

# The Eclipse Modeling Framework

## Moving into model-driven development

---

The idea of building applications by first modeling them, then transforming these models into implementation code has been around for many years. Providing a higher level abstraction for defining software would seem to be a natural evolution. Twenty or so years ago, structured programming languages replaced assembly language, or machine code, as the most popular way to write software. About 10 years ago, object-oriented programming languages became entrenched as the most predominant languages, again raising the abstraction level. Lately, there's been a lot of talk about model-driven development as the next higher level abstraction.

Each step in the evolution of software development has been accompanied by skepticism, and model-driven development is no different. The skepticism is usually the result of overly grandiose visions and promises, opening up the visionaries to attack from the more practical types. Many programmers think that class diagrams might be helpful to document their designs, but they know that implementing complex systems by simply "transforming a picture" is a pipe dream. They know that all the expressive power of a programming language can't be available in a model because if it was, it wouldn't be any simpler (higher level). It would just be another programming language.

That said, most programmers do recognize that generating some of the code that they write over and over must be possible. How many copy-and-paste operations do you need to do before you start to wonder if there couldn't be a way of specifying parameters for patterns that you want, and just have the code generated automatically? Clearly, these patterns must represent some higher level abstraction that, if only it could be specified (modeled), could enable us to write a lot less code.

When building desktop applications, you need to start with good design and architecture. Because there is no universally accepted desktop-application framework, most developers design their own architecture, then build it into a framework themselves. However, the costs of this approach are a considerable expense, time, debug effort, support, and aggravation expended on solving a problem that is peripheral to building the business functionality of the application.

Another approach is to find a framework that accommodates your needs to simplify and accelerate project development. A "wish list" for such a framework would (among other things):

- ?? Implement a clear, consistent, and cohesive architecture.
- ?? Support development and execution on all the major desktop platforms (Windows, Mac OS X, Linux, QNX Photon, Pocket PC, HP-UX, AIX, Solaris).
- ?? Have UI response that is "snappy," while maintaining the platform's native look-and-feel.
- ?? Provide a variety of widgets, both standard (buttons and checkboxes) and extended (toolbars, tree views, progress meters).
- ?? Provide extensive text processing that include editors, position/change management, rule-based styling, content completion, formatting, searching, and hover help.
- ?? Support the use of platform-specific features (ActiveX, for instance) and legacy software (if necessary).

## **Eclipse: A Developer's Story by Mary Kroening**

### *A case study showing Eclipse at work on a real project.*

We are the makers of a Prolog language development system that runs on a variety of Windows and Unix platforms. Prolog differs from conventional programming languages in that it is non-procedural and rule-based. It has built-in search and pattern-matching capabilities--so programs run forwards looking for a match, then backwards on failure or to find additional matches. And, unlike other Prolog implementations, we specialize in embedding Prolog modules in conventional (procedural) languages and tools such as Java, C++, .NET, Delphi and Web servers.

Our existing IDE dates from the early 90's and was showing its age. So last year we started thinking seriously about a replacement. We wanted an IDE that would run on multiple platforms, and we wanted to support all the modern conveniences.

The question was how to do that without a budget of the likes of Microsoft or Borland? When we learned of Eclipse, we were immediately intrigued. Eclipse offered:

- ?? True open source licensing where we could develop and own our Prolog-specific additions yet contribute to the base product for all to benefit.
- ?? Ready-to-run downloads for a wide variety of Windows and Unix platforms.
- ?? Full source code and remote debugging support. The latter is especially important to us because our customers develop Prolog components as part of larger applications instead of stand-alone Prolog programs. Debugging embedded components (especially those running on Web servers) is especially challenging without remote debugging.

Eclipse also allowed our users to develop both parts of their application using the same IDE. That is, they could develop the user interface in Java or C++ and an intelligent logic-base and/or rule-base in Prolog without ever leaving Eclipse.