Software Architecture

A pattern language for building sustainable software architectures

Version 1.0, Mar 10, 2005

(c) 2005 Markus Völter, Heidenheim, Germany
voelter@acm.org, www.voelter.de

Abstract

Recently, the business of software architecture has become one of technology hypes and technology geeks. An architecture often defines itself by the primary technology it is built upon. Developers are given a J2EE book and then let loose. And then the project fails, although “we used an industry standard” … How come?

The craft of defining an architecture – independent of buzzwords – has gone out of fashion. Designing architectures on a conceptual level is not something people learn, or read books about (there aren't many books on this topic!). The view for the essential aspects of an architecture is obstructed by all the technology crap.

This paper outlines a couple of best practices that I consider essential when building a real-world software architecture. It could be called an “architectural process” if you wish…

Introduction

Why write a paper on software architecture? There are several reasons. The most important is that I think the craft of software architecture in current industrial practice is not what it should be.

Before I start bashing current practice, I want to make what this paper is actually about. I – personally – think, there is a difference between the functional architecture of a system, and the technical architecture. The functional architecture is aligned with the domain. For example, it is about understanding processes, responsibilities, variabilities; in one word it's about what the system should do. Technical architecture on the other hand is about how the functional architecture is implemented: do we have components? Are we distributed? How do we scale? What about systems management? How do we realize the required QoS? How are processes rendered? Do we use a relational or a non-relational DB? In this paper, I focus primarily on technical architecture. Specifically, I want to show, how we can come up with a technical architecture that makes the development of the functional architecture (i.e. the realization of the use cases for the system) as pain-free as possible.

Current state of the practice

Software architecture is too much *technology driven*. You hear statements such as "we have a web-service architecture". Obviously, this statement is stupid because it describes only one aspect of the overall system (communication), and second, web services are a particular implementation technology for that aspect. There is much more to say about the architecture (even about the communication aspect), than just a realization technology. The same is true with "EJB Architectures" or a "Thin Client Architecture". A too early commitment to a specific technology usually results in blindness for the concepts and a too tight binding to the particular technology. The latter, in turn, results in a complicated programming model, bad testability and no flexibility to change the technology, as QoS requirements evolve. It disguises the view for the really important things.

Another problem is the *hype factor*. While it is good practice to characterize an architecture as implementing a certain architectural style or pattern [POSA1], some of the buzzwords used today are not even clearly defined. A "service based architecture" is a classic. Nobody knows what this *really* is, and how it is different from well-designed component-based systems. And there are many misunderstandings. People say "SOA", and others understand "web service"… Also, since technologies are often hyped, a hype-based architecture often leads to too early (and wrong) technology decisions – see above!

Another problem is what we usually call *industry standards*. A long time ago, the process of coming up with a standard was basically the following: try a couple of alternatives; see which one is best; set up a committee that defines the standard, based on the experiences made before, the standard is usually close to the solution that worked best. Today, this is different. Standards are often defined by a group of (future) vendors. Either they already have tools, and the standard must accommodate for all the solutions of all the tools of all the vendors in the group, or, there is no practical previous experience and the standard is defined "from scratch". As a consequence of this approach, standards are often not usable (because there was no previous experience), or overly complicated (because it must satisfy all the vendors…). Thus, if you use standards for too many aspects of your system, your system will be complicated!

All these things together prevent people from thinking about the really relevant aspects of an architecture. In my opinion, these include architectural patterns, logical structures (architectural metamodels), programming models for developers, testability, and the ability to realize key QoS concerns.

The following pages sketch something that I consider a reasonable approach to software architecture. It also paves the way to automating many aspects of the software development, a key ingredient to model-driven software development [SV05] and Product Line Engineering.

Of course I am not the only one seeing this problem in current software architecture. There are good architectural resources you should definitely read, such as [POSA1, 2 and 3] as well as [JB00], [VSW02] and [VKZ04].

Patterns Overview

The approach is structured into three phases.

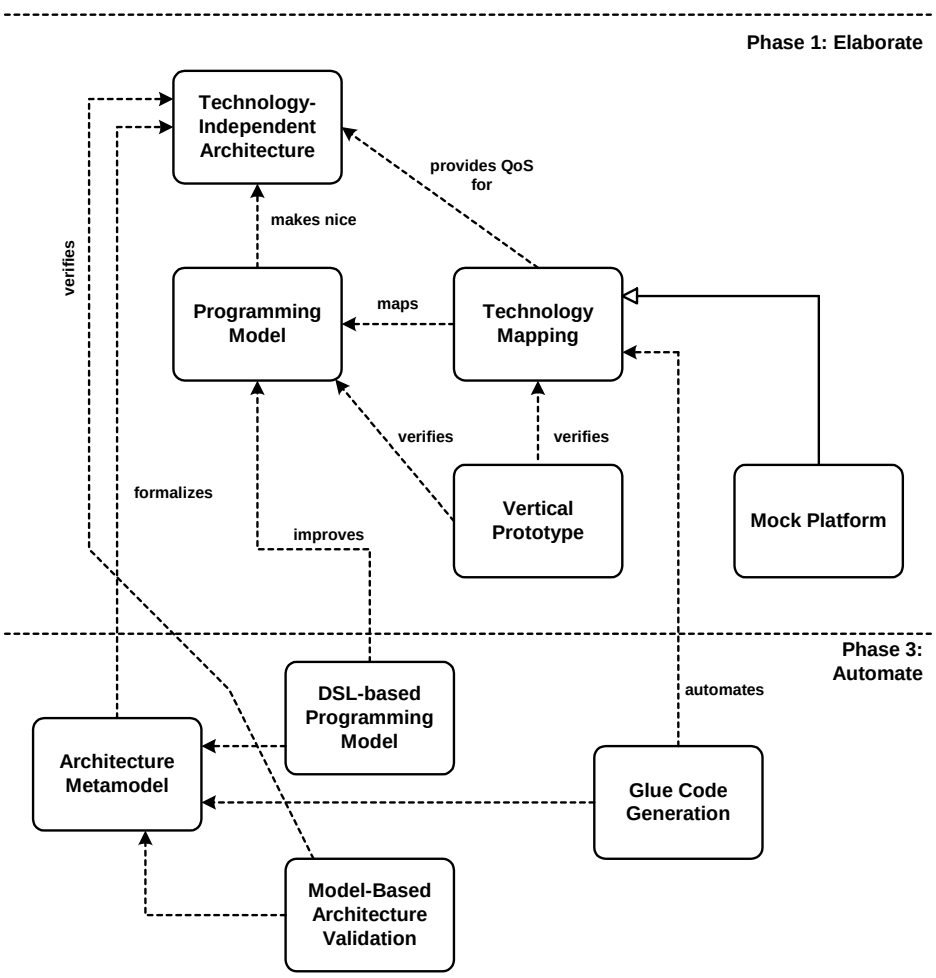
Elaboration: In the first phase, the elaboration, you define a TECHNOLOGY-INDEPENDENT ARCHITECTURE. Based on it, you define a nice and workable PROGRAMMING MODEL for the developers that work with the architecture. In order to let developers run their stuff locally, a MOCK PLATFORM is essential. Finally in this phase, you define one or more TECHNOLOGY MAPPINGS which project the TECHNOLOGY-INDEPENDENT ARCHITECTURE to a particular platform that provides the required/desired QoS features. A VERTICAL PROTOTYPE verifies that the system performs as desired – here is where you run the first load tests and optimize for performance – and that developers can work efficiently with the PROGRAMMING MODEL.

