The complete guide to (external) Domain Specific Languages

Written by Federico Tomassetti
in
Domain specific languages

Table of contents

This guide will show you:

- the **what**: after a definition we will look into 19 examples of DSLs
- the **why**: what are the concrete benefits you can achieve using DSLs
- the **how**: we will discuss the different ways to build a DSLs and what are the success factors

After that you will get a list of resources to learn even more: books, websites, papers, screencasts.
This is the most complete resource on Domain Specific Languages out there. I hope you will enjoy it!

Just one thing: here we try to be practical and understandable. If you are looking for formal definitions and theoretical discussions this is not the right place.

Send it to me!

What are Domain Specific Languages?

Domain Specific Languages are languages created to support a particular set of tasks, as they are performed in a specific domain.

You could be familiar with the typical programming languages (a.k.a. General Programming Languages or GPLs). They are tools good enough to create all sort of programs, but not really specific to anything. They are like hammers: good enough for many tasks, if you have the patience and ability to adapt them, but in most cases you would be better off using a more specific tool. You can open a beer with an hammer, it is just way more difficult, risky and lead to poorer results that using a specific tool like a bottle opener.

Ok, but what does it mean in practice? How do DSLs look like? Let's see plenty of examples.

19 Examples of Domain Specific Languages

Domain Specific Languages can serve all sort of purposes. They can be used in different contexts and by different kinds of users. Some DSLs are intended to be used by programmers, and therefore are more technical, while others are intended to be used by someone who is not a programmer and therefore they use less geeky concepts and syntax.

Domain Specific Languages can be extremely specific and being created only to be used within a company. I have built several of this kind of DSLs myself, but I am not allowed to share them. I can instead list several examples of public DSLs which are used by millions of persons.

1. DOT – A DSL to define graphs

```
digraph graphname {
yellow -> orange -> red;
orange -> green;
}
```

From this description images representing these graphs can be generated. To do that you use a program named `graphviz`, which works with the DOT language. From the previous example you would get this:

An image generated using
the dot DSL

The language permits also to define the shape of the nodes, their colors and many other characteristics. But the basics are pretty simple and almost everyone can learn how to use it in a matter of minutes.

PlantUML Can be used to define UML diagrams of different kinds. For example we can define a sequence diagram.

```
@startuml
actor MyUser
actor CustomerCare
database database
MyUser -> CustomerCare : Ask a refund
CustomerCare -> database : Verify the
data
CustomerCare -> MyUser : Issue a refund
@enduml
```

From this definition we can get a picture.

Image generated with the PlantUML DSL

With a similar syntax different kinds of diagrams can be

this one could be used during a meeting to support a discussion: as the participants argue different diagrams can be quickly defined and the corresponding images generated with a click.

3. Sed – A DSL to define text transformation

On UNIX-like operating systems (Linux, Mac, BSD, etc.) there are a set of command line tools, each one accepting instructions in their own format. This format can be considered a DSL that permits to specify the tasks to be executed. For example *sed* executes the text transformations indicated using its own DSL.

Do you want to replace the word “Jack” with the word “John”?

```
1.  s/Jack/John/g
```

Or do you want to delete all the lines of a file from line 10 until the word “stophere” is found?

```
1.  10,/stophere/d
```

computer users could learn the basic to execute common operation on files. Small things that they could need to do every day and that currently are doing manually. For example, you could have e-mail templates containing placeholders like “{FIRST_NAME}” and replace them with the proper test with one of this commands.

4. Gawk – A DSL to print and process text

Like sed, gawk is another UNIX utility accepting commands in its own language. For example you could print all the lines of a given file which are longer than 80 characters:

```
1.   length($0) > 80
```

Or count the lines in a file:

```
1.   END { print NR }
```

The UNIX philosophy is to use several or these little

5. Gherkin – A DSL to define functional tests

Gherkin is a DSL for defining functional tests. It has a very flexible syntax that makes it look almost like free text. Basically developers, analysts and clients can sit around a table and define some scenarios. These scenarios will be then executable as tests, to verify whether the application meet the expectations.
Here it is how we could define the expectations for withdrawing from an ATM:

```
1.  Scenario: Verify withdraw at the ATM
    works correctly
2.  Given John has 500$ on his account
3.  When John ask to withdraw 200$
4.  And John inserts the correct PIN
5.  Then 200$ are dispensed by the ATM
6.  And John has 300$ on his account
```

I really like this DSL because the bar for using it is very low. This DSL however requires a developer to define some code using a GPL. How it works in practice is that a developer define specific commands like: “{name} has {amount}$ on his account” and define the code that execute this command in the GPL chosen for the project (Ruby, Java, or others are supported). Once the

as you want, trying to capture the requirements and only later developers map each command to a corresponding function in a GPL.

In other words, this DSL is great for hiding the real code behind a surface that everyone can understand and everyone can contribute to. It is much better to sit at a table and discuss with a bank representative using the example we have displayed than showing him the hundreds of lines of Java which correspond to those commands, right?

6. Website-spec – A DSL for functional web testing

Gherkin is not the only DSL used to define tests. Website-spec can be used to define functional tests specific for web applications.
Here we define how to navigate on a certain website and what we expect to find.

```
1.  Open $url
2.  Clock on create
3.  # Select a store
4.  Within card-panel-store
5.  Select `[date-test=stores] label`
6.  Remember test as $StoreName
```

In this case there is no need for a developer to define the translation of commands to a GPL because this language users domain specific commands, like “Click”, which the interpreter knows how to execute. With a minimal training now anyone can describe a sispecific interaction with a website and the expected results. Pretty neat, eh?

7. SQL – databases

You have probably heard of SQL. It is a language used to define how to insert, modify or extract data from a relational database. Let’s get some stats from the *STATS* table:

```
1.  SELECT MAX(TEMP_F), MIN(TEMP_F),
    AVG(RAIN_I), ID
2.  FROM STATS
3.  GROUP BY ID;
```

For sure you do not expect the average Joe to be able to write complex queries: SQL is not a trivial language and it requires some time to be mastered. However you do not need to be trained as a developer to learn SQL.

to ask someone and wait to get an answer. Suppose he needs to know the maximum temperature in august in Atlanta:

```
1.   SELECT MAX(value) FROM TEMPERATURES
    WHERE city="Atlanta" AND month="August";
```

Maybe Joe will never reach the level of a DBA, but he can learn a few basic queries and adapt them to his needs, making him more independent and letting his colleagues focus on their job instead of helping him out.

8. HTML – web layout

I really hope you have heard of this quite successful language to define documents. It is amazing to think that we could have defined HTML pages 20 years ago, when most people had desktop computers attached to monitors with a resolution of 640×480 pixels and now those some pages can be rendered on the browser running on our smartphones. I guess it is a good example of what can be achieved with DSLs.

```
1.   <html>
```

```
<div>consid
</body>
</html>
```

Note that HTML is really about defining documents: their structure and the information they contain. The same document then it is rendered differently on a desktop computer, a tablet or a smartphone. The same document is consumed differently from people with disabilities. Specific browser for people with impaired sight help them consume a document defined with HTML by reading the content and support navigation to the different sections of the document.

9. CSS – style

The *Cascading Style Sheet* language defines the style to use to visualize a document. We can use it to define how an HTML document will appear on the screen or how it will appear when printed.

```
p.center {
text-align: center;
color: red;
}
@page :left {
margin: 0.5cm;
```

CSS is a not trivial to master but many persons, with basic or no knowledge of programming, can use it to change the appearance of a web page. This DSL has played an important role in democratizing web design.

10. XML – data encoding

Some years ago XML used to seem the solution to all problems in IT. Now the hype is long gone, but XML is here to stay. A solid DSL to represent data, and a quite flexible one.

```
1.  <library>
2.  <author firstName="John" lastName="Doe"
    id="JDOE" />
3.  <book title="The Story of Mr. Doe"
    author="JDOE" />
4.  </library>
```

While it is not the most readable or impressive language everyone is able to modify the data contained in an XML file.

11. UML – visual modeling

discussions. Someone also uses them to generate code or even to define an entire application (look for Model Driven Architecture, if you are interested in this kind of stuff).

UML: an example of DSL

UML is a vast language (someone said bloated?). There are many different kinds of different diagrams comprised after the UML umbrella. All of them share some commonalities.

Not everyone would agree that UML is a DSL. While it is definitely a language, someone would say it is not domain specific, but generic instead. Domain specificities can be added by means of UML profiles. I consider it instead a language specific to *modeling*.

