

# Software Architecture

A pattern language for building  
sustainable software architectures

Version 1.5, April 13, 2006

(c) 2006 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.

### Why is software architecture important

Software architecture has been, is, and will be an important discipline in software development. At some point, you have to come up with a consistent metaphor for how your system is structured and behaves. There are different opinions on *when* you have to define your architecture (at the beginning of a project, or on the fly), *who* should do that (one or more architects, the developer team as a whole), *how detailed* it should be defined (just a rough spec or detailed prescriptions) and *in what way* to specify it (powerpoints, word docs, code snippets, metamodels).

Also, in some circles, the word architecture itself has accumulated so much negative connotation, that it is not used at all: people use terms such as “strategic design” instead.

However, I think it is agreed that a non-trivial system has to stick to certain consistency rules internally in order to communicate its internal structure to (new) developers, keep the system maintainable and flexible, and be able to deliver on guaranteed quality-of-service properties such as scalability or timeliness.

### Current state of the practice

When talking about software architecture, it is important to distinguish the architectural concepts from the technology decisions made to implement them, although of course, the two aspects are related.

Today, software architecture is too much *technology driven*. You hear statements such as “we use a web-service architecture”. Obviously, this statement is not a description of *an architecture*, since it describes only one aspect of the overall

system (communication). Also, 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”. There are several reasons why a technology-driven architecture is problematic. For example, you are forced to use the (often invasive) programming model that the technology prescribed. Also, it is hard to migrate to another technology if you learn over time that the current decision is not the best anymore, for example, as a consequence of changing non-functional requirements. Also, testability often suffers, because when testing you have to carry around all the “stuff” introduced by the technology. An early commitment to a specific technology usually results in blindness for the concepts. Discussions in the team often center around “implementation details” as opposed to concepts.

To look at the Web Services example again: It is much more important to discuss things such as interaction patterns, quality of service or the programming model of the communication. In the case of “EJB architecture”, you might want to discuss about separation of concerns, container/component-based systems, component lifecycle and dependency management.

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 oriented architecture” is a classic. The community has not agreed yet what an SOA *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! Note that I am not arguing against standards. Rather, I think they should be used only iff they really fit to the particular problem at hand, if they are mature, and if using the standard gives you real benefits such interoperability or wide-spread tool/framework/platform availability.

Finally, there’s *politics*.

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

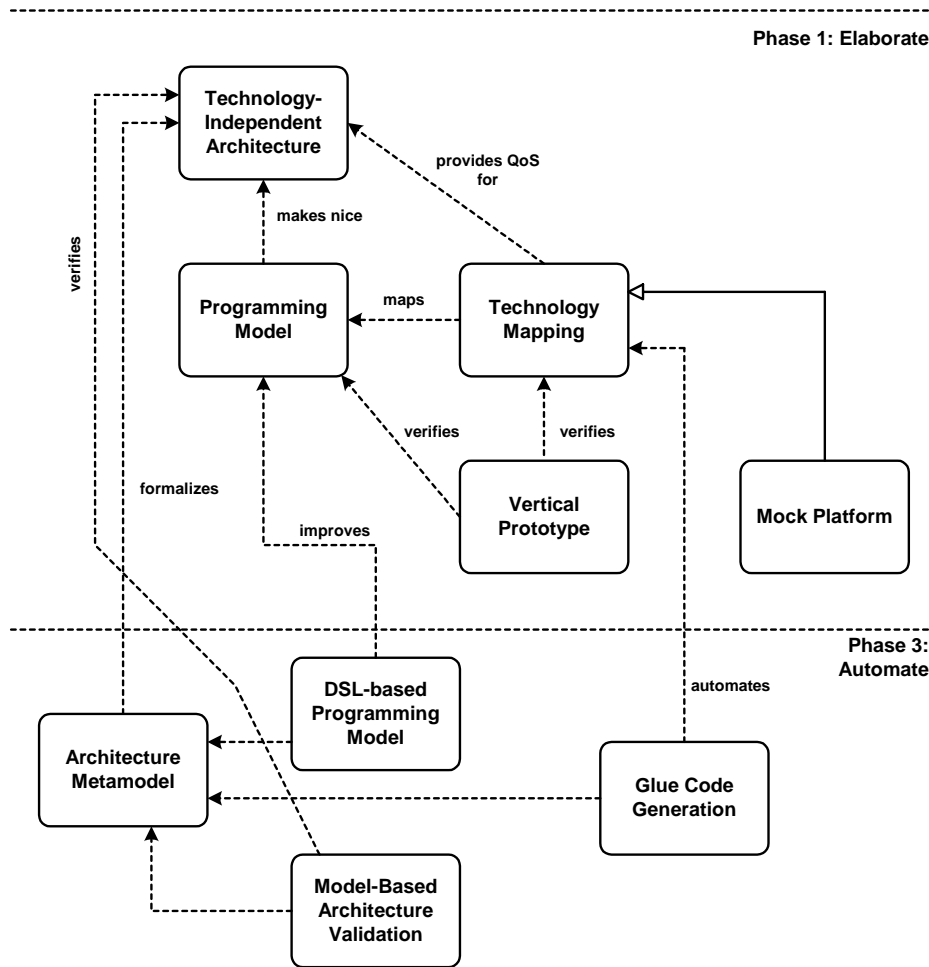
The approach to software architecture described in this paper is structured into three phases.