Iteration: The second phase iterates over the steps in the first phase. While I generally recommend an agile approach, I want to outline

explicitly the fact that you typically don't get it right the first time. You usually have to perform some of the steps several times, especially the TECHNOLOGY MAPPING and the resulting VERTICAL PROTOTYPE. It is important that you do this *before* you dive into phase 3: Automation.

Automation: The third phase aims at automating some of the steps defined in the first, and refined in the second phase, making the architecture useful for larger projects and teams. First, you will typically want to GENERATE GLUE CODE to automate the TECHNOLOGY MAPPING. Also, you often notice that even the PROGRAMMING MODEL involves some tedious repetitive implementation steps that could be expressed more briefly with a DSL-BASED PROGRAMMING MODEL. Finally, MODEL-BASED ARCHITECTURE VERIFICATION helps ensure that the architecture is used "correctly" even in large teams.

The following illustration shows the patterns and their dependencies.



Example and Known Uses

Throughout this pattern language I use a running example. The example is taken from the domain of business systems and should be readily understandable for everybody.

This pattern language has been used over and over again in successful software projects. As a consultant I have used it (or seen it being used) in various projects in different domains. It is especially interesting to see that this approach is not limited to enterprise architecture (as one might guess from the example). The following non-exhaustive list provides some pointers:

Embedded Components: The small components project [MV02] has basically outlined how to use components in embedded systems. In the context of the AUTOSAR standard [AS], I have contributed to a prototype project at BMW Car IT which has implemented the standard (for some information on it, see [RV05]. In this project it was clear from the beginning that a model-driven approach would be required.

Enterprise Systems: At a customer I cannot disclose at this time, a business system was built that resembles the example in this paper conceptually. Here it was *not* clear from the beginning, that models and code generation would be useful, the customer was quite skeptical. However, since the Phase 1 patterns had been used successfully, the potential for MDSD has been recognized, and the Phase 3 Patterns had been added later.

Radio Astronomy: In a project that develops management and control software for a future radio telescope array [ALMA] a distributed component infrastructure had been built that uses the Patterns in Phase 1, together with GLUE CODE GENERATION for remote transport using CORBA and transparent value object serialization to XML. The component infrastructure is available for Java and C++.

Phase 1 – Elaborate!

This section outlines best practices and approaches which I think are important and applicable for all kinds of projects – you don't want to go without these. This first elaboration phase should be handled by a small team, before the architecture is rolled out to the team as a whole.

Example. We want to build an enterprise system that contains various subsystems such as customer management, billing and catalogs. In addition to managing the data using a database, forms and the like, we also have to manage the

associated long-running business processes. We will look at how we can attack this problem below.

■ Technology-Independent Architecture

Context

You have to define a software architecture for a non-trivial system or product line.

Problem

How do you define a software architecture that is well-defined, long-lived and feasible for use in practice? The architecture has to be reasonable simply and explainable on a beer mat[1].

Forces

- You want to make sure that the architectural concepts can be communicated to stakeholders and developers
- Implementation of functional requirements should be as efficient as possible.
- The architecture must "survive" a long time, longer than the typical hype or technology cycles
- The architecture might have to evolve with respect to QoS levels such as performance, resource consumption or scalability.

Solution

Define the architectural concepts independent of specific technologies and implementation strategies. Clearly define concepts, constraints and relationships of the architectural building blocks – a glossary or an ARCHITECTURAL METAMODEL can help here. Define a TECHNOLOGY MAPPING in a later phase to map the artifacts defined here to a particular implementation platform.

> Use the well-known architectural styles and patterns here. Typically these are best practices for architecting certain kinds of systems independent of a particular technology. They provide a reasonable starting point for defining (aspects of) your system's architecture.

[1] …referencing a revolutionary idea for tax declarations in Germany ☺

Example. As part of our example, we decide that our system will be built from components. Each component can provide a number of interfaces. It can also use a number of interfaces (provided by other components). Communication is synchronous. Communication is also restricted to be local, no remoting is supported on this level. We design components to be stateless.

In addition to components, we also explicitly support business processes. These are modeled as a state machine. Components can trigger the state machine by supplying events to them. Other components can be triggered by the state machine, resulting in the invocation of certain operations. Communication to/from processes is asynchronous. Remote communication is supported.

Rationale, Discussion and Consequences

If you use less complicated technology, you can focus more on the structure, responsibilities and collaborations among the parts of your systems. Implementation of functionality becomes more efficient. And you don't have to educate all developers with all the details of the various technologies that you'll eventually use.

