*6*

## Process Issues

*Software development with DSLs requires a compatible development process. A lot of what’s required is similar to what’s required for working with any other reusable artifact such as a framework: a workable process must be established between those who build the reusable artifact and those who use it. Requirements have to flow in one direction, and a finished, stable, tested and documented product has to be delivered in the other direction. Also, using DSLs can be a fundamental change for all involved, especially the domain experts. In this chapter we provide some guidelines for the process.*

### 6.1 DSL Development

#### 6.1.1 Requirements for the Language

How do you find out what your DSL should express? What are the relevant abstractions and notations? This is a non-trivial issue; in fact it is one of the key issues in developing DSLs. It requires a lot of domain expertise, thought and iteration. The core problem is that you’re trying not just to understand one problem, but rather a *class* of problems. Understanding and defining the extent and nature of this class of problems can be a lot of work. There are three typical fundamentally different cases.

The first one conerns technical DSLs where the source for a language is often an existing framework, library, architecture or architectural pattern (the inductive approach). The knowledge often already exists, and building the DSL is mainly about factoring the knowledge into a language: defining a notation, putting it into a formal language, and building generators to generate (parts of) the implementation code. In the process, you often also want to put in place reasonable defaults for some of the framework features, thereby increasing the level of abstraction and making framework use easier.

**mbeddr C:** This was the approach taken by the extensible C case study. There is a lot of experience in embedded software development, and some of the most pressing challenges are the same throughout the industry. When the DSL was built, we talked to expert embedded software developers to find out what these central challenges were. We also used an inductive approach and looked at existing C code to indentify idioms and patterns. We then defined extensions to C that provided linguistic abstractions for the most important patterns and idioms. J

The second case addresses business domain DSLs. There you can often mine the existing (tacit) knowledge of domain experts (deductive approach). In domains like insurance, science or logistics, domain experts are absolutely capable of precisely expressing domain knowledge. They do it all the time, often using Excel or Word. Other domain artifacts can also be exploited in the same way: for example, hardware structures or device features are good candidates for abstractions in the respective domains. So are existing user interfaces: they face users directly, and so are likely to contain core domain abstractions. Other sources are standards for an industry, or training material. Some domains even have an agreed ontology containing concepts relevant to that domain and recognized as such by a community of stakeholders. DSLs can be (partly) derived from such domain ontologies.

**Pension Plans:** The company for which the pension DSL was built had a lot of experience with pension plans. This experience was mostly in the heads of (soon to be retiring) senior domain experts. They also already had the core of the DSL: a "rules language". The people who defined the pension plans would write rules as Word documents to "formally" describe the pension plan behavior. This was not terribly productive because of the missing tool support, but it meant that the core of the DSL was known. We still had to run a long series of workshops to work out necessary changes to the language, clean up loose ends and discuss modularization and reuse in pension plans. J

In the two cases discusses so far, it is pretty clear how the DSL is going to look in terms of core abstractions; discussions will be about details, notation, how to formalize things, viewpoints, partitioning and the like (although all these can be pretty nontrivial too!).

