MARKUS VOELTER, VOELTER@ACM.ORG

DSLS IN SOFTWARE ENGINEERING

DSLs and Software Architecture

What is Software Architecture

Definitions

Software architecture has many definitions by various groups of people. Here are a few. Wikipedia defines software architecture as *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships between them.*

This is a classic definition that refers to the structure of a software system, observable by analyzing an existing system. It emphasizes the structure (as opposed to the behavior) and focusses on the coarse grained building blocks found in the system. A definition by Boehm builds on this:

A software system architecture comprises

- *A collection of software and system components, connections, and constraints.*
- *A collection of system stakeholders' need statements.*
- *A rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements.*

In addition to the structure, this definition also emphasizes the relevance of the stakeholders and their needs, it ties the structure of the system to what the system is required to do.

Hayes-Roth introduce another concern: *The architecture of a complex software system is its style and method of design and construction.* Instead of looking at the structures, they emphasize that there are different architectural styles and they emphasize the "method of design and construction". Eoin Woods takes it one step further: *Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled.* He emphasizes the design decisions that lead to a given system. So he doesn't look at the system as a set of structures, but rather considers the architecture as a process - the design decisions - that lead to a given system.

Let us propose another definition: *Software architecture is everything that needs to be consistent throughout a software system*. This definition is useful because it includes structures and behavior, it doesn't say anything about coarse-grained vs. detailed (after all, a locking protocol is possibly a detail, but it is important to implement it consistently) and it implies that there needs to be some kind of process or method to achieve the consistency.

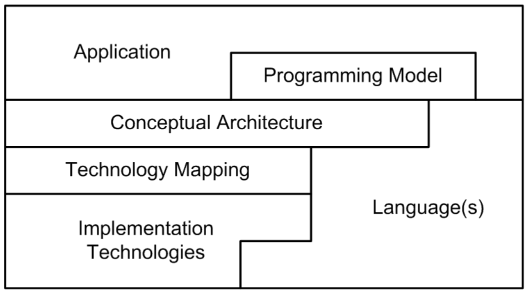


Figure 1: Conceptual structure of a software architecture

Figure 1 looks at software architecture in more detail. At the center of the diagram you can see the conceptual architecture. It defines architectural concepts from which systems are built - we will come back to this in the next section. Examples could include: task, message, queue, component, port, or replicated data structure. Note that these concepts are independent of specific implementation technologies: it is the technology mapping that defines how the architecture concepts are implemented with specific technologies. Separating the two has the advantage that the conceptual discussions aren't diluted by what certain technologies may or may not be able to do. The decision what technology to map to is then driven by non-functional concerns, i.e. whether and how a given technology can provide the required quality of service.

The programming model is the way how the architectural concepts are implemented in a given programming language. Ideally, the programming model should not change as long as the architectural concepts remain stable - even if the technology mapping or the implementation technologies change. Applications are implemented by instantiating the architectural concepts, and implementing them based on the programming model.

We want to reemphasize the importance of the conceptual architecture. When asking people about the architecture of systems, one often gets answers like: "it's a Web service architecture", or "it's an XML architecture" or "it's a JEE architecture". Obviously, all this conveys is that a certain technology is used. When talking about architectures per se, we want to talk about architectural concepts, how they relate to each other. In many cases, fundamental architectural concepts can and should be used to explain these. Only in a second step, a mapping to one or more technologies should be discussed.

Here are some of these fundamental architectural concepts:

Modularize Break big things down into smaller things, so they can be understood (and potentially reused) more easily. Examples: Procedures, Classes, Components, Services, User Stories.

Encapsulate Hide the innards of a module so they can be changed without affecting clients. Examples: Private Members, Facade Pattern, Components, Layers/Rings/Levels.

Contracts Describe clearly the external interface of a module. Examples: Interfaces, Pre/Post Conditions, Protocol State Machines, Message Exchange Patterns, Published APIs.

Decoupling Reduce dependencies in time, data structure or contention. Examples: Message Queues, Eventual Consistency, Compensating Transactions.

Isolate Crosscuts Encapsulate handling of cross-cutting concerns. Examples: Aspect Orientation, Interceptors, Application Servers, Exception Handling.

Separation of Concerns Separate different responsibilities into different modules.

. . . more to be added . . . TODO

Architecture DSLs

Conceptual Architecture and DSLs

So how does all of this relate to DSLs? An Architecture DSL (ADSL) is a language that expresses a system's architecture directly with "directly" meaning that the language's abstract syntax contains constructs for all the ingredients of the conceptual architecture. The language can hence be used to describe a system on architectural level such that no 3GL/GPL source code is required. Code generation is used to generate representations of the application architecture in the implementation language(s), automating the technology mapping (once the decisions about the mapping have been made manually, of course). Finally, the programming model is defined with regards to the generated code plus possibly additional frameworks.

We want to (re)emphasize an important point: we do not advocate the definition of a generic, reusable language such as the various ADLs, or UML (see below). Based on our experience, the approach works best if you define the ADSL in real time as you understand, define and evolve the conceptual architecture of a system! The process of defining the language actually helps the architecture/development team to better understand, clarify and refine the architectural abstractions as the language serves as a (formalized) ubiquitous language that lets you reason and discuss about the architecture.

An Example ADSL

This section contains an example of an Architecture DSL Markus has implemented for a real system together with a customer from the domain of airport management systems on one of his consulting "gigs".

The customer decided they wanted to build a new flight management system. Airlines use systems like these to track and publish information about whether airplanes have landed at airports, whether they are late, the technical status of the aircraft, etc. The system also populates the online-tracking system on the Web and information monitors at airports. This system is in many ways a typical distributed system: there is a central data center to do some of the heavy number crunching, but there's additional machines distributed over relatively large areas. Consequently you cannot simply shut down the whole system, introducing a requirement to be able to work with different versions of parts of the system at the same time. Different parts of the system will be built with different technologies: Java, C++, C#. This is not an untypical requirement for large distributed systems either. Often you use Java technology for the backend, and .NET technology for a Windows front end. They had decided that the backbone of the system would be a messaging infrastructure and they were evaluating different messaging tools for performance and throughput.

When Markus arrives they briefed him about all the details of the system and the architectural decisions they had already made, and then asked me whether all this made sense. It quickly turned out that, while they knew many of the requirements and had made specific decisions about certain architectural aspects, they didn't have a well-defined conceptual architecture. And it showed: when the team were discussing about their system, they stumbled into disagreements about architectural concepts all the time because they had no language for the architecture. Hence, we started building a language.

We would actually build the grammar, some constraints and an editor while we discussed the architecture in a two-day workshop.

We (i.e., the customer and Markus) started with the notion of a component. At that point the notion of components is defined relatively loosely, simply as the smallest architecturally relevant building block, a piece of encapsulated application functionality. We also assume that components can be instantiated, making components the architectural equivalent to classes in OO programming. To enable components to interact, we also introduce the notion of interfaces, as well as ports, which are named communication endpoints typed with an interface. Ports have a direction (provides, requires) as well as a cardinality.

