Architecture as Language: A story

Feb 21, 2008

Markus Völter
voelter@acm.org
www.voelter.de

Abstract

Architecture is typically either a very non-tangible, conceptual aspect of a software system that can primarily be found in Word documents, or it is entirely driven by technology ("we use an XML architecture"). Both are bad: the former makes it hard to work with, and the latter hides architectural concepts behind technology hype.

What can be done? As you develop the architecture, evolve a language that allows you to describe systems based on this architecture. Based on my experience in a number of real-world projects, this makes the architecture tangible and provides an unambiguous description of the architectural building blocks as well as the concrete system while still staying away from technology decisions (which then can be made consciously in a separate step).

The first part of this paper illustrates the idea using a real-world story. The second part summarizes the key points of the approach.

A story

System Background

So I was with a customer for one of my regular consulting gigs. The customer decided they wanted to build a new flight management system. Airlines use systems like these to track and publish various information about: whether airplanes have landed at a given airport, whether they are late, the technical status of the aircraft, etc. The system also populates the online-tracking system in the internet and information monitors at airports etc. This system is in many ways a typical distributed system, consisting of many machines running different parts of the overall 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.

My customer has been building systems like these for many years and they were planning to introduce a new generation of this system. The new system had to be able to evolve over a 15-20 year timeframe. It was clear from this requirement alone that they needed some kind of technology abstraction, because technology probably goes through 8 hype cycles over 15 – 20 years. Another good reason for abstracting technology is that 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 frontend.

Because of the distributed nature of the system, it is impossible to update all parts of the system at the same time. This resulted in a requirement for being able to evolve the system piece by piece, in turn resulting in requirements for versioning of the various system components (to make sure component A could still talk to component B after B had been upgraded to a new version).

The starting point

When I arrived at the project, they had already decided that the backbone of the system would be a messaging infrastructure (which is a good decision for this kind of system) and they were evaluating different messaging backbones for performance and throughput. They had also already decided they would go with a system-wide business object model to describe the data the system works with (this is actually not a very good decision for systems like these, but it's not important to the punch line of this story).

So when I arrived they briefed me about all the details of the system and the architectural decisions they had already made, and then basically asked me whether all this makes sense. It turned out quickly that, while they knew many of the requirements and had made pinpoint decisions about certain architectural aspects, they didn't have what I'd call a *consistent architecture*: a definition of the building blocks (types of things) from which the actual system will be built. They had no *language* to talk about the system.

This is actually a very common observation when I arrive in projects to help out. And of course this is something which I think is a *huge* problem: if you don't know the *kinds of things* from which your system will be composed it is very hard to actually talk about, describe, and of course build the system consistently. You need to define a language.

Background: What is a language?

You know you have a consistent architecture when you have a *language* to talk about the system from an architectural perspective[1]. So what is a language? Obviously, it is first and foremost a set of well-defined terms. *Well defined* means primarily that all stakeholders agree on the meaning of the terms. If you look at languages form an informal point of view, terms and their meaning are probably already enough to define *language*.

However – and that might come as a surprise here – I am advocating a *formal* language for architecture description[2]. To define a formal language, you need more than terms and meaning. You need a grammar on how to build "sentences" (or models) from those terms, and you need a concrete syntax to express them[3].

Using a formal language for describing your architecture provides several benefits that will become clear from the rest of the story. I will also recap toward the end of this paper section.

Developing a Language for describing architecture

So let's continue our story. My customer and I agreed that it might be worthwhile to spend a day going through some technical requirements and build a formal language for an architecture that could realize those

[1] Eric Evans talks about a domain language that provides a language for the domain, for the business functionality of the system. This is of course also important, but in this paper I talk about a language for the architecture.

[2] No, I am not talking about ADLs or UML. Read on.

[3] You also need some kind of tool for writing sentences in the language. More on that later.

requirements. We would actually build the grammar, some constraints and an editor (using oAW's Xtext tool) as we discussed the architecture.

Getting Started

We started with the notion of a component. At that point the notion of components is defined relatively loosely. It's simply the smallest architecturally relevant building block, a piece of encapsulated application functionality. We also assumed that components can be instantiated, making components the architectural equivalent to classes in OO programming. So here's an initial example model we built based on the initial grammar we defined:

```
component DelayCalculator {}
component InfoScreen {}
component AircraftModule {}
```

Note how we did two things here: we defined that the concept of a *component* exists (making a component a building block for the system we're about to build) and we also decided (preliminarily) that there are the three components *DelayCalculator*, *InfoScreen* and *AircraftModule*. We refer to the set of building blocks for an architecture as the *conceptual architecture* and to the set of concrete exemplars of those building blocks as the *application architecture*[4].

