Language - Editor – Generator

Bruce Trask

Intent

Raise the level of abstraction and domain specificity and thus programming power by providing new high fidelity domain specific languages, domain specific editors and powerful domain specific transformation engines and generators. Provide the three [8] in the form of domain specific tooling. These enable tackling of increased complexity and change while simultaneously opening the door to a new group of domain expert programmers. Separate the three entities, language, editor and generator while coupling them through the use of more basic and tactical design patterns. Particular emphasis is put on promoting the editor up to be a first class fully featured industrial strength development asset and artifact. Create this domain specific tool using a language workbench [8].

Also Known As

Programming with Models [1], Model Driven Engineering [2], Model Driven Development, Model Integrated Computing [3], Tool, Language Workbench [8], Domain Specific Language

Background and Introduction

The software industry continues to suffer from not so stellar financial statistics regarding completion of software projects on time, under budget and with sufficient quality and functionality [1][5]. The power of any software technology increases to the degree that it maps to domain concepts rather than to computer artifacts and concepts [4][2]. [2] also notes that current programming tools and languages have reached a complexity ceiling and that platform complexity has outpaced the ability of language and tool technology to deal with it. These facts and concepts are covered in extreme detail in [1], [2], [5], [6] and [7]. Recent critical innovations have laid the groundwork and foundation for the industrialization of software [1]. At the heart of this is the creation of Domain Specific Languages and Tools [1][2][7]. The purpose of this pattern is to formerly capture the essence and core anatomy of this approach. Additionally this Language-Editor-Generator approach forms meta-pattern or larger scale architectural pattern into which more lower level patterns can be plugged.

Context

- Systems today are more complex than ever, particularly in the distributed real-time and embedded domain. List of modifiers used to describe these systems include object-oriented, component-based, multithreaded/multiprocess, real-time,
embedded, multi-language, platform independent, high performance, heterogeneous, distributed, vital, secure, dynamic, fault tolerant, portable, and standardized. Each of these words implies complexities by themselves not to mention achieving these goals all at the same time. Add to this fact that solutions to some of these concerns are diametrically opposed to each other (e.g. high performance vs. secure, real-time versus fault tolerant, dynamic versus secure etc).

• Currently the most prevalent tools used by developers to tackle these systems are domain independent modeling and programming languages. Additionally, the most prevalent modeling technologies in use are frequently only used for documentation purposes and thus become artificially divorced from executable development artifacts [1]. This results in vital design artifacts not being machine processable and thus being woefully out of synch with the executable artifacts. These tools try to be everything to everybody by choosing a large horizontal scope that swings the pendulum towards the generality side of the generality/power tradeoff pair. They end up allowing users to express only the lowest common denominator of many many disrelated domains and thus suffers the fate of not being particularly powerful in any one domain. The result is also an increase in the time between when something is modeled and when it is actually meaningfully run or deployed. This is primarily a problem stemming from choosing too great of scope, a problem addressed by domain specific modeling. The value of a model is directly proportional to it ability to be executed and inversely proportional to the time and activitiy that intervenes between modeling and execution (either through direct execution of the model or execution of artifacts generated directly from the model).

• Many domain experts are not actively participating in the “programming” of complex systems due to the insurmountable complexity of the computing platform and inability of 3rd generation language to deal with it. Additionally, the learning curve of the domain independent 3rd generation languages themselves is quite high and barrier to entry for domain experts. The same is true with the learning curve presented by todays frameworks, libraries and component models. Currently the software industry manifests an artificial divorce, dichotomy and linguistic divide between domain experts and developers, between modeling and programming and between the languages used by programmers and the platform for which they are used as tools. These various separations form large gaps in the continuity of software development. This gap has reached critical mass in relation to the complexities of current systems such that a paradigm shift is needed to adjust the software development methods and tools we use to tackle these systems. These separations are accidental complexities not inherent complexities [BROOKES] and as such much can be done to tackle these. One inherent complexity is the “Domain experts don’t know how to encode the abstractions that characterize the domain, but developers don’t know the domain well enough to derive the abstractions”[http://st-www.cs.uiuc.edu/~droberts/evolve.html].
• The widespread use of components, patterns and frameworks has led to the need for much rote framework completion code and descriptors that in many cases are (or should be) decoupled from the business logic of the domain. This completion code and patterns and descriptors should be automatically generated.