Now, this is one case that demonstrates which there are not hard and easy rule to define what is a DSL and what

12. VHDL – hardware design

VHDL is a DSL used to define circuits. Once upon the time electronic engineers used to design complex systems directly deciding which gates to use and how to wire them together. VHDL changed all of this, providing higher level concepts that those engineers can used to define their systems.

```
1.  DFF : process(RST, CLK) is
2.  begin
3.  if RST = '1' then
4.  Q <= '0';
5.  elsif rising_edge(CLK) then
6.  Q <= D;
7.  end if;
8.  end process DFF;
```

Example taken from Wikipedia.

There are tools able to process these definitions to derive actual circuit layouts, ready to be printed. Verilog is another DSL similar to VHDL.

13. ANTLR – lexer and parser definitions

the structure of a piece of text.

For example this is a small snippet of a lexer grammar:

```
// Identifiers
ID : [_]*[a-z][A-Za-z0-9_]* ;
// Literals
INTLIT : '0'|[1-9][0-9]* ;
DECLIT : '0'|[1-9][0-9]* '.' [0-9]+ ;
STRINGLIT : '"' ~["]* '"' ;
```

JavaCC, Lex, Yacc and Bison are similar tools and all come with their slightly different DSL, inspired by the Backus-Naur form.

14. Make – build system

Make is a language to describe how to build something and the dependencies between different steps. For example you can define how to generate an executable and specifying that to do that you will first need 3 object files. Then you can define for each of those object files how to obtain it from a corresponding source file.

```
CC=gcc
CFLAGS=-I.
```

In this example we specify that to create the program *myExecutable* we will need the object files, and once we have them, we will use gcc to link them together.
We can also define some constants at the top of the file, so it is easy to change the Makefile later, if we need it.

15. Latex – document layout

Latex is used a lot, in the academy and in the publishing industry, to produce gorgeous looking papers and books.

```
documentclass[12pt]{article}
usepackage{lingmacros}
usepackage{tree-dvips}
begin{document}
section{Introduction}
Here it start my introduction
subsection{Details}
Here I go in more details.
end{document}
```

Once you have described a document in this format you typically generate a PDF

you are proficient with LaTeX you can get pretty nice results.

16. OCL – model constraints

OCL stands for Object Constraint Language and it can be used to define additional constraints on objects. Typically it is used together with UML.
For example, if you have a class *Appointment* with two properties *start* and *end* you may want to specify that the *end* of the appointment follow its *start*:

```
1.  context Meeting inv: self.end >
    self.start
```

You can also define preconditions and postconditions for your operations or invariants that apply to classes. If you are interested in this sort of stuff you may also want to look into QVT, a set of languages to define model transformations.

17. XPath – XML nodes

representing a list of restaurants and you want to get the last restaurant:

```
/restaurants/restaurant[last()]
```

XPath expressions are used into XSLT to define which elements to transform or they can be used in combination with many libraries for all sort of languages to define which elements to extract from a document. If you wonder what XSLT is, it is a language to define transformations of XML documents.

18. BPEL – Business processes

BPEL is a language to define the collaboration between web services to implement business processes. It used to be more popular when the world was going through its *Service Oriented Architecture* (SOA) phase.

The goal of this language is to permit to software architects, or even to analysts, to combine different web-services, or other components, to obtain complex systems.

Example from Eclipse BPEL
(https://eclipse.org/bpel/)

There are different implementation of the language, each one with extensions and mostly incompatible one with the other.

19. Actulus Modeling Language – A DSL to calculate life insurance and pensions

```
1.  riskmodel RiskLifeDeath(p : Person) :
    LifeDeath(p) where
2.  intensities =
3.  alive -> dead by
    gompertzMakehamDeath(p)
```

Financial Domain-Specific Language Listing where you can find many more similar examples.

So what can we use DSLs for?

After looking at these examples we can derive that DSLs can be used for a variety of goals:

- define commands to be executed: like sed or gawk
- describe documents or some of their specific aspects: like html, latex or CSS
- define rules or processes: like BPMN or Actulus

These are just some typical usages, but DSLs can be used for so many other reasons.

It is also important to notice that DSLs focus on one specific aspect of a system and often it makes sense to combine several of them to describe the different facets of a product. For example HTML and CSS are used together to describe the content and the style or a document, XML and XPath are used to define data and how to traverse that data. OCL is used to define

Ok, at this point you should have some understanding of how a DSL can look like and what it can be used for. Now let's see *why* you should use one and then how to build one.

DSLs vs GPLS: 5 Advantages of using Domain Specific Languages

You could be asking yourself the following question:

> *Why using a specific, limited language instead of a generic, powerful one?*

The short answer is that Domain Specific Languages are limited in the things they can do, but because of their specialization they can do much more in their own limited domain.

program written in C or in Java will respect certain characteristics (like not ending in an infinite loop) we can perform all sort of analyses when we use DSLs. Precisely because they are limited in what they can do they are easier to analyze.

2. They are more safe. Less things can possibly go wrong when using a DSL. When is the last time you had a Null Pointer Exception when working with HTML or SQL? Exactly, never. This is very important if we are doing something critical like dealing with the health of someone or his money.
3. When there are errors those are errors specific to the domain, so that they are easier to understand. They are domain specific: so errors are not about null pointers, they are about things that a domain expert can understand.
4. It also means that the interpretation is easier, so bringing them to a new platform is easy. The same applies to simulators. The same HTML documents we could open on a PDA in 2000 can now be open on an iPad pro.
5. We can teach them more easily: they are limited in scope so less time and less training are needed to master them simply because there is less stuff

Why Adopting Domain Specific Languages?

Ok, we have seen what Domain Specific Languages are and how they differ from GPL, now we should understand ***why*** we should consider adopting them.

What are the real benefits?

Domain Specific Languages are great because:

1. They let you *communicate with domain experts*. Do you write medical applications? Doctors do not understand when you talk about *arrays* or *for-loops*, but if you use a DSL that is about *patients, temperature measures* and *blood pressure* they could understand it better than you do
2. They let you focus on the important concepts. They hide the implementation or the technical details and expose just the information that really matters.

They are great tools to support reasoning on specific domains and all the other advantages derive from that. Let's look at them in details.

In many contexts you need to build software together with domain experts who are not themselves developers.
For example:

- You could build medical applications and need to communicate with doctors to understand the treatment a companion software should suggest
- You could build marketing automation software. You would need the marketing people to explain you how to identify clients matching a certain profile, to offer them a particular deal
- You could build software for the automotive industry. You would need to communicate with the engineers to understand how to control the brakes

You could build software for accountants. You

Now, the problem is that these domain experts do not have a background in software development and the way of communicating of developers and those domain experts can be very different, because they speak different languages.

> *Developers talk about software, while domain experts talk about their domain.*

By building a DSL we build a language to communicate between developers and domain experts. This is not too dissimilar to what is described in Domain Driven Design. The difference here is that we want to create a language understood by developers, domain experts and also by the software that will be able to execute the instructions specified in the DSL.

Now, the holy grail would be to create a language, give it to domain experts and have them go away and write their queries or logic alone. In practice, usually DSLs do not achieve that but prove very useful anyway: a typical interaction consists in having a domain expert describes what it wants to a developer, and the developer can immediately write down that description using a DSL. The domain expert could at this point **read it and criticize it**.

simulators or run queries on the fly so that the domain expert can look not only at the code itself, but also at the result.

These kinds of interactions in practice can have a **very short turnaround**: code can be written during a meeting or within days. While typically when using GPL the turnaround is measured at the very least in weeks, if not months or years.

By using DSLs you can very frequently:

- having domain experts read or write high level tests. Like requirements which are executable
- when doing co-development with domain experts developers can get feedback at a very fast pace

So the answer to the question:

> *Can domain experts (not programmers) write DSLs alone?*

Is, of course, “it depends”. But definitely DSLs permit to have **domain experts involved in the development process**. Reducing dramatically feedback cycles and reducing exponentially the risks of disalignments. You know, when you have developers talking a couple of

avoid that, by using a DSL.
So it is important to understand the advantages of that, and at the same time be realistic. You see, some decades ago there were enthusiast suggesting that all sort of people could write queries in SQL autonomously:

Forty years later it appears pretty clear that housewives are not going to write SQL.
The point is that many DSLs require to formalize processes in a way that demand significant analytic skills. Those skills are easily found in developers, but are not so common in people with a non scientific background.