Interfaces

Of course the above notion of a component is more or less useless because components cannot interact. It is clear from the domain that the *DelayCalculator* would have to receive information from the *AircraftModules*, calculate the delay status for flights, and then forward the results to *InfoScreens*. We knew that they would somehow exchange messages (remember: the messaging decision had already been made). But we decided to not introduce messages yet, but rather abstract a set of related messages into interfaces[5].

```
component DelayCalculator implements IDelayCalculator {}
component InfoScreen implements IInfoScreen {}
component AircraftModule implements IAircraftModule {}
interface IDelayCalculator {}
interface IInfoScreen {}
interface IAircraftModule {}
```

The above code, we realized, looks quite a bit like Java code. This is not surprising, since my customer had a Java background and the primary

[4] This also hints at the fact that this approach is especially suitable for large systems, product lines and platforms.

[5] Coming up with the concrete set of components, their responsibilities, and consequently, their interfaces is not necessarily trivial either. Techniques like CRC cards can help here.

target language for the system was Java. It is therefore likely that the well-known concepts from the language they were used to working with trickle into our own languages. However, we quickly noticed that this is not very useful: we could not express that a component *uses* a certain interface (as opposed to providing it). Knowing about the interface requirements of a component is important, because we want to be able to understand (and later: analyze with a tool) the dependencies a component has. This is important for any system, but especially important for the versioning requirement.

So we changed the grammar somewhat, supporting the following expressions:

```
component DelayCalculator {
  provides IDelayCalculator
  requires IInfoScreen
}
component InfoScreen {
  provides IInfoScreen
}
component AircraftModule {
  provides IAircraftModule
  requires IDelayCalculator
}
interface IDelayCalculator {}
interface IInfoScreen {}
interface IAircraftModule {}
```

Describing Systems

We then looked at how those components would be used. It became clear very quickly that the components needed to be instantiatable. Obviously, there are many aircraft, each of them running an *AircraftModule* component, and there are even more *InfoScreens*. It was not entirely clear whether we'd have many *DelayCalculators* or not, but we decided to postpone this discussion and go with the instantiation idea.

So we need to be able to express instances of components.

```
instance screen1: InfoScreen
instance screen2: InfoScreen
…
```

We were then discussing how to "wire" the system: how to express that a certain *InfoScreen* talks to a certain *DelayCalculator*? Somehow we would have to express a relationship between instances. The types, respectively, already had "compatible" interfaces, a *DelayCalculator* could talk to an *InfoScreen*. But this "talk to" relationship was hard to grasp as of yet. We also noticed that *one DelayCalculator* instance typically talks to *many InfoScreen* instances. So cardinalities had to get into the language, somehow.

After some tinkering around, I introduced the concept of *ports* (this is actually a well-known concept in component technology and also UML, but was relatively new to my customer). A port is a communication endpoint defined on a component type that is instantiated whenever the owning component is instantiated. So we refactored the component description language to allow us to express the following. Ports are defined with the *provides* and *requires* keywords, followed by the port name and cardinality, a colon, and the port's associated interface.

```
component DelayCalculator {
  provides default: IDelayCalculator
  requires screens[0..n]: IInfoScreen
}
component InfoScreen {
  provides default: IInfoScreen
}
component AircraftModule {
  provides default: IAircraftModule
  requires calculator[1]: IDelayCalculator
}
```

The above model expresses that any *DelayCalculator* instances has a connection to *many InfoScreens*. From the perspective of the *DelayCalculator* implementation code, this set of *InfoScreens* can be addressed via the *screens* port. The *AircraftModule* has to talk to *exactly one DelayCalculator,* which is what the *[1]* expresses.

This new notion of interfaces inspired my customers to change the *IDelayCalculator*, because they noticed that there should be different interfaces (and hence, ports) for different communication partners. We changed the application architecture to this:

```
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
}
```

Notice how the introduction of ports led to a better application architecture, because we now have role-specific interfaces (*IAircraftStatus*, *IManagementConsole*).

Now that we had ports, we could *name* communication endpoints. This allowed us to easily describe systems: connected instances of components. Note the new *connect* construct.

```
instance dc: DelayCalculator
instance screen1: InfoScreen
instance screen2: InfoScreen

connect dc.screens to (screen1.default, screen2.default)
```

Keeping the Overview

Of course 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 concept. And of course we can distribute things to different files (the tool support makes sure that *go to definition* and *find references* still works).