**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 various customers I cannot disclose at this time, business systems were built that resemble the example in this paper conceptually. Here it was *not* clear from the beginning, that models and code generation would be useful, and only in some of the cases MDS was used eventually. However, since the Phase 1 patterns had been used successfully, the potential for MDS 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++.

---

## Applicability of the patterns

The patterns described in this paper have two basic benefits: they help you come up with a good, technology-independent architecture (phase one patterns), and they help to manage and implement the architecture (phase three patterns). I do think that the phase one patterns are really applicable in any project. Of course, they become more useful with growing project size or infrastructure complexity. Actually, if there is no real infrastructure on which the system should run then the patterns are probably irrelevant. However, since that is true for only very simple systems, I really think that the phase one patterns are very widely applicable.

For the phase three patterns things are somewhat different. They address the enforcement of architectural guidelines in teams as well as the automation of repetitive development aspects. Also, implementing some of the patterns does require some (though not too much) pre-investment. Consequently, these patterns lend themselves to being used in scenarios where 5+ people develop for 3+ months. And of course, their usefulness increases with increasing team size, project duration and requirements/infrastructure complexity.

---

## The Patterns and Process

It is important to clarify the nature of the relationship between these patterns and software development processes. These patterns are not intended to be a substitute for a process – agile or not. The patterns just provide building blocks for designing an architecture. Specifically this paper says nothing about

- whether the architecture will be designed by the whole team, or only a few "special" developers who like to call themselves architects.
- whether you design the architecture iteratively (highly recommended) or up-front (usually doesn't work).

---

## 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. In really big projects, 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<sup>1</sup>.

### Forces

- You want to make sure that the architectural concepts can be communicated to stakeholders and developers
- You want to make sure that in the team everyone is united in a common vision of the style and architecture of the system (see *Unity of Purpose* pattern in [CH04])
- You want to separate the various concerns addressed by the architecture efficiently, specifically, you want to separate technical and functional concerns.
- 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.

---

<sup>1</sup> ...referencing a revolutionary idea for tax declarations in Germany ☺

## 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.

---

**Example.** The core conceptual building blocks of our example application are components, interfaces, data types business processes and communication channels. Here is a simple glossary that defines these terms:

|                             |  |
|-----------------------------|--|
| <b>Data type</b>            | Represents a certain chunk of data. Data types can either be simple types (string, int, boolean and the like) or <i>Complex Types</i> .  |
| <b>Complex Type</b>         | A complex data type is basically a like a struct in that it has named and typed attributes. There are two kinds of complex data types: <i>Entities</i> and <i>Data Transfer Objects</i>  |
| <b>Entity</b>               | persistent entities that have a well-defined identity (and can thus be searched), and that can have relationships to other entities.   |
| <b>Data Transfer Object</b> | and data transfer objects; they have no identity and are not persistent.   |
| <b>Interface</b>            | A contract that contains a number of operations; operations are defined as usual.  |
| <b>Component</b>            | A component is a well-defined piece of behaviour. It does not implement technical concerns. Each component can provide a number of <i>Interfaces</i> . It can also use a number of interfaces (provided by other components). Components are stateless.  |
| <b>Process</b>              | We also explicitly support business processes. These are considered to be expressible as state machines. Components can trigger the state machine by supplying events to them. In turn, other components can be triggered by the state machine, resulting in the invocation of certain operations defined by one of their provided interfaces. |

A number of constraints can also be identified. For example, communication among ordinary components is synchronous and local, no remoting is supported on this level. Communication to/from processes is asynchronous. Remote communication is supported here.

