**Part IV**

# DSLs in Software Engineering

dsl engineering 441

This part of the book looks at how DSLs can be used in various aspects of software engineering. In particular, we look at requirements engineering, software architecture, developer utilities, implementation, product line engineering and business DSLs. Some of the chapters also serve as case studies for interesting, non-trivial DSLs.

Note that this part has many contributory authors, so there may be slight variations in style.

*17*

## DSLs and Requirements

*This chapter looks at the role of DSLs in requirements engineering. In particular it explores the use of DSLs to specify requirements formally, at ways of representing requirements as models and at traceability between implementation artifacts and requirements.*

*17.1 What are Requirements?*

Wikipedia defines a requirements as follows:

A requirement is a singular documented need of what a particular product or service should be or perform.

Wiktionary says:

[A requirement] specifies a verifiable constraint on an implementation that it shall undeniably meet or (a) be deemed unacceptable, or (b) result in implementation failure, or (c) result in system failure.

In my own words I would probably define a requirement as

. . . a statement about what a system should do, and with what quality attributes, without presupposing a specific implementation.

Requirements are supposed to tell the programmers what the system they are about to implement should do[[1]](#footnote-1). Require-

ments are a means of communicating from humans (people who know what the system should do) to other humans (those that have to implement it). Of course, as well all know, there are a number of challenges with this:

implement some functionality: architecture, design, the use of patterns and idioms and the choice of a suitable implementation technology and language are up to the developer.

* Those who implement the requirements may have a different background than those who write them, making misunderstandings between the two groups likely.

|  |  |
| --- | --- |
|  |  |
| • Usually requirements are written in plain English (or whatever language you prefer). Writing things down precisely and completely in a non-formal language is next to impos- |  |
| sible3. |  |
| Traditional requirements documents are a means to communicate from people to people. However, in the end this is not really true. In an ideal world, the requirements (in the brain of the person who writes them down) should be communicated directly to the computer, without the intermediate programmer, to avoid the misunderstandings mentioned above. If we look at the problem in this way, requirements now become formal, computer-understandable.  Wikipedia has a nice list of characteristics that requirements should posses. Here is a slightly adapted version of this list:  *Complete* The requirement is fully stated in one place with no missing information. This makes the requirement easy to consume, because readers do not have to build the complete picture from scattered information. |  |
| *Consistent* The requirement does not contradict any other requirement and is fully consistent with all authoritative external documentation4.  *Cohesive & Atomic* The requirement is atomic, i.e., it does not contain conjunctions5. This ensures that traceability from |  |
| implementation artifacts back to the requirements is relatively simple.  *Current* The requirement has not been made obsolete by the |  |
| passage of time. Outdated requirements should be removed or marked as outdated.  *Feasible* The requirement can be implemented within the constraints of the project6. |  |

* Those who write the requirements may not actually really know what they want the system to do, at least initially. Requirements change over the course of a project, particularly as people start to "play" with early versions of the system2.

*Unambiguous* The requirement is concisely stated without recourse to technical jargon, acronyms (unless defined else-

to address the requirement. This is one reason why interaction with the implementers is critical.

|  |  |
| --- | --- |
|  |  |
| *Mandatory* The requirement represents a stakeholder-defined characteristic the absence of which will result in a deficiency that cannot be ameliorated. An optional requirement is a |  |
| contradiction in terms8. |  |
| *Verifiable* The implementation of the requirement can be determined through one of four possible methods: inspection, demonstration, test or analysis. Otherwise it is hard to tell if a system actually fulfills a requirement or not9. |  |
| If requirements are written as pure prose, then making sure all these characteristics are met boils down mostly to a manual review process. Of course, this is tedious and error-prone, and requirements end up in the sorry state we all know.  To get one step better, you can use controlled natural lan- |  |
| guage10 in which words like *must*, *may* or *should* have a well |  |
| defined meaning and are used consciously. Using tables and – to some extent – state machines, is also a good way to make some of the data less ambiguous; these formalisms also help to verify requirements for consistency and completeness. To manage large sets of requirements, tools should be used to support unique identification and naming of requirements, as well as the expression of relationships and hierarchies among re- |  |
| quirements11 However, the requirements themselves are still |  |
| expressed as plain text, so the fundamental problems mentioned above are not improved significantly.  In this chapter we will give you some ideas and examples |  |
| on how this situation can be improved with DSLs12. |  |

where in the requirements document), or other esoteric verbiage. It expresses objective facts, not subjective opinions. It is subject to one and only one interpretation. Vague subjects, adjectives, prepositions, verbs and subjective phrases are avoided. Negative statements and compound statements are prohibited7.