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

Of course it is a good idea to keep component and interface definition (essentially: type definitions) separate from system definitions (connected instances), so here we define a system:

```
namespace com.mycompany.test {
  system 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, here we go. We specify a query that is executed at runtime against some kind of naming/trader/lookup/registry infrastructure. It is reexecuted every 60 seconds to find *InfoScreens* that had just come online.

```
namespace com.mycompany.production {
  instance dc: DelayCalculator
  // InfoScreen instances are created and
  // started in other configurations
  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 reexecuted when the currently used system 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*.

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

Interfaces, Part II

Until now we didn't really define what an interface was. We knew that we'd like to build the system based on a messaging infrastructure, so it was clear that an interface had to be defined as a collection of messages. Here's our first idea: a collection of messages, where each message has a name and a list of typed parameters.

```
interface IInfoScreen {
  message expectedAircraftArrivalUpdate(id: ID, time: Time)
  message flightCancelled(flightID: ID)
  …
}
```

Of course this also requires the ability to define data structures. So we added that:

```
typedef long ID
struct Time {
  hour: int
  min: int
  seconds: int
}
```

Now, after discussing this interface thing for a while, we noticed that it's not enough to simply define an interface as a set of messages. The minimal thing we want to do is 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*:

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

Is it really messages?

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 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 of a certain status item, incremental updates, invalidations, etc.

And at some point it hit us: We were working with the wrong abstraction! While messaging is a suitable transport abstraction for these things, 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 than 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 included full update of all items in the collection, incremental updates of just one or several items, invalidations, etc.

Of course, once we understood that in addition to messages there's this additional core abstraction in the system, we added this to our architecture language 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.

```
struct FlightInfo {
  from: Airport
  to: Airport
  scheduled: Time
  expected: Time
  …
}

replicated singleton flights {
  flights: FlightInfo[]
}

component DelayCalculator {
  publishes flights
}

component InfoScreen {
  consumes flights
}
```

This is of course much more concise compared to a description based on messages. The system can automatically derive the kinds of messages needed for full update, incremental update, invalidation, etc. This description also much clearer reflects the actual architectural intent: this description 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).