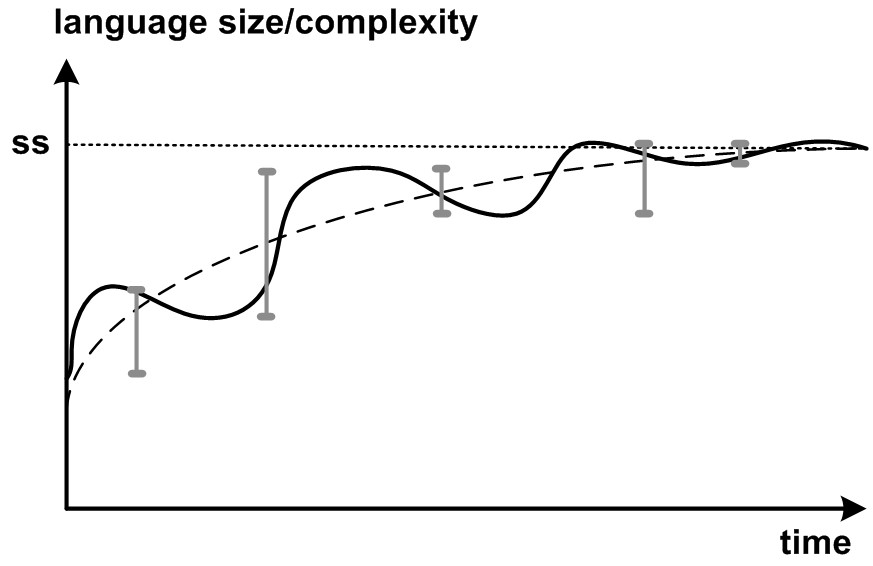
|  |
| --- |
| be knowledgeable, but they might be unable to conceptualize their domain in a structured way – it is then the job of the language designer to provide the structure and consistency that is necessary for defining a language. Co-evolving language and concepts (see below) is a successful technique, especially in this case.  **Refrigerators:** At the beginning of the project, all cooling algorithms were implemented in C. Specifications were written in Word documents as prose (with tables and some physical formulas). It was not really clear at the beginning what the right abstraction level would be for a DSL suitable for the thermodynamics experts. It took several iterations to settle on the asynchronous, state-based structure described earlier. J  For your first DSL, try to catch case one or two. Ideally, start with case one, since the people who build the DLSs – software architects and developers – are often the same as the domain experts.  *6.1.2 Iterative Development*  Some people use DSLs as an excuse to reintroduce waterfall processes. They spend months and months developing languages, tools and frameworks. Needless to say, this is not a very successful approach. You need to iterate when developing the language.  Start by developing some deep understanding of a small part of the domain for which you build the DSL. Then build a little bit of language, build a little bit of generator and develop a small example model to verify what you just did. Ideally, implement all aspects of the language and processor for each new domain requirement before focusing on new requirements[[1]](#footnote-1). |

In the remaining third case, however, we are not so lucky. If no domain knowledge is easily available we have to do an actual domain analysis, digging our way through requirements, stakeholder "war stories" and existing applications. People may in this case is to build "straw.

Novices to DSLs especially tend to get languages and meta models wrong because they are not used to "thinking meta". You can avoid this pitfall by immediately trying out your new language feature by building an example model and developing a compatible generator to verify that you can actually generate the relevant artifacts.

**Refrigerators:** To solidify our choices regarding language abstractions, we prototypically implemented several example refrigerators. During this process we found the need for more and more language abstractions. We noticed early on that we needed a way to test the example programs, so we implemented the interpreter and simulator relatively early. In each iteration, we extended the language as well as the interpreter, so the domain experts could experiment with the language even though we did not yet have a C code generator. J

It is important that the language approaches some kind of stable state over time (Fig. 6.1). As you iterate, you will encounter the following situation: domain experts express requirements that may sound inconsistent. You add all kinds of exceptions and corner cases to the language. You language grows in size and complexity. After a number of these exceptions and corner cases, ideally the language designer will spot the systematic nature behind them and refactor the language to reflect this deeper understanding of the domain. Language size and complexity is reduced. Over time, the amplitude of these changes in language size and complexity (the error bars in Fig. 6.1) should become smaller, and the language size and complexity should approach a stable level (*ss* in Fig. 6.1).



**Component Architecture:** A nice example of spotting a systematic nature behind a set of special cases was the introduction of data replication as a core abstraction in the architecture DSL (we also discuss this in Section 18). After modeling a number of message-based communication channels, we noticed that the interfaces all had the same set of methods, just for different data structures. When we finally saw the pattern behind it, we created new linguistic abstractions: data replication. J

#### 6.1.3 Co-evolve Concepts and Language

In cases in which you perform a real domain analysis, i.e. when you have to find out which concepts the language should contain, make sure you evolve the language in real-time as you discuss the concepts.

Defining a language requires formalization. It requires becoming very clear and unambiguous about the concepts that go into the language. In fact, building the language, because of the need for formalization, helps you become clear about the domain abstractions in the first place. Language construction acts as a catalyst for understanding the domain! I recommend actually building a language in real-time as you analyze your domain.

