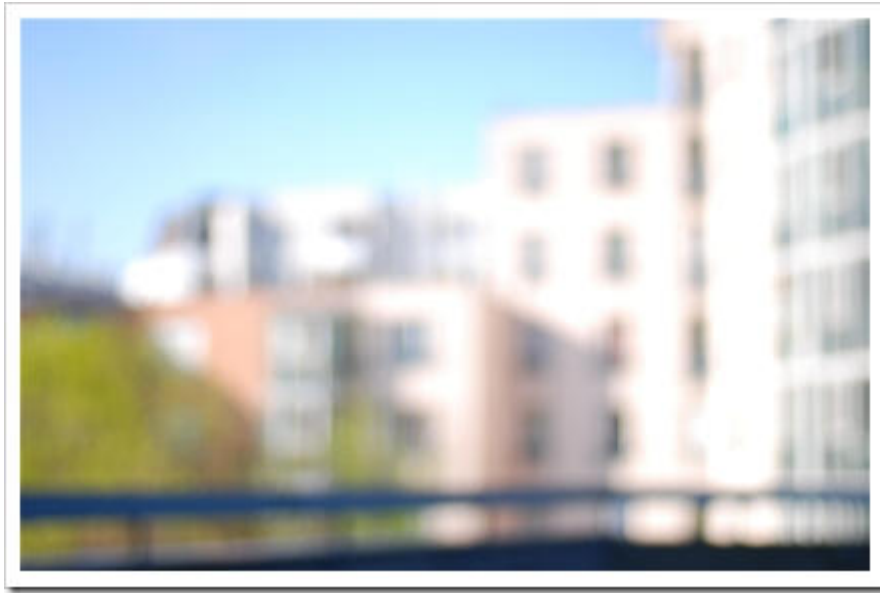


Patterns of Domain Specific Languages Use

By Sadek Drobi



Lately, involved in several discussions related to the subject, I noticed a global misunderstanding and an obvious blur about the Domain Specific Languages. This resulted in unbelief, generating a tangible doubt about its presumable use in a realistic context of nowadays industry.

Some of the blame goes to people like me, who hastily expose its ideal form, making it look like an unrealistic payoff for a lot of efforts. Such mistakes have been done frequently, particularly in IT field, recently in SOA, turning it into sort of imaginary! Other less recent examples exist, like Client-Server technology that was to allow users to draw their interfaces...

Arguments of some respectful people in today's industry had corrected my view, and made me focus on the core principle, letting the ultimate come naturally, as the appeal calls to it due to maturity.

Linking to examples from the real world can clarify some points here. I like an example of Erik Meijer in QCon 2007 London, stressing that software is like buildings. We don't build our own house ourselves. Rather we come in on later phases for decorating and arranging it. And in everyday life we keep configuring it, changing lights, paint color, carpets, maintaining and tuning it using simpler tools adapted to our needs. I'll cross this with a quote of Ted Neward "VB enabled managers to write programs. The problem is they really did! ". So the purpose, at least in the short term, is not to magically make business people write their own applications. But rather turning the software into softer, more flexible product, which can be bent and adjusted to shape the new ever-changing business needs, pursuing the ever-changing market.

DSLs are barely new, they always existed responding to the natural need of providing tools to humans for configuring their software. Their existence was indeed minimal, focused on satisfying these urgent needs of the client's business. Examples of common DSLs are xml

config file, Xslt, aspx, fit framework for unit tests, excel, CMSs templates, workflows, wikis, Sql ... etc.

Some of these DSLs have become quite mature in their domain, providing a unique tool for a specific purpose. I guess this maturity is due to their convergence with that domain. This convergence results from the fact that they were specifically designed for the domain, or at least found to fit perfectly in it, and reflect not only its vocabulary, but also its structure and interactions.

Given this observation, I would disagree about Xml as a proper wide use DSL. Xml in its own is a General Purpose Language. Proposed solutions, like XSD, DTD and other schemas, don't offer much more than rules and grammars. A DSL should have an easy understandable syntax that reflects the model. That doesn't often correspond to xml. One of the reasons can be its verbose syntax (caused by redundant opening and closing tags). Another is its structure that imposes a hierarchical thinking about the problem.