• Most software development tools today emphasize the *language* and the *generator* and not so much the *editor* (e.g. C++). Editors are usually text-based with compilers used to check correctness and transform the text into lower levels of more executable code. Particular emphasis is placed on this part of the L-E-G triad in this pattern as this is where the rubber meets the road between a tool and the developer.

• Much code today (around 80%) is not directly related to the problem domain at hand [Jacobsen & 1]. An much of that code is difficult yet rote code (e.g. framework completion code). The result of this is that many developers and domain experts have to swim in the complexity of code that is not directly related to their domain proper.

• Recent work in complex domains has shown a tremendous lack of tooling, particularly in the area of complex systems [Doug MDE book reference].

• “Frameworks and components are currently considered to be the most effective technologies for achieving software reuse. Unfortunately, none of the popular object-oriented (OO) analysis and design methods support their development.”[7]

• A short anecdote to make the context more concrete: The essence of the problem lies in bridging the dichotomy of the domain experts not being able to abstract the concepts and the tool experts not knowing the domain well enough to make the abstractions. It is almost always a two terminal/pole activity.

I have mostly been a consultant. In my walking the halls of (usually large) client companies as a consultant and coaching many different teams working on slightly different but similar projects, the implementation of this pattern usually begins with the Language. Here is what I mean. When talking with enthusiastic domain experts they are usually very quick to get up and grab a dry erase marker and put their ideas on the white board. Frequently, these domain experts (particular in the embedded world where I come from) are not UML experts and so don't usually draw UML diagrams (mind you, even the ones who do know UML don't like to use UML but rather prefer to write in domain specific pictures, images and diagrams instead as they are looking for the most direct expression of their design intent). I begin to see domain experts expressing their intent in common ways on white boards across many projects and also see them struggling to translate or get others to translate their intentions into computer digestable artifacts. So usually, I begin with the language and have a conversation with the domain experts along
the lines of "Would you like to program the systems the same way you are drawing on the white board?". Most times the answer is a resounding "YES" followed by a slight look of incredulity. To this we usually respond, "like this" and we break out our language workbench of choice (in our case eclipse) and whip off a quick metamodel getting input from the domain experts and create some quick graphics that lets them directly express their intent very similar to the images on the whiteboard (I keep reminding them of the whiteboard). I then ask them if there are any computer artifacts that could be generated from this "model". A slight pause and then a torrent of ideas rush forth from the domain experts and their developer's mouths. While I say the Language comes first, it really is the Editor that you have to get quickly into the hands of the Domain Experts as they are chomping at the bit to "express their design intent". The Generator usually falls out of the back of that in a big rush.

The actual making the pattern happen requires folks familiar with what Martin Fowler refers to as Language Workbenches [8]. The interplay between domain expert and language workbench expert is really something to behold; a fast iterative exchange back and forth between both resulting in the birth of Domain Specific *Power* Tool. (the language workbench expert has to be fast enough in relation to the “fire hose" of domain expertise coming his way, this is vital). I will go into this more in the Solution and Implementation sections below.

This is just one, although common, way I have seen this pattern come into being.

**Problem**

Despite the presence of patterns, frameworks, components and middleware, developing complex large scale systems is still very difficult. Patterns can be difficult to evaluate for particular domains, frameworks can be very complex and can make debugging difficult, components provide a unit of assembly and deployment but not many tools exist to perform this assembly and deployment, and middleware has matured but sometimes can be hard to configure properly [21].

Primarily, however, this pattern addresses the problem of the lack of language technology to keep up with platform complexity and technology resulting in the complexity ceiling observed by [2]. This pattern also seeks to walk the thin line of the tension of *generality* versus *power* of software tools. Many of today's software languages and tools have become too general in an effort to be everything to everyone and this has resulted in many of the difficulties the software industry faces today.

