Editor: Brian Brannon
bbrannon@computer.org

From Programming to Modeling—and Back Again

Markus Völter

WHAT'S THE DIFFERENCE between programming and modeling? And should there be one? A long time ago, I started programming with Pascal, C++, and Java, but in the past couple of years, I've focused on domain-specific languages and model-driven development (MDD). Modeling is a different world from programming, especially because of the mindset and tools involved. But as I thought more about the dichotomy between the two, I concluded that what we really need is a set of composable language modules that express different software concerns—some application-domain specific, others more related to technical concerns and thus more generic and reusable. This idea isn't new, but the time is right to discuss it again, especially as the necessary tools are maturing.

Programming and Modeling Today

I distinguish software development from programming because software development includes requirements engineering, development processes, design, and documentation, whereas programming is the act of actually implementing and testing a running program.

The main tools for programming are general-purpose languages such as Java, C#, C/C++, Scala, and Ruby. These languages typically use procedural, functional, or object-oriented abstractions in various combinations, but building nontrivial applications also requires any number of additional languages and formalisms, including XML, HTML, and state charts.

Developers also rely heavily on frameworks. In the Java world, prominent examples include JEE, Spring, and Hibernate, which come with their own—usually XML-based—configuration "languages." Integration among the various languages, frameworks, and technologies is often limited and based on name references; tool support is built specifically for each framework, or sometimes a combination of frameworks.

Programming languages can't represent architectural concepts as first-class entities, which leads to all kinds of maintainability problems as well as limited analyzability and tool support for architectural concepts such as ports, protocols, pre- and postconditions, messages, queues, data replication specifications, persistence mappings, and synchronization. Components, for example, are often represented as all the classes in a package, a façade class registered in some kind of configuration file, or even an XML descriptor that somehow captures the metadata describing the component.

Today's programming languages also aren't very good at representing abstractions, concepts, and notations borrowed from specific application domains—it isn't easy to work with vectors, matrices, or temporal data in general-purpose languages, for instance. Of course, you can use a library, but libraries don't provide for convenient nota-

0740-7459/11/$26.00 © 2011 IEEE

TABLE 1

The differences between modeling and programming.

Task	Modeling	Programming
Define your own notation and language	Easy	Sometimes possible
Syntactically integrate several languages	Possible, depends on tool	Hard
Graphical notations	Possible, depends on tool	Usually only visualizations
Customize generator/compiler	Easy	Sometimes possible w/ open compilers
Navigate/query	Easy	Sometimes possible w/ IDE and APIs
Constraints	Easy	Sometimes possible w/ Finbugs and other tools
Sophisticated, mature IDE	Sometimes	Standard
Debugger	Rarely	Almost always
Versioning/diff/merge	Depends on tooling	Standard

tions, static type checking, or compiler optimizations.

MDD, on the other hand, lets you create formal, tool-processable representations of certain aspects of software systems.[1] You can then use interpretation or code generation to map these representations to executable code expressed in a suitable programming language and the necessary XML/HTML/whatever files. To make the models semantically meaningful, it helps to develop domain-specific (modeling) languages that closely align with the particular concern they're supposed to describe. With today's tools, it's easy to define arbitrary abstractions to represent any aspect of a software system in a meaningful way. It's straightforward to built code generators that can create executable artifacts. Depending on the particular MDD tool used, it's also possible to define suitable notations that make the abstractions accessible to nonprogrammers (for example, opticians or thermodynamics engineers, two of my current projects).

Based on years of my own experience, I'm convinced that MDD is a step in the right direction and leads to a significant increase in productivity, quality, and overall system consistency. However, it has its limitations, the biggest one being that modeling languages, environments, and tools are distinct from programming languages, environments, and tools. The level of distinctiveness varies, but in many cases, it's big enough to cause integration issues that can make MDD adoption challenging.

Why the Difference?

So what, then, really is the difference between programming and modeling? Table 1 offers a few general observations, but the bigger question is *why* there's a difference. My guess is history—both worlds had different origins and evolved in different directions.

Programming languages have traditionally used textual syntax (I'll come back to this), whereas modeling languages have used graphical notations because of the ill-conceived generalized notion that something represented with pictures is always easy to understand. Of course, we've always had textual domain-specific languages (and failed graphical general-purpose programming languages), but the use of textual syntax for domain-specific modeling has only recently become more prominent.