I like thinking of DSL as a language created for the problem, not forcedly adapted to fit the problem. Forcing generates awkwardness and meaningless semantics that break the readability and the coherent side of a DSL.

Nowadays there is a sufficiently wide choice of DSL tools, they come in different flavors (list is not exhaustive):

- Xml based DSLs, meaning generating xml out of personalized graphical or specific purpose syntax. Usually used as a configuration representation, it can't escape hierarchical structure for long but still handy.
- Some General Purpose Languages friendly syntax permits an expression of domain semantics in a pronounced readable manner. We can link to this some design patterns that enhance the ubiquitous language produced by these languages. Like Expression Trees, Specifications, Fluent Interfaces...etc. Examples of such languages are Ruby and Prolog. A limitation of this kind of DSL is the compiler grammars that insist on some differences that are not obvious for humans.
- General Purpose Languages with extensible syntax, the possibility of adding keywords and semantics to the language through *macros* or *metadata*. Examples include Lisp, F#, Nemerle.net. Such kind of GPL urges discipline, as it might introduce ambiguousness and verbosity.
- Compiler-Compiler tools that are tools and APIs for declaring some grammar for a parser that will parse the customized syntax, verifies it against the grammar, then generate something useful. It might be code, config files or something else. Such tools, e.g. SableCC, Antlr, have no influence on the chosen structure of the built languages since we do it from scratch. Still they require much more efforts for validation, error checks and maintainability.

Clearly enough there is no single DSL that satisfies all needs. So, logically enough, tradeoffs should be done.

Now, as choices of DSL tools diverge, their applicability areas vary in different domains, applications and contexts. The aim is to introduce modifiability by identifying an important frequently changed area in the software, to abstract it and then introduce it into a friendly DSL. This modifiability adds flexibility, agility and dynamism.

Other possible advantage of that is a connection between *userstories* and code, where the user, together with developers, can write her needs and expectations in an appropriate domain language. That language can be interpreted into running code.

I had an interesting discussion with *Jim Coplien* - he is one of the kindest and most interesting speakers I met at QCon - about TDD. The discussion didn't mention DSLs at all, but it made me think of them as a possible solution to the absence of trace between *userstories* and code. (It can be called DSL driven design and development, which maps well to Domain Driven Design of *Eric Evans*...)

Like any software development practice, DSLs should be applied right and in the right context. This comes with experience (mostly domain experience, but also conceptual and architectural one), experience in finding the right abstraction and then mapping it to the right tool for a given context.

Such an exercise can be hard even for experienced developers and architects. The choice is not evident given the diversity of domains, diversity in the same domain and diversity of DSL tools styles.

Abstraction is the key; it comes from years of experience in identifying common places to extract DSLs. Together with the domain experience that is instrumental to point out the most profitable modifiability and the most appropriate DSL style, it could be translated into templates in form of patterns, Patterns of Domain Specific Languages Use. This way, instead of generalizing solutions, thus falling back into General Purpose Language, we extract best practices to be used for boosting learning and experience curve. Examples of common places where DSLs apply well are *Strategies*, *Expression trees*, *Specifications*, *Factories* in banking, finance, workflows...

To conclude this article, and to rescue it from the pitfall of excessive abstraction, I will try to demonstrate a scenario where pattern can be identified and abstracted. A whitepaper listing several discovered patterns is the subject of a research that I started with my colleague Julien Delhomme and will be posted on www.domainspecific.org. Updates about the project with code snippets would also be available on a wiki on the same website.

State Transition

Domain objects are statefull and quite frequently their behavior depends on their state. However, this dependency is oftentimes implicit and the client of the domain model finds himself forced to define it. This leak of the domain logic to other layers results in code redundancy, a more complex client code and a lot of not-reusable conditional checks.

Once identified, this problem can be solved by refactoring to *State Pattern* [GOF, Refactoring to Patterns/ Kerievsky]. State Pattern turns the dependency on the state into an explicit relation. And this explicitness renders the domain clearer, brings back the logic where it belongs to and introduces modifiability.

The GOF State-Pattern is a technical pattern. It introduces a lot of technical classes. However, the state notion is rather functional. So, introducing the state-pattern in the domain model partially solves the cohesion problem by encapsulating the logic into the domain layer, but, unfortunately, it spreads the logic into several units, which do not exist on their own in the

real world. This is very common in technical, general purpose, programming languages. They are close to the machine and good at talking to it, but they are not instrumental for expressing the domain logic.