The problem section here parallels the context in which the pattern and approach is being presented. Additionally, industrial strength domain specific tooling suffers from the same apparent dichotomy that framework development does: domain experts are frequently not language design experts, editor/graphical user interface experts, nor generative programming experts while expert domain specific modelers and tool developers don’t know the domain well enough to create the correct language, editors and
generators [1]. Ralph Johson expresses this problem in the context of evolving frameworks: “Developing reusable frameworks cannot occur by simply setting down and thinking about the problem domain. No one has the insight to come up with the proper abstractions. Domain experts won’t understand how to codify the abstractions that they have in their heads, and programmers won’t understand the domain well enough to derive the abstractions”. This is a huge problem in software development today.

On the subject of tooling, an accidental complexity [10] currently facing today’s developers is the multitude of tools needed and the lack of integration of these tools. The Language-Editor-Generator pattern and its domain specific approach, when combined with addition patterns like the Plug-In pattern language[17], go a long way in addressing this problem.

“The problem with current software engineering is that we usually end up with a concrete software system, but don’t know how we got there.” [7] The figure below depicts various ways in which software tools and artifacts exist in relation to each other.

Most of today’s software systems are being developed as diagrammed by the top two diagrams. Many gaps exist between the output of one tool and the input of another tool and in fact many of the outputs of some tools are not fully machine processable. Model driven and Software Factories are striving for the third diagram. The jury is still out as to whether it is worth it to provide or even strive for round trip synchronization of software artifacts as various level of abstraction.

**Forces**
Many of the forces to be reconciled with this pattern are again similar to those of framework development. I.e. do the economics of the particular situation warrant the investment of a new language, editor and generator? The answer to this question is very similar to the criteria used to design a software product lines and their production assets[6]. There have to be enough concrete systems that could benefit from the additional work to create a fully featured domain specific modeling tool.

In addition, one has to compare the benefits and expenses of such domain specific approach versus how software is being developed without such an approach in a particular domain and group.

To sum up, the forces to be reconciled are the viability variables observed by [5]. I.e. being able to increase quality and longevity of designs and code without an inordinate increase in cost or production [5].

The forces of the dichotomy between domain experts and software developers

**Applicability**

Use the L-E-G pattern when:

- You want to be able to express your design intent closer to the problem domain. I.e. you want to *model and program* in terms of concepts specific to the domain you are tackling
- You have a family of systems that have many commonalities and variabilities
- There is much rote and duplicated framework completion code and descriptors that are taking up much of the time and energy of your best developers and domain experts [1]
- You are tackling complex systems

**Solution**

Create a formal high fidelity domain specific language (L), a fully featured industrial strength domain specific graphical editor (E) and domain specific transformation engines and generators (G) and integrate them into a Domain Specific Modeling tool. Use existing “Language Workbenches” that make the job of language definition much easier.

**Language**

The anatomy of a language can be thought of as a composition of *abstract syntax*, *concrete syntax(es)* and *semantics*. The anatomy of a model driven development tool can be thought of as being a composition of a language *definition*, editors in which to *capture* designs in this language, and *generators* that can transform these programs into other software artificacts.
Each of the L-E-G form a higher level pattern language which can and should be implemented in terms of more basic tactical design patterns show in the diagram below.

In the L-E-G pattern, the ‘L’ part corresponds to the definition of a new domain specific language. Two terms are used to denote the definition of a language, grammar and metamodel. We will assume the use of a metamodel here. [1] defines metamodeling as “the construction of an object-oriented model of the abstract syntax of a language.”[1] The language “is the metamodel of the domain, itself derived from the feature analysis of the domain, models the key resuable abstractions in the domain “[SF 1].
There are a number of metamodeling languages today that are used by language workbenches including the Meta-Object Facility (MOF), ECORE and MetaGME[OMG][ECLIPSE][GME]. Various metamodeling languages can be used to define new languages (MOF, ECORE, MetaGME). [5] lists various techniques for defining new meta-models either via extending UML by way of a profile, extending UML via MOF, using MOF to define a completely new language. Add to this list the use of meta modeling other than MOF. In fact, recent release of “language workbenches”[8] including EMF, GME and Microsoft's DSL tools provide powerful platforms for language and tool design. In addition there are textual ways to define languages and parsers (boost::spirit, EBNF, lex and yacc) as well as defining embedded domain specific languages within 3rd generation languages them selves using metaprogramming facilities.