Additionally, programming languages typically use storage based on concrete syntax along with parsers to transform the character stream into an abstract syntax tree (AST) for further processing. Modeling languages use editors that directly manipulate the abstract syntax, using projection to show the concrete syntax in the form of diagrams. Modeling tools provide the ability to define views—that is, to show the same model elements in different contexts, often with different notations. This has never been a priority for programming languages.

But in spite of these surface differences, programming and modeling should be based on the same fundamental approach. Programmers don't want to model—they just want to express different system concerns with the appropriate abstractions and notations.

A Vision for Modular Programming Languages

A modular language has a minimal core language, as well as a library of language modules that you can import for use in a given program. A language module is like a framework or library, but it comes with its own syntax, editor, type system, and integrated development environment (IDE) tooling.

Once you import a language module, it behaves like an integral part of the composed language—that is, it integrates with other modules by referencing symbols or by being syntactically embedded in code expressed with another language module. Language modules are also integrated at the type system and semantic levels.

Decades of experience show that textual syntax, together with good tool support, is perfectly adequate for large and complex software systems.

As I mentioned earlier, this idea isn't new. Charles Simonyi (http://research.microsoft.com/apps/pubs/default.aspx?id=69540) and Sergey Dmitriev (www.onboard.jetbrains.com/is1/articles/04/10/lop) have written about it, as has Guy Steele in the context of Lisp (http://video.google.com/videoplay?docid=-8860158196198824415#). The idea also relates very much to language workbenches as introduced by Martin Fowler (http://martinfowler.com/articles/languageWorkbench.html). In particular, Fowler defines language workbenches as tools with the following characteristics:

- *Users can freely define languages that are fully integrated with each other.* This is the central idea for both language workbenches and modular languages because you can easily argue that each language module is what Fowler calls a language. "Full integration" can refer to referencing as well as embedding, and it includes type systems and semantics.
- *The primary source of information is a persistent abstract representation and language users manipulate a DSL through a projectional editor.* Storing programs in their abstract representation and then using projection to arrive at an editable representation is useful, and maybe even the best approach to achieving modular languages, but in the end I don't care as long as languages are modular. If this is possible with a different approach, such as scannerless parsers, it's fine with me.
- *Language designers define a DSL in three main parts: schema, editors, and generators.* I agree that ideally a language should be defined "metamodel first," meaning you first define a schema, then the editor or grammar, and then the generator to map your constructs to existing languages. However, it's also okay to start with the grammar and derive the metamodel. From the user's viewpoint, it doesn't make a big difference.
- *A language workbench can persist incomplete or contradictory information.* This is trivial if you store models in a concrete textual syntax, but it's not so trivial if you use a persistent representation based on the abstract syntax.

Let me add two additional characteristics for language workbenches. For the languages built with a workbench, I want tool support: syntax highlighting, code completion, static type checking, and, ideally, a debugger. A central idea of language workbenches is that language definition always includes IDE definition. The two should be integrated. I also want to be able to program complete systems within the language workbench. This means that together with DSLs, general-purpose languages must be available in the environment based on the same language definition/editing/processing infrastructure. Depending on the target domains, this language could be Java or C#, but it could also be C for the embedded community. Starting with an existing general-purpose language also makes the adoption of the approach simpler: incremental language extensions can be developed as the need arises.

Notation

Generally, I expect the notation to be textual. Decades of experience show that textual syntax, together with good tool support, is perfectly adequate for large and complex software systems. This becomes even truer if you consider that programmers will have to write less code: the abstractions available in the languages align much more closely with the domain than is the case for traditional languages. Programmers can always define a language module that fits a domain.

However, there are worthwhile additions to consider. Semigraphical syntax would be useful—that is, the ability to use graphical symbols as part of a fundamentally text-oriented editor; mathematical symbols, fraction bars, subscript/superscript, or even tables can make programs represent certain domains much more clearly. Custom visualizations are important as well. Visualizations are read-only, automatically laid out, and provide a way back to the program. They often illustrate certain global properties of the program or answer specific questions, typically those related to interesting metrics.

Finally, actual graphical editing would be useful for data structure relationships, state machine diagrams, and dataflow systems, to name a few. The textual and graphical notations must be integrated, though—you'll want to embed the expression language module into the state machine diagram to express guard conditions.

A Library

The importance of being able to build your own language varies depending on the concern at hand. As an example, let's assume that you work for an insurance company and want to build a DSL that supports your company's specific way of defining insurance contracts. You'll want the language to exactly align with your business, so you'll have to define the language yourself.