### 17.2 Requirements versus Design versus Implementation

Traditionally, we try to establish a clear line between requirements, architecture and design, and implementation. For example, a requirement may state that the system be 99.99% reliable. The design may use hot-standby and fail-over to continue service if a component breaks. The implementation would then

chapter that *all* requirements should be captured with DSLs. Instead, DSLs can be one important ingredient for a well thought out requirements management approach.

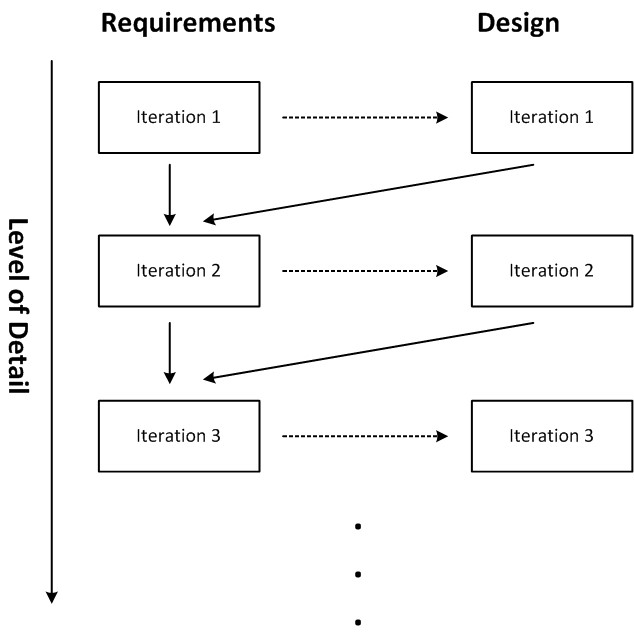
|  |  |
| --- | --- |
| ing to a separation between requirements and architecture that is driven by organizational constraints14: the OEM writes the |  |
| requirements, a systems integrator does the architecture, and some outsourcing company does the coding. In such a scenario it is important to draw precise boundaries between the activities. However, in some sense the boundaries are arbitrary. Consequently, the distinction between requirements and architecture are arbitrary as well: we could just as well state the following:  *Requirement* The system shall be 99.99% reliable by using hotstandby and fail-over to continue service if something breaks.  *Architecture/Design* We use two application servers running on two machines, using the XYZ messaging middleware as a replication engine for the hot-standby. We use a watchdog for detecting if the primary machine breaks, so we can fail over to the second one.  *Implementation* . . . all the code that is necessary to implement the design above.  From software development we know that it is very hard to get requirements right. In the real world, you have to elaborate |  |
| on the requirements incrementally15. In systems engineering |  |

select a specific standby/fail-over technology to realize the design. We make this distinction because we want to establish different roles in the software engineering process. For example, product management writes the requirements, a systems architect comes up with the architecture and design, and then a programmer writes the actual code and chooses the technologies13. Different organizations may even be involved, lead-

this approach is also very well established. For example, when satellites are built, the scientists come up with initial scientific requirements, for example, regarding the resolution a satellitebased radar antenna looking at the earth should have. Let’s look at some of the consequences:

* A given resolution requires a specific size of the antenna, and a specific amount of energy being sent out. (Actually, the two influence each other as well).
* A bigger antenna results in a heavier satellite, and more radar energy requires more solar panel area – increasing the size and weight even further.
* At some point, the size and weight of the satellite cannot be further increased, because a given launch vehicle reaches its limits – a different launch vehicle might be required.
* A bigger launch vehicle will be much more expensive, or you might have to change the launch provider. For example, you might have to use a Soyuz instead of an Ariane.

|  |  |
| --- | --- |
| • A Soyus launched at Baikonur cannot reach the same orbits as an Ariane launched from Courou, because of the higher latitude. As a consequence, the satellite might be "further away" from the area you want to inspect with your radar, |  |
| neglecting the advantages gained by the bigger antenna16. |  |
| . . . and this has just looked at size and weight! Similar problems exist with heat management, pointing accuracy or propulsion. As you can see, a change in any particular requirement can lead to non-trivial consequences you will only detect if you think about the *implementation* of the requirement. A strictly sequential approach (first write all the requirements, then think about the implementation) will not work. So what do the systems engineers do? They come up with a model of the satellite. Using mathematical formulas, they describe how the different properties discussed above relate. These might be approximations or based on past experience – after all, the real physics can be quite complex. They then run a *trade-off analysis*. In other words, they change the input values until a workable compromise is reached. Usually this is a manual process, but sometimes parts of it can be automated.  This example illustrates three things. First, requirements elicitation is incremental. Second, models can be a big help to precisely specify requirements and then "play" with them. And third, the boundary between requirements and design is blurred, and the two influence each other. Fig. 17.1 shows a multi-step approach to requirements definition, intertwined with incrementally more refined designs.  *17.3 Using DSLs for Requirements Engineering*  So here is the approach for using DSLs we suggest: identify a couple of core areas of the system to be built that lend them- |  |
| selves to specification with a formal language17. Then develop |  |
| one or more DSLs to express these areas and use them to describe the system. The rest of the system – i.e., the areas for |  |