Language workbenches supporting these facilities apply the L-E-G pattern themselves and thus recursively provide the ability to provide a tool that makes tools. In other words, they provide editors and generators for defining new languages and tools.

The purpose of the new language is to define the abstract syntax, or grammar, upon which the editor, serialization concrete syntax, and in-memory concrete syntax are based. In this pattern, the definition of this language is the meta-model while instances of this are models, that is, concrete expressions of that language to capture particular concrete designs.

Additionally, many constraints can be designed into the language definition so that concrete instances of the language are correct by construction [2]. This is the “domain model” [DDD] and [POEA] for this software product. This domain model is generative in nature [7] as it will be used as the basis for the Editor and Generator.

The following figure is some figures from the Eclipse Language Workbench known as the Eclipse Modeling Framework (EMF).
This diagram shows a simple domain model that is expressed in EMF’s metamodeling language ECORE. It is from this specification that a whole host of programming artifacts can be generated that thereafter form the basis of the entire L-E-G pattern. In other words we need programming language equivalents to what we see in this figure. These can be generated.
Show equivalent of ECORE, XSD, Java.

Now here is where the language workbench really shines. Having simple java class is not enough to proceed. The java classes that make up this domain model need to go beyond their nominal structure and provide facilities to be, observed, persisted, reflected, and changed dynamically. Language workbenches do this by automatically generating code that implements the Observer pattern [GOF], the Serializer pattern [PLOPDx], Reflection [POSA I] and changed dynamically. Along with the metamodeling, these patterns round out the pattern language that form the generative domain model that serves at the meta model, or the ‘L’ for the L-E-G pattern. With a complete domain model that contains all the necessary elements of a domain and that exists in a language that can be machine processed, is in sync with the other forms of the model (ecore, xsd), AND that can be observed, persisted, reflected and dynamically changed, we are ready to jump to the next element of the main pattern, the Editor. To sum up the definition of the ‘L’ in this pattern is quite precise and is described above. Lack of any of these aspects creates gaps in the software artifact chain and thus make domain specific tooling less viable. If complete, however, the viability of such an approach increases tremendously.

Editor

The purpose of the editor is to render, capture and express design intent [1]. The goal is to achieve this for the developer in as much a domain specific way as possible. In other words, we are trying to avoid think such as “well we *could* UML diagrams, so let’s”. This does not assist the domain experts as they are most times not familiar with UML and UML is a language for object oriented software not for particular domains in industry.

The following figure is an example of what I am talking about.
One could imagine trying to shoehorn this domain into a UML diagram. It would not be pretty, nor very domain specific and most likely would be rejected by the domain experts.

After establishing the domain model and language associated with the domain the idea is to provide and editor that provides the domain expert or user to program directly in terms of their domain, graphically if desired. In this diagram we provide a pallet of known domain object that can be rendered on the canvas on the right.

In reality the graphical editor is “just” a form of concrete syntax that conforms to the abstract syntax defined by the new domain specific language. The other two notable forms of concrete syntax are the serialization syntax used primarily for persistence and for tool information interchange [1] and the in-memory syntax. [1] properly points out that modeling language need not always be graphical and the programming languages need not always be textual [1]. The same applies to the type of specification, declarative and imperative as applied to modeling and programming. [1] even goes on to point out that the differences between programming and modeling may soon fade. The graphical editor can certainly be declarative in nature and thus allow the connected generators to produce pre-programmed pre-validated imperative pattern implementations [5].

To repeat, the editor is the prime tool with which the developer can capture and express design intent. Additionally, proper design of the editor can expose just enough of the variabilities of the domain while hiding the commonalities thus making the editor somewhat like a Façade [18], providing a “simpler interface to a complex subsystem”[18] (i.e. domain) and also making it easy to the right thing, hard to do the wrong thing, while providing some wiggle room to do some unusual things without exposing the unusual things to the every developer [11].

