From Programming To Modeling – and back again

Version 1.2, 25 Nov 2010

Markus Völter
(voelter@acm.org)

Abstract

So, what's the difference between programming and modeling? And should there even be one?

A long time ago I started programming using the usual general-purpose languages of the time: Pascal, C++, Java. In the last couple of years I turned to domain specific languages and model driven development, building my own languages and compilers specific to certain technical and application domains, usually driven by a broad range of customer from various fields. However, modeling is a world different from programming – at least regarding tools. So as I think more about this, I came to the conclusion that there should be no difference between programming and modeling. What we really need is a set of composable language modules that each express different concerns of a software system, some application domain specific, others more related to technical concerns, and hence more general and reusable. In this article I explore this idea. Of course the idea isn't really new. However, as the necessary tools are maturing, the time is right to discuss this idea again.

Programming and Modeling Today

Programming

I distinguish software development from programming. Software development includes aspects such as requirements engineering, development processes, software design, and documentation. I consider programming to be the act of actually creating an implementation, as well as testing it.

The main tools for programming today are general purpose programming languages such as Java, C#, C/C++, Scala or Ruby. They typically use procedural, functional, or object oriented abstractions in various combinations. However, to build nontrivial applications, any number of additional languages and formalisms are required, including XML, HTML or state charts.

Frameworks are used heavily. In the Java world, prominent examples include JEE, Spring or Hibernate. They typically come with their own configuration "languages", usually XML-based. The integration between the various languages, frameworks, and technologies is often limited and based on name references. Tool support is limited and built specifically for each (combination) of the frameworks.

Architectural concepts, such as the following list, cannot represented as first-class entities in programs, leading to all kinds of maintainability problems as well as limited analyzability and tool support: hierarchical components and instances, ports, protocols, pre- and post conditions, messages, queue, data replication specifications, persistence mappings, or synchronization. Consider components. They are often represented as *all the classes in a package*, *a façade class that is registered in some kind of configuration file* or maybe an *XML descriptor* that somehow captures the metadata about the component.

Another limitation of today's programming languages is that they are not very good at representing abstractions, concepts, and notations of specific application domains. For example, it is not easily possible to work with vectors, matrices or temporal data in general purpose languages. Of course you can use a library. But such a library does not provide for convenient notations, static type checking or compiler optimizations.

Model-Driven Development

In model driven development [1] we create formal, tool-processable representations of certain aspects of software systems. We then use interpretation or code generation to transform those representations into executable code in traditional programming languages and the associated XML/HTML/whatever files. To make the models semantically meaningful, it is useful to develop domain specific (modeling) languages that are closely aligned with the particular concern they are supposed to describe. With today's tools it is easily possible to define arbitrary abstractions to represent any aspect of a software system in a meaningful way. It is also straightforward to built code generators that generate the executable artifacts. Within limits (depending on the particular MDD tool used), it is also possible to define suitable notations that make the abstractions accessible to non-programmers (for example opticians or thermodynamics experts – two of my current projects).

Based on years of my own experience, as well as the experience of the overall MDD community, I am convinced that model driven development is a step in the right direction and brings about significant improvements in productivity, quality, and overall system consistence.

However, there are also limitations to the approach. The biggest one is that modeling languages, environments and tools are distinct from programming languages, environments and tools. The level of distinctness varies, but in many cases it is big enough to lead to integration issues that can make adoption of MDD challenging. Let me provide some specific

examples. Industry has settled on a limited number of meta meta models, EMF/EMOF being the most widespread one. Consequently, it is possible to navigate, query and constrain models with a common API. However, since every programming language and IDE has its own API for accessing the program abstract syntax tree (AST), interoperability with source code is challenging – you can't treats source code the same way as models. A similar problem exists regarding IDE support for model-code integrated systems: you cannot mix (DSL) models and (GPL) programs while retaining reasonable IDE support. This results in generating skeletons into which source code is inserted, or the arcance practice of pasting C snippets into 5-in sized text boxes graphical state machine tools (and getting the errors only once the integrated resulting C code is generated).

