepticism.

Recently the desire to provide a unified semantic basis for UML and UML extensions and other OMG related efforts such as EDOC, and Model Driven Architecture have resulted in renewed interest in meta models in the form of the OMG Meta Object Facility (MOF) [8]. The MOF also enables standards such as UML Diagram interchange and UML - XML Metadata Interchange (XMI). It is the MOF of course that makes UML a moving target since UML is easily morphed into something completely different by extending the MOF. Perhaps this will lead to a meaningful semantic account for UML, at least one can hope.

UML, MDA and Ontology in XML have resulted in a renewed research interest in Meta modelling under the name of Model Engineering (see www.metamodel.com/wisme-2002).

4 AOSD: COMPOSITION FILTERS, ASPECTS, AND MULTIPLE DIMENSIONS OF CONCERN

This once rarified idea of meta-programming is threatening to become an everyday practice in software development. The large attendance and excitement at the recent 1st International Conference on Aspect Oriented Software Development (AOSD) (see http://trese.cs.utwente.nl/aosd2002/) shows the considerable interest in AOSD in the academic and industrial research community." Aspect-oriented software development is a new technology for separation of concerns (SOC) in software development. The techniques of AOSD make it possible to modularize crosscutting aspects of a system" www.aosd.net.

AOSD (see also the October 2001 issue of Communications of the ACM) has its origins in the following research efforts –

Composition Filters (see http://trese.cs.utwente.nl/composition_filters/);

AspectJ (see www.aspectj.org/servlets/AJSite);

Multi-Dimensional Separation of Concerns (MDSOC) and HyperJ (see www.research.ibm.com/hyperspace/);

and Demeter DJ (see http://www.ccs.neu.edu/home/lieber/AOP.html and www.ccs.neu.edu/research/demeter/DJ).

Both Composition Filters and Aspect Oriented Programming have their roots in reflective computation whereas MSDOC and Demeter have evolved more from program design ideas based on separation of concerns. Each allows separate, *ideally* orthogonal concerns to be represented as a code fragment which is a statement about the changes to be made to the underlying program. For the purposes of this article they address the same issues with different approaches. However there are important differences in implementation and use (see urls above).

5 SEPARATION OF CONCERNS

The most important contribution of the reflection community is to clearly illustrate the potential of *separation of concerns (see Parnas http://www.acm.org/classics/may96/*) As described by IBM HyperJ researchers "Separation of concerns is simply an approach to decomposing software into modules, each of which deals with, and encapsulates, a particular area of interest, called a concern. Examples of concerns are functions, data types or classes, features (such as persistence, print, or concurrency control), variants, and roles. Object-oriented languages permit decomposition by class, but only by class. Unlike classes, other kinds of concerns cannot be encapsulated in single modules; instead, their implementations end up scattered across the class hierarchy." Researchers discovered that by making simple programmatic changes at the meta-level, profound changes in the underlying program could be effected without actually changing the program code.

Quickly other investigators showed that this provides elegant ways to separate complex programming issues including concurrency and parallelism, persistence, transactions, and security. For example, composition filters provides a simple software composition approach for applying these techniques.

Despite these successes meta-programming and run-time reflection was abhorrent to some software engineers and its use was hotly debated within the Smalltalk and Lisp communities. Further the whole idea of procedural runtime reflection seemed completely ridiculous for widely used languages such as C++. Many dismissed reflection as potentially interesting as a thinking tool, but impractical for commercial use. I suspect many of my readers still hold this view.

6 TAMING RUN-TIME REFLECTION

In the late 90s work on OpenC++ showed that many reflective changes could be achieved at compile time without the overhead and complexity of meta-level facilities. Similar efforts for Java, which use class file rewriting, have been proposed [9]. It is amazing what really bright people will do when faced with a technology that isn't amenable to their research needs.

The work on providing procedural reflection for Java and C++ was not lost on researchers at Xerox PARC who combined their interest in separation of concerns;

compile-time reflection; and improved program development and maintenance to develop the concept of Aspect Oriented Programming (AOP). The work on AOP and in particular AspectJ implements reflection using a program transformer called a *weaver*. The appropriately named weaver transforms the underlying program by weaving in each of the aspects at their defined point of application.

Independently, researchers at IBM developed HyperJ that evolved from work on subject oriented programming. Their research demonstrated that separate concerns could be captured as code fragments and organized into *hyperslices* and hypermodules that can be composed with tool support to construct programs that contain different concerns.

All of these systems use a set of orthogonal descriptions about the program and induce program transformations. This can be implemented at the source code or for some languages at the binary level, if the language has sufficient self-description such as Java class files).

These results clearly show that it is possible to bring the benefits of the earlier theoretical research into current languages such as Java, C# and UML. AspectJ and HyperJ allow programmers to develop programs using AOSD today and provide practical platforms for software engineers to explore the use of aspects/concerns.