Figure 2: Components, required ports and provided ports

```
component DelayCalculator {
  provides aircraft: IAircraftStatus
  provides managementConsole: IManagementConsole
  requires screens[0..n]: IInfoScreen
}
component Manager {
  requires backend[1]: IManagementConsole
}
component InfoScreen {
  provides default: IInfoScreen
}
component AircraftModule {
  requires calculator[1]: IAircraftStatus
}
```

It is important to not just state which interfaces a component provides, but also which interfaces it requires because we want to be able to understand (and later: analyze with a tool) component dependencies. This is important for any system, but especially important for the versioning requirement. We then looked at instantiation. There are many aircraft, each running an AircraftModule, and there are even more InfoScreens. So we need to express instances of components. Note that these are logical instances. Decisions about pooling and redundant physical instances had not been made yet. We also introduce connectors to define actual communication paths between components (and their ports).

At some point it became clear that in order to not get lost in all the components, instances and connectors, we need to introduce some kind of namespace. It became equally clear that we'd need to distribute things to different files - the tool support ought to make sure

Figure 3: Component instances and their connectors

```
instance dc: DelayCalculator
instance screen1: InfoScreen
instance screen2: InfoScreen
connect dc.screens to (screen1.default, screen2.default)
```

that editor functionality such as Go To Definition and Find References still works.

Figure 4: TODO

```
namespace com.mycompany {
  namespace datacenter {
    component DelayCalculator { ... }
    component Manager { ... }
  }
  namespace mobile {
    component InfoScreen { ... }
    component AircraftModule { ... }
  }
}
```

It is also a good idea to keep component and interface definition (essentially: type definitions) separate from system definitions (connected instances), so we introduced the concept of compositions.

Figure 5: TODO

```
namespace com.mycompany.test {
  composition testSystem {
    instance dc: DelayCalculator
    instance screen1: InfoScreen
    instance screen2: InfoScreen
    connect dc.screens to (screen1.default, screen2.default)
  }
}
```

Of course in a real system, the DelayCalculator would have to dynamically discover all the available InfoScreens at runtime. There is not much point in manually describing those connections. So, we specify a query that is executed at runtime against some kind of naming/trader/lookup/registry infrastructure. It is re-executed every 60 seconds to find the InfoScreens that had just come online.

Figure 6: TODO

```
namespace com.mycompany.production {
  instance dc: DelayCalculator
  dynamic connect dc.screens every 60 query {
    type = IInfoScreen
    status = active
  }
}
```

A similar approach can be used to realize load balancing or fault tolerance. A static connector can point to a primary as well as a backup instance. Or a dynamic query can be re-executed when the currently used component becomes unavailable. To support registration of instances, we add additional syntax to their definition. A registered instance automatically registers itself with the registry, using its name (qualified through the namespace) and all provided interfaces. Additional parameters can be specified, the following example registers a primary and a backup instance for the DelayCalculator.

Figure 7: TODO

```
namespace com.mycompany.datacenter {
  registered instance dc1: DelayCalculator {
    registration parameters {role = primary}
  }
  registered instance dc2: DelayCalculator {
    registration parameters {role = backup}
  }
}
```

Until now we didn't really define what an interface is. We knew that we'd like to build the system based on a messaging infrastructure. Here's our first idea: an interface is a collection of messages, where each message has a name and a list of typed parameters - this also requires the ability to define data structures, but in the interest of brevity, we won't show that. After discussing this notion of interfaces for a while, we noticed that it was too simplistic. We needed to be able to define the direction of a message: does it flow in or out of the port? More generally, which kinds of message interaction patterns are there? We identified several, here are examples of oneway and request-reply:

We talked a long time about various message interaction patterns. After a while it turned out that one of the core use cases for messages

Figure 8: TODO

```
interface IAircraftStatus {
  oneway message reportPosition(aircraft: ID, pos: Position )
  request-reply message reportProblem {
    request (aircraft: ID, problem: Problem, comment: String)
    reply (repairProcedure: ID)
  }
}
```

is to push status updates of various assets out to various interested parties. For example, if a flight is delayed because of a technical problem with an aircraft, then this information has to be pushed out to all the InfoScreens in the system. We prototyped several of the messages necessary for "broadcasting" complete updates, incremental updates and invalidations of a status item. And then it hit us: we were working with the wrong abstraction! While messaging is a suitable transport abstraction for these things, architecturally we're really talking about replicated data structures. It basically works the same way for all of those structures:

- You define a data structure (such as FlightInfo).
- The system then keeps track of a collection of such data structures.
- This collection is updated by a few components and typically read by many other components.
- The update strategies from publisher to receiver always include full update of all items in the collection, incremental updates of just one or a few items, invalidations, etc.

Once we understood that, in addition to messaging, there's this additional core abstraction in the system, we added this to our Architecture DSL and were able to write something like the following. We define data structures and replicated items. Components can then publish or consume those replicated data structures. We state that the publisher publishes the replicated data whenever something changes in the local data structure. However, the InfoScreen only needs an update every 60 seconds (as well as a full load of data when it is started up).

This is much more concise compared to a description based on messages. We can automatically derive the kinds of messages needed for full update, incremental update and invalidation and create these messages in the model using a model transformation. The description also much more clearly reflects the actual architectural intent:

Figure 9: TODO

```
struct FlightInfo {
   //  … attributes …
}

replicated singleton flights {
  flights: FlightInfo[]
}

component DelayCalculator {
  publishes flights { publication = onchange }
}

component InfoScreen {
  consumes flights { init = all update = every(60) }
}
```

it expresses better what we want to do (replicate state) compared to a lower level description of how we want to do it (sending around state update messages). While replication is a core concept for data, there's of course still a need for messages, not just as an implementation detail, but also as a way to express your architectural intent. It is useful to add more semantics to an interface, for example, defining valid sequencing of messages. A well-known way to do that is to use protocol state machines. Here is an example that expresses that you can only report positions and problems once the aircraft is registered. In other words, the first thing an aircraft has to do is register itself.

Initially, the protocol state machine is in the new state. Here, the only valid message is `registerAircraft`. If this is received, we transition into the registered state. In registered, you can either `unregisterAircraft` and go back to new, or receive a `reportProblem` or `reportPosition` message in which case you'll remain in the registered state. We mentioned above that the system is distributed geographically. This means it is not feasible to update all part of the systems (e.g. all InfoScreens or all AircraftModules) in one swoop. As a consequence, there might be several versions of the same component running in the system. To make this feasible, many non-trivial things need to be put in place in the runtime. But the basic requirement is this: you have to be able to mark up versions of components, and you have to be able to check them for compatibility with old versions. The following piece of code expresses that the DelayCalculatorV2 is a new implementation of DelayCalculator. `newImplOf` means that no externally visible aspects change which is why no ports and other externally-exposed details of the component are declared. For all in-

Figure 10: TODO

```
interface IAircraftStatus {
  oneway message registerAircraft(aircraft: ID )
  oneway message unregisterAircraft(aircraft: ID )
  oneway message reportPosition(aircraft: ID, pos: Position )
  request-reply message reportProblem {
    request (aircraft: ID, problem: Problem, comment: String)
    reply (repairProcedure: ID)
  }
  protocol initial = new {
    state new {
      registerAircraft => registered
    }
    state registered {
      unregisterAircraft => new
      reportPosition
      reportProblem
    }
  }
}
```

tents and purposes, it's the same thing - just maybe a couple of bugs are fixed.