However, for a large range of technical or architectural concerns, the abstractions are well known and could be made available for reuse (and adaptation) through a library of language modules, for example,

- hierarchical components, ports, component instances, and connectors;
- data structure definitions and XML schema, including specifications for persisting the data;
- definitions of rich contracts (interfaces, pre- and postconditions, protocol state machines, and the like);
- various communication paradigms such as message passing, synchronous and asynchronous remote procedure calls, and service invocations; and
- abstractions for concurrency based on transactional memory or actors.

This sounds like a lot of stuff to put into a programming language, but remember, it won't all be in one language. Each of these concerns could be a separate language module for use in a program only if needed.

Clearly, it's not possible to define all these language modules in isolation: modules must be designed to work with each other, and a clear dependency structure must be established. A minimal core language that supports primitive types, expressions, functions, and maybe object orientation will likely act as the focal point around which additional language modules are organized.

Many architectural abstractions have to interact with frameworks, platforms, and middleware, so it's crucial that abstractions in the language remain independent of specific technology solutions. Additional (hopefully declarative) specifications might also be necessary: such a technology mapping should be a separate model that references the core program. Language modules define a language for specifying persistence, distribution, or contract definition. Technology suppliers can support customized generators that map programs to the APIs defined by their technology, taking into account possible additional specifications that configure the mapping. This is a little bit like service provider interfaces (SPIs) in Java enterprise technology.

Comparing Existing Languages to the Vision

Although they aren't widely used yet, some existing language workbench technologies are suitable for the vision of modular programming that I've outlined here.

JetBrains MPS

JetBrains' Meta Programming System (http://jetbrains.com/mps) is an open source language workbench that supports most of the technical requirements I mentioned earlier. It uses a projectional editor that renders the AST in a notation that looks and feels textual while you directly edit the tree. By the time you read this, tables and graphical notations should be possible as well, integrated with textual notations.

Defining a language starts with defining the abstract syntax; you define the editor (projection rules) in a second step and then define the type system and generator, which provide semantics by mapping your new language constructs to one of several existing base languages (currently C, Java, XML, or plaintext). Because Java and C are available in MPS, they're treated exactly the same as any other DSL you would build. You can also easily extend them with new language constructs. MPS comes with support for debuggers, as well as with good integration for existing version control systems; it supports diff and merge using the projected, concrete syntax.

Although MPS does a great job of making textual notations editable as if they were real text, the illusion isn't perfect; it takes a couple of days to change your editing habits.

Intentional Software's Domain Workbench

Charles Simonyi worked with Microsoft Research on a project about intentional programming; his company Intentional Software (http://intentsoft.com) is now continuing this research and bringing the concept to market as the Intentional Domain Workbench. Intentional hasn't published a lot about what it's doing, but it's using a projectional approach similar in concept but quite different in detail to JetBrains MPS. The rendering engine is more powerful than the one MPS uses—for instance, I've seen examples where it uses logic diagrams or tables as part of (otherwise normal) C programs. Other examples include insurance mathematics mixed with "normal" programs, so the ability to mix and match notations seems to be more sophisticated.

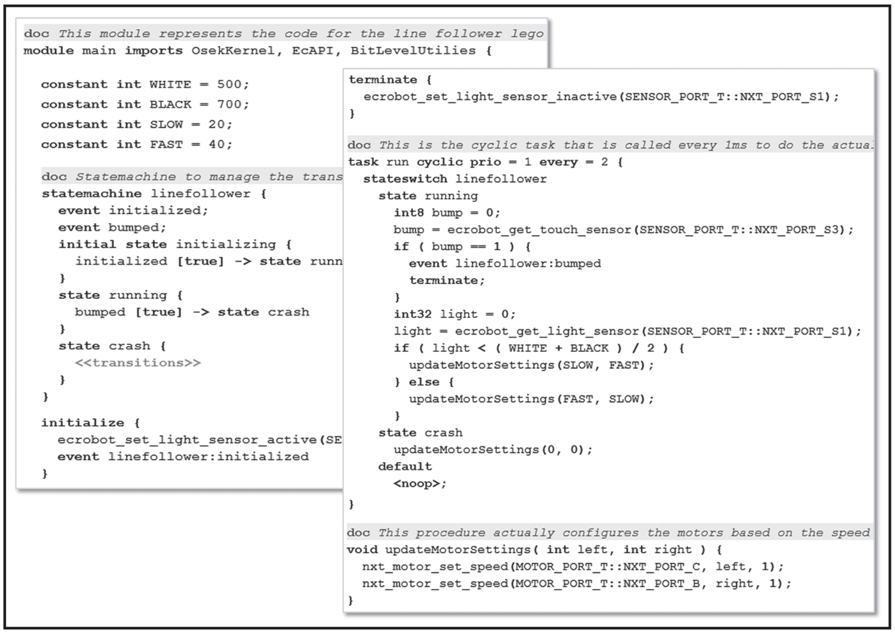
SDF, Stratego, and Spoofax

Eelco Visser and his group at TU Delft developed the SDF, Stratego, and Spoofax tools. SDF defines textual language syntax, Stratego is a term-rewriting engine used for transformation, and