Focus and productivity

The fact that the DSLs abstract some technical details

On one hand, it make the investments in the code written using DSLs something that **mantain value over time**. As the technology change you can change the interpreter processing DSL code, but the DSL code can stay the same. An HTML page written 20 years ago can still be opened using devices that no one was able to imagine 20 years ago. The browsers in the meantime have been completely rewritten multiple times. Then the logic can be ported to new technologies.

I want to share a story about a company I have worked with. This company has created its own DSL to define logic for accounting and tax calculations. They started building this DSL 30 years ago and at that time they used to generate console applications. Yes, applications that run in consoles of 80×25 cells. I worked with them re-engineering the compiler and the same code of their DSL is now used to generate reactive web applications. How this happened? Because the **DSL captured only the logic, the really valuable part of the programs**, an asset extremely important for the company. The technical details were abstracted in the compiler. So we just had to change the DSL compiler to preserve the value of the logic and make it usable in a more modern context.

This teach us that:

> *Domain logic is what has value and should preserved, while technology change over time*

By using a DSL we can decouple domain logic and technlogy and make them evolve separately.

Another advantage of hiding technical details is productivity. Think about the time spent thinking about deallocating memory or choosing implementation of a list would perform best for the case at hand. That time has a poor ROI. With a DSL instead you just focus on the relevant parts of the problem and get it solved.

The Lexical (Lexer)

> *If my language is Turing complete I can do everything with it*

Yes and no. You can tackle any problem but you cannot necessarily:

- write the most concise and clear solution to a problem
- you cannot write the solution quickly
- you cannot write a solution that is understandable
- you cannot provide errors which are understandable
- you cannot show it and discuss it with domain experts
- you cannot provide tool support which is meaningful for the problem at hand
- how reusable is the solution: if you write it in C you cannot later use that solution in other context. With a DSL you can start by building the solution once and then build several code generators or interpreters
- a DSL can in reality be faster because the generator can be specialized for a certain architecture

First of all, if a language is tailored for a specific domain, persons that know that domain should be very facilitated to learn the language, because it is about the concepts they are familiar with. It is true that learning has a cost and this cost can be reduced with good tool support: editors that provide auto-completion, proper error messages and quick fixes can reduce the learning time. The possibility to easily obtain feedback, for example by simulating the results of the code just wrote, also helps. There is also the possibility of creating interactive tutorials.

> *But I would be locked in into the DSL!*

I have some shocking news: you are already locked in whatever programming language you are using to express the logic of your systems. If those languages stop evolving you will need to sort your way out. How many companies are trapped by their Cobol, Visual Basic or Scheme codebases? The difference is that, if you are locked in into a language you build and control, you can decide if the language keeps evolving, if the compiler can target a new environment ("let's generate a web application instead of a console application!"). If you are locked in someone else's language there is not much you can do about it.

occasionally want to do but that the DSL will not support. Why is that? Because a DSL has to be specific and limited, to support in a great way a set of tasks others have to be left out. To strike the right balance is one of the greatest challenges you will face when designing a DSL, but if you get it right you should be able to cover all reasonable usages with your DSL. It is often possible to design a DSL to be extensible, supporting functionalities written in a GPL, in the case they are really, really needed. But this should be more seen as a way to reassure users, until they realize they would very rarely need it.

> *Build a DSL take a huge effort*

This is just not true, if you use the right approach. In the following section we are going to see different ways to build languages and all the necessary supporting tools with a reasonable effort.
You can find a list of other perceived obstacles to DSLs adoption here:
Stumbling Blocks For Domain Specific Languages or you may want to read this paper I coauthored about benefits and problems with adopting modeling in general (it mostly applies to DSLs as well).

Languages

Wonderful, you have seen why Domain Specific Languages are so cool and what benefits they can bring you.
Now there is only one question:

> *How do we build DSLs?*

Let's see how by looking at:

- what tools you can use to build DSLs
- what are the most important success factors
- which skills do you need

What tools can we use to build Domain Specific Languages?

There are different ways to build a DSL. The goal here is to build a language, with tool support, while keeping the effort reasonable. We are not building the next Java or C# so we are not going to pour tens of man years at building an extra complex compiler or an IDE with tons of features. We are going to build a useful language

So I am going to show you the menu and you can pick your own choice. If you need help you can look at the comparison I prepared at the end of this list.

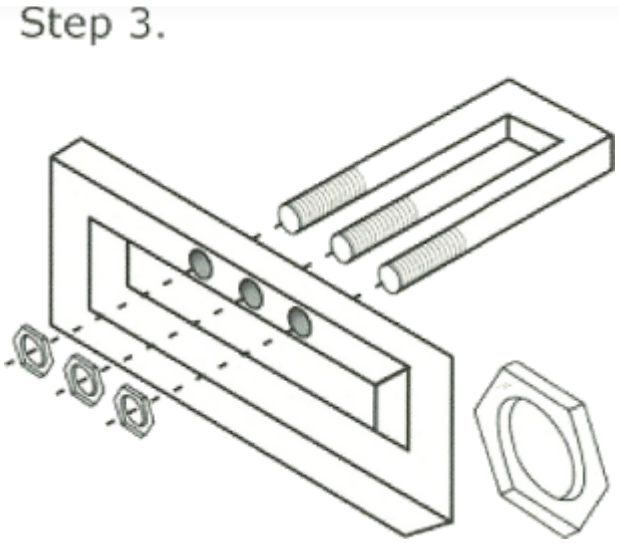
The approaches are divided in three groups, depending on the kind of DSLs you want to build:

- textual languages
- graphical languages
- projectional editors

You probably know what textual and graphical languages are but you may not have encountered projectional editors before. They are pretty interesting things so you should probably take a look.

Textual languages

These are the most classical languages. Most practicioners will not even conceive other kinds of languages. Admittedly we are all used to work with textual languages. They are easier to support and can be used in all sort of contexts. However to use them productively I think that a specific editor is mandatory. Let's see how to build textual languages and supporting tools.



Roll up your own solution: You could reuse a parser generator like ANTLR (or Lex & Yacc if you are an old-style guy) and write the rest of the processing yourself. This is doable but it requires knowing what your doing. If you don't know where to start you can take a look at my book on building languages.

I am sure you want to read it so I do now want to spoil it too much, but the path is more or less this:

1. You define the lexer and parser grammar using ANTLR. You do not know ANTLR? No problem, here it is a nice tutorial on ANTLR. In this blog there are many other articles about ANTLR.
2. You transform the parse tree produced by ANTLR in a format easier to work with. So you get the model of your code.

you know what you are doing you can get it done in a few hundreds of lines of code.

4. Finally you either interpret or compile the model of your code
5. You build a simple editor for your language. For how to do that look for tutorials on this blog or into the book. I tend to use an hackable editor I built myself. I named it Kanvas

What I like of this approach is that you are in control of what is happening. You can change the whole system, evolve it and it is simple enough that you can really understand it. Sure with this approach you are not going to get a super complex IDE with tens of refactoring operation. Or you are not going to get those things for free at least.

Xtext is a solid solution to build textual languages. In practice you define your grammar in a way similar to what you would do with ANTLR but instead of getting just a parser you get a nice editor. This editor is by default an Eclipse plugin. It means that you will be able

basically supporting only your language and removing a bunch of stuff that would not be useful to your users.

So Xtext give you an editor and a parser. This parser produces for you a model of your code using the Eclipse Modeling Framework (EMF). So it basically means that you have to study this technology. I remember the long days reading the EMF book as one of the most boring, mind-numbing experiences I have ever had. I also remember asking questions on the Eclipse forums and not getting any answer. I have open bug reports and I have received answers *three years after* (I am not joking). So it was disheartening at first but over time the community seems to be improved a lot. Right now the material available on the Xtext website is incomparably better than it used to be and the book from Lorenzo Bettini helped a lot (more on it later).

The editors generated by Xtext can be deeply customized, if you know what you are doing. You can get away with minor changes with a reasonable effort, but if you want to do advanced stuff you need to learn the Eclipse internals and this is not a walk in the park.

Recently Xtext escaped the “Eclipse trap” by adding the possibility of generating editors also for IntelliJ IDEA and… the web! When I first found out this I was extremely excited. Me, as many other developers

and battle-tested as the one for Eclipse. I did not try yet the web editor, but from what I understood it generates a server side application which is basically an headless Eclipse and on the client side it generates three different editors based on three technologies (each one with a different level of completeness). The one fully supported is Orion, an Eclipse project. While the other two are the well-known CodeMirror and ACE.

