JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 8, November-December 2006

Programming with Models – Modeling with Code
The Role of Models in Software Development

Dave Thomas, Bedarra Research Labs

1 MODELS AND MODEL RELATIONSHIPS

A model is a simplified description/representation of a more complex entity or process. It is used frequently to simplify and analyze a complex entity usually by focusing on one or more aspects. We use models to understand, communicate, explore alternatives, simulate, emulate, calibrate, evaluate and validate. All models lie a little; some lie a lot! By definition, a model is an abstraction of the real thing.

In practice, engineers work with partial descriptions (models) and add/remove model elements at each level of abstraction. Models describe a slice of life – not life itself. Modelers use many different models to explore different aspects of the thing being modeled such as requirements, data, behavior, event/time, security, flow, process, activity, performance, quality, usability, etc. These models can take many forms and be expressed through many different tools including systems dynamics, prototypes, textual and visual models of artifacts and their relationships. It is important to note that UML is just one of many models representations which may be used in this process.

The set of central relations in model-driven engineering and object-oriented technology are fundamentally different. The two central relations in object-oriented technology are *isA* and *isPartOf.* In model engineering the key relations are *isBasedOn, isLike,* and *isRepresentedBy*. These inherently one-to-many and partial relationships are often mistakenly confused with OOP relationships. Trying to use one set of relations, as a central interpretation, in the wrong context, can lead to serious problems. Models provide a representation of partial truths about the software system, but only the final code contains the actual truth.

Cite this column as follows: Dave Thomas, "Programming with Models – Modeling with Code. *The Role of Models in Software Development*", in *Journal of Object Technology*, vol. 5, no. 8, November -December 2006, pp. 15-19 http://www.jot.fm/issues/issue_2006_11/column2

2 MODEL-DRIVEN DEVELOPMENT – *EXPRESSING MODELS AS CODE*

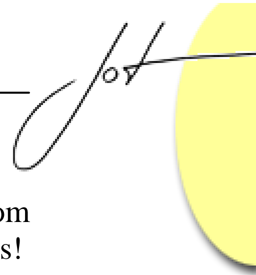
Simula [1], the source of today's object technology, originated an approach to software architecture, design and implementation that is based on building software by building simulation models. In this approach models are expressed in code, augmented where appropriate by tabular or graphical visualizations of the static or dynamic structure of the system. This Model-Driven Development approach advocates iterative modeling of the system to be built using successive levels of refinement – requirements, architecture, components and features are all refined from concept to final code. This very much parallels the approach in other engineering disciplines where one starts with conceptual models which allow for high level conceptual agreement, moving next to successively fine-grained prototypes and simulation models where one can achieve behavioral agreement, then building an emulation model where one can obtain functional and nonfunctional agreement, and finally refining that to the actual product.

One advantage that software developers have over physical engineers is that we can use our core tools and techniques throughout the development cycle. Other engineering disciplines are forced to use specialized tools such as simulators, CAD tools, physical prototypes, multiple languages, emulators, etc. We have the huge benefit of being able to program from start to finish. We note in passing that hardware engineers have largely abandoned visual notations as implementation tools and instead use high level languages such as VHDL with the associated gains in productivity.

The art of programming is about managing the half-truths at higher levels of abstraction in such a way that they are sufficiently correct to make progress, and sufficiently vague that they can be refined at the next level. Good software evolves like a book outline, starting with an initial set of key ideas which are ruthlessly refactored into the final form. Model refinements and compositions can at best be captured as a set of annotated hyperlinks which describe the evolution of the software as a literate program.

The functional and dynamic communities, as well as the rapid application development 4GL community, all have substantial experience with model-driven development. This is greatly facilitated by properties of these languages, which allow models to be created easily and changed quickly. People who have used these languages just think differently about software than those who only have experience with compiled procedural languages. Design patterns in these languages are language idioms, rather than mechanisms which must be emulated in less powerful languages. Indeed Agile development has evolved directly from best practices used in the Smalltalk and Lisp communities.

I remember the early days of Smalltalk, when people would stop me to ask how I could program before the analysis and design were done. I replied that the Smalltalk Browser was an active data dictionary and people were always impressed by such a neat



tool for doing documentation. Little did they know the functional specs published from Smalltalk were already implemented and the specs were just the class/method comments!

Modeling through code isn't an attempt to justify premature hacking of code. Rather it is about writing code to understand what one is building. It is unfortunate that many people don't seem to appreciate that often code can provide the best expression of a model.