We expect components to be deployed/hosted in some kind of container that takes care of the technical concerns. Selection of the appropriate container/platform is guided by the required quality of service and will be addressed in the TECHNOLOGY MAPPING.

---

## 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.
- Implementation of functional requirements should be as efficient as possible.
- 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. Typical ingredients of a programming model are:

- How do I access resources, who manages the lifecycle of certain artifacts
- In which chunks, and where, do I implement my application logic
- How do I interact with the platform and environment
- Which aspects of the underlying programming languages are disallowed
- Important conventions and idioms, including certain important naming conventions
- A How-To Guide that walks developers through the process of building an application.

Low-level details such as a style guide are typically not included.

---

**Example.** The programming model uses a simple dependency injection approach as exemplified by the Spring Framework<sup>2</sup> to define component

---

<sup>2</sup> Note that the Spring programming model does not introduce *any* dependencies to the Spring framework itself. This is why it can be safely here and does not constitute a technology binding.

dependencies. 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. Component implementation classes are marked up with the *@component* annotation, whereas the the setters for the resources use the *@resource* tag. The setters and annotations together make dependencies explicitly visible as part of the class signature, and doesn't hide them in implementation code

```
public @component class AddressManager
    implements IAddressStore { // provides AddressStore

    private IPersonDAO personDAO;

    public @resource void setPersonDAO( IPersonDAO d ) {
        this.personDAO = d;          // setter for dao
                                    // component interface

    public void addOrUpdateContact( Person p ) {
        ...                          // from IAddressStore
    }

    public void addAddress( Person p, Address a ) {
        ...                          // from IAddressStore
    }

    public Address[] getAddresses( Person p ) {
        ...                          // from IAddressStore
    }
}
```

Processes described by state machines are implemented within special kinds of components, the so-called process components. The state machine itself is implemented using the State pattern [GHJ+94]. To make the state machine triggers accessible for external clients, process components provide an interface that contains a void operations for each of the state machine's triggers (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 (*paymentMade* and *paymentTimeout*) and also has one guard that needs to be evaluated.

```
public @process class PaymentProcess
    implements IPaymentProcessTrigger {

    private ICustomerManager custMgt;

    public @resource void setCustomerManager(
        ICustomerManager mgr ) {
```

```

    this.custMgr = mgr;
}

public @trigger void paymentMade( int procID ) {
    PaymentProcessInstance i = loadProcess( procID );
    if ( amountCorrect() ) {
        // advance to another state...
    }
}

public @trigger void paymentTimeout( int procID ) {
    PaymentProcessInstance i = loadProcess( procID );
    ... send reminder using the custMgr ...
}
}

```

Since components are stateless, the process component shown above does not actually represent a specific instance of the state machine. Rather, it is an engine that can „advance“ any of the process instances. The actual process instance is loaded by the process component when a trigger is received. To identify the process instance on which we want to apply the trigger, the trigger operations 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

At some point, you have to run your software on with “real hardware and software”, you have to integrate it into a system architecture. Making this binding fixes certain aspects of the system, typically QoS concerns. So you want to make the decisions consciously.

### Forces

- You don't want to implement the expensive features that enable all the non-functional requirements yourself. You might not even have the necessary skills on the team.
- You want to keep the conceptual discussions, as well as the PROGRAMMING MODEL free from those technical issues.
- At some point, you have to marry your concepts and programming model with “real hardware and software”, integrate it into a system architecture.
- 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 subsequent 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.** The infrastructure for running the application itself will be kept as simple as possible. The Spring Framework will be used as long as no advanced load balancing, management or transaction policies are required. As discussed before, we will try to keep the technology mapping well separated from the application logic so we can easily move to a different technology platform should the need arise.