You may want to check out this list of projects implemented with Xtext to get an idea of what is possible to achieve using Xtext.

Textual languages: other tools

If I had to build a textual language in most cases I would go for one of the two approaches defined earlier: either my do-it-yourself approach or using Xtext. That said there are alternatives on which I think it makes sense to keep an eye on.

metaprogramming power of Python to define classes in memory. While it seems nice and easy to use, textX does not generate editor support like Xtext, so that is a major difference.

If you want to get a better feeling of how textX works take a look at this video.

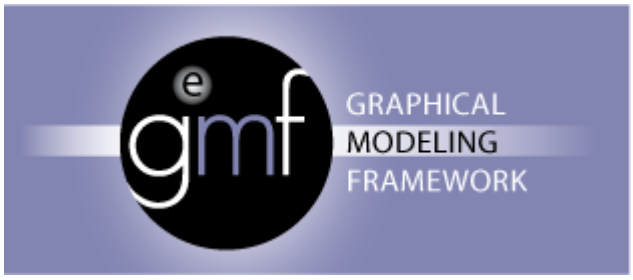
There are other tools like Spoofax. I did not use it myself so I cannot vouch for it. It is more academic stuff than an industrial-grade language workbench, so I would suggest a bit of caution.
Spoofax can be used inside Eclipse. It is based on a set of DSLs to use to create other DSLs.

If you want to look into Spoofax you may want to look at this free short book from Eelco Visser named Declare Your Language.

Graphical languages

and they are less flexible than textual languages. Also, they are less frequently used than textual languages and the tools to build graphical languages tend to be less mature and more clunky. Here I present you a list of a few options. If you want to read a more complete list you can look into this survey on graphical modeling editors.

GMF, the painful solution



There is one way to build graphical editors for your language that over time acquired quite a (not-exactly-positive) reputation. Its name is *GMF: Graphical Modeling Framework*.
Yes, you can use it to build editors which can be used inside Eclipse. Similarly to Xtext it is based on EMF. Basically you use EMF to define the structure of your data and then GMF permits to specify how the different elements are represented, how their connections are displayed and that sort of stuff. Typically you then edit

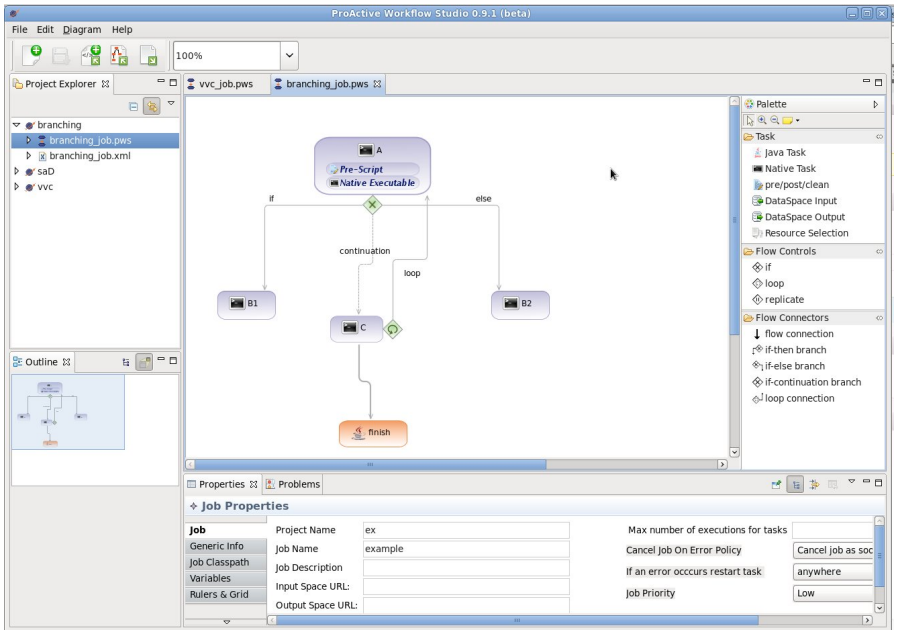


Image from https://esalagea.wordpress.com/2011/04/13/lets-solve-once-for-all-the-gmf-copy-paste-problem-and-then-forget-about-it/

Now, the documentation is basically unexisting and to make it work is a challenge which requires a great amount of patience and determination.

This framework has potential and it is powerful and flexible, but working with it is far from being an enjoyable experience.

Sirius, hiding GMF

experience less terrible for the language designer. A simple tool is Eclipse Eugenia, while a more complex one is Eclipse Sirius. Sirius reuses some pieces of GMF (GMF Notation, GMF Runtime) but it moves aways from its code generation approach and instead use model introspection. Let me stress this is what I have read about Sirius but I did not used it myself.

I have used Eclipse Eugenia and it helped jump starting my editor, but it is a limited tool and if you want to customize your editor you are back to GMF.

I have not used Eclipse Sirius myself but it seems to be decently supported by Obeo and being used at Thales, so I would expect it to have at least reasonable maturity and usability.

MetaEdit+, the commercial solution

MetaEdit+ is a language workbench for defining graphical languages. Contrary to all the other tools we discussed it is a commercial tool. Now, generally I prefer to base my languages on open-source solutions because I found them more reliable. I know that in the worst case I can always jump in and mantain the platform myself, if I really need to. With a commercial solution instead we have to consider what happens if the provider goes out of business. You can probably keep using the tool you bought for a while, until it is not compatible with the operating-system you are using. If you are about doing a large investment you could also consider setting up a source code escrow, to get access to the code in the unfortunate circumstance the provider shuts down. That said MetaCase (the company behind MetaEdit+) is a solid company which has been in business for quite a few years.

I have assisted in two occasions to a presentation from Juha-Pekka Tolvanen and I was positively impressed both of the time. They have a mature solution and they use it to build a bunch of interesting DSLs for their clients. I like very much to check their regular tweets on the *DSL of the week*. Here a few samples:

[av_gallery ids='3081,3080,3079,3078,3077,3076,3075'

imagelink=’lightbox’ lazyload=’avia_lazyload’ av_uid=’av-akiaf’]

So if you ever need a graphical language I would advise to consider this solution. The alternative is in my opinion to use a full-blown projectional editor, which permits to create *also* graphical languages, but not only those. Curious? Keep reading.

Projectional editors

Projectional editors are extremely powerful and exciting but they are unfamiliar to many users. I can give you the theory bit, and throw at you a definition, but if you really want to understand them watch the video below.

A projectional editor is an editor that show a projection of the content stored on file. The user interacts with such projection and the editor translates those interactions to changes to the persisted model. When you use a text editor you see characters, you add or delete characters and characters are actually saved on disk. In a projectional editor you could edit tables, diagrams and even what it looks like text, but those changes would be persisted in some format different from what you see on screen. Maybe in some XML

them around, connect lines and in the end the editor save some obscure format, not the nice pictures you see on the screen. The point with projectional editors is that they are much more flexible than your typical graphical language. You can combine different notations and support all sort of representation you need for your case.

Confused? That is expected, watch the video below, watch many more videos and things will appear clear at some time.

You could also take a look at this explanation of projectional editing written by Martin Fowler.

Jetbrains MPS

Jetbrains MPS is a tool that I have been using for some years, working on it daily on most of last 12 months. It is an extremely powerful tool and it is the most mature projectional editor available out there. It is no accident: Jetbrains has invested significantly on developing it

I find very useful MPS to build families of interoperable languages with advanced tooling. Imagine several DSLs to describe the logic of your problems, to define tests, to define documentation. Imagine all sort of simulators, debuggers, tools to analyze code coverage. Everything built on one platform.

Now, it means you need to be ready to embrace Jetbrains MPS and to invest a significant amount of time to properly learn it (or hire someone like me). However if you are ready to do the investment it can simply revolutionize your processes.

Intentional Platform

This one is the mysterious and intriguing one.

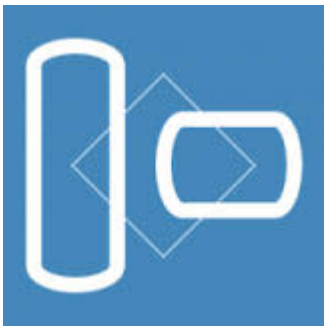
Charles Simonyi is the man who designed Excel, one of the first space tourists and one of the richiest men on earth. In 1995 he wrote the revolutionary paper The Death of Computer Languages, The Birth of Intentional Programming. In this paper he presents a new, more intuitive way of programming, fundamentally based on projectional

extremely scarce. They have published a few papers and given a few presentations.