**Refrigerators:** This is what we did in the cooling language. Everybody learned a lot about the possible structure of refrigerators and the limited feature combinations (based on limitations imposed by the way in which some of the hardware devices work). J

To make this feasible, your DSL tool must be lightweight enough to support language evolution during domain analysis workshops. Turnaround time should be minimal.

**Refrigerators:** The cooling DSL is built with Xtext. Xtext allows very fast turnaround regarding grammar evolution, and, to a lesser extent, scopes, validation and type systems. We typically evolved the grammar in real-time, during the language design workshops, together with the domain experts. We then spent a day offline finishing scopes, constraints and the type system, as well as the interpreter. J

#### 6.1.4 Let People Do What They are Good At

DSLs offer a chance to let everybody do what they are good at. There are several clearly defined roles, or tasks, that need to be done. Let me point out two, specifically.

Experts in a specific target technology can dig deep into the details of how to efficiently implement, configure and operate that technology. They can spend a lot of time testing, digging and tuning. Once they have found out what works best, they can put their knowledge into platforms and execution engines, efficiently spreading the knowledge across the team. For the latter task, they will collaborate with generator experts and language designers – our second example role.

**Component Architecture:** In building the language, an OSGi expert was involved in building the generation templates. J

The language designer works with domain experts to define abstractions, notations and constraints to capture domain knowledge accurately. The language designer also works with the architect and the platform experts in defining code generators or interpreters. Be aware that language designers need to have some kind of predisposition: not everybody is good at "thinking meta", some people are comfortable with concrete work. Make sure you use "meta people" to do the "meta work". And of course, the language designer must be fluent with the DSL tool used in the project.

The flip side is that you have to make sure that you actually have people on your team who are good at language design, know the domain and understand the target platforms, otherwise the benefits promised by using DSLs may not materialize.

#### 6.1.5 Domain Users vs. Domain Experts

When building business DSLs, people from the domain can play two different roles. They can either participate in the domain analysis and the definition of the DSL itself, or they can use the DSL to create domain-specific models or programs.

It is useful to distinguish these two roles explicitly. The first role (language definition) must be filled by a domain *expert*. These are people who have typically been working in the domain for a long time, often in different roles, and who have a deep understanding of the relevant concepts, which they are able to express precisely and maybe even formally. The second group of people are the domain *users*. They are of course familiar with the domain, but they are typically not as experienced as the domain experts.

This distinction is relevant because you want to work with the domain *experts* when defining the language, but you want to build a language that is suitable for use by the domain *users*. If the experts are too far ahead of the users, the users might not be able to "follow", and you will not be able to roll out the language to the actual target audience.

Hence, make sure that when defining the language that you actually cross-check with real domain users whether they are able to work with the language.

**Pension Plans:** The core domain abstractions were contributed by Herman. Herman was the most senior pension expert in the company. In workshops we worked with a number of other domain users who didn’t have as much experience. We used them to validate that our DSL would work for the average future user. Of course they also found actual problems with the language, so they contributed to the evolution of the DSL beyond just acting as guinea pigs. J

#### 6.1.6 DSL as a Product

The language, constraints, interpreters and generators are usually developed by one (smaller) group of people and used by another (larger) group of people. To make this work, consider the "language stuff" as a product developed by one group for use by another. Make sure there’s a well-defined release schedule, that development happens in short, predefined increments, that requirements and issues are reported and tracked, errors are fixed reasonably quickly, there is ample documentation and that support staff is available to help with problems and the unavoidable learning curve. These things are critical for acceptance!

A specific best practice is to exchange people: from time to time, make application developers part of the language team so that they can appreciate the challenges of "meta", and make people from the language development team participate in actual application development to make sure they understand if and how their work products suit the people who do the actual application development.

**mbeddr C:** One of our initial proof-of-concept projects didn’t really work out very well. So in order to try out our first C extensions and come up with a showcase for an upcoming exhibition, the language developers built the proof-ofconcept themselves. As it turned out, this was really helpful. We didn’t just find a lot of bugs, we also experienced first-hand some of the usability challenges of the system at the time. It was easy for us to fix, because it was we who experienced the problems in the first place. J

#### 6.1.7 Documentation is still necessary

Building the DSLs and execution engines is not enough to make the approach successful. You have to communicate to the users how to use these things in real-world contexts. Specifically, here’s what you have to document: the language structure and syntax, how to use the editors and the generators, how and where to write manual code and how to integrate it into generated code, as well as platform/framework decisions (if applicable).

