JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 5, May-June 2004

Message Oriented Programming
The Case for First Class Messages

Dave Thomas
Bedarra Corp., Carleton University and University of Queensland

1 MESSAGES SHOULD BE OBJECTS TOO!

The hallmark of object-orientation is the use of objects to model a computation. If something is interesting it should be modeled as an object (more properly a class or prototype). Naïve models treat exceptions, transactions and bitblt as methods. However, when one designs a real system, these objects are so interesting that they need to be modeled as classes or class collaborations with many methods.

In this article we argue that the message-centric view deserves more attention. The *message-object* perspective provides the natural dual to the more common *object-method* perspective. We call this Message Oriented Programming (MoP).

Ideally, Messages should have first class status in the language as metaclasses. However, one can apply MoP to design and implement using a generative approach as is done with aspects.

2 MOTIVATION

We do not claim that the MoP idea is new, rather it is a useful metaphor for many problems, which are thorny when viewed using popular object models. My first exposure to viewing a computation from the message perspective came from the inspirational paper on messages as active objects [1]. Wall showed that by using the *message perspective* versus the traditional *process perspective*, it was much easier to describe and reason about the correct behavior of many distributed algorithms. Over the years I have come across many situations where the problem was more easily modeled from the perspective of a message rather than an object.

Specifically, the following solutions can be viewed elegantly from the message perspective: use case extensions [9]; capability and role based security [18, 20]; synchronization [6, 10]; optimization [14,15]; method combination [13,16]; active

Cite this column as follows: Dave Thomas: “Message Oriented Programming”, in *Journal of Object Technology*, vol. 3, no. 5, May-June 2004, pp. 7-12.
http://www.jot.fm/issues/issue_2004_05/column1

messages [11]; message patterns [7, 20]; composition filters [8] and multicast communication [12], subject oriented programming [19]. In each of these situations it is critical that the programmer be able to refine the message dispatch without violating the encapsulation. Since this isn't possible in current languages, solutions such as above require either a new language or internal VM/compiler modifications or source code changes to the sender/receiver methods. We conjecture that a language that supports first class messages would provide a more elegant and trustworthy solution.

3 WHAT IS FIRST CLASS[1] AND WHY DOES IT MATTER?

One of the major challenges in the design and implementation of a programming language is the consistent definition and treatment of the major concepts in the language. First class abstractions in a language are usually considered the mark of clean language design. One can often notice the treatment of concepts as second class when there are lots of restrictions on the definition of sub concepts or their use. A language concept is called "first class"[1] when it can be used freely in the programs in all contexts in which this would be reasonable.

Functional languages, for example, are defined by their support for first class functions. Thus in functional languages it is possible to write functions that create other functions; functions can be assigned to variables; functions can be passed as parameters to other functions and finally functions can return functions as results. Scheme [2] for example, is a popular teaching language because it has first class functions and the static scoping allows lexical closures to be used to model numerous other computational mechanisms such as streams, objects etc.

Metaclasses support First Class Concepts

First class concepts typically are modeled in OO languages via metaclasses. In OO languages such as CLOS and Smalltalk the programmer can introduce new concepts by working with the classes that define the concepts in the language. Metaclasses are the essential underpinnings for these languages. Class *Class*, for example, defines semantics of class declaration and instantiation.

Meta modeling has been advocated as a vehicle for understanding advanced computational concepts in languages using computational reflection. While first considered purely academic and poorly understood by many, Meta modeling [4, 5] has proven to be a powerful tool for modelers, language designers and implementers. Once considered excess baggage by traditional language designers, metaclasses' first class status in C#, which supports compile-time reflection and Java reflection, was added to support tools that need runtime introspection. More recently, Aspect oriented

[1] Readers familiar with the concept of first class can skip this informal discussion that is provided for readers who are unfamiliar with the concept, which is often only taught in the context of functional languages such as Scheme, ML or Haskell or languages with metaclasses such as Smalltalk or CLOS.

development AOSD [3] and UML MDA tooling is based on reflective and model driven program generation.

Why Not Make Everything First Class?

While highly desirable, the treatment of all things as first class is often challenging both in terms of a clean semantic account and a clean and efficient implementation. Meta classes are typically implemented by run-time reflection. NeoClasstalk [5] provides one of the cleanest reflective implementations that truly support meta programming. The elegance of run-time reflection is very appealing but presents major implementation challenges[2] that will often be of limited use by the primary users of the language.