However, the interesting question is: How much technology is in a technology-independent architecture? Is AOP ok? In my opinion, all technologies or approaches that bring provide additional expressive concepts are useful in a TECHNOLOGY-INDEPENDENT ARCHITECTURE. AOP is such a candidate. The notion of components is also such a concept. Message queues, pipes and filters and in general, architectural patterns are also useful.

When documenting and communicating your TECHNOLOGY-INDEPENDENT ARCHITECTURE models are useful. I am *not* talking about formal models as they're used in model-driven software development – we'll take a look at these later. Simple box and line diagrams, layer diagrams, sequence, state or activity charts can help to describe what the architecture is about. They are used for illustrative purposes, to help reason about the system, or to communicate the architecture. For this very reason, they are often drawn on beer mats, flip charts or with the help of Visio or Powerpoint. While these are not formal, you should still make sure that you define what a particular visual element means intuitively – boxes and lines with no defined meaning are not very useful, even for non-formal diagrams.

■ Programming Model

Context

You have defined a TECHNOLOGY INDEPENDENT ARCHITECTURE. Your architecture is rolled out, developers have to implement functionality against this architecture.

Problem

The architecture is a consequence of many non-functional requirements and the basic functional application structure, which might make the architecture non-trivial and hard to comprehend for developers. How can you make the architecture accessible to (large numbers of) developers?

Forces

- You want to make sure the architecture is used "correctly" to make sure it's benefits can actually materialize.
- You have developers of different qualifications in the project team. All of them have to work with the architecture.
- You want to be able to review application code easily and effectively.
- Your applications must remain testable.

Solution

Define a simple and consistent programming model. A programming model describes how an architecture is used from a developer's perspective. It is the "architecture API". The programming model must be optimized for typical tasks, but allow for more advanced things if necessary. Note that a main constituent of a programming model is a How-To Guide that walks developers through the process of building an application.

Example. The programming model uses a simple IOC approach à la Spring to define component dependencies on an interface level. An external XML files takes care of the configuration of the instances. The following piece of code shows the implementation of a simple example component. Note how we use Java 5 annotations

```
public @component class ExampleComponent
  implements HelloWorld {// provides HelloWorld

  private IConsole console;
```

```
  public @resource void setConsole( IConsole c ) {
    this.console = c;            // setter for console
  }                              // component

  public void sayHello( String s ) {
    console.write( s );
  }
}
```

The process states are implemented using the State pattern (from the GoF) book. Processes engines are components like any other. For the triggers, they provide an interface that contains only void operations (which can easily be sent asynchronously). They also define interfaces with the actions (also implemented as void methods, for the same reason) that those components can implement that want to be notified of state changes. The following code shows the skeleton of a component that hosts a state machine; it has two triggers (T1 and T2) and calls a single action on a resource component. It also has one guard that needs to be evaluated.

```
public @process class SomeProcess
                implements ISomeProcessTrigger {

  private IHelloWorld resource;

  public @resource void setResource( IHelloWorld w ) {
    this.resource = w;
  }

  public @trigger void T1( int procID ) {
    SomeProcessInstance i = loadProcess( procID );
    if ( guardG1() ) {
      // advance to another state…
    }
  }

  public @trigger void T2( int procID ) {
    SomeProcessInstance i = loadProcess( procID );
    // …
    resource.sayHello( "hello" );
  }
}
```

The actual process instance is loaded by the process component upon a received trigger. Triggers (and as a consequence, the respective interface) contain a unique process ID.

Rationale, Discussion and Consequences

The most important guideline when defining a programming model is usability and understandability for the developer. This is the reason

why the documentation for the programming model should always be in the form of tutorials or walkthroughs, not as a reference manual! Frameworks, libraries, and as we'll see in DSL-BASED PROGRAMMING model, domain-specific languages are useful here.

Sometimes it's not possible to define a programming model completely unaware of the platform on which it will run (see TECHNOLOGY MAPPING). Sometimes the platform has consequences for the programming model. For example, if you want to be able to deploy something as an enterprise bean, you should not create objects yourself, since this will be done later by the application server. There are a couple of simple guidelines that help you come up with a programming model that stands a good chance that it can be mapped to various execution platforms:

- Always develop against interfaces, not implementations
- Never create objects yourself, always use factories
- Use factories to access resources (such as database connections)
- Stateless design is a good idea in enterprise systems
- Separate concerns: make sure a particular artifact does *one* thing, not five.

A good way to learn more about good PROGRAMMING MODELS and TECHNOLOGY-INDEPENDENT ARCHITECTURE can be found in Eric Evans wonderful book on Domain-Driven Design [EE03].

One of the reasons why a technology decision is made early in the project is the "political pressure" to use a certain technology. For example, your customer's company already has a global lifetime license for IBM's Websphere and DB2. You have no chance but to use those two. You might wonder whether the approach based on a TECHNOLOGY-INDEPENDENT ARCHITECTURE and explicit TECHNOLOGY MAPPINGS still work? In case the imposed technology is a good choice, the benefits of the approach described here still apply. In case the technology is not suitable (because it is overly complicated or unnecessarily powerful), life with the technology will be easier if you isolate it in the TECHNOLOGY MAPPING.

■ Technology Mapping

Context

You have defined a TECHNOLOGY INDEPENDENT ARCHITECTURE and a PROGRAMMING MODEL.

Problem

Your software has to deliver certain QoS levels. Implementing QoS as part of the project is costly. You might not even have the appropriate skills on the team. Also, your system might have to run with different levels of QoS, depending on the deployment scenario.

Forces

- You don't want to implement the advanced features that enable all the non-functional requirements yourself.
- You want to keep the conceptual discussions, as well as the PROGRAMMING MODEL free from those technical issues.
- You might want to run the system with various levels of QoS, with minimal cost for each.