Figure 11: TODO

```
component DelayCalculator {
  publishes flights { publication = onchange }
}
newImplOf component DelayCalculator: DelayCalculatorV2
```

If you really want to evolve a component (i.e., change its external signature) you can write it like this:

Figure 12: TODO

```
component DelayCalculator {
  publishes flights { publication = onchange }
}
newVersionOf component DelayCalculator: DelayCalculatorV3 {
  publishes flights { publication = onchange }
  provides somethingElse: ISomething
}
```

The keyword is `newVersionOf` and now you can provide additional features (like the `somethingElse` port) and you remove required ports. You cannot add additional required ports or remove any of the provided ports since that would destroy the "plug-in compatibility". Constraints make sure that these rules are enforced on model level. Using the approach shown here, we were able to quickly get a grip towards the overall architecture of the system. All of the above was actually done in the first day of the workshop. We defined the grammar, some important constraints, and a basic editor (without many bells and whistles). We were able to separate what we wanted the system to do from how it would achieve it: all the technology discussions were now merely an implementation detail of the conceptual descriptions given here (albeit of course, a very important implementation detail). We also had a clear and unambiguous definition of what the different terms meant. The nebulous concept of component has a formal, well-defined meaning in the context of this system. It didn't stop there: the next day involved a discussion of how to actually code the implementation for a component and which parts of the system could be automatically generated.

In this project, as well as in many other ones, we have used textual DSLs. We have argued in this book why textual DSLs are superior in many cases, and these arguments apply here as well. However, we did use visualization to show the relationships between the building blocks, and to communicate the architecture to stakeholders who were not willing to dive into the textual models. Figure 13 shows an example, created with Graphviz.

Architecture DSL Concepts

What we did in a nutshell

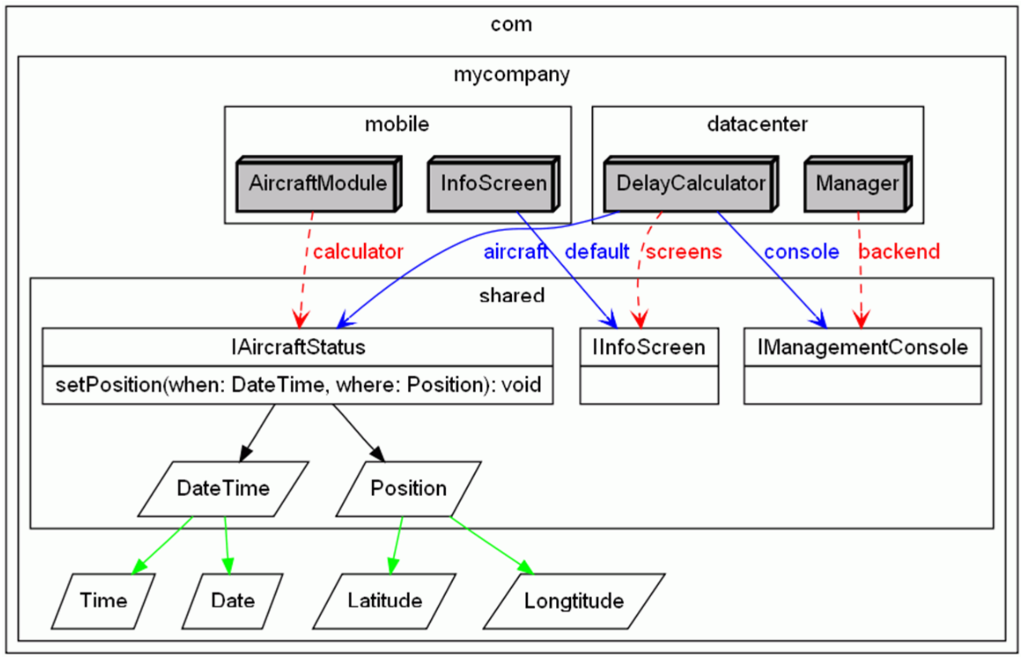
The approach recommends the definition of a formal language for your project's or system's conceptual architecture. You develop the language as the understanding of your architecture grows. The language therefore always resembles the complete understanding about your architecture in a clear and unambiguous way. As we enhance the language, we also describe the application architecture using that language.

Component Implementation

By default, component implementation code is written manually against the generated API code, using well-known composition techniques such as inheritance, delegation or partial classes.

However, there are other alternatives for component implementation that do not use a 3GL/GPL, but instead use formalisms that are specific to certain classes of behavior: state machines, business

Figure 13: TODO



rules or workflows. You can also define and use a domain-specific language for certain classes of functionality in a specific business domain. Note how this connects the dots to business domain DSLs mentioned below.

Be aware that the discussion in this section is only really relevant for application-specific behavior, not for all implementation code. Huge amounts of implementation code are related to the technical infrastructure (e.g., remoting, persistence, workflow) of an application. It can be derived from the architectural models, and generated automatically.

Standards, ADLs and UML

Describing architecture with formal languages is not a new idea. Various communities recommend using Architecture Description Languages (ADLs, for example [ADL1, ADL2, ADL3, ADL4]) or the Unified Modeling Language (UML) to this end. However, all of those approaches advocate using existing, generic languages for specifying architecture, although some of them, including the UML, can be customized to some degree.

Unfortunately, efforts like that completely miss the point. We have not experienced much benefit in shoehorning your architecture description into the (typically very limited as well as too generic) constructs provided by predefined languages - one of the core activities of the approach explained is this paper is the process of actually building your own language to capture your system's conceptual architecture.

So: are standards important? And if so, where?

In order to use any architecture modeling language successfully, people first and foremost have to understand the architectural concepts they are dealing with. Even if UML is used, people will still have to understand the architectural concepts and map them to the language - in the case of UML that often requires an architecture-specific profile. Is a profiled UML still standard?

Note that we do not propose to ignore standards altogether. The tools are built on MOF/EMOF, which is an OMG standard, just like the UML. It is just on a different meta level.

Of course, you can use the approach explained above with UML profiles. We have done this in several projects and my conclusion is that it doesn't work well in most environments. There are several reasons, three of them are:

- Instead of thinking about architectural concepts, working with UML makes you think more about how to use UML's existing constructs to express your architectural intentions. That is the wrong focus! It also takes a lot of effort just to learn UML well enough to make choices that are sensible (enough). In the end, you'll find that UML has very little actual semantics of its own.

- UML is a huge standard and allows quite a bit of variation in how it's used and implemented, so in order to be able to have *processable* models, you'll have to confine your users and yourself to a certain subset and implementation, which takes additional effort.