I have heard from people that have used it that it has a lot of potential but it is not there yet. For sure I would love to put my hands on it. The closest you can get is to read this somehow old review of the Intentional Platform from Martin Fowler. Maybe one day even us mortals will have the possibility to know more about this legendary tool.

As far as I know they work with selected companies but for now their tool is not publicly available.

Whole Platform



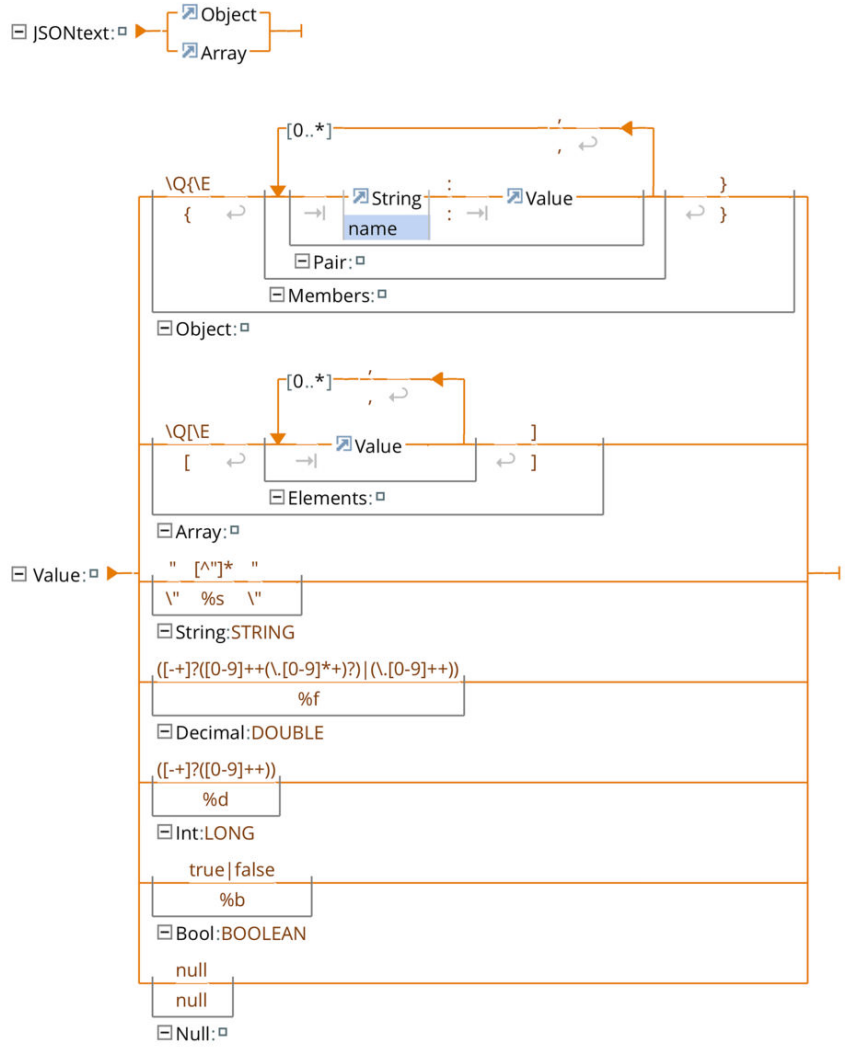
This is a good Language Workbench too frequently overlooked. While it has been used in production for many years in a large company in Italy, it is lacking a little bit on the documentation and marketing side. Yes.

There are a few concepts that I find quite interesting. I am not an expert on the Whole Platform: I have just played with it and talked with his authors.

One aspect that I find really interesting and different with respect to the other Language Workbenches is that the Whole Platform supports quite well working with existing formats, and evolve from existing processes to more advanced Language-Engineering approaches. For example, it is quite easy to define grammars for existing formats in order to parse them.

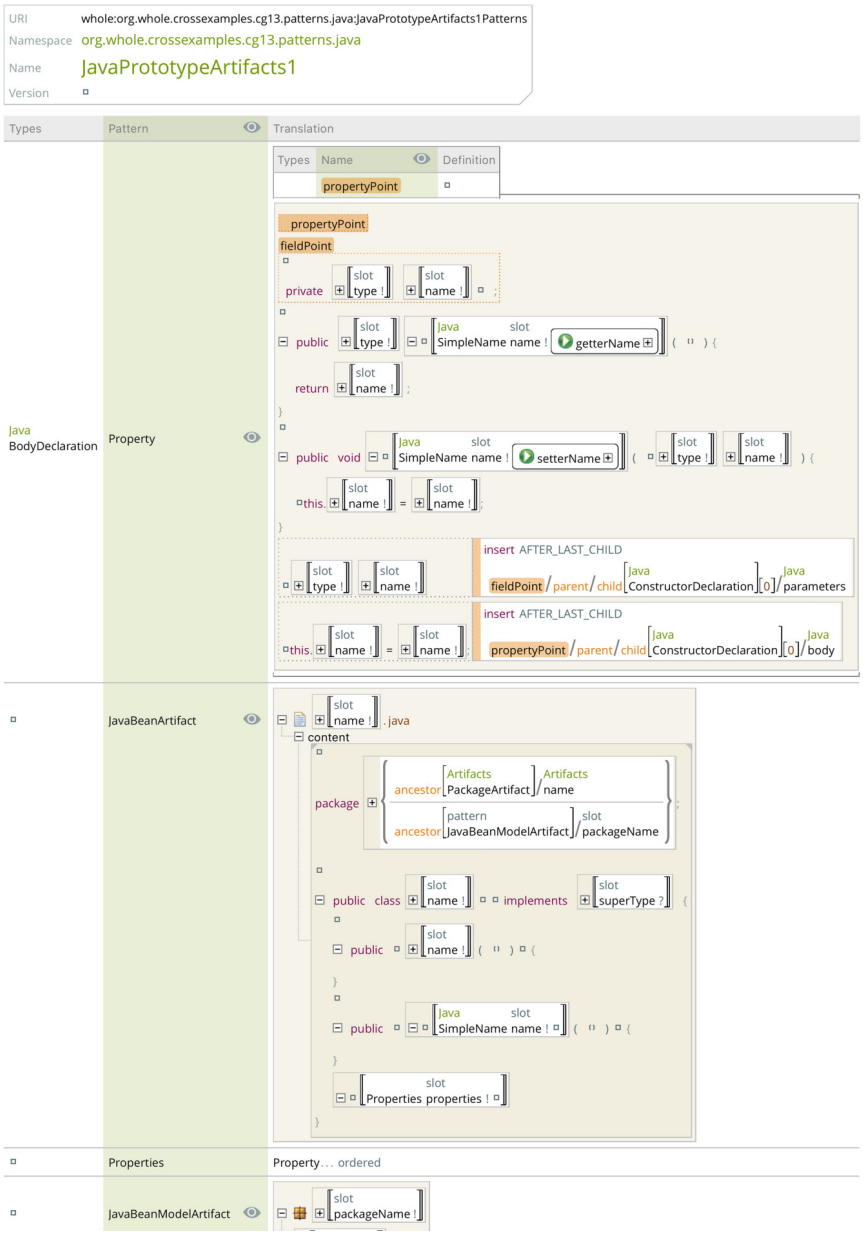
The following is an example of a grammar to parse JSON but the same approach has been used to parse very complex formats used in the banking domain.