Solution

Map the TECHNOLOGY-INDEPENDENT ARCHITECTURE to a specific platform that provides the requires QoS. Make the mapping to the technology explicit. Define rules how the conceptual structure of your system (the metamodel) can be mapped to the technology at hand. Define those rules clearly to make them amenable for GLUE CODE GENERATION.

Decide about standards usage here, not earlier. As mentioned, standards can be a problem, they can also be a huge benefit. For stuff that is not related to your core business, using standards is often useful. But keep in mind: First solve the problem. Then look for a standard. Not vice versa. And make sure PROGRAMMING MODEL hides the complexity.

Use technology-specific Design Patterns here. Once you decided on a certain platform, you have to make sure you use it correctly. Often, the platform is not really easy to use. If it is a commonly used platform, though, platform specific best practices and patterns are documented. Now is the time to look at these and use them as the basis for the TECHNOLOGY MAPPING.

Example. For the remote communication between business processes we will use web services. Since we transport rather simple trigger events implemented as asynchronous *oneway* methods, the mapping to the technology is trivial. So, from the business interfaces such as IHelloWorld, we generate a WSDL file, as well as the necessary endpoint implementation. Of

course we don't implement all the technology ourselves – we use on of the many available web service frameworks.

The infrastructure for running the application itself will be kept as simple as possible, i.e. Spring will be used as long a no advanced load balancing and transaction policies are required. The following is the spring configuration file for this simple example.

```
<beans>
  <bean id="proc" class="somePackage.SomeProcess">
    <property name="resource">
       <ref bean="hello"/>
    </property>
  </bean>
  <bean id="hello"
        class="somePackage.ExampleComponent">
    <property name="console">
      <ref bean="cons"/>
    </property>
  </bean>
  <bean id="cons" class="someframework.StdOutConsole">
</beans>
```

Once this becomes necessary, we will use Stateless Session EJBs. The necessary code to wrap our components inside beans is easy to write. So, for each bean, we write a remote/local interface, an implementation class that wraps our own implementation, as well as a deployment descriptor.

Persistence for the process instances – like any other persistent data – is managed using Hibernate. To make this possible, we create a data class for each process. It contains the id of the process's current state, as well as the values of the context attributes. Since this is a normal value object, using Hibernate to make it persistent is straight forward.

Rationale, Discussion and Consequences

Let's recap: The TECHNOLOGY-INDEPENDENT ARCHITECTURE defines the concepts that are available to build systems. The PROGRAMMING MODEL defines how these concepts are used from a developer's perspective. The TECHNOLOGY MAPPING defines rules how the PROGRAMMING MODEL artifacts are mapped to a particular technology.

The question is now, which technology do you chose? In general, this is determines by the QoS requirements you have to fulfill. Platforms are good at handling technical concerns such as transactions, distribution, threading, load-balancing, failover or persistence. You don't want to implement these yourself. So, always use the platform that provides the

services you need, in the QoS level you are required to deliver. Often this is deployment specific!

■ Mock Platform

Context

You have a nice PROGRAMMING MODEL in place.

Problem

Based on the PROGRAMMING MODEL, developers now know how to build applications. In addition to that, developers have to be able to run (parts of) the system locally, at least to run unit tests. How can you make sure developers can run "their stuff" locally without caring about the TECHNOLOGY MAPPING and its potentially non-trivial consequences for debugging and test setup?

Forces

- You want to make sure developers can run their code as early as possible
- You want to minimize dependencies of a particular developer on other project members, specifically those caring about non-functional requirements and the TECHNOLOGY MAPPING.
- You have to make sure developers can efficiently run unit tests.

Solution

Define the simplest TECHNOLOGY MAPPING that could possibly work. Provide a framework that mocks or stubs the architecture as far as possible. Make sure developers can test their application code without caring about QoS and technical infrastructure.

Example. Since we are already using a PROGRAMMING MODEL that resembles Spring, we use the Spring container to run the application components locally. Stubbing out parts is easy based on Springs XML configuration file. Since persistence is something that Hibernate takes care of for us, the MOCK PLATFORM simply ignores the persistence aspect.

Rationale, Discussion and Consequences

This pattern is essential in larger and potentially distributed teams to allow developers to run their own stuff without caring too much about

other people or infrastructure. This is essential for unit testing! Testing one's business logic is simply if you have your system well modularized. If you stick to the guidelines given in the PROGRAMMING MODEL pattern (interfaces, factories, separation of concerns) it is easy to mock technical infrastructure *and* other artifacts developed by other people.

Note that it's essential that you have a clearly defined programming model, otherwise you TECHNOLOGY MAPPING will not work reliably.

Note that the tests you run on the MOCK PLATFORM will not find QoS problems – QoS is provided by the execution platform.

■ Vertical Prototype

Context

You have a TECHNOLOGY INDEPENDENT ARCHITECTURE, a PROGRAMMING MODEL as well as a TECHNOLOGY MAPPING. The first implementations of functionality are available and tested using the MOCK PLATFORM.

Problem

Many of the non-functional requirements your architecture has to realize depend on the technology platform, which you selected only recently in the TECHNOLOGY MAPPING. This aspect cannot be verified using the MOCK PLATFORM, since it ignores most of these aspects. The mapping mechanism might even be inefficient. How do you make sure you don't run into dead-ends?

Forces

- You want to keep your architecture as free of technology specific stuff as possible.
- However, you want to be sure that you can address all the non-functional requirements.
- You want to make sure you don't invest into unworkable technology mappings

Solution

As soon as you have a reasonable understanding of the TECHNOLOGY INDEPENDENT ARCHITECTURE and the TECHNOLOGY MAPPING, make sure you test the non-functional requirements! Build a prototype application that uses all of the above and implements it only for a very small subset of the functional requirements. This specifically includes performance and load tests.

Work on performance improvements here, not earlier. It is bad practice to optimize design for performance from the beginning, since this often destroys good architectural practice. Of course, in certain domains, there are some really fundamental patterns to realize certain QoS properties (such as stateless design for large-scale business systems). You shouldn't ignore these intentionally at the beginning. Don't pretend to be dumber than you are!