Difference Programming vs. Modeling

So what really is the difference between programming and modeling today? The table contains some (general and broad) statements:

Task	Modeling	Programming
Define your own notation and language	Easy	Possible
Integrate various languages	possible, depends on tool	hard
Graphical Notations	possible, depends on tool	usually only visualizations
Customize Generator/Compiler	Easy	Sometimes possible based on open compilers
Navigate/Query	Easy	Sometimes possible, depends on IDE and APIs
Constraints	Easy	Sometimes possible with Finbugs & Co
Sophisticated Mature IDE	Sometimes	Standard
Debugger	Rarely	Almost always
Versioning/Diff/Merge	Good	Depends on tooling, usually not great

So of course one can and should ask: why is there a difference? I guess the primary reason is history, the two worlds have different origins and have evolved in different directions.

Programming languages have traditionally been using textual syntax (I will come back to this). Modeling languages traditionally used graphical notations (because of the ill-conceived idea that what is represented as pictures is *always* easy to understand). Of course there have always been textual domain specific languages (and (failed!) graphical general purpose programming languages), but the use of textual syntax for domain specific modeling is only now becoming more prominent.

Programming languages have traditionally used storage based on concrete syntax, together with parser technology to instantiate an abstract syntax tree for further processing. Modeling languages have traditionally used editors that directly manipulate the abstract syntax, using projection to show the concrete syntax (diagrams). Modeling tools have also provided the ability to define views, i.e. the ability to represent the same model elements in different contexts, often using different notations. This has never really been a priority for programming languages.

My central hypothesis is that there should be no difference, programming and modeling should be based on the same fundamental approach, enabling meaningful integration. No programmer really *wants* to model[1]. They just want to express different concerns of software systems with appropriate abstractions and notations. The next paragraph shows what I mean by this.

Modular Programming Languages

A modular language is made of a minimal language core, plus a library of language modules that can be imported for use in a given program. Each module addresses a specific concern of the system to be built (I will come back to this later). A language module is like a framework or library, but it comes with its own syntax, editor, type system, and IDE tooling. Once a language module is imported, it behaves as an integral part of the language, i.e. it is integrated with other modules by referencing symbols or by being syntactically embedded in code expressed with another module. Integration on the level of the type system and semantics is also provided.

This idea isn't new. Charles Simonyi has written about it in 1995 [11], and it is related to language workbenches as defined by Martin Fowler [2]. He defines language workbenches as tools where:

Users can freely define languages that are fully integrated with each other. This is the central idea for language workbenches, but also for modular languages since you can easily argue that each language module is what Martin calls a language. “Full integration” can refer to referencing as well as embedding.

[1] Of course there is a need for modeling in concept modeling, ontologies, systems engineering and for drawing bubbles on a flipchart. But *programmers* really don’t want to model as part of their work.

The primary source of information is a persistent abstract representation *and* **Language users manipulate a DSL through a projectional editor.** These two state that projectional editing be used. I don't agree. Storing programs in their abstract representation and then using projection to arrive at an editable representation is a very useful and maybe even the best approach (see below). However, 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 (see below), that is fine with me as well.

Language designers define a DSL in three main parts: schema, editor(s), and generator(s). I agree that ideally a language should be defined "meta model first", i.e., you first define a schema, then the editor or grammar, and then the generator to map your constructs to existing languages. However, it is also okay for me if I start with the grammar, and the meta model is derived. From the user's point of view, it does not make a big difference.

A language workbench can persist incomplete or contradictory information. I agree. This is trivial if the models are stored in a concrete textual syntax, is not so trivial if a persistent representation based on the abstract syntax is used.

Let me add two additional requirements. For all the languages I build in the workbench, I want to get tool support: syntax highlighting, code completion, static checking, and ideally also a debugger. A central idea of language workbenches is that language definition always includes IDE definition. The two should be integrated.

A final requirement is that I want to be able to program complete systems within the language workbench. This means that together with DSL, general-purpose languages must also 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: extensions can be developed as the need arises.

Notation

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

However, there are worthwhile additions. For example, semi-graphical syntax would be useful, i.e. the ability to use graphical symbols as part of a fundamentally text-oriented editor: mathematical symbols, tables, or

subscript/superscript can make programs resemble the domain much more closely.

Custom visualizations are important as well. Visualizations are read-only, automatically layouted and provide drill-down back to the program. They are used to illustrate certain global properties of the program or to answer specific questions, typically related to interesting metrics.

Finally, actual graphical editing is useful for certain cases. Examples include data structure relationships or state machine diagrams. The textual and graphical notations must be integrated, though: you will want to embed the expression language module into the state machine diagram to be able to express guard conditions.