3 LITERATE MODELING

The Agile Manifesto emphasizes the importance of code. Unfortunately many misinterpret "code" to mean just the program text, whereas what was meant was literate programming, where code is written to be read, and all of the artifacts are available to the reader. One of the major reasons for collective ownership and pair programming is to develop code which can be easily understood by others. Some AgileXP zealots insist that code needs no comments or documentation since the code should be easily understood from its own clarity and its associated unit and acceptance tests. They have a well-founded concern about the tendency for a comment to be outdated or not refactored when the code is changed. However, this concern doesn't, in my experience, justify the omission of properly structured comments which improve the readability and understandability of the code. Hopefully in the future IDEs will allow one to treat the comment as a true part of the code [7].

The focus in Agile development is in producing high quality code artifacts that are written well enough that the code itself serves as the primary artifact in the system. Documentation artifacts are associated with the code where necessary to enhance the models expressed in the code by providing the pragmatics and semantics that cannot be inferred directly from the code itself. For example, OO languages lack the ability to describe relationships between objects. Many to many relationships require an explicit relationship to be made manifest in the code, however, clever developers know that one to many relationships can be implemented directly using a collection. Unless the code is annotated appropriately or modeled as a relationship the implicit relationship will be hidden and can easily be overlooked in the relationships changes from one to many to many to many.

Since it is tedious to write effective documentation in a programming IDE, especially for requirements, we have found Wikis to be a simple and effective mechanism for capturing such information, with annotations linking to the code and vice versa (http://trac.edgewall.org/). Wikis and annotations should also reference more extensive artifacts, such as models, standards and prototypes, which can elaborate the requirements. Modern web technology allows code to be hyperlinked with documentation, providing the best of both worlds.

4 BENEFITS OF MODELS AS CODE

There are many advantages to expressing models in code. Models can be delivered to both customers and developers as a means for communicating the essence of the system both externally and internally. Models as code provide a single expression of the system in the form of a literate program. Since the models themselves are code, they can be readily understood by developers. This approach eliminates the need for multiple representations using ADL, UML, etc., while allowing such representations to be generated automatically from code models.

Models can be developed using powerful IDEs and can leverage modern language features such as annotations and documentation capabilities. Models are version-managed as part of the overall development process, which allows all artifacts to be traced through their evolution. Refactoring tools can be used to support model evolution. Finally, modern IDEs integrate with visual modeling and display tools to provide multiple visual perspectives. Recent work in the AOP community has demonstrated that architectural constrains/styles can be enforced using an architecture expressed in code.

REFERENCES

1. Ole-Johan Dahl, The Birth of Object-Orientation, http://folk.uio.no/olejohan/birth-of-oo.pdf June 2001

2. J.Sklenar, Introduction to OOP in Simula, http://staff.um.edu.mt/jskl1/talk.html, 1997

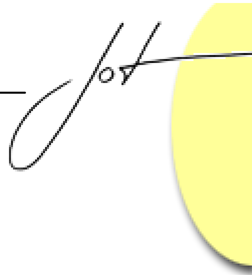
3. Ole Lehrmann Madsen, What mainstream OO can learn from SIMULA and BETA, http://www.jaoo.dk/file?path=/2006/slides/OleLehrmannMadsen_WhatMainstreamOOCanLearn.ppt, Oct 2006

4. Wikipedia, Literate programming, http://en.wikipedia.org/wiki/Literate_programming

5. Christopher Lee, Literate Programming – Propaganda and Tools, Carnegie Mellon University http://vasc.ri.cmu.edu/old_help/Programming/Literate/literate.html

6. Donald E. Knuth, Literate Programming, CSLI Lecture Notes, no. 27, 1992 http://www-cs-faculty.stanford.edu/~uno/lp.html

7. Hanspeter Mössenböck, Kai Koskimies, Active Text for Structuring and Understanding Source Code (1995)



About the author

Dave Thomas is cofounder/chairman of Bedarra Research Labs (www.bedarra.com), www.Online-Learning.com and the Open Augment Consortium (www.openaugment.org) and a founding director of the Agile Alliance (www.agilealliance.com). He is an adjunct research professor at Carleton University, Canada and the University of Queensland, Australia. Dave is the founder and past CEO of Object Technology International (www.oti.com) creator of the Eclipse IDE Platform, IBM VisualAge for Smalltalk, for Java, and MicroEdition for embedded systems. Contact him at dave@bedarra.com or www.davethomas.net.