- UML tools typically don't integrate very well with existing development infrastructure (editors, CVS/SVN, diff/merge). That is a huge problem now models play the role of source code (as opposed to being mere "analysis pictures").

Code Generation

It should have become clear from the paper that the primary benefit of developing the architecture DSL (and using it) is just that: understanding concepts by removing any ambiguity and defining them formally. It helps you understand your system and get rid of unwanted technology interference.

But of course, now that we have a formal model of the conceptual architecture (the language) and also a formal description of system(s) we're building - i.e., the sentences (or models) defined using the language - we might as well use it to do more good:

- We generate an API against which the implementation is coded. That API can be non-trivial, taking into account the various messaging paradigms, replicated state, etc. The generated API allows developers to code the implementation in a way that does not depend on any technological decisions (beyond the precise technical description format of the API): the generated API hides those from the component implementation code. We call this generated API and the set of idioms to use it the programming model.

- Remember that we expect some kind of component container or middleware platform to run the components, so we also generate the code that is necessary to run the components (including their technology-neutral implementation) on the implementation technology of choice. We call this layer of code the technology mapping code (or glue code). It typically also contains a whole bunch of configuration files for the various platforms involved. Sometimes this requires addition "mix-in models" (some call these "marking models") that specify configuration details for the platform. As a side effect, the generators capture best practices in working with the technologies you've decided to use.

It is of course completely feasible to generate APIs for several target languages (supporting component implementation in various languages) and/or generating glue code for several target platforms (supporting the execution of the same component on different middleware platforms). This nicely supports potential multi-platform requirements, and also provide a way to scale or evolve the infrastructure over time.

Another point worth making is that you typically generate in several phases: a first phase uses type definitions (components, data structures, interfaces) to generate the API code so you can code the implementation. A second phase generates the glue code and the system configuration code. As a consequence, it is often sensible to separate type definitions from system definitions in models: they are used at different times in the overall process, and also often created, modified and processed by different people. This shows that language (and model) modularization is useful.

In summary, the generated code supports an efficient and technology independent implementation and hides much of the underlying technological complexity, making development more efficient and less error-prone.

The Role of Patterns

Patterns are an important part of today's software engineering practice. They are a proven way of capturing working solutions to recurring problems, including their applicability, trade-offs and consequences. So how do patterns factor into the approach described above?

Architecture Patterns and Pattern Languages describe blueprints for architectures that have been used successfully. They can serve as an inspiration for building you own system's architecture. Once you have decided on using a pattern (and have adapted it to your specific context) you can make concepts defined in the pattern first class citizens of your DSL. In other words, patterns influence the architecture, and hence the grammar of the DSL.

Design Patterns, as their name implies, are more concrete, more implementation-specific than architectural patterns. It is unlikely that they will end up being central concepts in your architecture DSL. However, when generating code from the models, your code generator will typically generate code that resembles the solution structure of a number of patterns. Note, however, that the generator cannot decide on whether a pattern should be used: this is a tradeoff the (generator) developer has to make manually.

When talking about DSLs, generation and patterns, it is important to mention that you cannot completely automate patterns: a pattern doesn't just consist of the solution's UML diagram nor is it a mere code template. Significant parts of a pattern explain which forces affect the pattern's solution, when a pattern can be applied and when it cannot, as well as the consequences of using the pattern. A pattern often also documents many variations of itself that may all have different advantages and disadvantages. A pattern that has been implemented in the context of a transformation does not account for these aspects - the developer of the transformations must take them into account, assess them and make decisions accordingly.

What needs to be documented?

I advertise the above approach as a way to formally describe your system's conceptual and application architecture. So, this means it serves as some kind of documentation, right?

Right, but it does not mean that you don't have to document anything else. Here's a bunch of things you still need to document:

Rationales/Architectural Decisions The DSLs describe what your architecture(s) look like, but it does not explain why. You still need to document the rationales for your architectural and technological decisions. Typically you should refer back to your (non-functional) requirements here. Note that the grammar is a really good baseline. Each of the constructs in your architecture DSL grammar is the result of a number of architectural decisions. So, if you explain for each grammar element why it is there (and why maybe certain other alternatives have not been chosen) you are well on your way to document the important architectural decisions. A similar approach can be used for the application architecture, i.e. the instances of the DSL.

User Guides A language grammar can serve as a well-defined and formal way of capturing an architecture, but it is not a good teaching tool. So you need to create tutorials for your users (i.e., the application programmers) on how to use the architecture. This includes what and how to model (using your DSL) and also how to generate code and how to use the programming model (how to fill in the implementation code into the generated skeletons).

There are more aspects of an architecture that might be worth documenting, but the above two are the most important.

Embedding Business Logic via DSLs

Architecture DSLs as introduced above focus on the description of software architecture. As the section on component implementation explains, we do not concern ourselves much with the question of how the components are implemented, i.e. how business logic is expressed.

Of course this is essential for building the actual software system. As we outline above, one can either implement the application logic with manually written code, or one can use a DSL for that. This leads us to application domain DSLs, to which we have given a separate chapter.

If application domain DSLs are used together with architecture DSLs, then the code generator for the application domain DSL has to act as the "implementor" of the components (or whatever other architectural abstractions are defined in the architecture DSL), and the generated code must fit in. This is an example of DSL cascading, described here TODO.

Component Models

There are many (more or less formal) definitions of what a components is. They range from a building block of software systems to something with explicitly defined context dependencies to something that contains business logic and is run inside a container.

Our understanding (notice we are not saying we have a real definition) is that a component is the smallest architecture building block. When defining a system's architecture, you don't look inside components. Components have to specify all their architecturally relevant properties declaratively (aka in meta data, or models). As a consequence, components become analyzable and composable by tools. Typically they run inside a container that serves as a framework to act on the runtime-relevant parts of the meta data. The component boundary is the level at which the container can provide technical services such as as logging, monitoring, or failover.

We don't have any specific requirements towards what meta data a component actually contains (and hence, which properties are described). We think that the concrete notion of components has to be defined for each (system/platform/product line) architecture separately: this is exactly what we do with the language approach introduced above.

From our experience in development projects, we find that we almost always start by modeling the component structure of the system to be built. To do that, we start by defining what a component actually is - that is, by defining a meta model for component-based development. Independent of the domain in which the development project resides, these meta models are quite similar across application domains - insurance, e-commerce, embedded, radio astronomy and so on - as opposed to technical domains such as persistence, transaction processing, security. We therefore show parts of these meta models here to give you a head start when defining your own component architecture.

Three Typical Viewpoints

It is useful to look at a component-based system from several viewpoints. Three viewpoints form the backbone of the description. These viewpoints should end up being separate models in their own separate DSL, so in the following text we use "model" and "interchangeably".

The **Type** viewpoint describes *component types*, *interfaces*, and *data structures*. A component provides a number of interfaces and references a number of required interfaces. An interface owns a number of operations, each with a return type, parameters, and exceptions. Figure 14 shows this.