Phrase Structure

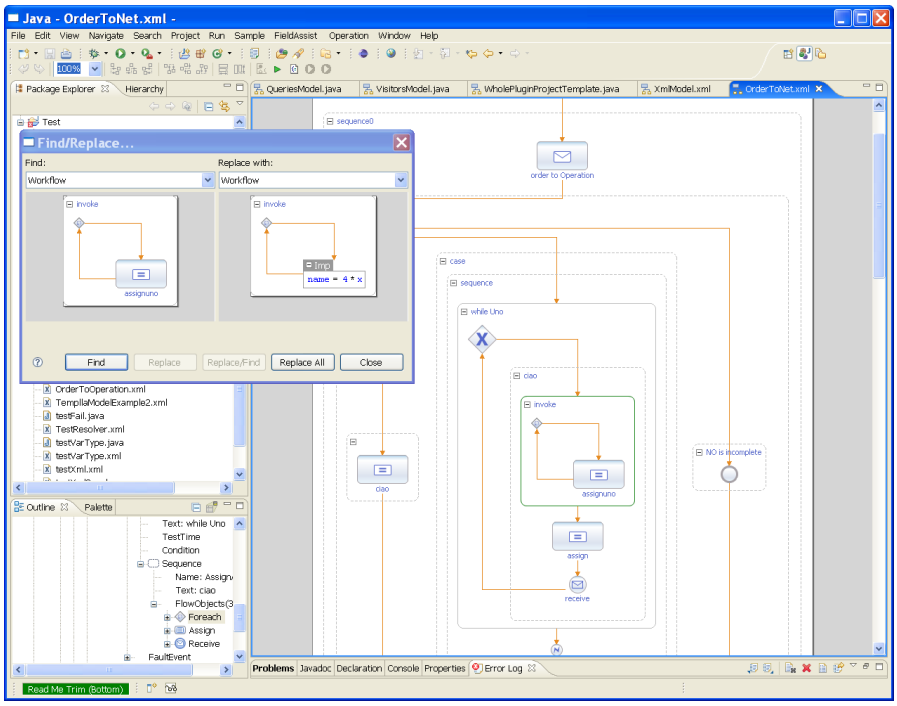


Lexical Structure

Another idea I love about the Whole Platform is the Pattern Language: the possibility to take a model (in this sense a piece of Java code is a model) and define variability points that could be filled with values from another model.



Also, the Whole Platform is quite rich and it supports also graphical languages.



Whole Platform – Graphical Language

Riccardo Solmi and Enrico Persiani are the minds behind the Whole Platform and you should probably talk to them if you are interested in using this Language Workbench.

The images of the Whole Platform I have used are released under the CC Attribution 2.0 license (https://creativecommons.org/licenses/by/2.0/)

Comparing the different

Do-it-yourself	Textual	You need to be in absolute control of the platforms supported and you do not want any vendor lock-in
Xtext	Textual	You want a textual language with good editors and you want it fast
textX	Textual	You love the flexibility of dynamic languages while editor support is not important to you
Spoofax	Textual	You like to work with a sound theoretical approach and you are not afraid of a few bumps in the road
GMF	Graphical	You need extreme flexibility to build your very own graphical editor
Eclipse Sirius	Graphical	You are ready to trade some flexibility to get things done quicker and saving some mind sanity

Jetbrains MPS	Projectional	You want to build family of languages with powerful and complex tooling like simulators, debuggers, testing support and more
Intentional Workbench	Projectional	You have some connection that give you access to the most mysterios and hyped Language Workbench
Whole Platform	Projectional	You want to support existing formats and transition smoothly to a new approach

One thing I would like to stress is that projectional editing is a superset of the graphical editing. So you can define graphical languages using Jetbrains MPS. Given there is not a clear and great alternative to build only graphical DSLs I would use Jetbrains MPS in that case. Alternatively I would consider build the tooling myself targeting the web. Another interesting option could be looking into something like FXDiagram.

Still not enough?

typically colocated with the Software Language Engineering conference. I co-authored a paper and attended the last edition and it was awesome.

What do I need to make my DSL succeed?

There are just two things that will seem obvious but are not:

1. You need your users to use it
2. You need your users to get benefits from using it

To achieve this you will need to build the right tool support and adopt the right skills. We are going to discuss all of this in this section.

Get users to use it

The first point means that you need to win the support of users. During my PhD I conducted a survey on the reasons why DSLs are not adopted. There are several causes, but one important factor is **resistance from users, especially when they are developers**.

If you target a DSL to developers, they could resist because they feel they are not in control as when they use a General Purpose Language (GPL). They could also fear that a DSL lowers the bar, being simpler to use than, let's say, Java. In addition to that, as all innovations, a new DSL is threatening for seasoned developers because it reduces the importance of some of their skills, like the vast experience in dealing with the quirks of the current GPL you are using at your company.

If your DSL is intended to non-developers generally it is easier to win their support. Why? Because you are giving them a superpower: **the ability to do something on their own**. They could use a DSL to automatize a procedure that previously was done manually. Maybe before the DSL was available the only possibility for them to do something was bothering some developer to write custom code. Now they get a DSL which means more power and independence because of it. Still they could resist adopting it if they perceive it as too difficult

design of the DSL or the tooling around it. Involve them in the design of the DSL. When talking with your users, technical or not, communicate that the DSL will be a tool for them, designed to support them and derived by their understanding of the domain at hand. When designing DSLs the cowboy approach does not work, you need to succeed as a team or not succeed at all.

Give benefits to users

If you get the support of users and people start using it you win only if they get a significant advantage from using the DSL.

We have discussed the importance of a DSL as communication tool, as a medium to support co-design. This is vital.

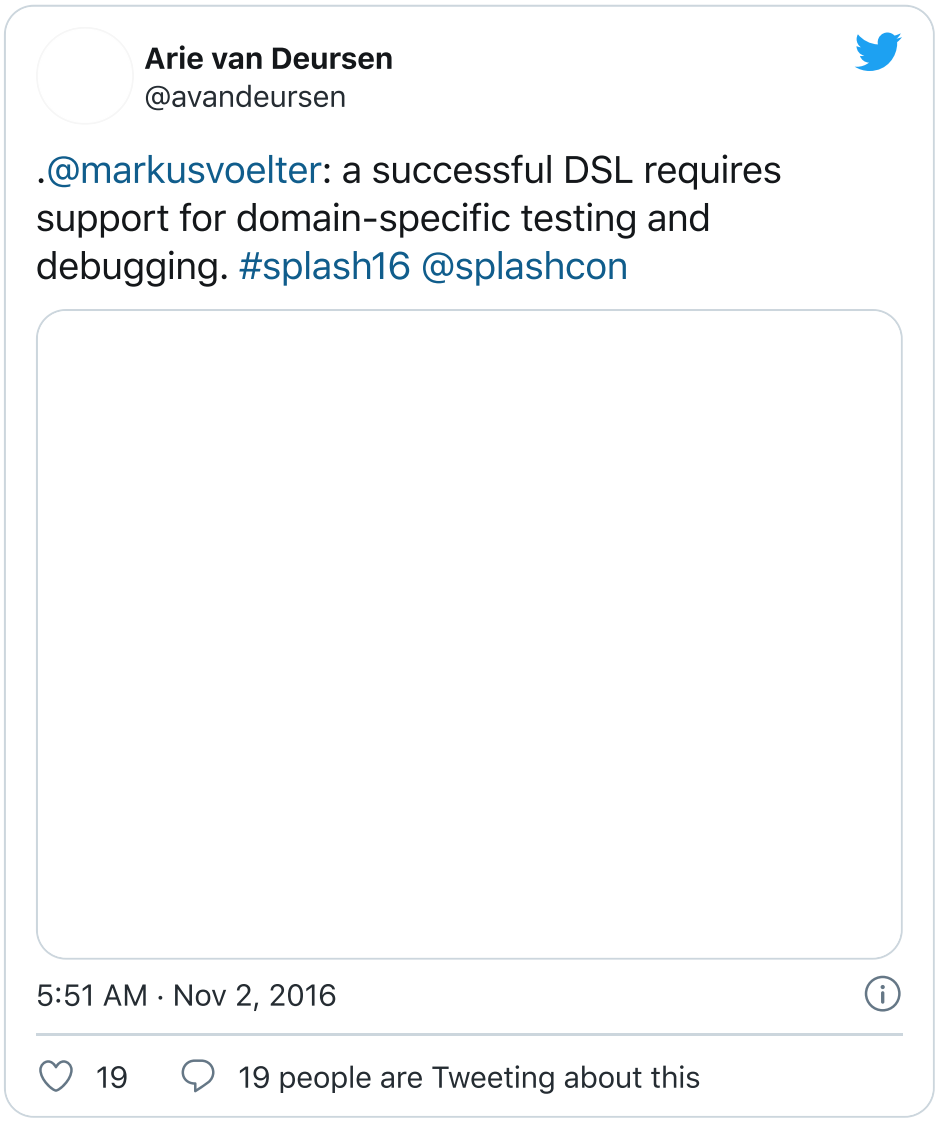
A few examples:

- a great editor with syntax highlighting and auto completion: so that learning the language and using it feel like a breeze
- great error messages: a DSL is an high level language and error can be very significant for users
- simulators: nothing helps users as the possibility to interact with a simulator and see the results of what they are writing
- static analysis: in some contexts the possibility to analyze the code and reassure against possible mistake is a big win

These are a few ideas but more can be adopted, depending on the specific case. Specific tools support for specific languages.

