Architecture as Language

Markus Völter, *independent contractor*

Architects can develop domain-specific languages that express the desired architecture during the definition process and use them to describe systems based on the architecture.

Software architecture is a funny thing. Most developers agree that we need it in some way, shape, or form. However, they can't agree on a definition, don't know how to manage it efficiently in nontrivial projects, and usually can't express a system's architectural abstractions precisely and concisely. When asking customers to describe a system's architecture, I get responses that include specific technologies, buzzwords such as AJAX (asynchronous JavaScript and XML) or SOA (service-oriented architecture), or vague notions of "components" (such as publishing, catalog, or payment). Some have wallpaper-sized UML diagrams in which the meanings of the boxes and lines aren't clear. These are all aspects of a system's architecture, but none of them represent a concise, unambiguous, and "formal" description of a system's core abstractions.

This isn't necessarily surprising because we don't have a language that directly expresses software architecture. And as we all know, if we don't have a language to express something, we have a hard time grasping it. Today's mainstream programming languages don't even have a way to express building blocks larger than classes and the relationships among them, the very basics of software architecture.

To address this issue, I'll explain the usefulness of domain-specific languages (DSLs) for describing, managing, and validating software architecture. The approach centers around building a DSL that captures the core architectural abstractions of the particular architecture, system, or platform, and using code generation to help implement it consistently.

I've used this approach with customers from various domains, including embedded systems, finance, and large-scale Web applications. I can't name the customers here, but they're global players in their respective domains. The projects aim at rebuilding core platforms that will be used over at least the next decade—explaining the drive toward a stable, well-defined, and technology-agnostic architecture definition; technology goes through at least three hype cycles each decade.

My Notion of Software Architecture

So what's software architecture? The industry has many definitions (see www.sei.cmu.edu/architecture/published_definitions.html). My working definition comprises the best of many of these definitions: software architecture is all those aspects of a system that we want to keep consistent throughout the system for reasons ranging from meeting nonfunctional requirements, to technology best practices, to maintainability, to developer training. The definition also implies that some aspects don't need system-wide consistency. As the person (or team) responsible for architecture, I don't care how those aspects are implemented, as long as they fulfill the stated requirements.

My working definition doesn't state anything about granularity. There might be certain nitty-gritty details of the locking protocol used in a concurrent system that I consider part of the system architecture because if they aren't implemented consistently, the overall system might encounter resource contention.

0740-7459/10/$26.00 © 2010 IEEE

Here's how I break down software architecture in practice. The conceptual architecture defines the concepts used to describe the system on an architectural level and the relationships among these concepts. Examples include *component, message queue, interface, async RPC call*, and *actor*. The application architecture uses instances of these concepts to define a concrete system—for example, the component `CustomerManagement` might implement the `CustomerLookup` interface. Once we've defined the conceptual architecture, we can then use the nonfunctional requirements to determine a technology mapping for these concepts—for example, we can map components to Enterprise JavaBeans (EJBs) or Windows Communication Foundation (WCF) components. Finally, we define a programming model, which is how developers express the application architecture. This includes ways (to stick with the previous examples) to describe and implement components, an API to create messages and post them to a queue, or the actual protocol for acquiring and locking resources.

Of course, down-to-earth team members (not ivory-tower architects) should develop all this incrementally. However, even in an agile/iterative/incremental development approach, you should ensure that at any given time the system consistently conforms to whatever is defined as the architecture. The notion of the architecture evolves over time, but at any relevant point in time—for example, an incremental release—you want the system to be consistent. Although this article isn't on process, it's important to keep this in mind.

Architecture DSLs

So how does all this relate to DSLs? An architecture DSL (ADSL) is a language that directly expresses a system's architecture. "Directly" means that the language's abstract syntax contains constructs for all the ingredients of the conceptual architecture. Developers can thus use the language to describe a system on the architectural level. Code generation creates representations of the application architecture in the implementation languages, automating the technology mapping (once the architects and developers have manually made the decisions about the mapping, of course). Finally, developers implement the remaining, nongenerated functionality using regular programming languages, based on the programming model.

To enable architecture model checking and validation, and to support meaningful code generation, the ADSL must be formal. This means that the semantics are unambiguous, and checkers and generators can process the models. "Just pictures" isn't enough.