To describe the data structures with which the components work (figure 15), we start out with the abstract type Type. We use primitive types as well as complex types. A complex type has a number of named and typed attributes. There are two kinds of complex types. Data transfer objects are simple (C-style) structs that are used to exchange data among components. Entities have a unique ID and can be made persistent (this is not visible from the meta model). Entities can reference each other and thus build more complex data graphs. Each reference has to specify whether it is navigable in only one or in both dimensions. A reference also specifies the cardinalities of the

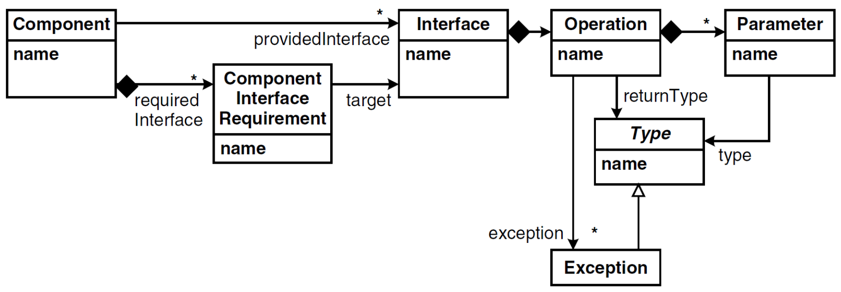


Figure 14: TODO

entities at the respective ends.

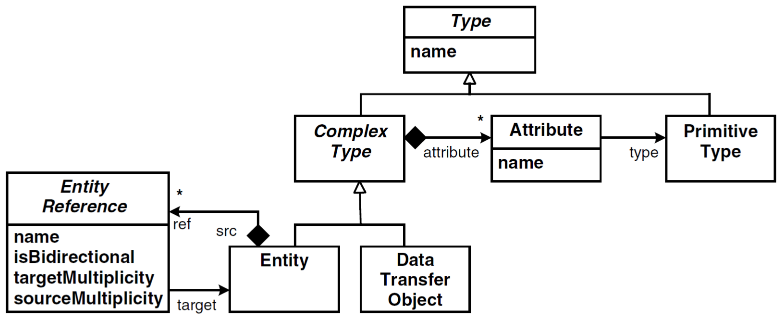


Figure 15: TODO

The **Composition** viewpoint, illustrated in figure 16, describes component instances and how they are connected. A configuration consists of a number of component instances, each referencing their type (from the Type viewpoint). An instance has a number of wires: a wire is an instance of a component interface requirement. Note the constraints defined in the meta model:

- For each component interface requirement defined in the instance's type, we need to supply a wire.
- The type of the component instance at the target end of a wire needs to provide the interface to which the wire's component interface requirement points.

Using the Type and Composition viewpoints, it is possible to define component types as well as their collaborations. Logical models of applications can be defined. From the Composition viewpoint, you can for example generate or configure a container that instantiates the component instances.

The **System** viewpoint describes the system infrastructure onto which the logical system defined with the two previous viewpoints is

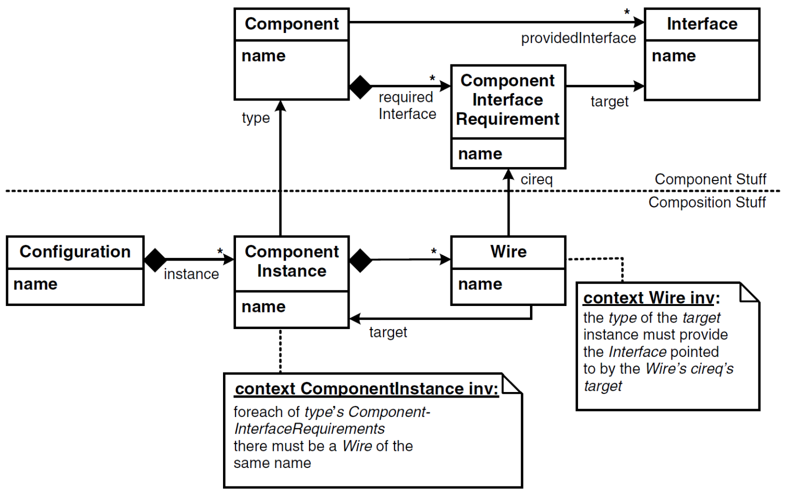

Component
name
providedInterface
Interface
name
required Interface
Component Interface Requirement
name
target
type
cireq
Component Stuff
Composition Stuff
Configuration
name
instance
Component Instance
name
Wire
name
target
context Wire inv:
the type of the target instance must provide the Interface pointed to by the Wire's cireq's target
context ComponentInstance inv:
foreach of type's Component-InterfaceRequirements there must be a Wire of the same name

Figure 16: TODO

deployed (figure 17).

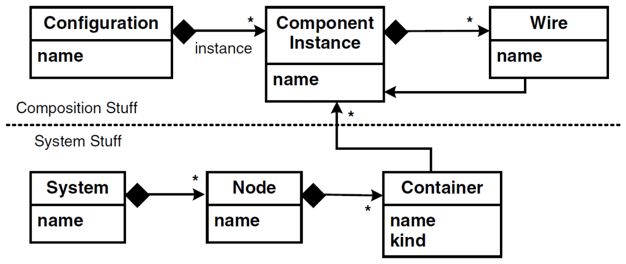


Figure 17: TODO

A system consists of a number of nodes, each one hosting containers. A container hosts a number of component instances. Note that a container also defines its kind - this could be things like OSGi, JEE, Eclipse or Spring. Based on this information, you can generate the necessary "glue" code to run the components in that kind of container. The node information, together with the connections defined in the composition model, allows you to generate all kinds of things, from remote communication infrastructure code and configuration to build and packaging scripts.

You may have observed that the dependencies among the viewpoints are well-structured. Since you want to be able to define several compositions using the same components and interfaces, and since you want to be able to run the same compositions on several infrastructures, dependencies are only legal in the directions shown in figure 18.

Aspect Models

The three viewpoints described above are a good starting point for modeling and building component-based systems. However, in

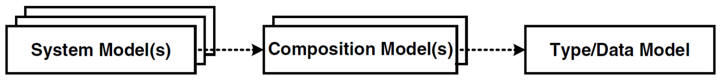


Figure 18: TODO

most cases these three models are not enough. Additional aspects of the system have to be described using specific aspect models that are arranged around the three core viewpoint models. The following aspects are typically handled in separate aspect models:

- Persistence
- Authorization and Authentication (important in enterprise systems)
- Forms, layout, page flow (for Web applications)
- Timing, scheduling and other quality of service aspects (especially in embedded systems)
- Packaging and deployment
- Diagnostics and monitoring

The idea of aspect models is that the information is not added to the three core viewpoints, but rather is described using a separate model with a suitable concrete syntax. Again, the meta model dependencies are important: the aspects may depend on the core viewpoint models and maybe even on one another, but the core viewpoints must not depend on any of the aspect models. Figure 19 illustrates a simplified persistence aspect meta model.