Example. The vertical prototype includes parts of the customer and billing systems. Both kinds of interactions are required here. For creating an invoice, the billing system uses normal interfaces to query the customer subsystem for customer details. The invoicing process – incl. payment receipt and optional reminder management is based on a long-running process.

A scalability test was executed and resulted in two problems: For short running processes, the repeated loading and saving of persistent process state had become a problem. A caching layer was added. Second, web-service based communication with process components was a problem. Communication was changed to CORBA for remote cases that were inside the company – the external processes are still based on web services. Note that the application code did not have to be changed, only the adapters that mapped the logical communication to web services had to be extended to use CORBA.

Rationale, Discussion and Consequences

Vertical prototypes are a well-known approach to risk reduction. In the approach to architecture suggested in this paper, the vertical prototype is, however, even more critical than in other approaches since you have to verify that the (nice) programming model does not result in problems with regards to QoS later. You have to make sure the various aspects you define in your architecture really work together!

Phase 2 – Iterate!

Now that you have the basic mechanisms in place you should make sure that they actually work for your project. Therefore, iterate over the steps given above until they are reasonable stable and useful.

Then, roll out the architecture to the overall team. In case you have larger project teams, the TECHNOLOGY MAPPING is still too much work,

or if you don't arrive at a suitable PROGRAMMING MODEL, you should consider Part 3, Automate!

Example. There was the idea to use Spring not just as the MOCK PLATFORM, but also for the production environment. However, as a consequence of new requirements, this has become infeasible. Spring does not support two important features: Dynamic installation/de-installation of components, and isolations of components from each other, specifically with regards to using different classloaders. Both of these problems arose as a consequence the additional non-functional requirement that several versions of the same component have to run in one system.

As a consequence, the Eclipse platform has been chosen as the new execution framework. The PROGRAMMING MODEL did not change; the TECHNOLOGY MAPPING, however had to be adapted.

Phase 3 – Automate!

The steps outlined above are useful in any kind of project. In case your project is really large (i.e. you have a large number of developers), or in case your TECHNOLOGY MAPPING or the PROGRAMMING MODEL is too tedious to use, you should consider automating the development. The next set of patterns describes how.

■ Architecture Metamodel

Context

You have a TECHNOLOGY-INDEPENDENT ARCHITECTURE. You want to automate various tasks of the software development processes.

Problem

In order to be able to automate, you have to codify the rules of the TECHNOLOGY MAPPING and define a DSL-BASED PROGRAMMING MODEL. For both aspects, you have to be very clear and precise about the artifacts defined in your TECHNOLOGY-INDEPENENT ARCHITECTURE.

Forces

- Automation cannot happen if you can't formalize translation rules.

- An architecture definition based on prose text is not formal enough.
- When building models (as part of the DSL-BASED PROGRAMMING MODEL and for MODEL-BASED ARCHITECTURE VALIDATION) you have to have a formal basis.

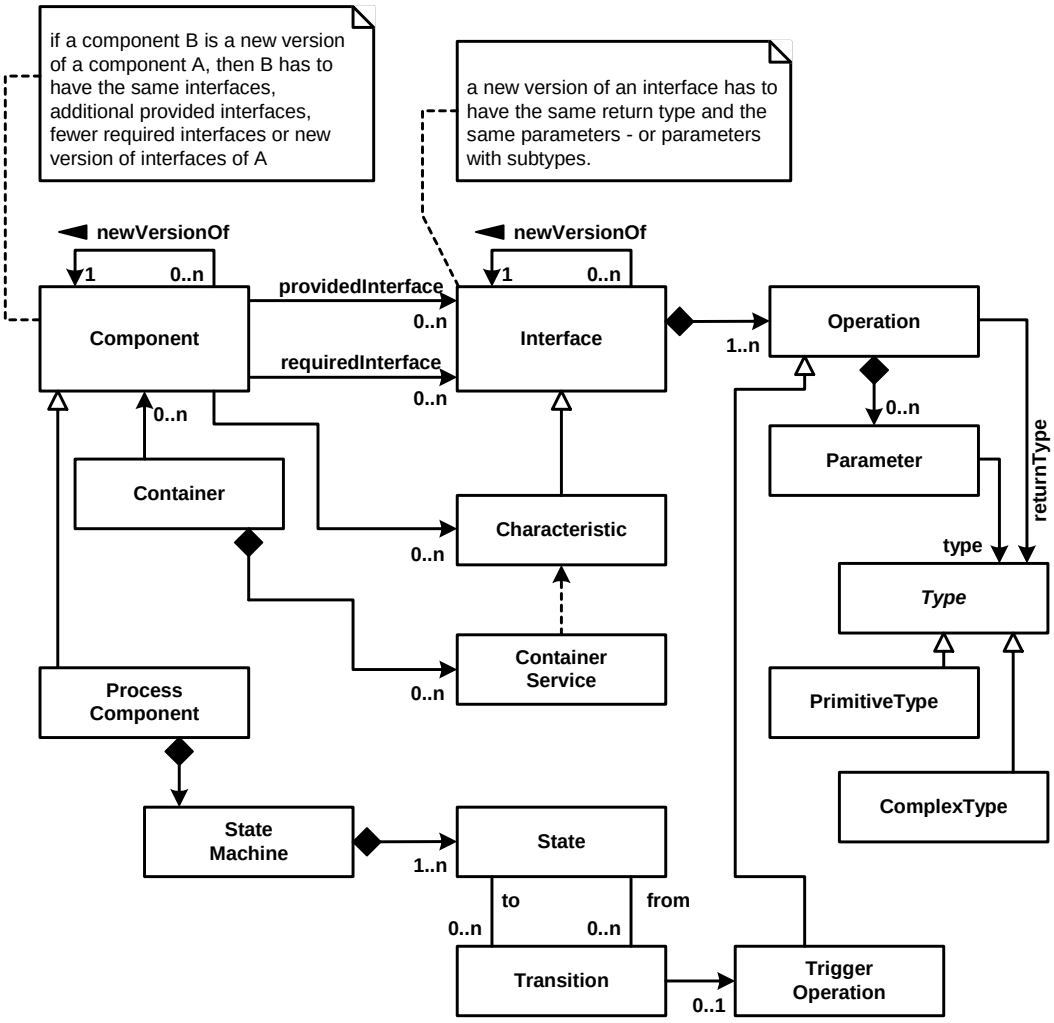
Solution

Define a formal architecture metamodel. An architecture metamodel formally defines the concepts of the TECHNOLOGY-INDEPENDENT ARCHITECTURE. Ideally this metamodel is also useful in the transformers/generators that are used to automate development.