Tool support: why we do not care about internal Domain Specific Languages

express, with everything else being a detail.



This is just fundamentally wrong because the tool support can multiply exponentially the productivity when using any language, especially a DSL. This is a crucial aspect to consider, because the language should be designed considering tool support.

"yes yes yes!" #splash16 #hazelnut

11:58 AM · Nov 4, 2016

21 27 people are Tweeting about this

Because when building Domain Specific Languages, if you want to get serious, you have to build good tool support. Tool support is an essential key in delivering value.

I recorded this short video on this very subject.

Building a language: tool support

from **Strumenta**

Tool support is the reason why internal domain specific languages (i.e., fake DSLs) are irrelevant: they do not have any significant tool support.

When using some host languages you can bend them enough of getting some sort of feeling of having your own little language, but that is it. There are some languages that are flexible enough to give a sort of decent … like ruby. With other languages you get very poor results. I feel pity for the people trying to build "internal DSLs" for languages like Scala or Java. The worst of all is lispers. I understand their philosophy "if you want to solve a problem in LISP, first you create your LISP dialect and then solve it using it". I understand and I think it is a great technique. Just let's not pretend this is a real DSL. This is not something you can share and work with closely with domain experts. It can be your trick to be more productive, but that is it.

You see those ridicuosouly long chains of method calls and you hear someone presenting those as Domain Specific Language. That makes me feel a mix of two emotions:

- pity for him and his users
- rage for the confusion it creates. Real DSLs are very different and they can bring real benefits. Stop mixing them with this … thing

Just build a real DSL, so an external DSL!

What skills are needed to write a DSL?

Typically you need to be able to have **high abstraction skills**, the same you need typically for metaprogramming. If writing a library is 3 times harder than writing a program, writing a framework or a DSL is typically 3 times harder than writing a library.

You need to be **humble**: you may need a developer, but typically you need to create this DSL for other professional that are going to use it for their job. Listen

the single best way to create a failed DSL, not useful in practice and therefore not used.

Aside from this, practice and learn. Keeping doing it for a few years should do the trick.

Resources

Now that you have seen what DSLs can bring you and you have an idea how to build them you should be happy.

But you are not, you want more, you want to understand better DSLs, you want to learn everything about them.

Well, I do not know about everything, but definitely I can give you some pointers. I can tell you where to find the good stuff.

Let's see what we can find:

- books
- websites and papers
- companies

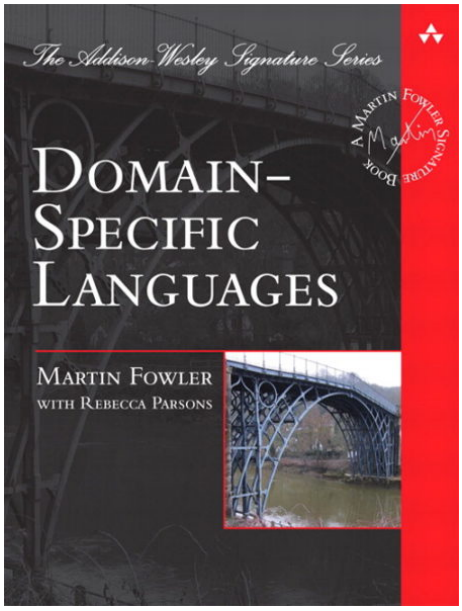
Books

I would start suggesting to read the DSL Engineering by Markus Völter. This PDF version of the book is donation-ware. So just read it and donate. Alternatively you can find the printed version on Amazon.

The book start with an introduction part: it is very useful to set your terminology straight. After that it comes the DSL design part, focusing on different aspects separately. If you do not have direct access to an expert to teach you how to design DSLs reading this part of this book is the best alternative I can recommend (together with as much practice as you can, of course).

Then it comes the part about implementation: remember that Markus has a PhD, but he is first of all someone who gets things done so this part is very well

Part IV is about scenarios in which DSLs are useful. Given this is based on his large experience in this field there are a lot of interesting comments.



I had the occasion to work with Markus. I used to admire him a lot before meet him and I now I admire him even more. He is simply the best one on this field, so if you can learn something from him do it. Read his books, watch his presentations, follow his projects. It will be a good way to invest your time. He is lately working on Jetbrains MPS stuff, so you should follow what is going on with mbeddr and IETS3. Mbeddr is both a set of plugins for MPS and an implementation of the C language in MPS with special domain-specific extensions to support development of embedded software. IETS3 is instead an expression language built

author of Domain Specific Languages, a book about both internal and external DSLs.

I find the mental models presented in the book quite useful and elegant. In practice however I find internal DSLs irrelevant, so I am interested in only some portions of this book.

There are 15 chapters dedicated specifically to external domain specific languages. While those chapters are organized around implementation techniques there are comments and remarks from which you can learn some design principles.

I think the sections on the alternative computational models and code generation are very valuable. You will have an hard time finding an exploration to these topics at this level of detail anywhere else.

The book is 7 years old and the techniques may have

well explained, as you would expect from Martin Fowler.

If you are interested in textual languages and in particular on ANTLR you should definitely look into the Language Implementation Patterns from Terence Parr. I like the author, I like the publisher (the Pragmatic Bookshelf) and unsurprisingly I love the book.

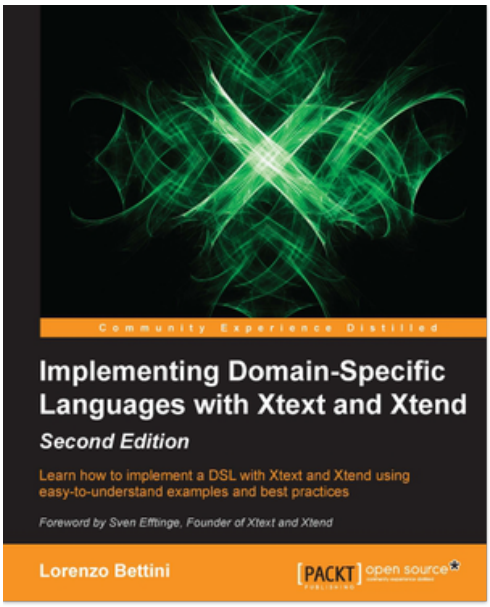
The book starts discussing different parsing algorithms. If you like to learn how stuff works you should take a look at these chapters. Then there are chapters about working with the Abstract Syntax Tree, extracting information, transforming it. This is the kind of stuff you need to learn if you want to become a Language Engineer. Also, there chapters on resolving references, building symbol tables or implementing a typesystem. These are the foundations to learn how to process the information expressed in your DSL.

Finally Terence explains you how to use the information you have processed by building an interpreter or a code generator. At this point you end your journey, having seen how to build a useful language from start to finish. This book will give you solid basis to learn how to implement DSLs. The only thing missing is a discussion on how to design DSLs, but this is not the goal of this book.

If you are looking into MPS there are not many resources around. It could make sense to buy the two books from Fabien Campagne on the MPS Language Workbench. They explain in details all the many features of MPS (admittedly some are a bit obscure). If I would have to find one thing missing is more advices on language design. These books are very good references to learn how MPS works, but there is not much guidance on how to combine these features to get your results. One reason for that is that MPS is an extremely powerful tool, which can be used in very different ways so it is not easy to give general directions.

Volume I explains separately the different aspects of a language: how to define the structure (the metamodel), how to define the editors, the behavior, the constraints, the typesystem rules and so on. Most of the chapters

Volume II is mostly about the advanced stuff that you can safely ignore at the beginning. I suggest looking into this book only when you feel comfortable with all the topics explained in Volume I. If you have never used MPS before it will take some time. Volume II explaine you how to use the build framework to define complex building configurations, it gives you an overview of all the different kinds of testing you may want to use for your languages. It also show you how to define custom aspects for your language or custom persistence.