**Generator**

The generators provides automatic ways to get to lower levels of more executable forms of code. The generators provide the actual meaning or semantics to the higher levels of abstractions. This is where the LEG pattern rubber meets the road with regard to the tool and the computer. Being able to traverse the model as rendered in the editor and automatically produce more tactical imperative [2] development artifacts is the key fact that makes the L-E-G and Model Driven Development viable. Lack of this full automatic transformation is the reason the CASE tools of days past and some modeling tools of present have not been as effective as they should have been. Figure 1 is an example of a meta-model, its associated graphical concrete syntax and its serialization syntax as used in the software defined radio domain. Figure 2 is another example of a domain specific modeling tool used for complex distributed real time and embedded systems [3].

Many patterns exist for traversing models, including abstract syntax trees and graphs and generating various “reports” on the backend. Patterns like Visitor[18] and facilities like string templates [JET from eclipse and the String framework from ANTLR].
deal extensively with various forms of transformation in this regards including, horizontal, vertical and oblique mappings. The generators are another key area to leverage the “correct by construction” [2] provided elsewhere by this pattern.

The generators become a classic application of the Strategy[18] pattern. Various degrees of freedom can be generated by traversing the model and passing that information across a fixed interface to concrete generators. 3\textsuperscript{rd} generation language compilers do this by iterating over the platform independent “model” and generate code for various types of processors on the back end.

The generator is all about writing code that writes code. Below is an example of some java code using EMF JET template language, which implements the Template View pattern [POEAA]

```java
public final class <%=enum.getClassName()%> {

    <% for (Iterator i = enum.instances(); i.hasNext(); ) { %>
        <%     Instance instance = (Instance) i.next(); %>

        // instance definition
        public static final <%=enum.getClassName()%> <%=instance.getName()%> =
            new <%=enum.getClassName()%>(<%=instance.constructorValues()%>);
    <% } %>

[http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.html]
Visitor pattern

Samples

The following figures illustrate a couple of examples from two existing language workbenches.
Figure 1 – Eclipse based Language, Editor and Generator
Consequences

The consequences of the L-E-G pattern are all the consequences one might expect from raising the level of abstraction and level of domain specificity and providing a high fidelity tool to deal with them.

- Increased productivity
- Increased correctness
- Seamless and automatic machine processable integration of models with lower perhaps more executable forms of code.

All of the viability variables that [5] observes are touched and improved upon. Quality is increased due to the constraints and grammar of the metamodel, the engineered limited control provided by the editor and the pattern replication and correct framework completion code provided by the generators. Longevity of design is increased because it is no longer coupled to particular implementation technologies that are now hidden on the other side of the generators. The cost of production of concrete systems is dramatically reduced as result of all three aspects of L-E-G. The only calculation needed is the
amortization of the cost of producing the L-E-G domain specific modeling tool over the lifetime of the concrete implementations it produces.

Opening the world to a whole new set of programmers that are domain experts.

A navigation map

Save money, laying the foundation for the industrialization of software (Schmidt quote from Doug off of SF reviews)

The quote from SF about the eventual irrelevance between programming and modeling. As it should be. It just become looking at it from different views of greater or lesser abstraction levels etc. But all still in synch and machine processable. The key is not choosing too great a scope.

Before and After the pattern
Consequences of falling short on each of the above, L – E - G

Related Patterns

MVC, MVP, Command, Observer, Strategy, visitor, mediator. Layers.

Known Uses

The Software Defined Radio Modeling Spectra tools from PrismTech are Eclipse based tools that provide fully featured Language, Editors and Generators for the Software Defined Radio domain. MORE DETAIL HERE ON THE USE

COSMIC is a suite of GME based tools and are used for many complex distributed real-time and embedded systems modeling, editing and generating.

MOBIES is another project uses this pattern

One could consider 3 GL languages (e.g. C++) falling into this pattern with the formal BNF of the language being the abstract syntax, with the concrete programs developer write via the editor and the subsequent compilation of the “model” transforming it into lower level more executable code. Editor not so present,

The language workbenches recursively themselves apply this pattern and provide tools for applying this pattern, that is, a tool for making tools.

Expand the Known Uses section here

[17] Pattern Language of Program Design 5, Manolescu et. al. eds. Addison Wesley 2006
[18] Design Patterns, Gamma et. al. Addision Wesley 1995
[20] The Eclipse Foundation