Think about semantics rather than code forms:

How to specify the relation between the state and the behavior in a more business-real world-friendly way? Abstraction will help us to express business requirements using its *semantics*. In this case, the *state pattern* becomes an implementation of the notion of state. And it will be encapsulated behind the friendly code, the Domain Specific Language.

Consider the following domain code for defining a shipment order:

Order definition:

N: ID
Status: State

Supported Operations:

- "Cancel"
- "Add Order Line"
- "Register"

States:

- "New Order"
- "Registered"
- "Canceled"

State Transitions:

- Register changes state "New Order" to "Registered"
- Cancel changes state "Registered" to "Canceled"
- Cancel changes state "New Order" to "Canceled"
- Add Order Line changes state "Registered" to "New Order"

This code is in English, and it is validated by MS-Word spelling and grammar check. It contains all the information needed to implement the *state pattern* for this case. And this implementation can be automatically generated out of this code. Moreover, it is modifiable and easily understandable. Introducing states and methods is quite simple and modifying the state transition condition can be done in a rather flexible way.

Consumable domain code:

This code, unlike *strategies* and *specifications*, should be consumable to be useful. Bearing that in the mind, it is quite a bad idea to ask client code to interact with the generated code, even if it has implementation *interfaces*. Client code should interact at the same level of abstraction as the domain code. Otherwise, it will be strongly coupled to the specific implementation of the generated code.

To achieve that, you need to introduce an interface code. The term "interface" is used heavily in OOP languages but it doesn't yield what I exactly mean here. It rather yields a technical term for decoupling classes and tires. What I mean by interface code is the code that is used, whatever the underlying technology is, to access our implementation of the order. This code should be provided by the same tool that provided the domain definition one. And that tool is the only one to know about the implementation behind it.

The client code could look like this:

```
Register Order N 30.
```

This piece of code contains all the information needed for doing the task, described in “register the order number 30”.

The DSLs client interface code makes it easier to change the technology, because all what is domain is expressed in a neutral language. It also results in a high readability and helps, together with the domain implementation language, to really bridge the gap between developers and business people.

Common questions can be raised about this code: “what is the use of this domain model if it is not persisted anywhere?” or “how can you persist that?”

This gives me the chance to talk about some nonsense that is spreading in the enterprise applications programming industry. Aspect Oriented Programming is a very powerful concept to eliminate technical code, and it leads towards a cleaner code. The problem is that most examples listed in tutorials are in fact the wrong use of the concepts. As I stated before in my weblog entry “Transactions Are Not Automatic, Log Neither!”, transactions are not technical, they are rather part of the business logic and should be explicit in the code. But I believe that object persistence should be an *Aspect*. The fact that it is harder to achieve doesn’t make a worse example. Ideally, objects should not care about their persistence, actually no one should. This can be a pretty good and useful example of an *aspect*. So, the code can be modified to introduce the new aspect:

Persisted Order definition:

```
N: ID  
Status: State
```

Supported Operations:

- “Cancel”
- “Add Order Line”
- “Register”

States:

- “New Order”
- “Registered”
- “Canceled”

State Transitions:

- Register changes state “New Order” to “Registered”
- Cancel changes state “Registered” to “Canceled”
- Cancel changes state “New Order” to “Canceled”
- Add Order Line changes state “Registered” to “New Order”

I even guess that ORM mapping are not always necessary, since implementations like *Active Record* offer a simpler solution, which fits pretty much with the domain model and with introducing transparently the aspect.

Conclusion:

DSLs provide solutions to the mismatch between technology and business requirements. Such a solution can't be installed using a tool but it can be introduced through practices and patterns. Concepts like DDD [Evans] and AOP can help achieving the goal, but experience is needed and that's where patterns come to help. As we are talking about Domain Specific Languages, there is no generic solution, at least not before one proves its utility being extracted from the solution and not forced into it.

My research subject will be "Discovering and Extracting Domain Specific Languages Best Practices, Introducing them as Patterns and Mapping them to the Appropriate Contexts and Tools". And as I stated before, I'll be posting patterns on www.domainspecific.org/wiki , even in early drafts form, so that I can get feedback from people's experience and their points of view.