Example. The metamodel for the system is shown below, it is rendered as a MOF model[2]. It is interesting to see that even the container is modular with respect to its services. Characteristics (special kinds of interfaces) are used to mark components with respect to the services they require. A container service (such as persistence of lifecycles) will take care of components that have a specific characteristic.

---

[2] In case you think it looks like UML: this is true, since UML and MOF share a common core.



Example models (at least some) are shown in the DSL-BASED PROGRAMMING MODEL pattern.

Rationale, Discussion and Consequences

Formalization is a double-edged sword. While it has some obvious benefits, it also requires a lot more work than informal models. The only way to justify the extra effort is additional benefits. The most useful benefit is if the metamodel doesn't just collect dust in a drawer, but is really used by tools in the development process. It is therefore essential that the metamodel is used, for example as part of the code generation in DSL-BASED PROGRAMMING MODELS and ARCHITECTURE-BASED MODEL VERIFICATION. See the *Implement the Metamodel* pattern in [MV04]

■ Glue Code Generation

Context

You have a TECHNOLOGY INDEPENDENT ARCHITECTURE, as well as a working TECHNOLOGY MAPPING.

Problem

The TECHNOLOGY MAPPING – if sufficiently stable – is typically repetitive and thus tedious and error prone to implement. Also, often information that is already defined in the artifacts of the PROGRAMMING MODEL have to be repeated in the TECHNOLOGY MAPPING code (method signatures are typical examples).

Forces

- A repetitive, standardized technology mapping is good since it is a sign of a well though-out architecture
- Repetitive implementations always tend to lead to errors and frustration.

Solution

Based on the specifications of the TECHNOLOGY MAPPING, use code generation to generate a glue code layer, and other adaptation artifacts such as descriptors, configuration files, etc. To make that feasible you might have to formalize your TECHNOLOGY INDEPENDENT ARCHITECTURE into an ARCHITECTURAL METAMODEL. In order to be able to get access to the necessary information for code generation, you might have to use a DSL-BASED PROGRAMMING MODEL.

Example. Our scenario has several useful locations for glue code generation.

- We generate the Hibernate mapping files
- We generate the web service and CORBA adapters based on the interfaces and data types that are used for communication. The generator uses reflection to obtain the necessary type information.
- Finally, we generate the process interfaces from the state machine implementations.

In the programming model, we use Java 5 annotations to mark up those aspects that cannot be derived by using reflection alone. Annotations can help a code generator to "know what to generate" without making the programming model overly ugly.

Rationale, Discussion and Consequences

Build and test automation is an established best practice in current software development. The natural next step is to automate programming – at least those issues that are repetitive and governed by clearly defined rules. The code and configuration files that are necessary for the TECHNOLOGY MAPPING are a classic candidate. Generating these artifacts has several advantages. First of all, it's simply more efficient. Second, the requirement to "implement" the TECHNOLOGY MAPPING in the form of a generator helps refine the TECHNOLOGY MAPPING rules. Code quality will typically improve, since a code generator doesn't make any accidental errors – it may well be wrong, but then the generated code is typically *always* wrong, making errors easier to find. Finally, developers are relieved from having to implement tedious glue code over and over again, a boring, frustrating, and thus error prone task.

■ DSL-based Programming Model

Context

You have a PROGRAMMING MODEL defined.

Problem

Your PROGRAMMING MODEL is still too complicated, with a lot of domain-specific algorithms implemented over and over again. It is hard for your domain experts to use the PROGRAMMING MODEL in their everyday work. And the GLUE CODE GENERATION needs information about the program structure that is hard or impossible to derive from the code written as part of the PROGRAMMING MODEL.

Forces

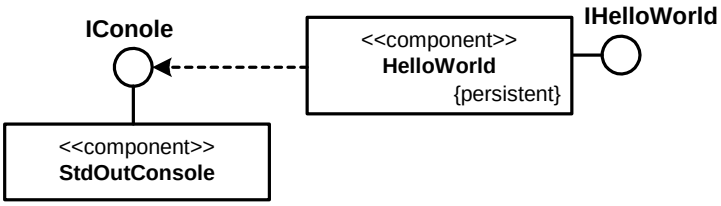
- The PROGRAMMING MODEL is still on the abstraction level of a programming language. Domain-specific language features cannot be realized.
- Parsing code in order to gain information on what kind of glue code to generate is tedious, and the code also does not have the necessary semantic richness.

Solution

Define Domain-Specific Languages that developers use to describe application structure and behavior in a brief and concise manner.