A library of language modules

The importance of being able to build your own languages and generators varies depending on the particular concern. Assume that you work for an insurance company and you want to build a domain specific language that supports your company's specific way of defining insurance contracts. It is essential that the language is exactly aligned with *your* business, so you have to define the language yourself. There are other similar examples: building DSLs to describe radio astronomy observations for a given telescope, a language to describe cooling algorithms for refrigerators, or a language for describing telecom billing rules (all of these are actual projects I have worked on).

However, for a large range of technical or architectural concerns, the abstractions are well known. They could be made available for reuse (and adaptation) in a library of language modules. Here are some examples.

- Hierarchical components, ports, component instances, and connectors.
- Data structure definition à la relational model or hierarchical data à la XML and XML schema, including specifications for persisting that data
- definition of rich contracts, i.e. interfaces, pre- and post conditions, protocol state machines, and the like
- various communication paradigms such as message passing, synchronous and asynchronous remote procedure calls, and service invocations

This sounds like a lot of stuff to put into a programming language. But remember: it will not all be in one language. Each of those concerns will be a separate language module that can be used in a program as needed.

It is certainly not possible to define all these language modules completely independent of each. It is necessary to come up with a layering or dependency structure between them. The modules have to be designed to work with each other. Interfaces on language level support "plugging in"

new language constructs. The modules might for example rely on a minimal core language that comes with object orientation and expression support, maybe relatively close to languages like Java or C# today. Some of the tools outlined in the next section provide first implementations of this approach.

Many of these architectural concerns interact with frameworks, platforms and middleware. It is crucial that the abstractions in the language remain independent of specific technology solutions. In addition, when interfacing with a specific technology, additional (hopefully declarative) specifications might be necessary: such a technology mapping should be a separate model that references the core program. The 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 detail this technology mapping. This is a little bit like service provider interfaces (SPIs) in Java enterprise technology.

Existing Language Workbenches Technologies

In this section I want to look at a couple of existing language workbench technologies and their suitability to the vision of modular programming outlined above. In the section after that one, I will discuss my own experience with the approach.

JetBrains MPS

JetBrains' Meta Programming System [3] is an Open Source language workbench that supports most of the technical requirements mentioned above. It uses a projectional editor that renders the abstract syntax tree in a notation that looks and feels textual while the user directly edits the tree. By the time this is published, tables and graphical notations will be possible as well, completely integrated with textual notations. Defining a language begins with defining the abstract syntax, the editor (projection rules) is defined in a second step. You then define the generator which provide semantics by mapping your new language constructs to one of several existing base languages (currently C, Java, XML, or plain text). Because Java and C are available in MPS, they are treated exactly the same as any DSL you build. You can easily extend them with new language constructs. MPS also comes with support for building debuggers, as well as with good integration for existing version control systems - diff and merge is supported on the level of the projected syntax.

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

MPS has been used within JetBrains for a number of years and on several projects. JetBrains employees have built various languages, for example for persistence specification or Web user interfaces. Their new bugtracker, YouTrack, is built completely with MPS. A commercial tool for editing ActionScript called Realaxy has been built by an independent company. I have worked with it quite a bit as well, see below.

Intentional Software's Domain Workbench

Charles Simonyi has been working for Microsoft Research on a project called Intentional Programming. His company Intentional Software is now continuing this research and is productizing the system as the Intentional Domain Workbench [4]. Intentional has not published a lot about what they are doing, but a number of things are known:

Like JetBrains, Intentional uses a projectional approach. It is similar in concept, but quite different in detail. The layouting and rendering engine is more powerful than the one used by MPS. I have seen examples where logical diagrams or fraction bars are used 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

These tools are developed by Eelco Visser and his group at the TU Delft. SDF is a tool to define grammars and languages, Stratego is a term rewriting tool used for translation, and Spoofax is an IDE framework for SDF and Stratego based on Eclipse [5].

Traditional parsers use two phases: in phase one, tokenization, the character stream is broken into the tokens defined by the language. In phase two the parser consumes the tokens, checks the token sequence for conformance to the grammar and builds an AST. Since tokens are defined without any context, ambiguous grammars can arise if grammars are combined that define different tokens for the same sequence of characters

SDF in contrast has no separate tokenization phase. The parser directly consumes the character stream, everything is context-aware. If language modules are combined, there can never be a problem with overlapping token definitions. In case several interpretations of the same text stream is possible, SDF returns several syntax trees, and additional (compositional) grammar rules can be added for disambiguation. Language composition is therefore no problem. Please take a look at this Onward paper [12] for details.