7 GREAT MINDS OFTEN THINK ALIKE

To my knowledge the earliest use of such a technique appeared in a technical report on Trellis-S (1988) where systematic substitution similar in concept to weaving was applied to the language Trellis OWL to allow sequential programs to be automatically transformed into distributed programs in the spirit of Emerald. It clearly showed the benefits of such an approach for introducing the distribution aspect into a sequential program. For many years the AI/Lisp community has used *mixins* and *method combination* to adapt code without requiring explicit editing of the code. For Smalltalk, the ENVY/Developer class extension, subapplication and image builder provide support for weaving at image build time

The systematic generation of programs from descriptions has of course also been the major focus of research in generative programming (see Generative Programming - Methods, Tools and Applications", Addison-Wesley 2000) some of which also has its roots in meta-programming and reflection. One of the interesting aspects of this work has been the tailoring of algorithms. It is important to note that AOSD is not limited to OOP and like pair programming and refactoring can also be applied to functional languages [15].

8 ASPECTS VIOLATE ENCAPSULATION?

One of the major concerns of AOSD sceptics is the apparent violation of encapsulation using AOSD. Program generation and weaving which appears to obliterate the original program clearly violates encapsulation, the principal benefit of object orientation. AOSD proponents argue that encapsulation happens at the level above the program where the concerns are clearly and separately modeled. What does this mean for unit testing, aspect testing and program correctness? This requires some new thinking on components, especially for those who feel the clear need and benefit of binary components. Will it really be possible to debug at the level of the abstraction? All of these challenges have of course appeared before with the use of macros and program generators. This led at least two of the participants in my ECOOP 2000 panel to argue that Aspects were really nothing more than *smart macro programming.* It will take a very complete tool set to convince many software engineers that they can count on AOSD to regenerate their programs.

9 ORTHOGONALITY AND NUMBER OF DIMENSIONS?

Most of the reflective and AOSD examples are based on a small number of “orthogonal” aspects. However, it isn’t clear as yet if large numbers of additional aspects will be discovered. Perhaps like design patterns there will turn out to be a small set of well known concerns which are used and reused in the practice of AOP. One clear challenge is assuring that developers are aware of any interaction (non-orthogonal) between aspects since this greatly increases the difficulty of understanding an AOSD design or program. One of the reasons multiple inheritance has such a bad rap is that extensions beyond specification inheritance (interfaces) and simple mixins were fraught with complex rules and contributed to programs that were difficult to understand and maintain. HyperJ provides tools to identify potential interactions between aspects that are essential if AOSD is to move in this direction.

10 DYNAMIC ASPECTS

Some aspects approaches provide very flexible dynamic aspects that support what amounts to predicate or instance based aspect execution. This can come very close to the kinds of things done in fully reflective systems, but can also greatly complicate program understanding, testing and verification. The presence of dynamic aspects in systems, which throw exceptions and execute in concurrent and transactional environments seems very daunting unless they are limited to a trusted infrastructure. In effect dynamic aspects can result in self modifying programs which are very complex and difficult to test and reason about (many similar techniques have been explored in the AI community http://www.dreamsongs.com/NewFiles/HOPL2-Uncut.pdf.)

11 WHO WILL USE AOSD?

AOSD is one more of a number of concepts and practices that must be understood by software professionals. It seems increasingly clear that all these advanced concepts such as interfaces (separation of specification and implementation inheritance), patterns, conformance and refinement hierarchies, and refactoring in addition to Java, C#, UML, XML, MDA, SOAP/JINI/Messaging, and scripting, may collectively be just too much stuff for day to day developers. Several of my colleagues argue that their students know all of these concepts and that they can be taught to any programmer who wants to learn, however my experience says that while students may have indeed seen all of them, most of them become proficient in only a few techniques.

The reality is that we are growing an increasingly large gap between the skills of professional software engineers and those of day-to-day application developers. If AOSD can contribute to reducing the complexity of the software that day-to-day developers see, it will be a wonderful contribution. It will be important to gain further experience to see where it fits in the software development process – analysis, design, implementation, reverse engineering etc. (see www.cs.ubc.ca/labs/spl/papers/2002/aosd02-concerns.html).

12 APPLICATIONS – CAN ANYONE REALLY USE THIS STUFF?

As usual with any new development AOSD has yet to prove itself and find it's place in the design, development and maintenance of large software systems. However preliminary experience reports from early adopters are encouraging. Given AspectJ, HyperJ, DJ, AspectS and AspectR [10] developers can now gain practical experience. It is clear that good tool support is essential and the very thought of debugging and maintaining generated programs is going to require a convincing and compelling tool set.