For humans to be able to use the language, language developers must define a suitable concrete syntax. "Suitable" means that it must efficiently support the abstraction of architectural concerns by developers. It must also integrate well with existing development tools—specifically, source code management systems.

I don't advocate defining a generic, reusable language such as the various existing ADLs or UML. On the basis of my experience, the best approach is to define the ADSL in real time as you understand, define, and evolve a system's conceptual architecture. The process of defining the language actually helps the architecture/development team better understand, clarify, and refine architectural abstractions. This goes back to the previous notion: having a language to express architectural aspects of a system helps you reason about and discuss the architecture.

The best approach is to define the ADSL in real time as you understand, define, and evolve a system's conceptual architecture.

An Example

This section describes an ADSL I've implemented for a real system with a customer from the airport management systems domain. The example, as well as the real-world systems mentioned in the first section, uses textual syntax to express the architecture. I'll explain later why I made this decision.

A customer for one of my consulting gigs decided to build a new airport management system. Airports use such systems to track and publish information about whether airplanes have landed, whether they're late, the aircraft's technical status, and so on. The system also populates the online-tracking system on the Web and at airport information monitors.

In many ways, this is a typical distributed system, consisting of many machines running different parts of the overall system. There's a central data center to do some of the heavy number crunching, but there's additional machines distributed over relatively large areas.

Consequently, you can't simply shut down or update the system as a whole, thus introducing the requirement 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++, and C#. This isn't an atypical requirement for large systems either: often, you use Java technology for the back end and .NET technology for a Windows front end. My customer had decided that the system's backbone would be a messaging infrastructure and

Figure 1. Initial set of components, their ports, and the interfaces they use.

```
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
}
```

Figure 2. A couple of component instances and their connections.

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

Figure 3. Namespaces are used to structure the components into logical units.

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

Figure 4. A composition groups instances and their connections into a named entity.

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

was evaluating different messaging tools for performance and throughput.

When I arrived, they briefed me on the system's details and the architectural decisions they'd already made, and asked whether all this made sense. It turned out quickly that, although 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 discussed the system, they stumbled over disagreements and misunderstandings all the time. They had no language for the architecture.

So, we started building a language. We built the grammar, some constraints, and an editor while we discussed the architecture during a two-day workshop.

We started with the notion of a component. At that point, this notion is relatively loosely defined. It's simply the smallest architecturally relevant building block, a piece of encapsulated functionality. We also assumed that components can be instantiated, making components the architectural equivalent to classes in OO programming. (Coming up with the concrete set of components, their responsibilities, and consequently their interfaces isn't necessarily trivial either. Techniques such as CRC cards can help here.[1])

To enable components to interact, we also introduced the notion of interfaces, as well as ports, which are named-communication end points typed with an interface. Ports have a direction (`provides`, `requires`) as well as a cardinality (see Figure 1). It's important to state not just which interfaces a component provides but also which interfaces it requires because we want to understand (and later, analyze with a tool) component dependencies.

We then looked at instantiation. There are many aircraft, each running an `AircraftModule`, and even more `InfoScreens`. So we needed to express instances of components. These are logical instances. We haven't yet made decisions about pooling and redundant physical instances. We also introduced connectors to define actual communication paths between components and their ports (see Figure 2).

At some point, it became clear that to avoid getting lost in all the components, instances, and connectors, we needed some kind of namespace. We'd also need to distribute things to different files (the tool support ensures that `go to definition` and `find references` still work; see Figure 3).

It's 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 (see Figure 4).

Of course in a real system, `DelayCalculator` would have to dynamically discover all the available `InfoScreens` at runtime. There isn't much point in manually describing those connections individually. So we introduced dynamic connectors: we specify a query that executes at runtime against some kind of naming/trader/lookup/registry infrastructure. It reexecutes every 60 seconds to find the `InfoScreens` that had just come online (see Figure 5).

We can use a similar approach to realize load balancing or fault tolerance. A static connector can point to a primary and a backup instance, or a dynamic query can reexecute when the currently used component becomes unavailable.

To support registration of instances, we add additional syntax to their definitions. A registered instance registers itself with the registry, using its name (qualified through the namespace) and all provided interfaces. Additional parameters can be specified—Figure 6 registers a primary and a backup instance for DelayCalculator.