The following is the Spring configuration file for this simple example. It instantiates three beans and „wires“ their resources accordingly.

```
<beans>
  <bean id="proc" class="com.pany.PaymentProcess">
    <property name="customerManager">
      <ref bean="cm"/>
    </property>
  </bean>
  <bean id="hello"
    class="com.pany.CustomerManager">
    <property name="personDAO">
      <ref bean="personDAO"/>
    </property>
  </bean>
  <bean id="personDAO" class="com.pany.PersonDAO">
</beans>
```

Once a more sophisticated platform becomes necessary, we will implement Stateless Session EJBs to run the components inside a J2EE application server. The necessary code to wrap our components inside EJBs is easy to write: 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 data entities is implemented using Hibernate. The persistent, long running business processes are implemented along the same lines; for each process we implement an entity that contains all the data that represents the current state of the respective process: the id of the process instance, the identifier of its current state, as well as the values of the context attributes.

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 *>HelloWorld*, we generate a WSDL file, as well as the necessary endpoint implementation. Of course we don't implement all the technology ourselves – we use one of the many available web service frameworks.

---

## 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 determined 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!

Of course, sometimes you have to decide on your platform based on politics. If a company builds everything on Oracle and Websphere, you'll have a hard time arguing against. However, the process based on this and the two aforementioned patterns is still *useful*, because it will allow you to understand the consequences of *not* using the ideal platform. You might have to use a compromise, but at least you know it is one!

---

## ■ 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 Spring as the technology mapping, we use that same platform to run the application components locally for test purposes. Stubbing out parts is easy based on Springs XML configuration file. To make testing as easy and fast as possible, we use the Hypersonic in-memory database. Whenever we run a test, the schema is created anew - Hibernate can do this for us with one line of code. We will talk about the mock platform more further below.

---

### 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. These parts of the system require both kinds of interactions: 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.

- After implementing the vertical prototype, a load test was executed. This unearthed 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 for both changes the application code did not have to be changed, only the adapters that mapped the logical communication to web services had to be changed to be able to also 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. For example, you might notice that the TECHNOLOGY MAPPING or you might find that the TECHNOLOGY-INDEPENDENT ARCHITECTURE is not well-enough defined to allow for a succinct mapping to a certain platform. In that case, you have to go back and evolve the technology-independent architecture.

Of course, it can be annoying to redo the TECHNOLOGY INDEPENDENT ARCHITECTURE after you have already done the TECHNOLOGY MAPPING. However, sometimes this is unavoidable. To reduce the associated pain, make sure that you break down your architecture in a number of sub-parts and run phase one for each of them separately. Necessary iterations in phase two can often be limited to one part.

Once you're confident in Phase one artefacts, 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. The next step would be to automate the programming model and make it more domain-related. The patterns in this section require some additional effort – the specific numbers depend on your experience with the required tooling.

Here are some guidelines of when the automation will be useful. The patterns will elaborate on these points.

- The more repetition you have to cope with as part of your PROGRAMMING MODEL and TECHNOLOGY MAPPING, the more benefit you'll get from automating these aspects e.g. using GLUE CODE GENERATION.
- Formalizing your TECHNOLOGY-INDEPENDENT ARCHITECTURE will help to define it even more sharply, most likely improving your architecture as you go.
- The larger your team is, the more you have to make sure that people use the PROGRAMMING MODEL correctly. Using DSL-BASED PROGRAMMING MODELS and GLUE CODE GENERATION can help to further tighten the PROGRAMMING MODEL and avoid bugs.
- Finally, if you have complex deployment scenarios, product lines or variant management issues, DSL-BASED PROGRAMMING MODELS and GLUE CODE GENERATION based on an ARCHITECTURAL METAMODEL can help. Specifically, MODEL-BASED ARCHITECTURE VALIDATION can become really essential.

So if all of these things do not apply to your scenario, you might want to stop reading here. Otherwise, things will become all the more interesting below.

