Editor: Robert L. Glass ■ Computing Trends ■ rglass@indiana.edu

On the Declarative Specification of Models

Diomidis Spinellis

… in which I oppose the uncritical overuse of graphical drawing tools for modeling

Colleagues in my research group and in collaborating institutions typically model software designs using graphical tools such as Rational Rose, Together, and Visio. I often witness them toiling to adjust a graph's appearance with the mouse or laboriously visiting each class to change a single field's type. This need not be so. Design models should be composed textually, and graphs should be automatically generated. You might find it perverse to employ two different representations (textual and graphical) for the same underlying model. So, to substantiate my view, I'll outline the advantages of graphical models and describe the benefits gained from directly manipulating a textual representation, illustrating my point using a prototype implementation.

Graph-based models

No rule specifies that models should appear in a graphical form. A model is a simplification of reality, so a model for a software artifact could really be an outline of that artifact; think of a class definition without code in the method bodies. However, we usually prefer to examine many of our models in a graphical representation: UML employs nine different diagrams for visualizing different perspectives of a system.

Using a diagram to represent a model has several advantages. When we examine a model's graphical representation, we use our visual cognitive apparatus, which has millions of years of evolutionary advantage over our text-reading abilities. A diagram's 2D representation is a lot more expressive than text, which readers typically scan from left to right and from top to bottom. We can view diagrams from different directions to gain distinct insights, while using a larger symbol set makes them more expressive. In addition, we can obtain different levels of detail from the same diagram: a bird's-eye view will easily convey a system's structure, while examining a class in detail can reveal its collaborators. Finally, a diagram can let us identify patterns; again, 2D pattern matching is an activity we humans are particularly good at.

The drawing-editor approach

Designers typically create their model diagrams using a drawing editor. The semantic distance between the editor's graphical model representation and the underlying software artifact can vary enormously. Some tools, such as Visio, are purely drawing aids. Others, such as Rational Rose, offer round-trip engineering (code-to-model and model-to-code generation), while tools such as ArgoUML provide domain-specific advice during design. However, all drawing editors require you to place and manipulate shapes on the canvas, which, regardless of the help that tools such as ArgoUML's broom alignment tool provide, is tedious and time consuming. The effort and the motor coordination skills required for this activity are mostly irrelevant to the end result. Unlike

Continued on p. 94

0740-7459/03/$17.00 © 2003 IEEE

Continued from p. 96

architectural or mechanical-engineering models, the appearance of a software system's model diagram is only marginally related to the represented software design's quality.

The drawing activity is, however, a creative task providing immediate feedback; software engineers thus often focus on delivering a nice picture rather than an effective design. Furthermore, the model's internal representation is typically opaque or under the drawing-editor tool's control, and thus at odds with vertical software process activities such as configuration and revision control. Finally, the semantic distance between the model and the artifact is large enough to burden activities that are naturally performed on software code such as refactoring, automatic code generation, and metric extraction. This is true even for design tools that support round-trip engineering of models, such as Rational XDE and Jbuilder 6.

Declarative modeling

Computer power and automatic-graph-drawing algorithms[1] have sufficiently advanced so as to allow the automatic placement of graph nodes on the canvas and the near optimal routing of the respective edges. So, we can design models using a declarative textual representation and subsequently view, publish, and share them in graphical form. Building architects employ a similar technique when they create realistic ray-traced pictures of a building out of "2-1/2 dimensional" ground plans (drawings where numerous *xy* coordinates share a single height datum, such as all of a house's 3-meter-high walls). My prototype uses a model expressed in a Java-like notation (see Figure 1a) to automatically create the diagram (see Figure 1b).

Creating models in a declarative, textual notation offers several advantages. The model composition mechanism matches well both a programmer's high-level skills (the textual, abstract formalization of concrete concepts) and low-level skills (the manipulation of text using an editor and other text-based tools).

Also, the declarative notation, by being closer to the program's representation, forces the designer to distinguish between the model and the respective implementation and between essential system characteristics and trivial adornments. In addition, designers using a declarative model will find it more difficult to get away with, as they often do now, drawing for a model a nice picture of the implementation they have in mind.

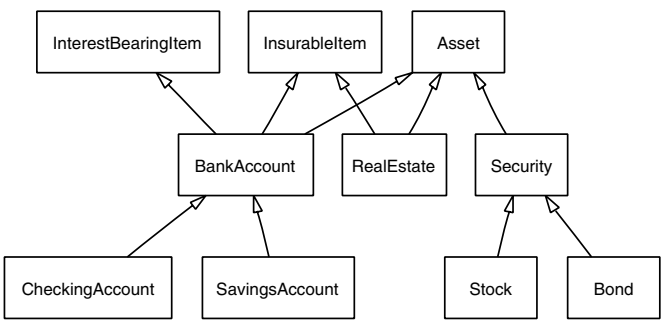
Furthermore, the declarative representation is highly malleable. The existing visual structure does not hinder drastic changes, and users don't waste effort on the tidy arrangement of graph nodes, thus lifting a psychological barrier against massive design refactoring.