Until then, we hadn't really defined an interface. We knew that we'd like to build the system on the basis of a messaging infrastructure. Our first idea was that an interface is a collection of messages, in which 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). We also defined several message interaction patterns; Figure 7 shows examples of oneway and request-reply.

We talked a long time about suitable message interaction patterns. After a while, it turned out that a core use case 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 must be pushed out to all 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. Although messaging is a suitable transport abstraction for these things, architecturally we're really talking about replicated data 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 include full updates of all items in the collection, incremental updates of just one or a few items, invalidations, and so on.

Once we understood that in addition to messaging there's this additional communication abstraction in the system, we added this to our ADSL and were able to write something like Figure 8.

We define data structures and replicated items. Components can then publish or consume those replicated data items. We state that the publisher publishes the replicated data whenever something changes in the local data structure. However, the InfoScreen needs an update only every 60 seconds (as well as a full load of data when it starts up).

A description based on data replication is much more concise than a description based on messages. We can automatically derive the kinds of messages needed for a full update, an incremental update, and invalidation. The description also much more clearly reflects the actual architectural intent: it

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

Figure 5. A dynamic connector finds its targets by querying a registry at runtime.

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

Figure 6. Instances use registration parameters to make themselves known to the registry.

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

Figure 7. An interface that contains a one-way and a request/reply message.

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

replicated singleton flights {
   flights: FlightInfo[]
}

component DelayCalculator {
   publishes flights { publication = onchange }
}

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

Figure 8. Data replication as a new interaction paradigm.

Figure 9. Protocol state machines are used to define valid message sequences.

```
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
    }
  }
}
```

Figure 10. A new implementation of the same component DelayCalculator.

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

Figure 11. A new version of the existing DelayCalculator component.

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

expresses what we want to do (replicate data) better than a lower-level description of how we want to do it (sending messages).

Although replication is a core concept for data, there's still a need for messages, not just as an implementation detail but also as a way to express architectural intent. It's useful to add more semantics to an interface—for example, defining valid sequencing of messages. A well-known method is to use protocol state machines.

Figure 9 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. The only valid message is registerAircraft. Once this is received, it transitions 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 registered.

I mentioned earlier that the system is distributed geographically. This means you can't update all parts of the systems (for example, all InfoScreens or all AircraftModules) in one swoop. So, several versions of the same component might be running in the system. To make this feasible, we need to put many nontrivial things in place at runtime. But the basic requirement is that you need the ability to mark up versions of components and check them for compatibility with old versions.

Figure 10 expresses that DelayCalculatorV2 is a new implementation of DelayCalculator. newImplOf means that no externally visible aspects change. This is why no ports are declared. For all intents and purposes, it's the same thing. Just maybe a couple of bugs are fixed.

If you really want to evolve a component (that is, change its external signature), you can write it like Figure 11.

The keyword is newVersionOf, and now you can provide additional features (such as the somethingElse port) and remove required ports. You can't add additional required ports or remove any of the provided ports because that would destroy the "plug-in compatibility." Constraints enforce these rules on the model level.

What We Did in a Nutshell

My approach proposes the definition of a formal language for a system's conceptual architecture. You develop the language as your understanding of the architecture grows, in real time: we built the previous example during a two-day architecture exploration workshop. The language therefore always resembles the complete understanding of your architecture in a clear and unambiguous way.

We separated what we wanted the system to do from how it would achieve it. All the technology discussions then become an implementation detail of the conceptual descriptions given here (albeit of course, an important implementation detail). We also had a clear, 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.

As we enhance the language, we also describe the application architecture using that language. We're building ADSLs.