---

## ■ 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-INDEPENDENT ARCHITECTURE.

## Forces

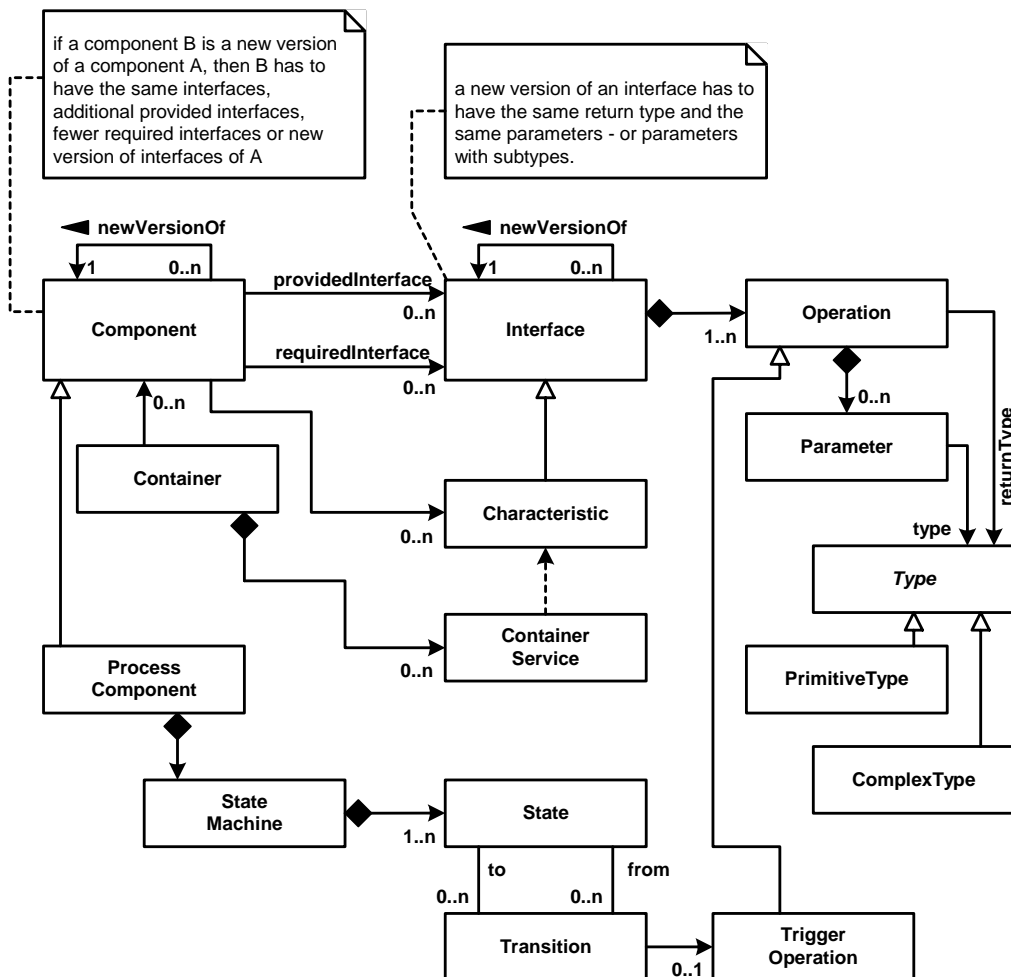
- Automation cannot happen if you can't formalize translation rules.
- An architecture definition based on prose text is not formal enough.
- You want to be able to check models for architectural consistency.

## 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<sup>3</sup>.




---

<sup>3</sup> 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.

It is interesting to see that even the component container which hosts the components is modular with respect to its services. Characteristics (special kinds of interfaces) are used to mark up components with respect to the services they require. A container service (such as persistence or lifecycle) will take care of components that provide the respective characteristics interface.

---

## 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 thought-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

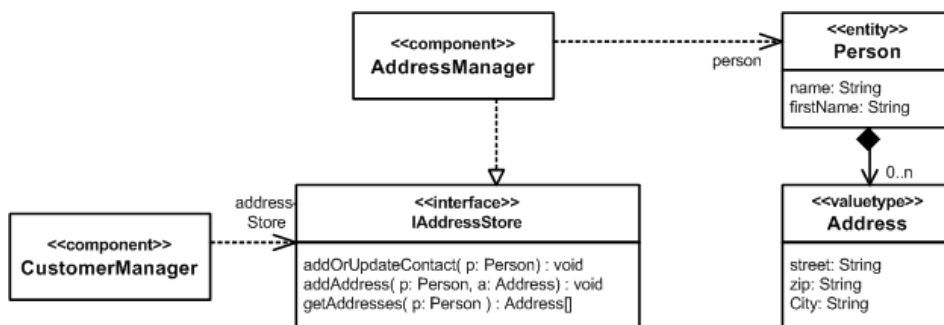
- The PROGRAMMING MODEL is still on the abstraction level of a programming language. Domain-specific language features cannot be realized.
- Parsing code in order to gain information on what kind of glue code to generate is tedious, and the code also does not have the necessary semantic richness.