Of course it does not stop there. Now that we have state replication as a first class citizen, we can add more information to its specification:

```
component DelayCalculator {
  publishes flights { publication = onchange }
}

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

We describe that the publisher publishes the replicated data whenever something changes in the underlying 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). Based on that information we can derive all the required messages and also an update schedule for the participants.

More?

During the rest of the discussions we identified several other architectural aspects and we added language abstractions for them:

- To support versioning, we added ways of specifying that a component should act as a new version (replacement) of an existing component. The tooling would ensure "plug in compatibility".
- To be able to express semantics of messages and their effects on system state we introduced pre- and postconditions. We also extended the notion of components to optionally be stateful.
- Finally, we added configuration parameters to components. Components specify the parameters, and component instances have to specify values for them.

Conclusion

Using the approach, we were able to quickly get a grip towards the overall architecture of the system. We also 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 clear and unambiguous definition of what the different terms mean. The nebulous concept of *component* has a formal, well-defined meaning in the context of this system.

And of course, it didn't stop here. The next steps involved a discussion of how to actually code the implementation for a component and which parts of the system could be automatically generated. More on this in the next section.

Recap & Benefits

What we did in a nutshell

The approach includes 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.

Background: DSLs

The language we built above is a DSL, a domain-specific language. Here is how I like to define DSLs:

A DSL is a focused, processable language for describing a specific concern when building a system in a specific domain. The abstractions and notations used are tailored to the stakeholders who specify that particular concern.

DSLs can be used to specify any aspect of a software system. The big hype is around using a DSL to describe the business functionality (for example, calculation rules in an insurance system). While is this a very worthwhile use of DSL, it is also worthwhile to use DSLs to describe software architecture: this is what we do above.

So the architecture language built above – and the approach I am generally advocating in this paper – is to use DSL technology to define a DSL that expresses your architecture.

Benefits

Everybody involved will have clear understanding of the concepts used to describe the system. There's a clear and unambiguous vocabulary to describe applications. The created models can be analyzed and used as a basis for code generation (see below). The architecture is freed from implementation details, or in other words: conceptual architecture and technology decisions are decoupled, making both easier to evolve. We can also define a clear programming model based on the conceptual architecture (how to model and code components using all the architectural features defined above). Last but not least, the architect can now contribute directly to the project, by building (or helping to build) an artifact that the rest of the team can actually use.

Why textual?

… or why not use a graphical notation? Textual DSLs have several advantages. First of all, languages as well as nice editors are much easier to build. Second, textual artifacts integrate much better with the existing developer tooling (CVS/SVN diff/merge) than graphical models based on some kind of repository. Third, textual DSLs are often more appreciated by developers, since "real developers don't draw pictures".

For aspects of the system where a graphical notation is useful to see relationships between architectural elements, you can use tools like Graphviz or Prefuse. Since the model contains the relevant data in a clear and unpolluted form, you can easily export the model data in a form that tools like GraphViz or Prefuse can read.

Tooling

To make the approach introduced above feasible, you need tooling that supports the efficient definition of DSLs. We use openArchitectureWare's Xtext. Xtext does the following things for you:

- It provides a way of specifying grammars
- From this grammar the tool generates an antlr grammar to do the actual parsing

- It also generates an EMF Ecore metamodel; the generated parser instantiates this metamodel from sentences of the language. You can then use all EMF-based tools to process the model
- You can also specify constraints based on the generated Ecore model. Constraints are specified using oAW's Check language (essentially a streamlined OCL)
- Finally, the tool also generates a powerful editor for your DSL that provides code folding, syntax coloring and customizable code completion as well as an outline view and cross-file *go-to-definition* and *find references*. It also evaluates your language constraints in real time and outputs error messages.

After a little bite of practice, Xtext gets out of your way and really allows you to specify the language as you understand architectural details and make architectural decisions. Customizing code completion might take a little bit longer, but you can do that when the language exploration phase has reached a plateau.

Validating the Model

If we want to describe an architecture formally and correctly, we need to put validation rules into place that constrain the model even further than what can be expressed via the grammar. Simple examples include the typical name-uniqueness constraints, type checks or non-nullness. Expressing such (relatively) local constraints is straight forward using OCL or OCL-like languages.

However, we also want to verify more complex and less local constraints. For example, in the context of our story above, the constraints check that new versions of components and interfaces are actually compatible with their predecessor and hence can be used in the same context. To be able to implement such non-trivial constraints, two preconditions are necessary:

- The constraint itself must actually be formally describable, i.e. there must be some kind of algorithm that deteremines whether the constraint holds or not. Once you know the algorithm, you can implement it in whatever constraint language your tooling supports (in our case here, the OCL-like Xtend or Java)
- The other precondition is that the data that is needed to run the constraint-checking algorithm defined above is actually available in the model. For example, if you want to verify whether a certain deployment scheme is feasible, you might have to put the available network bandwidth and the timings of certain messages as well as

the size of primitive date types into the model[6]. However, while capturing those data sounds like a burden, this is actually an advantage, since this is core architectural knowledge.

Generating Code

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 (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: 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 (incl. 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" 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.

[6] you might actually want to put them into a different file, so this aspect does not pollute the core models. But this is a tooling issue.

Another note 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.