We can use DSLs to specify any aspect of a software system. There's a lot of hype about using a DSL to describe business functionalities (for

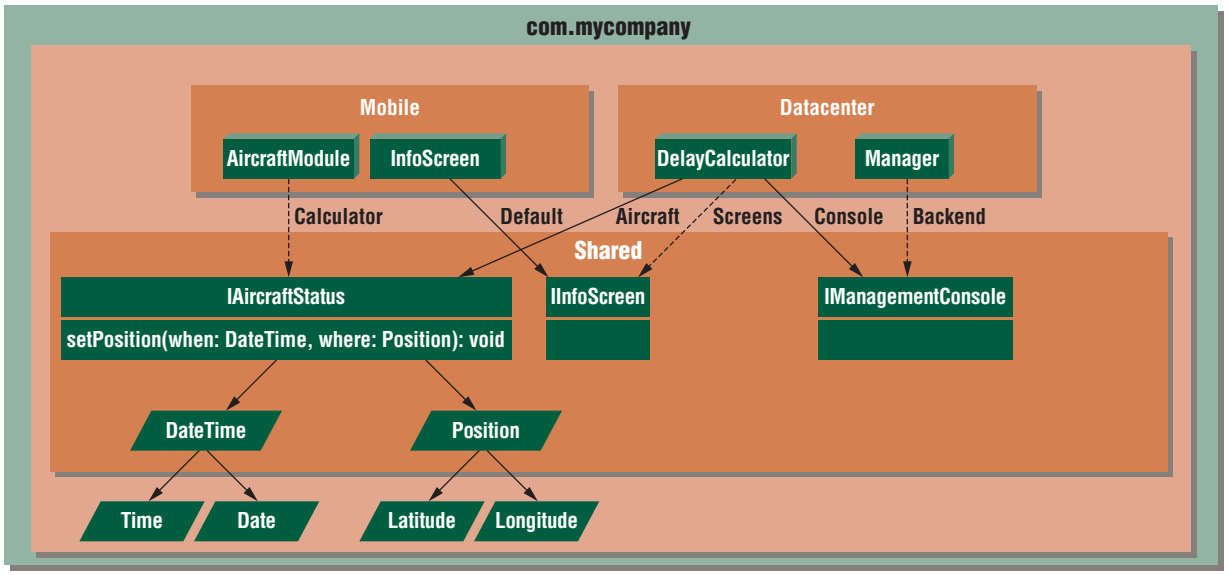


Figure 12. A Graphviz-generated diagram shows relationships between some components, interfaces, and data structures.

example, calculation rules in an insurance system). Although this is a worthwhile use of DSLs, it's also worthwhile to use them to describe software architecture.

Benefits

This approach has the following benefits. Everyone involved will clearly understand the concepts describing the system; there's an unambiguous vocabulary to describe applications. Models can be analyzed and used as a basis for code generation. The architecture is freed from implementation details—in other words, conceptual architecture and technology decisions are decoupled, making both easier to evolve. We can define a clear programming model based on the conceptual architecture. Last but not least, the architects can contribute directly to the project by building (or helping to build) the languages and related tools—an artifact that the rest of the team can actually use. In a very real sense, you could call this executable architecture.

Why Textual?

Textual DSLs have several advantages. Here are some of them:

- Languages as well as editors are easier to build than custom graphical editors (although this statement's validity depends on the tooling used).
- Textual artifacts integrate much better with existing developer tooling than repository-based models. You can use well-known diff/merge tools, and it's much easier to version, tag, and branch models and code together. Generally, the tooling is more lightweight.
- Model evolution (that is, adaptation of models when the DSL evolves over time) is much simpler. Although you can use the standard approach—a model-to-model transformation from the old to the new version—you can always use search and replace or grep, technologies familiar to everybody, as a fallback.

In cases where a graphical notation is useful to see relationships between architectural elements, you can use tools such as Graphviz (http://graphviz.net) or Prefuse (http://prefuse.org). Because the model contains all the relevant data, you can easily transform the model into a format these tools can process. Figure 1 illustrates a Graphviz-generated diagram showing namespaces, components, interfaces, and datatypes as well as their relationships. It was created by transforming the textual model into a dot file, Graphviz's way to describe graphs.

Tooling

Several textual or graphical tools support the definition of DSLs. All of them support the definition of abstract and concrete syntax, as well as user-friendly editors. In the textual category, you might want to look at the Eclipse Textual Modeling Framework Xtext (http://eclipse.org/xtext), Jetbrains' Meta Programming System (http://jetbrains.com/mps), or Intentional's Domain Workbench

We aren't ultimately interested in correct models but in correctly running systems.

(www.intentsoft.com). In the graphical world, take a look at MetaEdit+ (www.metacase.com).