MY EXPERIENCES WITH MODULAR LANGUAGES

In 2007, I was involved with a research project called AMPLE (http://ample.holos.pt). Together with Christa Schwanninger of Siemens and Iris Groher of TU Linz, I started creating a configurable language for architecture description based on the then-current oAW Xtext 4.3. This version of Xtext didn't support language modularization, not even the one-language-inheritance I described earlier, so I tried to implement modularization by using a C-style preprocessor to customize the language definition. From a user's perspective, the experiment worked: it was possible to select architectural features from a configuration model and then create a customized DSL that supported exactly these features. However, the preprocessor-based implementation was way too complex, and I had to give up on the experiment.

In winter 2008, I worked with Intentional Software on the implementation of the Pension Workbench for CapGemini (http://voelter.de/data/presentations/KolkVoelter_IntentionalSoftware.pdf). This product integrated text editing with a notation for insurance mathematics, a textual rule language for high-level pension plan specification, and a spreadsheet-like language for expressing unit tests for pension calculation rules. The spreadsheet language had been developed before and was reused for the unit tests. This is an example of successful language reuse and embedding. More recently, I've worked on a modular language for embedded software development (http://mbeddr.com). Based on MPS, my team and I have implemented several language extensions for C, including state machines, tasks, sensor access, and a DSL for robot control (see Figure A for some example code). This attempt has (so far) been much more successful than my initial one with the old Xtext. Because of MPS's powerful support for language modularization and composition, the implementation is well structured and easily extensible. The user experience is good as well, since the resulting languages are completely integrated. In July 2011, we started a government-sponsored research project that will investigate this approach further.



```
doc This module represents the code for the line follower lego
module main imports OsekKernel, EcAPI, BitLevelUtilies {

  constant int WHITE = 500;
  constant int BLACK = 700;
  constant int SLOW = 20;
  constant int FAST = 40;

  doc Statemachine to manage the trans
  statemachine linefollower {
    event initialized;
    event bumped;
    initial state initializing {
      initialized [true] -> state runn
    }
    state running {
      bumped [true] -> state crash
    }
    state crash {
      <<transitions>>
    }
  }

  initialize {
    ecrobot_set_light_sensor_active(SE
    event linefollower:initialized
  }
```

```
terminate {
  ecrobot_set_light_sensor_inactive(SENSOR_PORT_T::NXT_PORT_S1);
}

doc This is the cyclic task that is called every 1ms to do the actua
task run cyclic prio = 1 every = 2 {
  stateswitch linefollower
    state running
      int8 bump = 0;
      bump = ecrobot_get_touch_sensor(SENSOR_PORT_T::NXT_PORT_S3);
      if ( bump == 1 ) {
        event linefollower:bumped
        terminate;
      }
      int32 light = 0;
      light = ecrobot_get_light_sensor(SENSOR_PORT_T::NXT_PORT_S1);
      if ( light < ( WHITE + BLACK ) / 2 ) {
        updateMotorSettings(SLOW, FAST);
      } else {
        updateMotorSettings(FAST, SLOW);
      }
    state crash
      updateMotorSettings(0, 0);
    default
      <noop>;
}

doc This procedure actually configures the motors based on the speed
void updateMotorSettings( int left, int right ) {
  nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_C, left, 1);
  nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_B, right, 1);
}
```

FIGURE A. This example code from the mbeddr.com project shows C code extended with state machines and tasks. Each of these extensions are implemented as a language module that can be included as necessary in a given program.

Spoofax is an IDE framework for SDF and Stratego based on Eclipse (http://strategoxt.org/Spoofax).

