

Experiences with Domain Specific Languages

An Essay

Sjors E.F. van Berkel
Student number: 1262882

Delft University of Technology, 2011

1 Introduction

In my first year in college, at one of my first lectures, a teacher of mine said: “People are very good at building cars. They know exactly how to engineer them, how to optimize them, and if they are broken, fix them. . .”. As he continued, he said something that would stay with me for a long time, and I am pretty sure it will never again leave my head: “We have, however, not yet reached this level of expertise in building computer programs. . . In fact, the odds of there being a single programming paradigm that suits a majority of computer problems, are very small.”

In this essay I will layout what my experiences with Domain Specific Languages (DSLs) are, and how I think they are better suited for a large range of problems and people than the current generation of General Purpose Languages (GPLs).

I will do this in two separate ways. In the first part of the essay, I will use my experiences in using different website frameworks to show how many of the structural problems with these frameworks would be solved if these frameworks were built around a DSL like WebDSL.

In the second part, I will reflect on my experiences with building DSLs, and highlight the benefits, as well as the problems that arise when building such a DSL. I will use three cases I have encountered over the years, finishing with deLightDSL, our custom built DSL for programming interactions.

2 Why use DSLs?

As many of my colleagues in college, I have spent a considerable amount of time in website development. Because of that, I would like to take the opportunity to reflect upon my experience in website development with regard to DSLs, where they were applied, or how they would have helped development when they were (partially) absent. As a comparison for the other systems or frameworks, I will use WebDSL[1] as an example of how DSLs can help development of websites.

2.1 Other frameworks / systems

In the following section I will introduce three types of systems I have worked with in the past, and specify the aspects that could be improved by using a domain specific approach.

Green Valley CMS / Smartsite Green Valley CMS (GVCMS) is a system implemented, used, and sold by a company called Green Valley¹ in Delft. Smartsite² is a system implemented by the company Seneca³ that resides in Delft. It is both used to serve some of their own clients, but it is mostly licensed to other companies who do implementations for their clients. Both GVCMS and Smartsite are typical “old fashioned” content management systems. They are built to provide a way for content publishers to put their articles or other content into a backend system, and this content is then presented in a predefined front-end which is almost never adjustable by the publisher. Both systems provide a way to define content types, but the developer is concerned with mapping these types to a database table. Furthermore, both systems use nested templates to render the content and markup at the front-end. Checking the integrity of the front-end and content is the concern of the developer, and whenever he or she fails to deliver a correct implementation, a page, or even sometimes a whole website becomes unresponsive.

Both systems use a proprietary language for writing templates. GVCMS uses an XML based template framework called the Open Template Framework, for which the only possible editor services have to come from XML schema definitions, and this of course depends on the support of your favorite editor. In practice the templates are very verbose, and lack the contextual information to substantially support the developer in his or her work coding.

Smartsite uses a mix of XML based and short (`{` and `}` delimited) function calls. It stores every template in the database, and editing is done inside the browser (Internet Explorer only) via a plugin. The plugin provides some autocompletion functions and consistency checking, but only provides this functionality for the default functionality, and hence it does not help with self defined content types or variables and such.

Django <http://www.djangoproject.com/> is a Model-view-controller paradigm framework for building websites, built in Python. It provides the ability to define models that are automatically mapped to database structures using a built-in object relational mapper. Because Django is specifically built for the web, its developers constantly try to uncover what web developers want to do most, and simplify these things by providing default, easily modifiable implementations. In [2], Django is described as an internal DSL, using reflection and runtime code generation to provide an API that feels like a DSL.

¹ Green Valley: <http://www.greenvalley.nl/>

² Smartsite: <http://www.smartsite.nl>

³ Seneca: <http://www.seneca.nl/>

Because Django is built in Python, it inherits its untyped nature. Because of this, editorial services are already limited at the core, and in practice, the result is that most editors provide syntax coloring, but very little or bad autocompletion functions, no references, and no consistency checking. Although Django provides the ability to change the template framework used, I will consider the default template framework here. The templates depend on the content provided by the so-called views, that provide the template with a dictionary of objects and simple typed variables to use in the view. This content provided by the view can then be rendered by the template using simple tags that specify how the content is rendered (e.g. using loops and decorators etc.). However, the relations between views and templates are never checked, and will result in runtime failures if not correct.

.NET Framework The .NET framework is Microsoft's answer to web development, and in typical Microsoft fashion ("We view a tablet as a PC"⁴), it is a minimal deviation from normal programming practices used by Microsoft. The most commonly used structures are ASP.NET pages, and user controls, which are reusable components that can be embedded on pages. Although there are many different ways to link code to the HTML, the most common approach is to have one so called Code-behind file per page or user control. This Code-behind file (written in C# or another Microsoft supported language) contains the logic for adjusting the generic HTML of that page or user control to the specific situation (e.g. content or user status etc.).