Most conventional implementations do not allow you to really tinker with all details of the implementation. The limitations all have practical and rational justification that in most cases is one of efficiency. Hence even most dynamic OO languages restrict what can be done via metaclasses.

For example, although one can override method lookup by redefining the lookup function in Common Lisp or use the `doesNotUnderstand: hack` in Smalltalk, both practices are frowned upon because of performance and maintainability. Similarly, while control structures can in principle can be defined as methods, in practice all implementations treat `ifTrue:ifFalse` and `whileTrue:` as fixed and inline them. Scheme has first class functions and second-class objects/methods (lexical closures trap environments and functions) while Smalltalk has first class objects and second-class functions (blocks) and methods.

Language Engineering

Sometimes a refinement of a powerful concept can meet the needs of the majority of applications. For example, blocks in Smalltalk are not first class lexical closures and exceptions have varied semantics in different dialects rather than being first class continuations. The majority of blocks are used in simple control structures and callbacks hence do not require binding the lexical scope beyond the enclosing method instance.

C# gets by without lexical closures through the concepts of *delegates* and *iterators*. C# handles event callbacks with its *delegates* that are bound to an object instance versus a lexical closure or block. C# iterators provide a language mechanism which is extensible in that it provides a means for dealing with the common case of sequencing through a collection of objects and applying a function to the contents of the collection. These concepts cover the most common usage of lexical closures and hence merit special treatment. The use of two more limited, clean concepts, even if they are more limited in functionality, simplifies the language implementation. (Note - This is not to suggest that we think the omission of first class closures from Smalltalk, the CLR or JVM is a good idea!)

[2] We conjecture that by using the dynamic optimization techniques available today and given the speed of current and future processors, one could indeed build an efficient language that supports runtime reflection.

Compile Time Reflection and Generative Programming

In cases where the language lacks reflective capabilities, a preprocessor or generator may implement first class concepts. This is called generative programming or compile-time reflection. Aspects [8] for example, are usually implemented by a tool called a *weaver* that inserts the aspects into the source code of the application program. This creates a transformed source program that then is compiled and executed.

4 TOWARDS MOP

Compile Time Message Tailoring

One very important use of compile time MOP is the ability to tailor a specific call site so that the actual code compiled for that site is tailored. This allows the programmer to talk to the compiler in an intimate way to allow, for example, more efficient code or foreign calling sequences. I believe that the ability to treat a specific call site specifically first appeared in an industrial research project called Intrigue at Xerox Parc. This project explored reflection as means for tailoring C code generation for embedded devices.

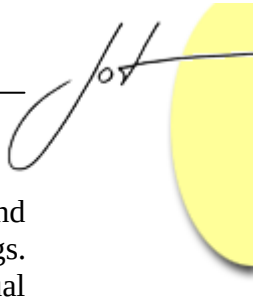
In C#, metaclasses are used to provide information to language tools such as browsers, debuggers, code generators etc. For example, meta information is used to annotate a standard method call site causing it to be compiled as a SOAP message rather than invoking the default C# method dispatch. Unfortunately, implementing such optimizations must be done by customizing the JIT, so this approach is not for the faint at heart.

Toward MoP

In object languages the user provides a declaration of the class and method signature. Method dispatch is abstractly defined as: `send(object,methodname,arguments)`,- where message is seen as a data structure on the runtime stack. However, message sending is magic, as it is performed using internal machinery hidden[3] in the compiler or virtual machine.

Unfortunately, few meta models reify messages as a concept in the meta model. To support MoP we need to allow both the definition and evaluation of messages. We need to introduce an explicit metaclass *Message* with a method: `send(sender,receiver,args)`. By default methods are created as instance of StandardMessage whose send implements the standard OO dispatch. In MoP, messages evaluation replaces method dispatch as the essential mechanism which can be abstractly

[3] Technically CLOS provides the ability to override the method lookup function and Smalltalk developers have used method wrappers or doesNotUnderstand to insert code into intervene in the method dispatch. Both of these techniques however are outside the bounds of normal CLOS or Smalltalk programming and should only be used by experts, if at all.



defined as: `send(message,sender,receiver,args)`, where the sending and receiving objects are explicit parameters to the message evaluation in addition to the args. Additional message types can be defined for things like security enforcement, unusual inheritance, proxy, multicast etc.