.

|  |  |
| --- | --- |
| which a DSL-based description makes no sense – is described |  |
| textually, with the usual tools18. |  |
| Once a suitable DSL has been found and implemented, those people who have the requirements in mind can *directly* express them – the lossy human-to-human communication is no longer a problem. Various constraint checks, tests and simulations can be used to allow the requirements owners to "play" with the requirements models to see if they really express what they had in mind.  Of course there is one significant caveat: we first have to build this DSL. So how do we go about that? We could have somebody write prose requirements and hand them over to the DSL developer . . . back to square one!  There is a much better approach, though. Since today’s language workbenches support extremely rapid prototyping, you can actually build the DSLs interactively with the requirements owner. Since you are not capturing the specific require- |  |
| ments, but rather try to define how specific requirements are described, you essentially perform a domain analysis: you try to understand the degrees of freedom in the domain to be able to represent the domain with the DSL19. Here is the process I |  |

have use successfully many times:

1. Have the requirements owner explain some particular aspect of the domain.

which makes it executable, in the sense that you can always turn around and have the requirements owner try to express specific requirements with the DSL you’re building, verifying the suitability of the DSL.

1. Try to understand that aspect and change your DSL so it can express that aspect.
2. Make the requirements owner try to express a couple of specific, but representative, requirements with the DSL.
3. Most likely you will run into problems, some things cannot be expressed with the DSL. If so, go back to 1 and reiterate. A complete iteration should take no more than 60 minutes.
4. After half a day, stop working with the requirements owner and clean up/refactor the DSL.
5. Start another of the language design sessions with the requirements owner and iterate – over time, you should get closer to *the* DSL for the domain.

Once the DSL is finished, the requirements owners will be able to express domain requirements without involvement of the software developers.

This approach to requirements engineering is very close to regular DSL usage. We identify an aspect of the domain that lends itself to formalization, iteratively build a language, and then let the domain experts – who are the requirements owners for many of the business requirements – express the system directly. The *classical* requirements document is gone20.

Using DSLs to specify (some parts of the) requirements formally helps achieve some of the desirable characteristics for requirements discussed above. The following lists only those characteristics for which DSLs make a difference.

*Consistent* Consistency is enforced by the language. If the DSL is crafted correctly, no inconsistent requirements can be expressed.

*Feasible* Specific requirements are checked for feasibility by being expressible with the DSL: they are within the scope of what the DSL – hence, the domain for which we write the requirements – is intended.

*Unambiguous* A description of requirements – or application functionality in general – with a DSL always unambiguous, provided the DSL has well-defined semantics.

*Verifiable* Constraints, tests, verification or simulation can be used to verify the requirements regarding various properties. Inspection and review is simplified, because DSL programs are less verbose than implementation code, and clearer than prose.

### 17.4 Integration with Plain Text Requirements

You will probably not be able to describe all the requirements of a system using the approach described above. There will always be aspects that cannot be formalized, or that are so specific that the effort for building a DSL does not pay off. You therefore have to find some way of integrating plain text requirements with DSL code. Here are some approaches to how this can be done.