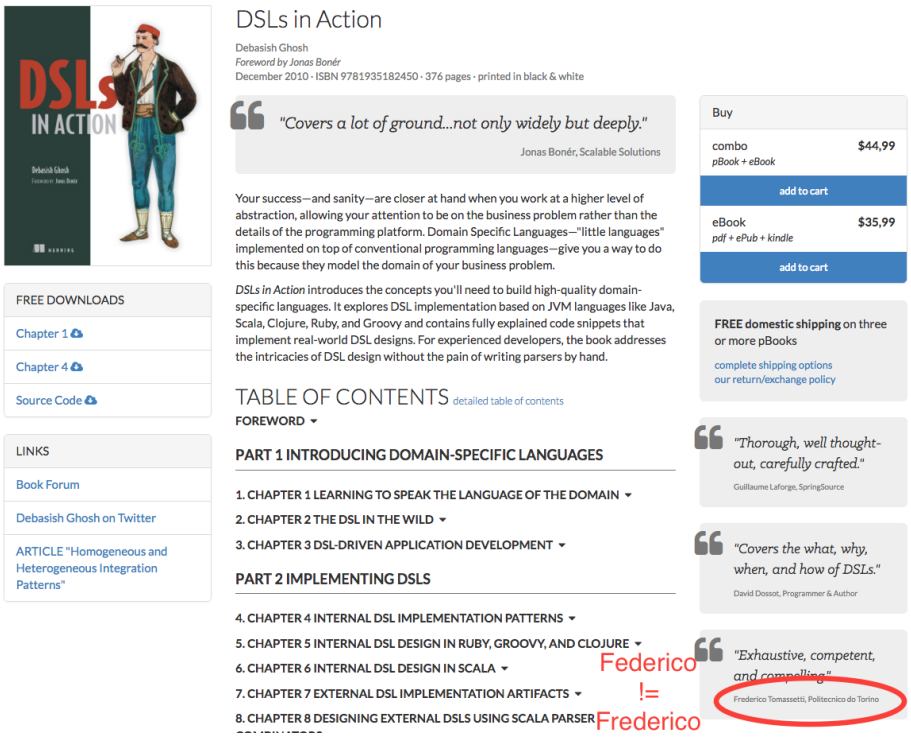
I have bought the Google Play version but they are available also in print form.

On Xtext there is a quite practical and enjoyable book from Lorenzo Bettini:
Implementing Domain-Specific Languages with Xtext

If you want to learn how to write textual languages with good tool support you could start following a couple of tutorials on Xtext and then jump to this book. This book will explain you everything you need to know to use Xtext to build rather complex editors for your language. The only caveat is that Xtext is part of a complex ecosytem so if you really want to become an expert of Xtext you need to learn EMF and Xtend. The book does a good job in teaching you what you need to know to get started on these subjects but you may have to complete your education with other resources too, when you want to progress.

What I like about this book is that is not a reference manual, but it contains indications and opinions on topics like Scoping or building a typesystem rules (the author has some significative experience on this specific topic). Also, the author is interested in best practices so you will read his take on testing and continuos integration. The kind of stuff you should not

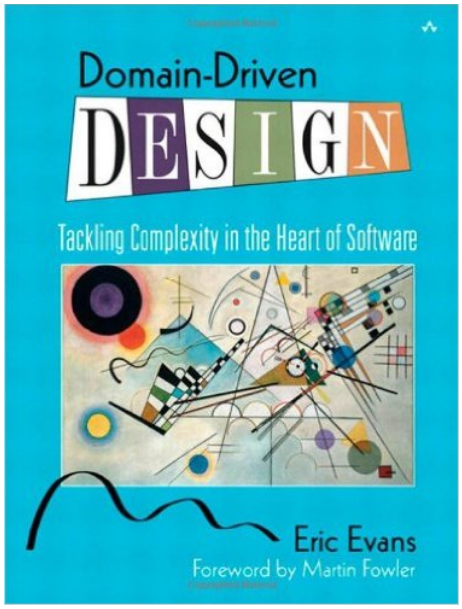
ignore if you are serious about language engineering.



If you are interested in DSLs in general you can take a look at DSLs in Action. The book is interesting but I have two comments:

1. It focus way too much on internal DSLs, which are, as we all know, not the real thing
2. They have misspelled my name. -1 point for that

Specifically on external DSLs there is not much: the author briefly discuss Xtext and then spend a chapter on using Scala parser combinators to build external DSLs. That would not have been my first choice. So if you are interested in learning how to implement an external DSL do not pick this book. However if you want a gentle introduction to the topic of DSLs, if you are a



Domain Driven Design is a relevant and important book to read. Because you need skills to understand the domain, in order to be able to represent it in your language, and to design your language so that it can capture it. Now, I should probably just praise this book and stress how much I have enjoyed it. Unfortunately I tend to err on the honesty side so I warn you: this is one of the most boring books I have ever read. It is important, it is useful, it is great but it just so plain and long. It stayed on my night stand for months.

What you should get how of this book is the importance of capturing the Domain in all of your software artifacts. The book stress the importance of building a common language to be shared among the stakeholders. This is completely and absolutely relevant if you want to build

specific to DSLs design. This book is a good complement to any of them.

Websites and papers

- modeling-languages.com: a blog from Jordi Cabot about modeling in general. Many topics are of interest for DSLs practicioners. I would suggest to keep an eye on it to learn about new stuff in the language engineering field.
- DSL development: 7 recommendations for Domain Specific Language design based on Domain-Driven Design: A concise list of suggestions on designing DSLs. While it was published some years ago I think it is still relevant. This article has been written by Johan den Haan who is the CTO at Mendix. Mendix is a company activing in the model-driven field.
- Xtext Screencasts: a collection of screencasts about Xtext. It is not very recent but still very valuable
- Design Guidelines for Domain Specific Languages: 26 guidelines condensed in a short paper. The guidelines are organized around language

When and How to Develop Domain-Specific Languages
: A paper over 30 pages with an interesting bibliography of almost 100 entries. The paper presents a few patterns to be used when designing languages and list a set of open problems.

- WebDSL: A Case Study in Domain-Specific Language Engineering
: A very detailed case study on a very practical DSL. The paper is authored by Eelco Visser, who is very well known in the field.

Companies

I co-founded Strumenta, and we design and implement Domain Specific Languagesfor a living. If you want to learn more about us visit Strumenta.com:

However this section is not about telling you how great Strumenta is, I will instead list other companies that you could work with.

The first, obvious name is Itemis. They are a German company with small offices in France and Switzerland. They employ some of the best in the field: I have met and worked with Markus Völter and Bernd Kolb and I am seriously impressed by the level at which they work. They have worked with so many companies and have done so many projects that the list will be scarely too long. I would just say that in the later years they have done amazing work using Jetbrains MPS. They have created the mbeddr project and from it they have derived a set of utilities, named the mbeddr platform, which contributed enormously to the growth of MPS. They have contributed early and significantly to the Language Workbench community, so if you need some help on a DSL you should seriously consider working with them.

TypeFox is another German company. I have

platform in general. If you need Xtext training or consulting I would consider them the top-choice. They are also a Solution Member of Eclipse and I would expect them to be able to build complex Eclipse-based solutions. If you want to hear more you can read this interview to Jan Köhnlein. Jan is one of the founders of the company and we talked a few months after the company was created.

Jetbrains is obviously the company behind Jetbrains MPS. Recently they started offering training on MPS. I asked about this during my interview at Vaclav Pech. They offer training, either basic or advanced, on their premises in Prague or on-site.

At the moment I do not think they offer MPS consulting (for that you can talk to me or to Itemis).

Conclusions

There are many reasons why you should really consider Domain Specific Languages. I have seen companies benefit enormously from DSLs. Most of the people I have worked with used DSLs as a key differentiator that helped them increase productivity by 10-20 times, reduce time-to-market and feedback cycles, increase

DSLs we build powerful tools that help other people do their job. As language designers we act as enablers, our languages can be used by skilled professionals to achieve great things and this is an amazing feeling for me.

Could you help me, please?

If you found this guide useful please share it, spread the work and link it. I spent several years working on this subject and a few weeks working on this guide. I would be very happy if you could help me reach others who could find it useful.

Thank you so much!

A few ideas:

- Share it on Twitter
- Share it on Facebook
- Share it on LinkedIN
- Share it on Google+
- Write about it in your blog
- Send an e-mail to your colleagues

The complete guide to (external) Domain Specific Languages

Receive the **guide** by email and get more tips on DSLs

Name

Email

Send it to me!

ANTLR
Application modernization
Code processing
Consulting
Create a programming language
Domain specific languages
Editors
Jetbrains MPS
Language design
Language Engineering
Miscellany
Model driven development
Natural Language Processing
Non software development
Open-source Parsing
Research
Software Development
Software Engineering
Turin Programming Language
Whole Platform Xtext

Interview with Meinte Boersma on Domain-Specific Languages

04 November 2020

Domain Specific Languages for smart contracts

25 March 2020