Keep in mind that there are other media than paper. Screencasts, videos that show flip chart discussions, or even a regular podcast that talks about how the tools change are good choices, too. Also keep in mind that hardly anybody reads reference documentation. If you want to be successful, make sure the majority of your documentation consists of example-driven or task-based tutorials.

**Component Architecture:** The documentation for the component architecture DSL contains a set of example applications. Each of them guides a new user through building an increasingly complex application. It explains installation of the DSL into Eclipse, concepts of the target architecture and how they map to language syntax, use of the editor and generator, as well as how to integrated manually written code into the generated base classes. J

### 6.2 Using DSLs

#### 6.2.1 Reviews

A DSL limits the user’s freedom in some respect: they can only express things that are within the limits of DSLs. Specifically, low-level implementation decisions are not under a DSL user’s control because they are handled by the execution engine.

However, even with the nicest DSL, users can still make mistakes, the DSL users can still misuse the DSL – the more expressive the DSL, the bigger this risk. So, as part of your development process, make sure you perform regular model reviews. This is critical, especially for the adoption phase, when people are still learning the language and the overall approach.

Reviews are easier on the DSL level than on the code level. Since DSL programs are more concise and support better separation of concerns than their equivalent specification in GPL code, reviews become more efficient.

If you notice recurring mistakes, things that people do in the

"wrong" way regularly, you can either add a constraint check

that detects the problem automatically, or (maybe even better) consider this as input to your language designers: maybe what the users expect is actually correct, and the language needs to be adapted.

#### 6.2.2 Compatible Organization

Done right, using DSLs requires a lot of cross-project work. In many settings the same language (module) will be used in several projects or contexts. While this is of course a big plus, it also requires that the organization is able to organize, staff, schedule and pay for cross-cutting work. A strictly projectfocused organization will have a very hard time finding resources for these kinds of activities. DSLs, beyond the small ad-hoc utility DSL, are very hard to introduce into such environments.

In particular, make sure that the organizational structure, and the way project cost is handled, is compatible with crosscutting activities. Any given project will not invest in assets that are reusable in other projects if the cost for developing the asset is billed only to the particular project. Assets that are useful for several projects (or the company as a whole) must also paid for by those several projects (or the company in general).

*6.2.3 Domain Users Programming?*

Technical DSLs are intended for use by programmers. Application domain DSLs are targeted towards domain users, nonprogrammers who are knowledgeable in the domain covered by the DSL. Can they actually work with DSLs?

In many domains, usually those that have a scientific or mathematical flavor, users can precisely describe domain knowledge. In other domains you might want to aim for a somewhat lesser goal. Instead of expecting domain users and experts to independently specify domain knowledge using a DSL, you might want to pair a developer and a domain expert. The developer can help the domain expert to be precise enough to "feed" the DSL. Because the notation is free of implementation clutter, the domain expert feels much more at home than when staring at GPL source code.

Initially, you might even want to reduce your aspirations to the point where the developer does the DSL coding based on discussions with domain users, then showing them the resulting model and asking confirming or disproving questions about it. Putting knowledge into formal models helps you

point out decisions that need to be made, or language extensions that might be necessary.

If you are not able to teach a business domain DSL to the domain users, it might not necessarily be the domain users’ fault. Maybe your language isn’t really suitable to the domain. If you encounter this problem, take it as a warning sign and consider changing the language.