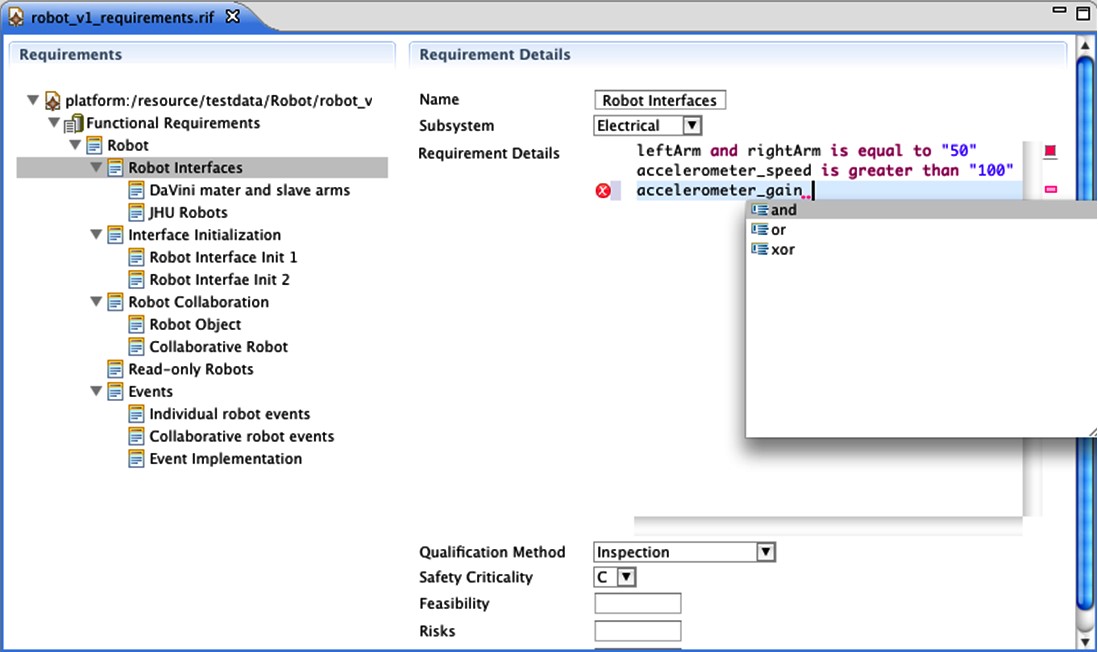
#### 17.4.1 Embedding DSL Code in a Requirements Tool

One approach is to mix prose requirements with formalized, DSL-based requirements. We show examples with Xtext and MPS.

|  |  |
| --- | --- |
| *Xtext Example* Eclipse-based tooling for requirements en- |  |
| gineering is being developed as part of the VERDE21 and ProR22 |  |
| research projects. This includes Eclipse RMF23, a "classical" |  |
| requirements engineering tool, in which textual requirements are classified, structured and put into relationships with each |  |
| other24. The requirements structure is represented as an EMF |  |
| model, to make integration with other model-based artifacts simple. In addition to plain text, requirements can have parameters with well-defined types. The types of these parameters can be primitive (string, int), but they can also be any other Ecore meta class, so any additional model structure can be embedded into a requirement. Integration with Xtext is available, which provides textual concrete syntax for these data structures. In other words, it is possible to enrich prose requirements specifications with additional formal specifications expressed in arbitrary DSLs. Fig. 17.2 shows a screenshot. |  |

*MPS Example* We have built a similar solution for MPS in the mbeddr project. The solution supports collecting trees of requirements, where each requirement has an ID, a kind and a short summary. Fig. 17.3 shows an example. In addition, the one-line summary can be expanded to reveal additional details (Fig. 17.4). There users can enter a detailed prose description, as well as additional constraints among requirements (**requires also**, **conflicts with**.) In the **Additional Specifications** section, users can enter arbitrary DSL programs: since MPS supports language modularization and composition (Section 16.2), embedding arbitrary languages with

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | additional specification with a particular requirements kind. This means that if a requirement has a particular kind, the additional data associated with that kind must be present in the |  | | **Additional Specifications** section25. |  | | *17.4.2 Requirements Traceability*  Requirements traceability establishes links, or traces, between implementation (or design or test) artifacts and requirements. This allows each (part of) an artifact to be traced back to the |  | | requirements that drive the artifact26. Once such pointers are |  | |

arbitrary syntax into the requirements language is trivial and works out of the box. It is also possible to associate a specific Figure 17.2: An Xtext DSL embedded in a requirements engineering tool based on the Eclipse Requirements Modeling Framework (RMF).

available, various analyses become possible. For example, it is easy to find out whether each requirement has been implemented (or tested), and we know which implementation artifacts may have to change if a requirement changes.