Declarative models can also be automated easily: they can be generated from even higher-level descriptions by trivial scripts and tools operating on design process inputs such as database schemas, existing code, or structured requirements documents.[2] Text macro processors can provide configuration management, while revision control and team integration activities can utilize the same proven tools and processes that are used for managing source code. So, with a tool such as RCS (Revision Control System), you can keep track of design revisions, create and merge branches, and monitor model changes, while a system such as CVS (Concurrent Versions System) lets you split work into teams.

Finally, the declarative approach

```
class Asset {}
class InterestBearingItem {}
class InsurableItem {}
/**
 * @extends InsurableItem
 * @extends InterestBearingItem
 */
class BankAccount extends Asset {}
/** @extends InsurableItem */
class RealEstate extends Asset {}
class Security extends Asset {}
class Stock extends Security {}
class Bond extends Security {}
class CheckingAccount extends BankAccount {}
class SavingsAccount extends BankAccount {}
```

(a)



(b)

Figure 1. Automatic graph drawing: (a) a model expressed in Java-like notation; (b) the diagram created from the model.

can readily utilize existing text-processing tools for tasks that a drawing editor might not provide. Consider how your favorite model editor handles the following tasks and how you could handle them using a simple Perl script or a text-processing pipeline applied to the declarative model specification:

- Identify all classes containing a given field (as a prelude to an aspect-oriented cross-cut).
- Count the total number of private fields in a given design.
- Order methods appearing in multiple classes by their degree of commonality.
- Identify differences between two design versions.

The declarative specification of software models is clearly not a panacea. My current UML diagram design prototype sometimes stresses *dot*, the underlying graph layout generator, into generating graphs with overlapping edges and nodes. For example, when dot draws UML association relationships, the multiplicity and visibility adornments might overlap with the respective edges. Furthermore, learning the declarative notation might be more difficult than experimenting with the toolbars of a GUI-based diagram editor competing for the designer's attention. However, because a profession's maturity is also judged by the tools its practitioners use, I believe that building and adopting a sharp declarative-modeling toolset will enrich and advance software engineering. You can download the tools I used to generate the diagram in this article from www.spinellis.gr/sw/umlgraph.

Acknowledgments

The prototype I describe could not exist without the Graphviz graph visualization system; John Elson and Stephen C. North graciously incorporated my changes for UML arrow styles into the tool's source distribution. Spyros Oikonomopoulos provided feedback during the development of this work.

References

1. E.R. Gansner et al., "A Technique for Drawing Directed Graphs," *IEEE Trans. Software Eng.*, vol. 19, no. 3, May 1993, pp. 214–230.
2. D. Spinellis and V. Guruprasad, "Lightweight Languages as Software Engineering Tools," *Proc. Usenix Conf. Domain-Specific Languages*, Usenix Assoc., Berkeley, Calif., 1997, pp. 67–76.

Diomidis Spinellis is an assistant professor in the Department of Management Science and Technology of the Athens University of Economics and Business. Contact him at dds@aueb.gr.

Circulation: *IEEE Software* (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., PO Box 3014, Los Alamitos, CA 90720-1314; (714) 821-8380; fax (714) 821-4010. IEEE Computer Society headquarters: 1730 Massachusetts Ave. NW, Washington, DC 20036-1903. Subscription rates: IEEE Computer Society members get the lowest rates and choice of media option—$43/34/56 US print/electronic/combination; go to http://computer.org/subscribe to order and for more information on other subscription prices. Back issues: $10 for members, $20 for nonmembers (plus shipping and handling). This magazine is available on microfiche.

Postmaster: Send undelivered copies and address changes to Circulation Dept., *IEEE Software*, PO Box 3014, Los Alamitos, CA 90720-1314. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Product (Canadian Distribution) Sales Agreement Number 0487805. Printed in the USA.

IEEE Software

How to Reach Us

Writers
For detailed information on submitting articles, write for our Editorial Guidelines (software@computer.org) or access http://computer.org/software/author.htm.

Letters to the Editor
Send letters to

Editor, *IEEE Software*
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
software@computer.org

Please provide an email address or daytime phone number with your letter.

On the Web
Access http://computer.org/software for information about *IEEE Software*.

Subscribe
Visit http://computer.org/subscribe.

Subscription Change of Address
Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software*.

Membership Change of Address
Send change-of-address requests for IEEE and Computer Society membership to member.services@ieee.org.

Missing or Damaged Copies
If you are missing an issue or you received a damaged copy, contact help@computer.org.

Reprints of Articles
For price information or to order reprints, send email to software@computer.org or fax +1 714 821 4010.

Reprint Permission
To obtain permission to reprint an article, contact William Hagen, IEEE Copyrights and Trademarks Manager, at whagen@ieee.org.