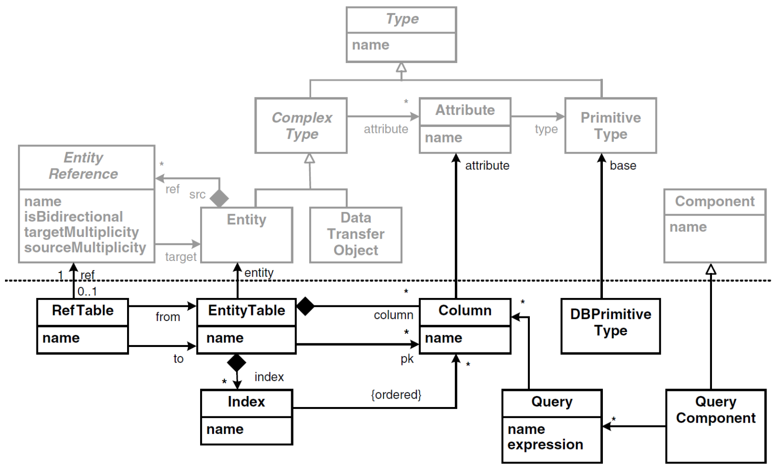


Figure 19: TODO

Variations

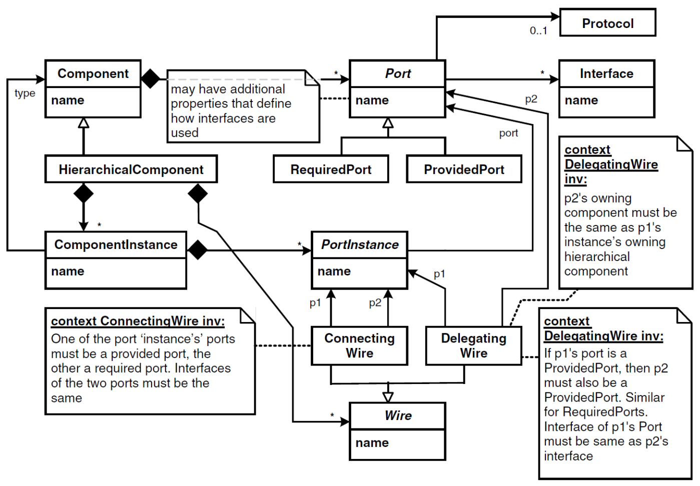
The meta models we describe above cannot be used in exactly this way in every project. Also, in many cases the notion of what constitutes a component needs to be extended. So there are many variations of these meta models. However, judging from practice, even these variations are limited. In this section we want to illustrate some of the variations we've come across.

You might not need to use interfaces to declare *all* of a component's behavior: Operations could be added directly to the components. As a consequence, of course, you cannot reuse the interface's "contracts" separately, independently of the supplier or consumer components.

Often you'll need different kinds of components, such as domain components, data access (DAO) components, process components, or business rules components. Depending on this component classification you can come up with valid dependency structures between components. You will typically also use different ways of implementing component functionality, depending on the component types.

Another way of managing dependencies is to mark each component with a layer tag, such as domain, service, GUI, or facade, and define constraints on how components in these layers may depend on each other.

Hierarchical components, as illustrated in figure 20, are a very powerful tool. Here a component is structured internally as a composition of other component instances. Ports define how components may be connected: a port has an optional protocol definition that allows for port compatibility checks that go beyond simple interface equality. While this approach is powerful, it is also non-trivial, since it blurs the formerly clear distinction between Type and Composition viewpoints.



may have additional properties that define how interfaces are used

context DelegatingWire inv: p2's owning component must be the same as p1's instance's owning hierarchical component

context ConnectingWire inv: One of the port 'instance's' ports must be a provided port, the other a required port. Interfaces of the two ports must be the same

context DelegatingWire inv: If p1's port is a ProvidedPort, then p2 must also be a ProvidedPort. Similar for RequiredPorts. Interface of p1's Port must be same as p2's interface

Figure 20: TODO

A component might have a number of configuration parameters - comparable to command line arguments in console programs - that help configure the behavior of components. The parameters and their types are defined in the type model, and values for the parameters can be specified later, for example in the models for the Composition or the System viewpoints.

You might want to say something about whether the components are stateless or stateful, whether they are thread-safe or not, and what their lifecycle should look like (for example, whether they are passive or active, whether they want to be notified of lifecycle events such as activation, and so on).

It is not always enough to use simple synchronous communication. Instead, one of the various asynchronous communication patterns, such as those described in [VKZ04], might be applicable. Because using these patterns affects the APIs of the components, the pattern to be used has to be marked up in the model for the Type viewpoint, as shown in Figure 7.13.

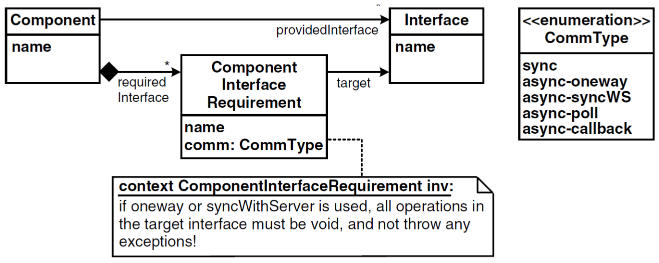

Component
name
providedInterface
Interface
name
required Interface
*
Component Interface Requirement
name
comm: CommType
target
<<enumeration>> CommType
sync
async-oneway
async-syncWS
async-poll
async-callback
context ComponentInterfaceRequirement inv:
if oneway or syncWithServer is used, all operations in the target interface must be void, and not throw any exceptions!

Figure 21: TODO

In addition to the communication through interfaces, you might need (asynchronous) events using a static or dynamic publisher/subscriber infrastructure. It is often useful that the "direction of flow" of these events is the opposite of the dependencies discussed above.

The (model for the) Composition viewpoint connects component instances statically. This is not always feasible. If dynamic wiring is necessary, the best way is to embed the information that determines which instance to connect to at runtime into the static wiring model. So, instead of specifying in the model that instance A must be wired to instance B, the model only specifies that A needs to connect to a component with the following properties: needs to provide a certain interface, and for example offer a certain reliability. At runtime, the wire is "deferenced" to a suitable instance using an instance repository. This approach is similar to CORBA's trader service.

Finally, it is often necessary to provide additional means of structuring complex systems. The terms business component or sub system are often used. Such a higher-level structure consists of a number of components. Optionally, constraints define which kinds of components may be contained in a specific kind of higher-level structure. For example, you might want to define that a business component always consists of exactly one facade component and any number of domain components. The approach laid out above works for engineering a suitable DSL for the higher-level situation as well, as it's basically a matter of finding/identifying the "right" abstractions. You might have to come up with a way to parametrize the lower-level structure to a small extent and to reference such parametrized structures from the higher-level DSL to make this really useful.