Stratego is a term rewriting framework based on SDF. It maps terms (think: tree fragments) of one tree to terms of an output tree. As a consequence of how Stratego is built, it is possible to use the concrete syntax of the source and target languages when defining term rewriting rules: a rewriting rule

looks like "text pattern mapping". However, what really happens is that a model to model transformation is executed, where source and target model are written down in their respective concrete syntaxes.

Showing that SDF can handle non-trivial languages, Eelco and his group have implemented Java, XML, and HTML based on SDF. They have also built a set of languages called WebDSL that showcase the idea of using different language modules to address different aspects of developing web applications.

Eelco's group now develops the Eclipse-based Spoofax tooling for SDF and Stratego, providing editor support for building and using SDF-based languages and Stratego-based transformations.

Eclipse Modeling and Xtext

The Eclipse Modeling Project provides a wide range of tools for developing domain specific languages. Historically, they have been mainly graphical. More recently, the Xtext [6] project supports textual domain specific languages. It supports the definition of textual DSL 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 - a language can inherit (and reuse and redefine concepts from) one base language. Direct integration with or extension of Java is not supported. However, using the so-called *JavaVMMetamodel*, it is easily possible to reference and navigate to Java types in the Eclipse workspace. Xtext has created a lot of buzz and is used widely. It is very mature and scales beyond trivial languages.

Internal DSLs in Dynamic Languages

To say it up front: I don't think the current crop of dynamic languages and their support for domain specific languages are up to the job. At first glance, languages like Ruby, which supports meta programming and provides a relatively flexible syntax can be used as a host language for modularized internal DSLs. For example, in Rails, you can extend certain base classes and you get additional language constructs[2] that can be used in your own program (e.g. data structure definitions, state-based behavior).

However, there are a number of limitations. First of all, the syntax of your language module can only be what the syntax of the host language supports. Depending on the host language, that might be a quite big space, but it is still limited. Second, the implementation of the language usually happens via meta programming. This is potentially a quite powerful approach, but the implementation of nontrivial language extensions is typically quite scattered and not very maintainable. The biggest disadvantage, however, is that there is no IDE support for the DSL. No

[2] Of course these are not really new language constructs, but rather the clever use of static methods, hash literals and blocks. But they look the part.

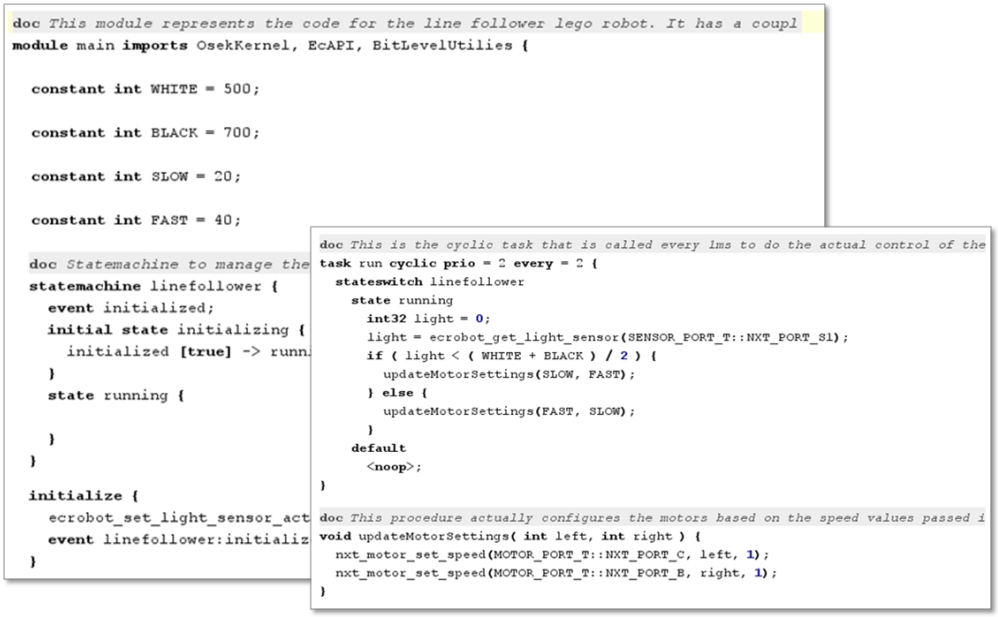
syntax highlighting, no code completion, and no type checking. The approach violates one of the axioms mentioned above: here, language development is *not* IDE development. While that might not be a problem for users of dynamic languages (after all, they usually don't have very good tool support for their host language), I am not willing to accept this limitation.