The advantage of the .NET framework is that it is tightly coupled with Microsoft Visual Studio, which provides excellent editor services, including autocompletion, type checking, code outline, syntax coloring, hovering information et cetera. It also provides excellent consistency checking, so that at compile time, everything that can be checked is actually checked.

The main disadvantage of the .NET framework is its minimal deviation from general purpose practices, leaving a very big programming task to the programmer. For example, the way the URL structure of a site is coupled to the data structure of the site, is totally up to the programmer. Another example is that the implementation of object persistence, or other database tasks are left totally to the developer, (letting them use a very generic framework SQL framework Linq for example,) while these are a very common requirement for web applications.

2.2 WebDSL, a DSL approach to website development

Next to working with the previously mentioned systems, I have also done two projects in WebDSL[1], a DSL for building web applications. In this section, I will try to point out the differences between WebDSL and the previously mentioned systems.

⁴ <http://www.zdnet.com/blog/mobile-news/-8216we-view-a-tablet-as-a-pc-microsoft-chooses-to-repeat-h-3267>

I think the keyword that best describes the advantage of WebDSL over other systems is integration. Because web development requires multiple techniques (i.e. HTML, CSS, Javascript, models, database content et cetera) to be integrated, acquiring and maintaining consistency is usually an onerous task. As mentioned in [3], it is possible to map one problem space, in this case web application development, to multiple solution spaces, the multiple techniques. Although the current version of WebDSL does not concern CSS markup (you can only import it), the other techniques are tightly integrated into one, problem defining language.

Because of the problem domain being captured in an external DSL[2], WebDSL attains advantages over a GPL as mentioned in [3]: abstractions (e.g. pages, page parts), concrete syntax (e.g. automatic Javascript, HTML and JSF creation from a tidy self defined input language), error checking, tool support, and optimizations (although I am not in possession of any examples while writing).

Access Control Apart from these “explicit” advantages due to WebDSL being a DSL, there are also a lot of “implicit” advantages to WebDSL. In [4], WebDSL is extended with an access control system to restrict or grant users access to certain parts of a web application. Because the access control is expressed in the same language as the web application, it can integrate with the web application content, and therefore the access restriction decisions can depend on content. Leveraging this, it is hard to build an application where content nevertheless “slips through the cracks”, even though a user is not allowed to see it. In many web application frameworks or systems, access control is abstract, and very often, it takes a lot of effort to perfectly configure it perfectly per user or group, if it is possible at all.

In comparison with the other systems mentioned, the WebDSL approach is really a luxury for web application developers. In GVCMS and Smartsite, the user rights are configured per object instance, giving a website editor or moderator the cumbersome task to make sure everyone sees exactly what they are entitled to, while it may for example already be very obvious in terms of company structure that someone should or should not be able to see something. In Django, arbitrary permissions and user groups are built in, so these can be checked throughout the code, but the remainder of the implementation is up to the developer. This is more likely to be a burden than an asset to the developer. In .NET, developers are again free to completely provide their own implementation for access control, while it should be mentioned the connection to Windows user management is easily setup, but this barely gives more information than a username and group.

Workflow In [5], WebDSL is extended with WebWorkFlow, a workflow system. WebWorkFlow enables developers to define procedures that encompass multiple steps that have to be fulfilled by multiple actors. In the same fashion as the access control system from [4], the workflow is tightly integrated into WebDSL, providing dedicated workflow constructs through the existing language. Because

of the tight integration with WebDSL, the workflow system can access all aspects of WebDSL and other WebDSL components can access the workflow entities, which can come in handy for access control, or showing the status of a process on a webpage, for example.

In comparison, the only workflow available in GVCMS is the publication cycle of articles written by editors. In Smartsite, there is a Workflow Definition object type. I personally have never used this, which is not a disqualification of some kind, but the fact that it uses a language (XPDL, a standardized workflow format) different from the default Smartsite implementation language is negative with regard to the integration possibilities with other parts of Smartsite. Django has multiple plugins to realize workflow, but the two prominent plugins, GoFlow⁵ and django-workflows⁶, have not been updated for at least a year. This is an obvious advantage of integrating such a feature into a DSL, because compiling and unit-testing the DSL will likely already point out that some aspects of the language have to be updated for the language to be consistent. For the .NET framework, Windows Workflow Engine⁷, a very generic workflow engine is available. While I think it is positive that Microsoft chose to include it into the .NET framework, as opposed to the Django plugins, it is only very loosely coupled to web development. This leaves the incorporation of workflow into a website to the developer.