#### 6.2.4 DSL Evolution

A DSL that is successfully used will have to be evolved. Just as for any other software artifact, requirements evolve over time and the software has to reflect these changes. In the context of DSLs, the changes can be driven by several different concerns:

|  |  |
| --- | --- |
|  |  |
| *Domain Changes* As the domain evolves, it is likely that the language has to evolve as well3. The problem then is: what do |  |
| you do with existing models? You have two fundamental options: keep the old language and don’t change the models, or evolve the existing models to work with the new (version of the) language. The former is often not really practical, especially in the face of several such changes.  The amount of pain in evolving existing models depends |  |
| a lot on the nature of the change4. The most pragmatic ap- |  |
| proach keeps the new version of the language backward compatible, so that existing models can still be edited and processed. Under this premise, adding new language concepts is never a problem. However, you must never just delete existing concepts or change them in an incompatible way. Instead, these old concepts should be marked as *deprecated*, and the editor will show a corresponding warning in |  |

*Target Platform Changes* The target platform may change because of the availability of new technologies that provide better performance, scalability or usability. Ideally, no changes to either the language or the models are necessary: a new execution engine for the changed target platform can be created. In practice it is not always so clean: the DSL may make assumptions about the target platform that are no longer true for the changed or new platform. These may have to be removed from the languages and existing models. Also, the new platform may support different execution options, and the existing models do not contain enough information to make the decision of which option to take. In this case, additional annotation models may become necessary2.

the IDE. The IDE may also provide a quick fix to change the old, deprecated concept to a new (version of the) concept, if such a mapping is straightforward. Otherwise the migration must be done manually. If you have access to *all* models, you may also run a batch transformation during a quiet period to migrate them all at once. Note that, although deprecation has a bad reputation from programming languages from which deprecated concepts are never removed, this is not necessarily comparable to DSLs: if, after a while, people still use the deprecated concepts, you can have the IDE send an email to the language developers, who can then work with the "offending user" to migrate the programs.

Note that for the above approach to work, you have to have a structure process for versioning the languages and tools, otherwise you will quickly end up in version chaos.

*DSL Tool Changes* The third change is driven by evolution of the DSL tool. Of course, the language definition (and potentially, the existing models) may have to evolve if the DSL tool changes in an incompatible way (which, one could argue, it shouldn’t!). This is similar to every other tool, library or framework you may use. People seem particularly afraid of the situation in which they have to switch to a completely new DSL tool because the current one is no longer supported, or a new one is just better. Of course it is very likely that you’ll have to completely redo the language definitions: there is no portability in terms of language definitions among DSL tools (not even among those that reside on Eclipse). However, if you had designed your languages well you will probably be able to *automatically transform existing models* into the new tool’s data structures[[2]](#footnote-2).

One central pillar of using DSLs is the high degree to

which they support separation of concerns and the expression of domain knowledge at a level of abstraction that makes

the domain semantics obvious, thus avoiding complex reverse engineering problems. Consequently you can generate all kinds of artifacts from the models. This characteristic also means that it is relatively straightforward to write a generator that creates a representation of the model in a new tool’s data structures[[3]](#footnote-3).

#### 6.2.5 Avoiding Uncontrolled Growth and Fragmentation

|  |  |
| --- | --- |
| The organizational approach requires putting in place governance structures for language development. Maybe developers have to coordinate with a central entity before they are "allowed" to define a new language. Or an open-source like model is used, in which languages are developed in public and the most successful ones will survive and attract contributions. Maybe you want to limit language development to some |  |
| central "language team"8. Larger organizations in which un- |  |
| controlled language growth and fragmentation might become a problem are likely to already have established processes for coordinating reusable or cross-cutting work. You should just plug into these processes.  The technical approach (which should be used together with the organizational one) exploits language modularization, extension and composition. If (parts of) languages can be reused, the drive to develop something completely new (that does more or less the same as somebody else’s language) is reduced. Of course this requires that language reuse actually works with your tool of choice. It also requires that the potentially reusable languages are robust, stable and documented – otherwise nobody will use them. In a large organization I would assume that a few languages will be strategic: aligned with the needs of the whole organization, well-designed, well tested and documented, implemented by a central group, used by many de- |  |
| velopers and reusable by design9. In addition, small teams may |  |
| decide to develop their own smaller languages or extensions, reusing the strategic ones. Their focus is much more local, and the development requires much less coordination. |  |

If you use DSLs successfully, there may be the danger of uncontrolled growth and diversification in languages, with the obvious problems for maintenance, training and interoperability7. To avoid this, there is an organizational approach.

1. . [↑](#footnote-ref-1)
2. . [↑](#footnote-ref-2)
3. . [↑](#footnote-ref-3)