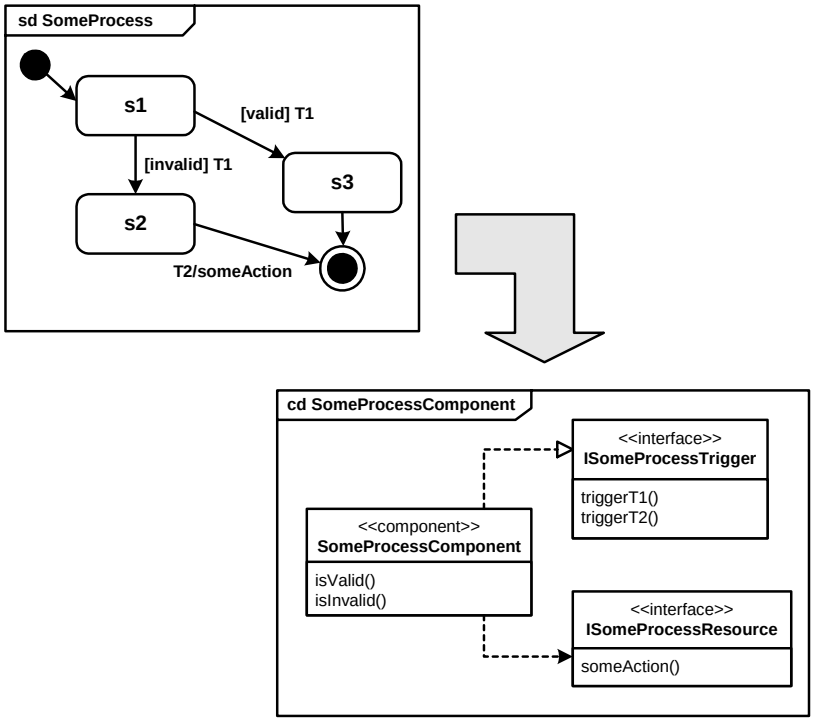
Generate the lower-level implementation code from these models. Generate a skeleton against which developers can code those aspects that cannot be completely generated from the models.

Example. There are at least two rather obvious places, where using a DSL makes a lot of sense. One place is components, interfaces and dependencies. Describing this aspect in a model has two benefits: First, the GLUE CODE GENERATION can use a more semantically rich model as its input, and the model allows for very powerful MODEL-BASED ARCHITECTURE VALIDATION (see below).



From these diagrams, we can generate a skeleton component class as well as all the necessary interfaces. Developers simply inherit from the generate skeleton and implement the operations defined by the provided interfaces.

A second place is the processes. Here, the necessary state machines can be "drawn" using UML state machines. This is much simpler than coding the State pattern manually. To integrate processes with the other components (e.g. those that use the processes) can easily be rendered by "black-boxing" the state machine with a component and using it in component diagrams. The component is derived from the state chart automatically.

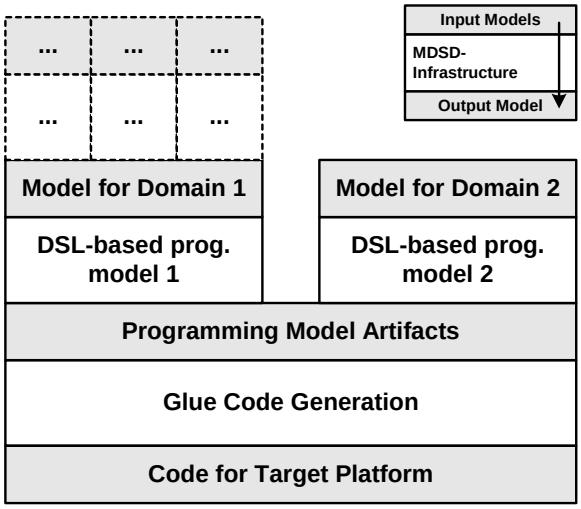


Verifying the consistency of these models and generating the necessary code is standard, and no particular problem with today's tools.

Rationale, Discussion and Consequences

This pattern marks the entrance into the model-driven software development arena. Defining DSLs for various aspects of a system and then generating the implementation code – fitting into the PROGRAMMING MODEL defined above – is a very powerful approach. On the other hand, defining useful DSLs, providing a suitable editor, and implementing an generator creates efficient code is a non-trivial task. So this step only makes sense if the generator is reused often, the "normal" PROGRAMMING MODEL is so intricate, that a DSL boosts productivity, or if you want to do complex MODEL-BASED ARCHITECTURE VALIDATION.

The deeper your understanding of the domain becomes, the more expressive your DSL can become (and the more powerful your generators have to be). In order to manage the complexity, you should build cascades of DSL/Generator pairs. The lowest layer is basically the GLUE CODE GENERATOR; higher layers provide more and more powerful DSL-BASED PROGRAMMING MODELS. The following illustration shows the approach.



■ Model-Based Architecture Verification

Context

You have all the things from above in place and you roll out your architecture to a larger number of developers.

Problem

You have to make sure that the PROGRAMMING MODEL is used in the intended way. Different people might have different qualifications. Using the programming model correctly is also crucial for the architecture to deliver it QoS promises.

Forces

- Checking a system for "architectural compliance" is critical!
- Using only manual reviews for that does not scale to large and potentially distributed teams.
- Since a lot technical complexity is taken away from developers (it is in the GENERATED GLUE CODE) these issues need not be checked.
- Checking the use of the PROGRAMMING MODEL on source level is complicated, mostly as a consequence of the intricate details of the programming language used.

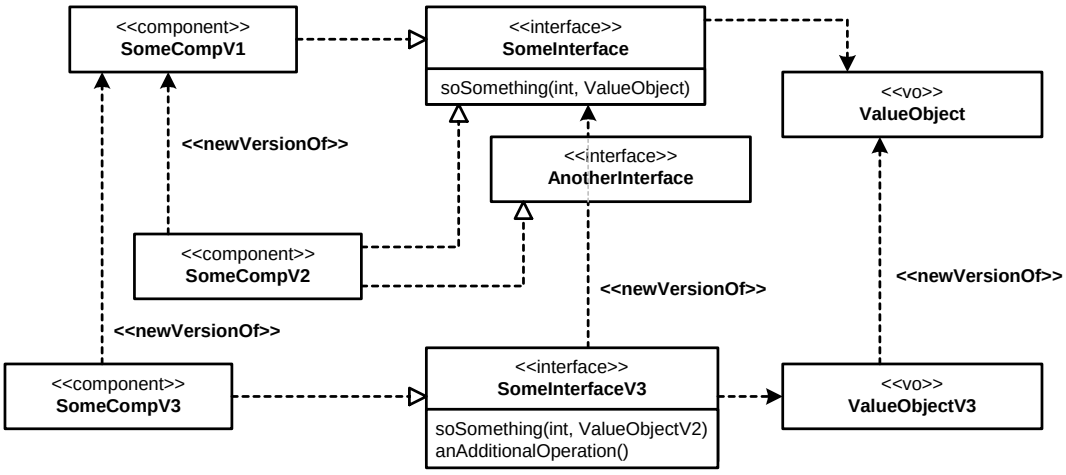
Solution

Make sure critical architectural things are either specified as part of the DSL-BASED PROGRAMMING MODEL, or the developers are restricted in what they can do be the generated skeleton, into which they add their

3GL code. Architectural verifications can then be done on model level, which is quite simple: it can be specified against the constraints defined in the ARCHITECTURE METAMODEL.