3 Building DSLs

The last section was about reflecting upon the difference between DSLs and other types of languages / systems. This section will highlight my own experiences in using DSL techniques to empower developers.

3.1 The retry statement

In the process of writing a piece of code, it suddenly appeared to me that a common programming pattern is always left to be manually coded by the developer: many languages have `try` and `catch` blocks, where the `catch` blocks are often used to fix an anticipated situation that occurred inside the `try` block. However, when this situation is fixed, there is no straight forward manner to return to the `try` block. Hence I wrote the following tweet:

“Just had an idea for pr. languages in general, a retry function in a catch block that restarts execution of the corresponding try block” –sjors1986

The only response I received was not quite what I hoped, but it was expected:

```
@sjors1986 bool didwork = false; for(int rc = 0; rc<3 && !didwork; rc++){try{...didwork = true;}catch(Exception ex){...}}?" –roelspruit
```

⁵ GoFlow: <https://code.djangoproject.com/wiki/GoFlow>

⁶ django-workflows: <http://pypi.python.org/pypi/django-workflows>

⁷ Windows Workflow Engine: <http://msdn.microsoft.com/en-us/netframework/aa663328>

Although it was not the intention of the author, in my opinion his tweet captures why a dedicated language construct would be a great idea, to remove the arbitrary nature of such structures from a program.

There are two good ways to good ways to achieve such a language construct to be integrated in a language. Ideally it should be integrated into the real language, which is not that strange because it is an asset for all programmers, not just for people in a very specific domain. Apparently this was wishful thinking, because Oracle, the current owner of the Java language, redirected me to a Java community site to submit my idea. It would take years, not only because of the community, but Java is generally slow with adopting new constructs.

My other attempt would be to get it integrated into Python, which is generally faster at adopting new ideas. To achieve this, I decided to send Guido van Rossum, the Google employed maintainer and founder of the Python language, a message with my idea. He decided not to reply.

Although Java (partially) and Python are both open source projects, overseeing the whole source code is very hard, and adjusting it is even harder. A better idea would be to apply a technique leveraged in [6]. In that paper, MetaBorg is used to embed Domain Specific constructs into the Java language, extending the language with (among other things) more intuitive user interface constructs (called JavaSwul). Although the retry statement is not really Domain Specific, the same methods could be applied here. MetaBorg uses a language definition for Java 1.5, which it extends with custom constructs. This is an important step in the process because without the language definition, MetaBorg would be able to translate the custom piece of code into Java-code, but it would never be able to check whether the program would compile on a Java compiler. Since compiler errors in generated code are generally hard to trace back to the original code, this is unwanted behavior.

For the retry statement, we would also have to have a language definition for the target language to overcome this. This is also an obvious disadvantage of this technique because it comes down to proving the equality of two language definitions, which is difficult to do. Another disadvantage is that the developer of the extension always has to manually upgrade the code to the new version of the language. Most of the time this results in limited support of the extension for different versions of the language, or support for a limited time. This is alright if the extensions are written in a limited setting, for a single project for example. In this case however, it would be best if the idea is maintained by the developers of the language, because it would be an integral part of the language.

3.2 A simple query language

For the course IN4325 Information Retrieval, a custom search engine had to be built on top of Lucene⁸, a generic search engine. Because the assignment was to be executed by groups of 6 people, we as a team decided to split the project up into separate parts. Because I was supposed to take care of the queries for

⁸ Lucene: <http://lucene.apache.org/java/docs/index.html>

the system, and the default Lucene queries were already good enough for the assignment, I decided to challenge myself in replacing the default query language with a custom query language.

Because Lucene is built in Java, a query is ultimately represented to Lucene by a query object that can contain multiple query objects and so on. To parse a text query into such an object, I decided to use ANTLR⁹, because it is a very widely used parser generator, and it comes with some tools for analyzing syntax definitions, but the most important factor was that it could output Java code.

Even though I had some experience with Bison¹⁰ and Flex¹¹ from a previous course, defining a query language that is not ambiguous and has no left recursion was still quite difficult, but eventually I succeeded to create a syntax definition that was accepted by ANTLR (and was useful, too!). Because of the apparent simplicity of text queries in a search engine, I think it is more difficult to create such a language, because users just want to type words with maybe some syntactic indicators in between them. For programming languages for example, the structure is more important and constructs are more verbose, leading to less ambiguity.