While MoP has not been extensively explored, several researchers are working on first class messages. Variations of the above approach have been explored in part in Coda [10] and more recently in Pico [21] and Pic% [22] and Classtalk [5]. The Pico and Pic% research has only recently come to my attention and is very interesting. Researchers in subject oriented programming are investigating the use of first class messages in component engineering in a project called Message Central [19].

REFERENCES

[1] D.W. Wall. "Messages as Active Agents". In *ACM Symposium on Principles of Programming Languages (POPL)*, Albuquerque, New Mexico, January 1982.

[2] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1986

[3] Dave Thomas: "Reflective Software Engineering - From MOPS to AOSD", in *Journal of Object Technology*, vol. 1, no. 4, September-October 2002, pp. 17-26. http://www.jot.fm/issues/issue_2002_09/column1

[4] Kiczales, G., des Rivires, J., and Bobrow, D.G., *The Art of the Metaobject Protoco*, MIT Press, 1991.

[5] F. Rivard. "Smalltalk: a reflective language". In *Proceedings, Reflection*, 1996.

[6] L. Bergmans and M. Aksit, "Composing Synchronisation and Real-Time Constraints", *Journal of Parallel and Distributed Computing 36*, pp. 32-52, 1996. http://trese.cs.utwente.nl/publications/paperinfo/compsyncandrt.pi.top.htm

[7] Phillippe Mougin and Stephane Ducasse. *OOPAL: integrating array programming in object-oriented programming* http://portal.acm.org/citation.cfm?doid=949305.949312

[8] Publications on Aspect-Oriented Software Development and Composition Filters http://trese.cs.utwente.nl/publications/publication_topics/aosd.htm

[9] Ivar Jacobson: "Use Cases and Aspects – Working Seamlessly Together", in *Journal of Object Technology*, vol. 2, no. 4, July-August 2003, pp. 7-28. http://www.jot.fm/issues/issue_2003_07/column1

[10] Jeff McAffer. "Meta-Level Programming with CodA". In *Proceedings of ECOOP'95*, LNCS 952, pages 190-214. Springer-Verlag, 1995.

[11] D. Tennenhouse and D. Wetherall. "Towards an active network architecture". *Computer Communication Review*, 26(2):5--18, August 1995.

[12] R. Guerraoui and A. Schiper. “Genuine Atomic Multicast in Asynchronous Distributed Systems”. *Theoretical Computer Science*, 254(1--2):297--316, Mar. 2001.

[13] Ingalls, D.H.H: “A Simple Technique for Handling Multiple Polymorphism”. In *OOPSLA'86 Conference Proceedings*, 1986.

[14] Urs Holzle, Craig Chambers, and David Unger. “Optimizating dynamically-typed object-oriented languages with polymorphic inlined caches”. In *ECCOP '91 Proceedings*, pages 21--38. Springer-Verlag, July 1991.

[15] Karel Driesen, Urs Holzle, Jan Vitek. “Message Dispatch on Modern Computer Architectures”. *ECOOP ‘95 Conference Proceedings*, p. 253-282, Arhus, Denmark, August 1995.

[16] Chambers, C.: “Object-Oriented Multi-Methods in Cecil”, *ECOOP '92 Conference Proceedings*, Utrecht, The Netherlands, July 1992.

[17] H. Minsky and D. Rozenshtein, “A Law-Based Approach to Object-Oriented Programming”, *ACM SIGPLAN Notices, Proceedings OOPSLA '87*, vol. 22, no. 12, pp. 482-493, Dec 1987.

[18] E, the secure distributed pure-object platform and p2p scripting language for writing Capability-based Smart Contracts http://www.erights.org/

[19] Message Central http://www.research.ibm.com/messagecentral/mop.htm

[20] Markus A. Hof, *Using Reflection for Composable Message Semantics*, http://www.ssw.uni-linz.ac.at/General/Staff/MH/Research/Thesis/index.html

[21] Theo D’Hondt. *The Pico Project*, http://pico.vub.ac.be/

[22] Theo D’Hondt and Wolfgang De Meuter. “Of first-class messages and dynamic Scope”, *RSTI, L’Object – LMO 2003*, p. 137 – 149.

About the author

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