Another case where people talk about language definition and language extension is Lisp. There is a long tradition in Lisp, to first define a language for a specific problem area and to then use this language to solve the concrete problem (see Guy Steele's *Growing a Language* talk [13]). However, Lisp doesn't have much of a syntax in the first place. Everything is basically lists (written it Lisp's particular parenthesis-rich way) and except for macros, you can't customize this very much. However, the Lispers are eager to tell you how useful this way of programming is. 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 are on to something.

My own Experience with Modular Languages

During 2007, I was involved with a research project called AMPLE [7]. 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 [TODO]. This version of Xtext did not support language modularization, not even the one-language-inheritance described above. I tried to implement the modularization by using a C-style preprocessor to customize the language definitions byswd on a configuration. From a user's perspective the experiment worked and proved the idea: it was possible to select architectural features from a configuration and then a DSL was "customized" that contained support for exactly these features. However, the preprocessor-based implementation didn't scale at all and I had to give up on the experiment.

More recently I have been working on a modular language for embedded software development at mbeddr.com [9, 10]. Based on MPS we have implemented language modules for C, state machines, tasks, sensor access and a DSL for robot control. This attempt is much more successful than my initial attempt based on the old Xtext: because of MPS' great support for language modularization and composition, the implementation is very well structured and easily extensible. The user experience is good as well, since the resulting languages are completely integrated. Next steps will include the support for decision tables and graphical notations for state machines and block diagrams.



During winter of 2008 I was working with Intentional Software, being involved with the implementation of the Pension Workbench for CapGemini [8]. This product integrated text editing (think: Word) 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 these pension calculation rules. The spreadsheet language had been developed before and was reused for the unit tests.

Finally, my current set of projects based on Xtext 1.0 use several languages (referencing, and one-language inheritance). While the language modularization features of Xtext are limited compared to other language workbenches, those that are supported work well and have proven very useful in practice.

Summary

Because of historical reasons there is a distinction between modeling and programming that I think is not helpful (anymore). Rather, I would like to see modular languages, where some modules can be used from libraries, and domain specific modules can be developed per project or domain. The main reason why modular languages haven't been available until now is the lack of suitable tools. This is changing now. I think programmers should start elaborating how modular languages can benefit their work, and which concerns can be provided as a reusable module. The tools are certainly coming around!

Acknowledgements

I would like to thank the participants the workshop on textual DSLs in February 2009 for the as inspiration for this paper. Participants of the

workshop included Axel Uhl, Bernd Kolb, Thomas Goldschmidt, Lennart Kats, Eelco Visser, Konstantin Solomatov and Jos Warmer.

References

[1] Völter, Stahl, Model-Driven Software Development, Wiley, 2006

[2] Martin Fowler, Language Workbenches: The Killer-App for Domain Specific Languages? http://martinfowler.com/articles/languageWorkbench.html

[3] JetBrains, Meta Programming System, http://jetbrains.com/mps

[4] Intentional Software, Intentional Domain Workbench, http://intentsoft.com

[5] TU Delft, http://strategoxt.org/Spoofax

[6] Eclipse.org, http://www.eclipse.org/Xtext/

[7] AMPLE Project, http://ample.holos.pt/

[8] Henk Kolk, Markus Voelter, Democratizing Software Creation, OOP 2008, http://voelter.de/data/presentations/KolkVoelter_IntentionalSoftware.pdf

[9] Modular Embedded DSL, http://mbeddr.com

[10] Markus Voelter, Embedded Software Development with Projectional Language Workbenches, Proceedings of MODELS 2010 and at http://www.voelter.de/data/pub/Voelter-EmbeddedSystemsDevelopmentWithProjectionalLanguageWorkbenches.pdf

[11] Charles Simonyi, The Death of Computer Languages, The Birth of Intentional Programming, http://research.microsoft.com/apps/pubs/default.aspx?id=69540

[12] Lennart C. L. Kats, Eelco Visser, Guido Wachsmuth. Pure and Declarative Syntax Definition: Paradise Lost and Regained. In Proceedings of Onward! 2010. ACM, 2010, http://www.lclnet.nl/publications/pure-and-declarative-syntax-definition.pdf

[13] Guy Steele, Growing a Language, Keynote at OOPSLA 1998, Video at http://video.google.com/videoplay?docid=-8860158196198824415#