Clearly AOSD like reflection will have an impact on the next generation of developers and will give them a much better ability to deal with the complexities associated with transactions, persistence, security, logging, tracing and exception handling. At a minimum they should induce a discipline of program transformation in the spirit that ER models are used to work above the level of the relational database implementation.

One of the largest opportunities may be for EJB developers where simple business logic is often tangled inside complex Java code for threads, transactions, security, exception handling, logging etc. Perhaps it might be possible using AOSD to present the business developer with a much simpler programming model, similar to transaction monitors such as CICS or Tuxedo, where the complexities of SMP, threads, caching, security, transactional integrity, distribution and persistence are provided underneath by a friendly and helpful weaver. AOSD techniques should be useful for those implementing and reasoning about such middleware [13] [14]. AOSD should also be applicable at the design level with UML and the implementation level for MDA. One simple example is the Usecase concept of *extensions* or *built on* relationship that needs either runtime or compile time modification of program behaviour.

13 THE ASPECT REFACTORING OPPORTUNITY

Most of the current application focus for AOSD is design, development or runtime usage. My personal opinion is that the most exciting opportunity is an *Aspect Refactoring* tool in the spirit of Aspect Browser [11] and Refactoring Browser for Smalltalk and Java [12]

Complex tangled programs evolve naturally due to the humans and circumstances that create them. Usually different developers are unaware or unable to discern the separate concerns. These tangled programs are largely responsible for holding companies hostage to legacy code and further they induce fear in new developers who see significant risk in attempting to enhance or repair defects in such complex systems. Surely aspect refactoring will be a key technique for Agile/XP developers. It should allow them to systematically reduce the complexity and rigidity of large legacy applications.

14 CONCLUSIONS

AOSD has the promise to unravel the tangled programs [16] we weave as software evolves over the life cycle. Even more promising it may allow us to be free from the "tyranny of the dominant decomposition" [17] that forces us to prematurely select one design approach over another. AOSD tools provide new ways to describe, factor and compose software, and perhaps most importantly to reverse engineer and/or refactor legacy code. This combined with Agile/XP approaches to development could finally lead to the end of "death march" projects and turn maintenance into a more creative development activity. We will learn much more in the coming years as the early adopters gain experience using AOSD, and associated tools become integrated into popular tool sets.

REFERENCES

[1] B. C. Smith. *Reflection and semantics in lisp*. Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, pages 23--35, 1984.

[2] *Objvlisp - Metaclasses are First Class: The ObjVlisp Model*, P. Cointe, SIGPLAN Notices 22(121): 156-167 (Dec 1987) (OOPSLA '87).

[3] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, MA, 1991.

[4] Briot J.-P., Cointe P. Programming with Explicit Metaclasses in Smalltalk-80. In *Proceedings of OOPSLA '89*, ACM SIGPLAN Notices, Vol. 23 No. 10, pp. 419--431, 1989.

[5] Pattie Maes: Concepts and Experiments in Computational Reflection. OOPSLA 1987: 147-155

[6] Takuo Watanabe and Akinori Yonezawa. Reflection in an Object Oriented Concurrent Language. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, September 1988.

[7] www.openjit.org/reflection2001

[8] www.omg.org/technology/documents/formal/mof.htm

[9] www.csg.is.titech.ac.jp/~chiba/pub/chiba-ecoop00.pdf, www.cs.ncl.ac.uk/research/dependability/reflection/, http://www.csg.is.titech.ac.jp/openjava/)

[10] www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/

http://aspectr.sf.net/

[11] www.cs.ucsd.edu/users/wgg/Software/AB/ and http://www.cs.ubc.ca/~jan/amt/

[12] http://st-www.cs.uiuc.edu/users/brant/Refactory/, www.instantiations.com/jfactor/default.htm, www-106.ibm.com/developerworks/library/l-eclipse.html

[13] www.ccs.neu.edu/research/demeter/evaluation/gte-labs/aosd2002/CommercialAOSD_AOSD2002.ppt

[14] http://trese.cs.utwente.nl/Workshops/adc2000/papers/Filman.pdf

[15] Filman, R. E. and Friedman, D. P. *Aspect-oriented programming is quantification and obliviousness*. Workshop on Advanced Separation of Concerns, OOPSLA.

[16] www.technologyreview.com/tr10_kiczales0101.asp

[17] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton, Jr. *N* Degrees of Separation: Multi-Dimensional Separation of Concerns. *Proceedings of the International Conference on Software Engineering (ICSE'99)*, May, 1999.

About the Autor

Dave Thomas is CEO of Bedarra Corp., Adjunct Professor at Carleton University, Canada and University of Queensland, Australia, founding Director of AgileAllinace.com, and founder of Object Technology International. Bedarra works with research labs and commercial partners to transition innovations into products and practices.