A detailed discussion on using these tools is beyond this article's scope.

Validating the Model

We need validation rules that constrain the model even further than what we can express with the language grammar. Examples include name-uniqueness, type checks, or versioning constraints. To implement such constraints, we need two things: First, the constraint itself must be describable via a formal algorithm (expressed, for example, in the Object Constraint Language). Second, the model must contain the data needed to evaluate the algorithm. For example, if you want to verify whether a certain deployment scheme is feasible, you might have to put the available network bandwidth, the timing and frequency of messages, and the size of primitive data types into the model (maybe in a separate file). Although capturing this data sounds like a burden, this is actually an advantage: it's core architectural knowledge.

Generating Code

The primary benefit of developing and using the ADSL is to better understand concepts by removing any ambiguity and defining them formally. It helps you understand the system and get rid of unwanted technology dependencies. However, we aren't ultimately interested in correct models but in correctly running systems. To make this possible, use code generation—it simplifies and constrains application implementation:

- We generate an API and code the implementation against it. That API can be nontrivial—for example, taking into account the various messaging paradigms or data replication. The generated API lets developers implement components in a way that doesn't depend on specific technologies (it does depend on a specific programming language, though). The generated API hides technology specifics from the component implementation code.
- We also generate the code that's necessary to run the components (including their technology-neutral implementation) on a suitable middleware infrastructure. This code is called the technology-mapping layer or glue code. It typically also contains configuration files for the target platform. Sometimes this requires additional "mix-in models" that specify configuration details for the particular platform. As an additional benefit, the generators capture best practices in working with the selected technologies.

It's feasible to generate APIs for several target languages (supporting component implementation in various languages) or generate glue code for several target platforms (supporting the execution of the same component on different middleware platforms—for example, for local testing). This nicely supports potential multiplatform requirements and provides a way to scale or evolve the infrastructure over time.

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, alternatives exist for component implementation. Instead of using general-purpose programming languages, you can use formalisms specific to certain classes of behavior: state machines, business rules, or workflows. You can also define and use a DSL for certain classes of functionality in a specific business domain. Note how this connects the dots to business DSLs.

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

Standards, ADLs, and UML

Describing architecture with formal languages isn't new. Various communities recommend using architecture description languages (ADLs)—for example, Acme (www-2.cs.cmu.edu/~acme), EAST-ADL (http://en.wikipedia.org/wiki/EAST-ADL), xADL (www.isr.uci.edu/projects/xarchuci), and Cosmic (www.dre.vanderbilt.edu/CoSMIC) or UML—to this end. However, those approaches advocate using existing, generic languages for specifying architecture, although some, including UML, can be customized somewhat.

However, this completely misses my point. I haven't experienced much benefit in shoehorning architecture descriptions into the (typically, very limited) constructs provided by predefined languages. One of the core activities of the approach I propose is actually building your own language to capture your system's conceptual architecture.

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

To use any architecture modeling language successfully, people must first understand the architectural concepts they're dealing with. Even if they use UML, people must still map architectural concepts to the language, which often requires an architecture-specific profile. Is a profiled UML still standard?

I don't propose to ignore standards altogether. The tools I use are built on the Essential Meta Object Facility (EMOF), which is an Object Management Group standard, just like UML. It's just on a different metalevel.

Of course, you can use my approach with UML profiles. It works, but not as well as the approach based on textual languages I explained in this article. Reasons for this include the following:

- 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's the wrong focus.
- Customizing UML tools (beyond different colors or icons for stereotyped elements) is restricted, cumbersome, and tool-specific (that is, nonstandard). You must expend much effort to make profiled UML look and feel as if it was a DSL. And, of course, you still have to deal with the complex UML metamodel when processing the models.
- UML tools typically don't integrate well with existing development infrastructure (editors, source code management systems, and diff/merge). That's a huge problem now that models play the role of source code (as opposed to being "analysis pictures"). Tool integration issues can make developer acceptance very challenging.

I've done architecture modeling and code generation based on UML in several projects. However, after I started using the textual languages together with automatically generated editors, I never looked back: using textual languages is much lighter weight and more productive.

A Few Words on Process and People

Define the language iteratively. Build example models all the time, and keep a reference model (one that exercises all parts of the language) current. Interleave language definition and generator construction to verify you have all the data in the models you need for generating meaningful code.