In an ideal scenario, we would first like to create an abstract syntax tree (AST) from parsed input. An AST can come in handy for representing the structure of a program (or query in this case), and optimizing parts of the program before outputting code. However, because of the limited amount of time given for that part of the assignment, and a feeling we did not necessarily need an AST, I decided to generate Java code for the query on the fly. This worked because the precedence of parsing the query turned out to be equal to the hierarchy in the query, so for example, if a part of a query was grouped between brackets, that part would first be completely parsed before moving on. During the execution of the parsing, I used a stack to keep intermediate query objects, the first element on the stack being the root of the tree. Every time when a non-terminal was encountered on the way back, 1 or 2 elements were pop'ed from the stack, and added to a query object representing that non-terminal. In the end, all the queries would collapse into one query object (the root), and this would be returned as the query object representing the text query.

The assignment prescribed that we should be able to use single-word-queries, multiple-word-queries, phrase-queries, and wildcard-queries, all of which were implemented in my language. The language got even better when we wanted to restrict the fields that were searched by a certain query part, something that I could adjust in the implementation generated by the language. This is again a typical advantage of using a DSL, because other groups doing the assignment, that did not build a DSL, were limited to the default Lucene querying behavior, already justifying its build.

⁹ ANTLR: <http://www.antlr.org/>

¹⁰ Bison: <http://www.gnu.org/s/bison/>

¹¹ Flex: <http://flex.sourceforge.net/>

3.3 deLightDSL: An interaction language

deLight is a small startup company in Delft building applications that use interactive elements to let the user control the program. The programs created vary per customer, but common aspects are presenting objects to a camera to influence the program, and presenting the user input by projecting the influenced program back to a screen, typical virtual reality and augmented reality applications. A typical example that was presented to us was the projection of the inside of a 3D building, which was empty. By placing miniature models of interior elements, such as a chair and a closet, on a opal glass plate (the camera underneath recognizes the markers on the back of the objects), the interior elements appeared in the projection, and hence you could use this to design an interior.

Identifying useful language elements Even though at the beginning we were not sure what the deLight programs would exactly look like, we identified that of all modules composing an application (appearance, interface, interaction, and media), interaction would be an interesting module to work on. There are a few reasons for this. The first reason is that these applications really revolve around interaction, the other modules depend on it, and hence we found it would be the most interesting module to optimize. The second (somewhat abstract) reason is that, while most programming constructs operate in a linear fashion, interaction is action-reaction based, and should be executed at all time during the program. In practice, the code consists of adding and removing listeners objects to and from the drawing cycle of the program, and this can happen throughout the entire code, and for the structure of the program, it makes sense to separate it from the program flow. The third, and most important reason, is that, while interaction may be explained very simply in human language, this does not mean it translates into easy code. Because the interactions are not necessarily, and most likely, not designed by programmers but by designers of the program, it would be wise to try to enable designers to write their own code.

The paper [7] describes different types of DSLs. Our DSL is contained under different patterns mentioned, Notation, Interaction, and Task Automation. The pattern Notation is described as “Add new or existing domain notation”, and “Add user-friendly notation to existing API”, both are the case. The pattern Interaction is described as “Make interaction programmable”, which is the main purpose of our language. The pattern Task automation, described as “Eliminate repetitive” is also applicable because the language eliminates boiler plate code to add reactions to actions (by automating adding events to the draw cycle).

Once we decided we would focus on interaction, we tried to identify the core of interaction code (this was without having seen deLight code) using informal analysis[7]. It turns out Newton already described the core of everything we could come up with, starting with:

```
when screen-touch boxA display graphB and play soundC
when gesture circleA move eye to pointB with speed number
```

So basically, fire (re)actions when certain conditions are true, possibly due to another reaction having fired previously. This would be the premise of our language, but we would later extend it to become more useful.

Because our language is designed to be used separately, by people who are likely to be new to programming, we chose to invent a new language instead of exploiting an existing language[7], so that we were free to come up with a syntax best suited to the problem domain, and compile this to C# later on. Also because our language would only fulfill a part of the total program, the generated code would not be executable on itself, but it does have well-defined execution semantics[7] (an initialization block, and different methods that can be called).

Building the language Once we had an idea of what we wanted to contain in our language, we went to deLight to continue on the process of informal analysis. For the already existing programs, the code for interaction was somewhat scattered through the code, making it difficult to clearly see which of the parts were to be replaced by our language. In consultation with René Elstgeest, CTO of deLight, we decided to create a simple application to replace camera/objects input by a more simple mouse driven input to drag and drop 2D objects on a plane, to trigger reactions based on the distance between two objects and the absolute position of objects.