Most traditional grammar specifications aren't closed under composition. This is a consequence of their use of a limited grammar class (such as ll(k)) and the fact that they work in two phases, where the first, tokenization, doesn't take into account the grammar structure. In contrast, SDF has no separate tokenization phase: the parser directly consumes the character stream. Its GLR parser can parse the complete class of context-free grammars. Consequently, SDF can deal with composed grammars much more gracefully. Additional declarative specifications can be added for disambiguation.[2]

Stratego maps the terms (think: fragments) of one tree to terms of an output tree. And because of the way Visser's group built Stratego, you can use the concrete syntax of the source and target languages when defining term-rewriting rules: a rewriting rule might look like "text pattern mapping," however, what really happens is that Stratego executes a model-to-model transformation, in which source and target model are written down in their respective concrete syntaxes.

The group is now focusing on its Eclipse-based Spoofax tooling to provide editor support for building and using SDF-based languages and Stratego-based transformations.

Eclipse Modeling and Xtext

Historically, the Eclipse Modeling Project's wide range of tools for building a DSL have been graphical, but the Xtext project (www.eclipse.org/Xtext) started supporting textual DSLs a couple of years ago, including the necessary tooling (syntax highlighting, code completion, and constraint checks—but no debugger). Because of the underlying ANTLR-based parser, language modularization and composition is limited, so a language can only inherit (and reuse/redefine concepts from) one other language. Although Xtext doesn't support direct integration with or extension of Java, it's possible to use the so-called JavaVM metamodel to reference and navigate to Java types in the Eclipse workspace. As of version 2.0, the reusable Xbase expression language can be embedded into DSLs via inheritance from the Xbase language. Xbase also provides an extensible type system, generator, and interpreter.

Internal DSLs

I'll say it up front: I don't think the current crop of general-purpose languages and their support for internal DSLs are up to the task. At first glance, it seems like you can use languages like Scala or Ruby, which support various forms of metaprogramming and provide a relatively flexible syntax, as a host language for modularized internal DSLs. For example, in Rails, you can extend certain base classes and get additional language constructs for your own program (data structure definitions, state-based behavior).

However, you face several limitations. First, your language module's syntax can only be what the host language's syntax supports. Depending on the host language, this might be quite a big space, but it's still limited. Second, language implementation usually happens via metaprogramming, which isn't based on a well-structured language definition. It's a potentially powerful approach, but the implementation of nontrivial language extensions is typically scattered and not very maintainable. The biggest disadvantage is the lack of IDE support for the DSL: no DSL-specific syntax highlighting, no code completion, and no type checking. This might not be a problem for users of dynamic languages (after all, they usually don't have this kind of IDE support for the host language, either), but I'm not willing to accept this limitation for situations where nonprogrammers have to use the DSL.

Another area where people talk about language definition and extension is Lisp, where you first define a language (extension) for a specific problem area and then use this language to solve a concrete problem (see Guy Steele's "Growing a Language" talk; http://video.google.com/videoplay?docid=-8860158196198824415#). However, Lisp hasn't caught on as a mainstream language for various reasons, and it doesn't have much of a syntax in the first place. To me, this proves that the approach of modular and extensible languages is indeed very useful. And if we make it "accessible to the masses" by providing good tool support, I think we're really on to something.

For historical reasons, software practitioners have made a distinction between modeling and programming, but I don't think it's helpful anymore. In the past, we lacked suitable tools to unify the two, but that's changing now. Take a look at the Language Workbench competition at http://languageworkbenches.net to get an overview of 13 (at the time of this writing) different language workbenches. Now is a great time to start!

Acknowledgments

I thank the participants of the workshop on textual DSLs held in February 2009 for the inspiration for this article: Axel Uhl, Bernd Kolb, Thomas Goldschmidt, Lennart Kats, Eelco Visser, Konstantin Solomatov, and Jos Warmer.

References

1. T. Stahl, *Model-Driven Software Development*, Wiley, 2006.
2. L.C.L. Kats, E. Visser, and G. Wachsmuth, "Pure and Declarative Syntax Definition: Paradise Lost and Regained," *Proc. Onward 2010*, ACM Press, 2010; www.lclnet.nl/publications/pure-and-declarative-syntax-definition.pdf.

MARKUS VÖLTER works as an independent researcher, consultant, and coach in Stuttgart, Germany. His focus is on software architecture, model-driven software development, and domain-specific languages as well as on product line engineering. Völter also regularly writes (articles, patterns, books) and speaks (trainings, conferences) on those subjects. Contact him at voelter@acm.org.

ERRATUM

The Viewpoints column, "Naïveté Squared: In Search of Two Taxonomies and a Mapping between Them" contained an error in "The Dark Side" sidebar on page 15. The correct link to Margaret-Anne Storey's interview with Robert L. Glass is http://doi.ieeecomputersociety.org/10.1109/MS.2011.89. *IEEE Software* regrets the error.