Don't define just a nice language but also a user-friendly editor for the application developers who will use the language. Although language and editor definition is quite straightforward with the modern tools mentioned earlier and much simpler that what you might know from traditional parser generators such as lex/yacc or antlr, make sure the developers who define the language can "think meta." For some people, this can be a challenge.

Please document how to use the language, editor, and code generator using examples and walkthroughs. Nobody reads reference manuals. In the documentation, also explain the architectural concepts expressed via the language. People won't understand the language unless they understand what it describes. And please set aside time for taking into account feedback (bugs and improvements) from the language users. Throwing something such as a custom language "over the fence" and then not supporting it is a death sentence for the approach.

This article doesn't contain a consistent methodology. I believe that

- a good idea proven in real projects (as suggested in this article),
- general software development experience and common sense (such as iterative and incremental development and testing),
- (architectural) experience (which I expect you to bring to the table), and (of course)
- learning how to use the language engineering tools

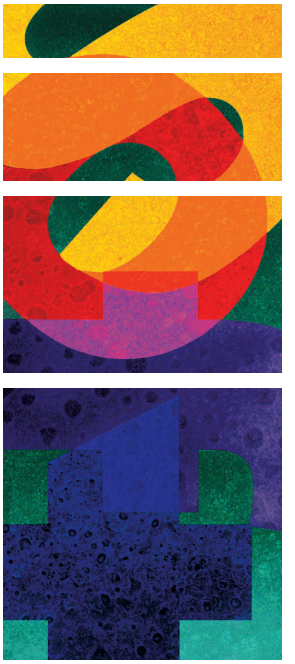
are enough to apply this approach successfully in the real world.

Challenges

The approach does have some challenges.

Although the tools for defining DSLs, editors, and code generators are improving, building the language tools involves some overhead. I don't count the analytical part of distilling the architectural concepts and defining the language as overhead, because you'll have to do that in some form anyway during architecture definition. But tool construction is additional work. In my experience it pays off in most settings, but it's an initial investment that you must make.

Convincing developers and especially management to abandon the use of standards (UML and ADLs) for architecture description can be a challenge. To overcome it, I like to create small but meaningful prototypes and show the benefits of being in control of your language and not being constrained by a predefined language.



IEEE Software

About the Author

Markus Völter is an independent researcher, consultant, and coach for Itemis. His focus is on software architecture, model-driven software development, domain-specific languages, and product-line engineering. He is also a member of the *IEEE Software* Advisory Board. Contact him at voelter@acm.org.

Another challenge is that DSL semantics usually aren't formally specified. Typically, you write documentation that explains the meaning of language constructs. The code generator also serves as a vehicle for semantics definition because it maps the language constructs to executable concepts in a target language with known semantics. However, if you generate the same models into code in several languages, then you need one central semantics definition against which you validate the various implementations. In practice, such a definition isn't feasible. To overcome this, you write (or generate) tests.

Finally, if you have many related but not identical systems, there will be common parts between several architecture DSLs. So, you might want to modularize your DSL into several combinable language modules. You can then extend this "base language" with concepts specific to the system in question. However, because of limitations in today's language-engineering tools, such an approach is still a bit challenging. I've written about my prototypical implementations in a previous paper.[2] Tools such as JetBrain's MPS (Meta Programming System) mandate another try, though.

I want to reemphasize the importance of defining the language specifically for your context, together with the architecture team as the architecture is built—a generic approach doesn't reap the same benefits. Also, experience with my customers shows one thing consistently: once the team is familiar with the capabilities of today's tools, they consider this approach much more feasible than they had initially thought.

References

1. K. Beck and W. Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," *Proc. Conf. Object-Oriented Programming Systems Languages and Applications* (OOPSLA 89), ACM Press, 1989, pp. 1–6; http://c2.com/doc/oopsla89/paper.html.
2. M. Voelter, "A Family of Languages for Architecture Description," *Proc. 8th Ann. Conf. Object-Oriented Programming Systems Languages and Applications* (OOPSLA 08), Workshop Domain-Specific Modeling, Univ. Alabama at Birmingham, 2008; www.voelter.de/data/pub/DSM2008.pdf.