Summing Up

DSLs and Requirements

What is Requirements Engineering

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: *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 our own words we would probably define a requirement as a statement about *what a system should do, and with which quality attributes, without presupposing a specific implementation.* Requirements are supposed to tell the programmers what the system they are about to implement, should do. They 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 in this:

- Those who implement the system may have a different background than those who write them, making misunderstandings likely.
- Those who write the requirements may not actually really know what they want the system to do. Requirements change and learning about these and writing them down is hard.
- 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 impossible.

Traditional requirements (documents) are a means of communication between 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. If we look at the problem in this way, requirements now become a formal, computer-understandable document.

Wikipedia has a nice list of characteristics that requirements should posses:

Cohesive The requirement addresses one and only one thing.

Complete The requirement is fully stated in one place with no missing information.

Consistent The requirement does not contradict any other requirement and is fully consistent with all authoritative external documentation.

Atomic The requirement is atomic, i.e., it does not contain conjunctions. E.g., "The postal code field must validate American and Canadian postal codes" should be written as two separate requirements: (1) "The postal code field must validate American postal codes" and (2) "The postal code field must validate Canadian postal codes".

Traceable The requirement meets all or part of a business need as stated by stakeholders and authoritatively documented.

Current The requirement has not been made obsolete by the passage of time.

Feasible The requirement can be implemented within the constraints of the project.

Unambiguous The requirement is concisely stated without recourse to technical jargon, acronyms (unless defined elsewhere 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 prohibited.

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 terms.

Verifiable The implementation of the requirement can be determined through one of four possible methods: inspection, demonstration, test or analysis.

If requirements are written as pure prose text, then making sure all these characteristics are met, boils down mostly to a manual review process. Of course, this is tedious and error prone (often exacerbated through over-specification even though the prose text isn't a specification at all), so it's often not done rigorous enough or not at all and requirements documents end up in the sorry state we all know.

If you want to get one step better, you use somewhat controlled language: 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 a bit more unambiguous. To manage large sets of requirements, tools like DOORS or RequisitePro are used: they allow the unique naming of requirements, as well as the expression of relationships and hierarchies among requirements. 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 DSLs.

Requirements vs. Design vs. Implementation

Traditionally, we try to establish a clear line between requirements, architecture and design, and implementation. For example, consider the following:

Requirement The system shall be 100% reliable.

Design Use hot-standby and failover to continue service if something breaks.

Implementation . . . all the code that is necessary to implement the design above.

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. Different organizations might be involved: the OEM writes the requirements, a systems integrator does the architecture, and some cheap outsourcing company does the coding. In such a scenario it is important to draw precise boundaries between the activities, of course.

However, in some sense the boundaries are arbitrary. We could just as well do the following:

Requirement The system shall be 100% reliable by using hot-standby and failover to continue service if something breaks.

Design We use two application servers running on two hosts, using XYZ messaging queue as a replication engine for the hot-standby. We use a watchdog for detecting if the primary machine breaks.

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 incrementally: you write some requirements, then you write a prototype and check if the requirements make sense, then you refine the requirements, write a (maybe more detailed) prototype, and so on.

In systems engineering 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 satellite-based 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 prohibiting a strict hierarchy.)
- A bigger antenna results in a heavier satellite, and more radar energy requires more solar panel area - increasing the size and weight again.
- 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 might 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 Baikonur cannot reach the same orbits as an Ariane launched from Courou. 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 antenna.
- Instead of a bigger antenna, you might want to use several smaller ones, maybe even on different satellites.

Now this has just looked at size and weight. Similar problems exist with heat management, pointing accuracy, propulsion, etc. It is plainly impossible to write requirements, throw them over the fence, and have somebody build the satellite.

So what do the systems engineers do? They come up with a model of the satellite. Using mathematic 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. And then they run a so-called 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 shows 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.

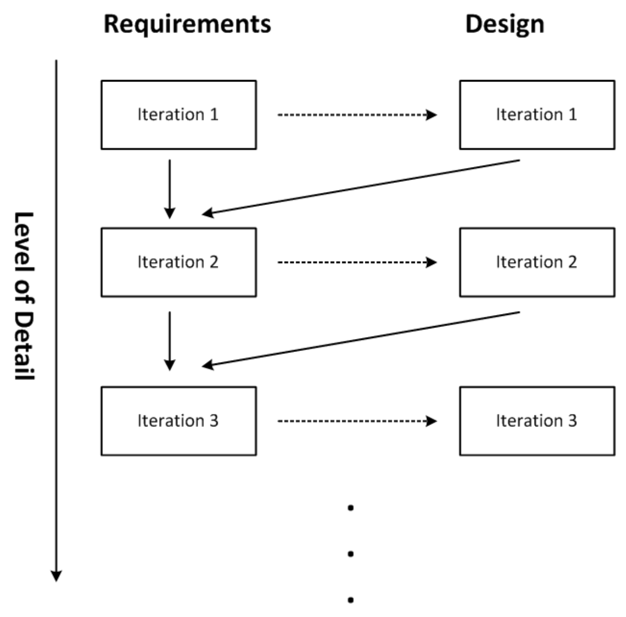


Figure 22: Requirements and Design influence each other and are thus best done iteratively, and in parallel

Using DSLs for Requirements Engineering

So here is the approach for using DSLs we suggest: identify a couple of core areas of the to-be-built system that lend themselves to specification with a formal language. Then develop a DSL 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 tools. We will discuss the integration between the textual requirements and the DSL-based requirements below.

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 have the requirements owners "play" with the requirements models to see if they really express what they had in mind.

At this point you may say: well, that's not really requirements engineering anymore, that's just plain old DSL usage. We agree to some extent. However, these DSLs might not be complete enough to support execution. In many cases they are, but in some cases checking, analysis and/or simulation might be all that's required. More detailed models may be required to make them executable.

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're not capturing the specific requirements but rather try to describe how specific requirements are described, you are performing what's called a domain analysis: you try to understand the degrees of freedom in the domain to be able to represent it with the DSL. This is similar to what has been done with analysis models (back in the day ...). However, instead of drawing UML analysis diagrams, you capture the domain into a language definition. These are if you will "executable analysis models", since 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.