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

How does this compare to ADLs and UML?

Describing architecture with formal languages is not a new idea. Various communities recommend using Architecture Description Languages (ADLs) or the Unified Modeling Language (UML) for describing architecture. Some even (try to) generate code from the resulting models. However, all of those approaches advocate using *existing generic* languages for documenting the architecture (although some of them, including the UML, can be customized).

However (as you probably can tell from the story above) this *completely misses the point*! I don't see much benefit in shoehorning your architecture description into the (typically very limited) constructs provided by predefined/standardized 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*. Adapting your architecture to the few concepts provided by the ADL or UML is not very helpful.

A note on UML and profiles: yes, you could use the approach explained above with UML, building a profile as opposed to a textual language. I have done this in several projects and my conclusion is that it doesn't work well in most environments. Here are some of the reasons:

- Instead of thinking about your architectural concepts, working with UML requires you to think more about how you can use UML's existing constructs to more or less sensibly express your intentions. That's the wrong focus!
- Also, UML tools typically don't integrate very well with your existing development infrastructure (editors, CVS/SVN, diff/merge). That's not much of a problem when you use UML during some kind of analysis or design phase, but once you use your models as source code (they accurately reflect the architecture

of your system, and you generate real code from them) this becomes a big issue.

- Finally, UML tools are often quite heavyweight and complex, and are often perceived as "bloatware" or "drawing tools" by "real" developers. Using a nice textual language can be a much lower acceptance hurdle.

Why not simply use a programming language

Architectural abstractions, such as messages or components are not first class citizens in today's 3GL programming languages. Of course you can use classes to represent all of them. Using annotations (also called attributes) you can even associate meta data to classes and some of their contents (operations, fields). Consequently, you can express quite a lot with a 3GL, somehow. But this approach has problems:

- Just like in the case of UML explained above, this approach forces you to shoehorn clear domain specific concepts into prebuilt abstractions. In many ways, annotations/attributes are comparable to UML stereotypes and tagged values, and you will experience similar problems.
- Analyzability of the model is limited. While there are tools like Spoon for Java, there is nothing easier to work with and process than a formal model.
- Finally, expressing things with "architecture-enhanced Java or C#" also means that you are tempted to mix architectural concerns with implementation concerns. This blurs the clear distinction, and potentially sacrifices technology independence.

My Notions of Components

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*.

My understanding (notice I am not saying I 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 do 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.

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

Component Implementation

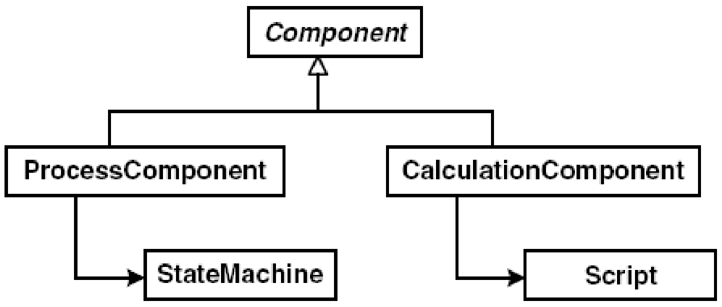
By default, component implementation happens manually. The implementation code is written against the generated API code introduced in the previous section, Developers add manually-written code to the component skeleton, either by adding the code directly to the generated class, or – a much better approach – by using other means of composition such as inheritance or partial classes.

However, there are other alternatives for component implementation that do not use a 3GL programming language for component implementation, but instead use formalisms that are specific to the kind of behavior that should be specified.

- Behavior that is very regular can be implemented using the generator, after parameterizing it in the model by setting a small number of well-defined variability points. Feature models are good at expressing the variabilities that need to be decided so that an implementation can be generated.
- For state-based behavior, state machines can be used.
- For things such as business rules, you can define a DSL that directly express these rules and use a rules engine to evaluate them. Several rule engines are available off-the-shelf.
- For domain-specific calculations, such as those common in the insurance domain, you might want to provide a specific notation that supports the mathematical operations required for the domain directly. Such languages are often interpreted: the component implementation technically consists of an interpreter that is parameterized with the program it should run.

There is also the alternative of using Action Semantics Languages (ASLs). However, it is important to point out that they don’t provide domain-specific abstractions, they are generic in the same way as for example UML is a generic modeling language. However, even if you use more specific notations, there might still be a need to specify small snippets of behavior generically. A good example are actions in state machines.

To combine the various ways of specifying behavior with the notion of components, it is useful to define various kinds of components, using subtyping at the metalevel, that each have their own notation for specifying behavior. The following diagram illustrates the idea.



Since component implementation is about behavior, technically, it is often useful to use an interpreter encapsulated inside the component.

As a final note, 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 is related to the technical infrastructure – remoting, persistence, workflow and so on – of an application, and can be derived from the architectural models.

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 influencethe 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! 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 the the grammar is a really good baseline. Each of the constructs in your architecture DSL grammar results from 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 wrt. to documenting 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.

Further Reading

If you tend to like the approach explained in this paper, you might want to read my collection of *Architecture Patterns*. They look at essentially the same topic from a patterns perspective and provide rationales for the stuff explained here. It is a somewhat older paper, but essentially looks at the same topic.

Another thing to look at is the whole area of domain-specific languages and model-driven software development. I have written a lot of stuff on this topic, primarily I have co-authored a book called *Model-Driven Software Development – Technology, Engineering, Management* which you might want to look at.

Of course you might also want to look at more details about Eclipse Modeling, openArchitectureWare and Xtext in general. There's a lot of information available at *eclipse.org/gmt/oaw*, including the official oAW docs and a large collection of introductory videos.

Acknowledgements

I would like to thank (in no particular order) Iris Groher, Alex Chatziparaskewas, Axel Uhl, Michael Kircher, Tom Quas and Stefan Tilkov for their very useful comments on previous versions of this paper.