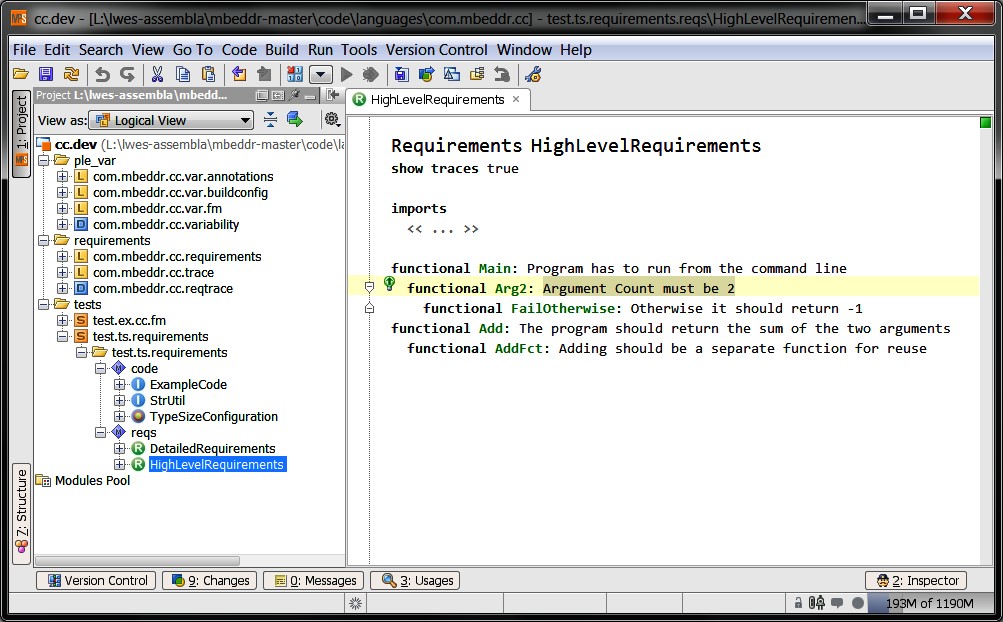
Requirements traceability has two challenges. The first one is social, the second one is technical. The social problem is that, while traces are easy to analyze once they are available, they still have to be established manually. This requires discipline by the people, typically developers, whose job it is to establish the traces.

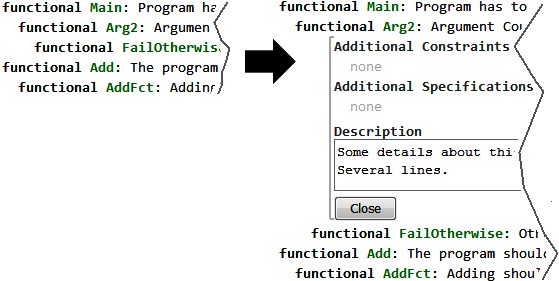
The technical problem addresses how to actually establish the pointers technically. In a world in which requirements –

various different relationships can be established.

as well as design, implementation and test artifacts – are all model-based, establishing these pointers becomes trivial. In mixed environments with many different tools built on many different foundations, it can become arbitrarily complicated.

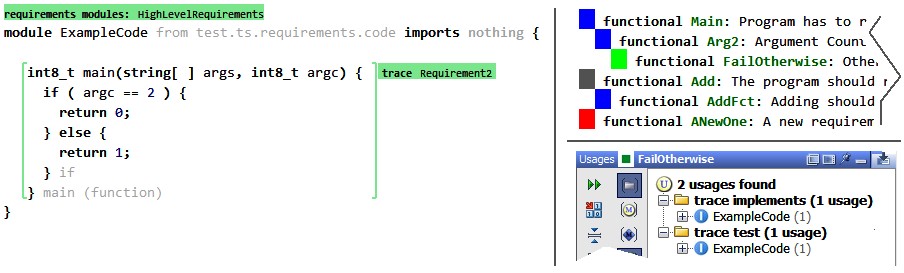
|  |
| --- |
| Again, we show tooling examples for Xtext and MPS. |

F



*Xtext Example* The VERDE project mentioned above also develops a traceability framework for Eclipse. Various trace kinds can be defined, which can subsequently be used to establish links between arbitrary EMF-based models[[2]](#footnote-2). The trace-

|  |  |
| --- | --- |
| ability links are kept external to the actual models, so no mod- |  |
| ifications to existing meta models or languages are required28. |  |
| *MPS Example* As part of mbeddr, we have implemented a generic tracing framework base on MPS’ language annotations (discussed in Section 16.2.7). Arbitrary models – independent |  |
| of the language used to create them29 – can be annotated with |  |
| traceability links, as shown in the left part of figure 17.5.  A context menu action adds a new trace to any model element: **Ctrl-Space** allows the selection of one or more requirements at which to point. Each trace has a **kind**. The traced requirements are color coded to reflect their trace status (Fig. 17.5, top right). Finally, the **Find Usages** functionality of MPS has been customized to show traces directly (Fig. 17.5, bottom right). |  |



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