Here is the process we use:

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

2. Try to understand that aspect and change your DSL so it can express that aspect.

3. Have the requirements owner try to express a couple of specific, but representative, requirements with the DSL.

4. You'll run into problems, some things cannot be expressed with the DSL.

5. Go back to 1. and reiterate. A complete iteration should take no more than 60 minutes.

6. After half a day, stop working with the requirements owner and clean up/refactor the DSL.

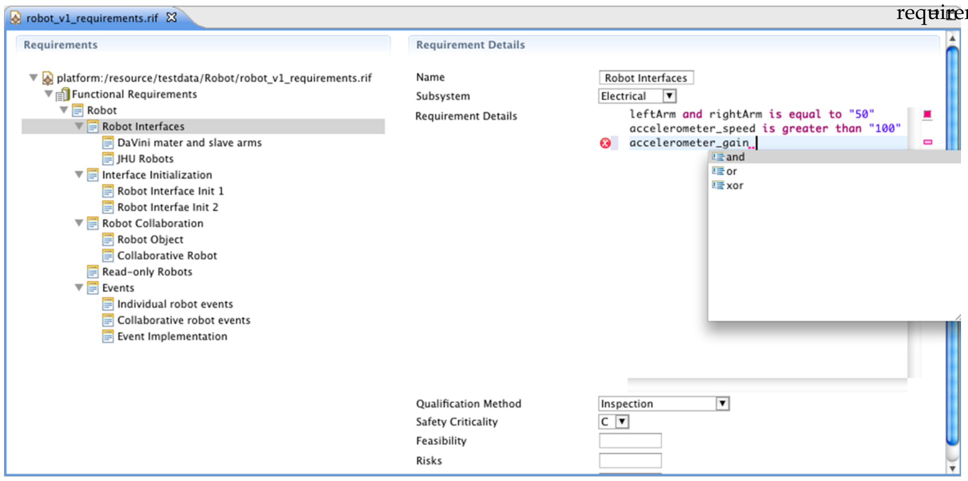
7. Start another of the language design sessions with the requirements owner and iterate – over time, you should get closer to *the* DSL for the domain.

Integration with Plain Text Requirements

Using DSLs, you will not be able to describe all requirements of a systems. There will always be aspects that cannot be formalized, or that are so specific, that the effort of building a DSL does not pay off. You have to find some way of integrating plain text requirements with DSL code. Here are some ideas of how you can do this.

Embedding DSL code in a requirements tool

As part of the VERDE research project, Eclipse-based tooling for requirements engineering is developed. This includes a "classical" requirements engineering tool in which textual requirements are classified, structured and put into relationships with each other. The requirements structure is represented as an EMF model. In addition to plain text, requirements can have parameters with well-defined types (strings, integers).



Name: Robot Interfaces
Subsystem: Electrical
Requirement Details:

```
leftArm and rightArm is equal to "50"
accelerometer_speed is greater than "100"
accelerometer_gain
```

Qualification Method: Inspection
Safety Criticality: C
Feasibility:
Risks:

Figure 23: An Xtext DSL embedded in a requirements engineering tool

In addition, the type of an attribute can also be a DSL - in other words, it is possible to associate DSL programs with requirements. As figure 23 shows, these DSL fragments are integrated with the UI of the requirements management tool.

Integrating Plain Text and Models

If DSLs are used to describe requirements, these can obviously annotated with comments, providing some form of integration with non-formal requirements. However, this is a bad solution for various reasons. In most cases it is better to keep the text and the models separate, and then refer from one to the using actual links (not just text references).

Using the Xdoc extension to Xtext

In MPS, one can use the generic documentation language and embed references

Requirements Traceability

This means: being able to trace how requirements lead to design, implementation and test artifacts. This allows you to see if and how requirements have influenced the design, whether they are implemented and whether tests for the requirements exist. This is especially important in safety-critical environments where you have to "prove" that certain requirements have been taken care of adequately.

Traceability links are basically "pointers" from requirements to other artifacts. In a world where requirements 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, this can become arbitrarily complicated.

In this section we want to show you two approaches to traceability.

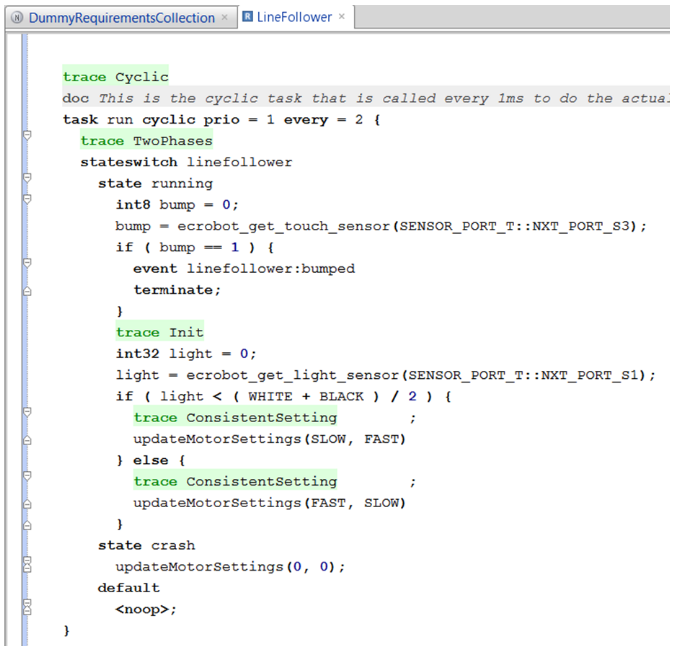
Traceability Framework for Eclipse

The VERDE project mentioned above also develops a traceability framework based on Eclipse. Various types of traceability links can be defined after which they can be used to establish links between arbitrary EMF based models. In addition to the generic EMF-based version, tool adapters can also be plugged in. These provide specific support for links from/into a number of specific tools or formats, for example plain text files, non-EMF UML tools or AUTOSAR models.

The links are kept external to the actual models, so no modifications to existing meta models or languages are required. This is also the reason why non-EMF artifacts can be integrated, albeit requiring the special tool adapter.

Traceability in MPS

MPS supports language annotations. Arbitrary models - independent of the language - can be annotated with any other data. One use case of this approach is the annotation with traceability links, as shown in figure 24.



```
trace Cyclic
doc This is the cyclic task that is called every 1ms to do the actua
task run cyclic prio = 1 every = 2 {
  trace TwoPhases
  stateswitch linefollower
    state running
      int8 bump = 0;
      bump = ecrobot_get_touch_sensor(SENSOR_PORT_T::NXT_PORT_S3);
      if ( bump == 1 ) {
        event linefollower:bumped
        terminate;
      }
      trace Init
      int32 light = 0;
      light = ecrobot_get_light_sensor(SENSOR_PORT_T::NXT_PORT_S1);
      if ( light < ( WHITE + BLACK ) / 2 ) {
        trace ConsistentSetting        ;
        updateMotorSettings(SLOW, FAST)
      } else {
        trace ConsistentSetting        ;
        updateMotorSettings(FAST, SLOW)
      }
    state crash
      updateMotorSettings(0, 0);
    default
      <noop>;
}
```

Figure 24: Traceability annotations on a C-like language

A context menu action adds a new link to any other model element: Control-Space allows the selection of one or more requirements to which to point. The requirements themselves are imported into an MPS model from a requirements engineering tool via RIF files.

Alternatively it would also be possible to create tooling that points from requirements to implementation (as opposed to the opposite direction, shown above).

Summing Up

This chapter describes how DSLs play into requirements engineering. Let us wrap up by revisiting the desirable characteristics for requirements as outlined by Wikipedia, and see how DSLs can improve the situation. 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 programs can be expressed.

Traceable As shown in the last section above, traceability is at least simplified.

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 that the requirements regarding various properties. Inspection and review is simplified, because DSL programs are less verbose than implementation code, and clearer than prose.

Bibliography