Example. Since this system will be built by a large number of developers, architectural constraint checking is essential. A number of basic model checks are done, for example, that for triggers in processes there is a component that calls the trigger. Other checks include dependency management. It is easy to detect circular dependencies among components. Also, components are assigned to layers (app, service, base) and dependencies are only allowed in certain directions. The IOC-programming, combined with the fact that the component signature is generated from the model prevents developers from creating dependencies to components that are not described in the model – and in the model, invalid dependencies can be detected easily.

Another really important aspect in our example system is evolution of interfaces. Take a look at the following diagram:



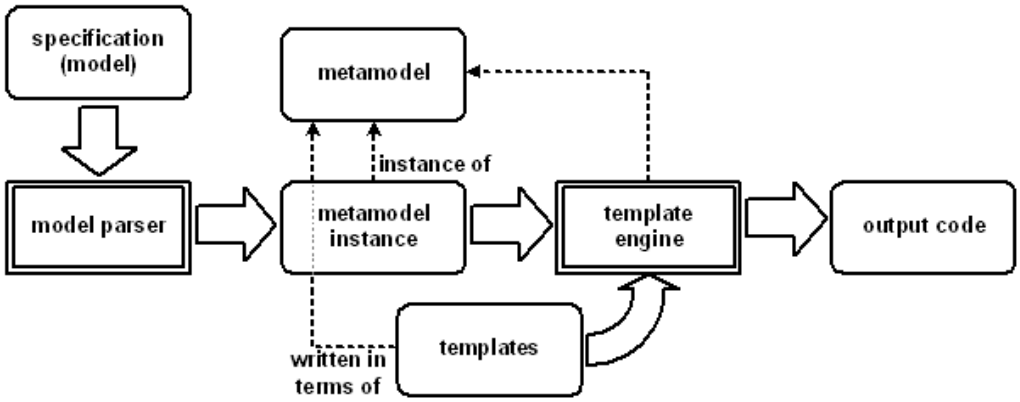
Note how this diagram makes new versions of things explicit! This is essential to check and enforce compatibility rules that make sure that a client that expects *SomeInterface* can also deal with a new version, i.e. *SomeInterfaceV3*. The generated implementation of *SomeInterfaceV3* inherits from *SomeInterface*. This makes the interface types compatible. The generator also makes sure that a new version of an interface has the same operations (plus maybe additional ones). An interface can refine an operation by using a new version of a value object – the new version of which inherits from the old one. So, in one sentence: The verification phase of the generator

enforces rules that *make sure* that new versions of components and interfaces are always compatible with previous versions.

Rationale, Discussion and Consequences

This where you want to get in the end! In larger projects, you have to be able to verify the properties of your system (from an architectural point of view) via automated checks. Some of them can be done on code level (using metrics, etc.). However, if you have the system's critical aspects described in models, you have much more powerful verification and validation tools at hand.

As pointed out earlier, it is essential that you can use the ARCHITECTURE METAMODEL to verify models/specifications. Good tools for model-driven software development (such as the openArchitectureWare generator [OAW]) can read (architecture) metamodels and use them to validate input models. This way, a metamodel is not "just documentation", it is an artifact used by development tools. The following illustration shows how this tool works.



Summary

The approach to software architecture described in this papers is a tried and trusted one. However, it is often not used … Why? People think it is too complicated to use. And it's not "standard". Well, to some extend this is true. Defining your own PROGRAMMING MODEL certainly means, that not all developers will learn each and every J2EE detail. While this might be considered a problem by some developers (for their CVs), it is certainly a good thing wrt. productivity.

References

ALMA European Southern Observatory, *The Atacama Large Millimeter Array*, http://www.eso.org/projects/alma/

AS Autosar Consortium, *Automotive Open Systems Architecture*, http://www.autosar.org

EE03 Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley 2003

JB00 Jan Bosch, *Design and Use of Software Architectures*, Addison-Wesley, 2000

MV04 Markus Völter, *Patterns for Model-Driven Software Development*, EuroPLoP 2004 proceedings and http://www.voelter.de/data/pub/MDDPatterns.pdf

MV02 Markus Völter, *A Generative Component Infrastructure for Embedded Systems*, http://www.voelter.de/data/pub/SmallComponents.pdf

OAW openarchitectureware.org, *The openArchitectureWare Generator Framework*, http://www.openarchitectureware.org

POSA1 Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, Michael Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Wiley, 1996

POSA2 Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann , *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*, Wiley, 2000

POSA3 Michael Kircher, Prashant Jain, *Pattern-Oriented Software Architecture, Volume 3, Patterns for Resource Management*, Wiley 2004

RV05 Michael Rudorfer, Markus Völter, Domain-specific IDEs in embedded automotive software, EclipseCon 2005 and http://www.voelter.de/data/presentations/EclipseCon.pdf

SV05 Tom Stahl, Markus Völter, *Modellgetriebene Softwareentwicklung*, dPunkt, 2005

VKZ04 Markus Voelter, Michael Kircher, Uwe Zdun, *Remoting Patterns : Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*, Wiley 2004

WSV02 Markus Völter, Alexander Schmid, Eberhard Wolff, *Server Component Patterns : Component Infrastructures Illustrated with EJB,* Wiley, 2002