By implementing a game, in which a product-line assembly of multiple products over multiple tables was modeled, we were able to gradually figure out steps we could automate in our language using elegant language constructs. One thing we found out this way was that it is sometimes very undesirable to declare behavior per object because sometimes it is the same for a whole class of objects, so we designed a language construct for that. Another thing we noticed was that some actions should be disabled by other actions, and hence we had to label the actions such that we could enable and disable them at will. We also found out that it would be useful to be able to call program functions in our language that were not part of our part of the program (e.g. a function to exit the program), so we decided we should provide functionality for a library file in which the programmer can provide a definition of such a function, such that we could use this in code completion, type checking et cetera.

To build our language, we used the Spoofox[8] language workbench. Using this, we were able to create an Eclipse plugin with syntax coloring, type checking, code completion, code folding, and references. Because the possibilities of the code outline were heavily dependent on the syntax definition, we were unable to implement this in a later stage without heavily revising our entire code, so for future projects we should take that into consideration when defining the syntax. For now we disabled the code outline because it yielded strange and almost useless results in its default behavior.

During the build process of the language, me and my lab partner André were surprised by how arbitrary some aspects of language building seemed to be. For example, type checking always took a considerable amount of time to get

right for every language construction, and there are a lot of error checking rules that look very similar but are slightly different. While this demands creativity of the programmer, which I think is generally a good case, it is questionable whether this is desirable when building DSLs for a professional purpose, because it diminishes the predictability of the time required for the project to be finished. Of course this is partly due to us expecting all these editor services, but it also raises questions about whether more experience in language building yields more consequent languages in which less checks are needed to assure compilable code. I personally think that this “problem” is partly avoidable for people who have more experience in the field, and if so, that could also be leveraged by publishing “language templates”, for building consistent languages. It is also in line with [7], which mentions that language invention can be substantially more difficult than exploiting an existing language definition.

4 Conclusion

In this essay I have essentially shown two things. Firstly I have shown that a methodology such as WebDSL really benefits from being built as a DSL in many ways, by showing how other, competing frameworks are always one step behind because of their fundamental internal incoherency. Secondly, I have reflected on my experiences building DSLs to show that although DSLs are an interesting option for a multitude of applications and people, whether it is suitable for a project is, I think, very dependent on the time (and money) available, because building a good DSL takes time, effort, and cooperation.

Concluding, I think DSLs are an answer to what many people have not yet considered a problem: They can connect the right people to a problem domain, and let them write code that is not broken to begin with.

References

1. E. Visser, “WebDSL: A case study in domain-specific language engineering,” in *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007* (R. Lmmel, J. Visser, and J. Saraiva, eds.), vol. 5235 of *Lecture Notes in Computer Science*, (Braga, Portugal), pp. 291–373, Springer, 2007.
2. D. M. Groenewegen, Z. Hemel, and E. Visser, “Separation of concerns and linguistic integration in WebDSL,” *IEEE Software*, vol. 27, September/October 2010.
3. K. Czarnecki, “Overview of generative software development,” in *Unconventional Programming Paradigms, International Workshop UPP 2004, Le Mont Saint Michel, France, September 15-17, 2004, Revised Selected and Invited Papers* (J.-P. Bantre, P. Fradet, J.-L. Giavitto, and O. Michel, eds.), vol. 3566 of *Lecture Notes in Computer Science*, pp. 326–341, Springer, 2004.
4. D. M. Groenewegen and E. Visser, “Declarative access control for WebDSL: Combining language integration and separation of concerns,” in *Proceedings of the Eighth International Conference on Web Engineering, ICWE 2008, 14-18 July 2008, Yorktown Heights, New York, USA* (D. Schwabe, F. Curbera, and P. Dantzig, eds.), pp. 175–188, IEEE, 2008.

5. Z. Hemel, R. Verhaaf, and E. Visser, “WebWorkFlow: An object-oriented workflow modeling language for web applications,” in *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings* (K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Vltter, eds.), vol. 5301 of *Lecture Notes in Computer Science*, pp. 113–127, Springer, 2008.
6. M. Bravenboer, R. de Groot, and E. Visser, “MetaBorg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT,” in *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers* (R. Lmmel, J. Saraiva, and J. Visser, eds.), vol. 4143 of *Lecture Notes in Computer Science*, pp. 297–311, Springer, 2006.
7. M. Mernik, J. Heering, and A. M. Sloane, “When and how to develop domain-specific languages,” *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.
8. L. C. L. Kats and E. Visser, “The Spoofox language workbench: rules for declarative specification of languages and IDEs,” in *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010* (W. R. Cook, S. Clarke, and M. C. Rinard, eds.), (Reno/Tahoe, Nevada), pp. 444–463, ACM, 2010.