## Solution

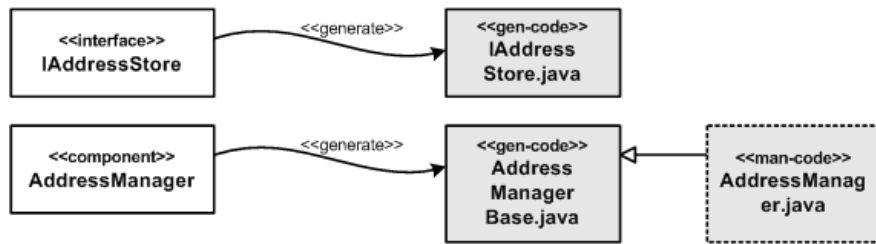
Define Domain-Specific Languages that developers use to describe application structure and behavior in a brief and concise manner. Generate the lower-level implementation code from these models. Generate a skeleton against which developers can code those aspects that cannot be completely generated from 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 various things. From the components and their interfaces we can generate a skeleton component implementation class as well as all the necessary Java interfaces. Developers simply inherit from the generated skeleton and implement the operations defined by the provided interfaces. The following illustration shows this.

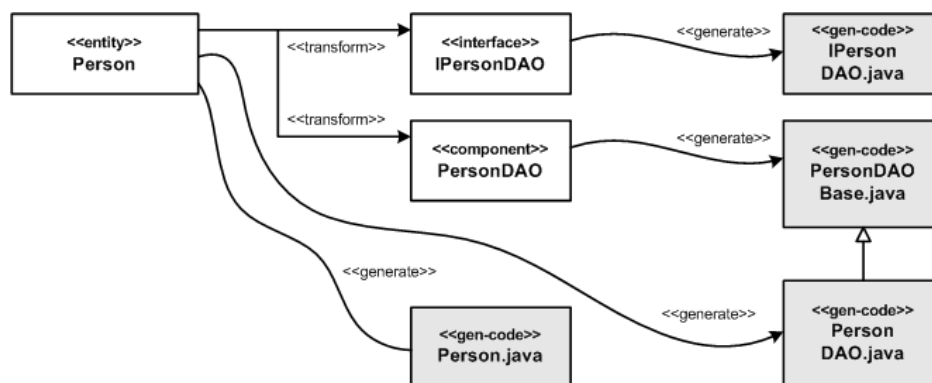


Handling entities is a bit more interesting. First of all, we generate the respective Java bean (*SomeEntity.java*) including their Hibernate mapping file (*SomeEntity.xbm.xml*). In addition to that, we want to have DAO component for each of the entities. The DAO component provides operations to create, read, update and delete instances of the respective entity. Instead of directly generating the code for these components from the entity, we use a model-to-model transformation to create model elements that resemble the DAO component as well as its interface. So, after that transformation, the model contains an additional interface and an additional component. These are treated just as any other interface/component, i.e. the existing code generation template generate the Java interface as well as the implementation skeleton class from them. We don't have to write new templates!

What is still missing, however, is the implementation for the DAO components. In general, implementation code is written manually by developers into an implementation class that extends the generated implementation skeleton class. In case of DAOs, however, we can also generate the implementation for their operations - these simply create, read, update and delete instances of the respective entity, a couple of lines of (Hibernate) code.

So now we create an additional template that generates the implementation for the DAO components following the same rule as the developers, when they create their implementation class: the implementation class extends the generated implementation skeleton.

The following illustration shows the process that handles entities.



We will now create our own models of how the system is composed and how it is deployed. This allows us to generate many more useful artefacts. Let us start with the composition model. In this model, we define various named configurations. Each of these configurations contains a number of component instances and their wiring. The test configuration is special in that it does not define its own instances but rather combines the two other configurations into a single one for testing. It is also interesting to note that we can create instances of the DAO components that we created from the entities (see last paragraph). Since we didn't just create code for these components but instead created real model elements by using a model-to-model transformation we can now „grab“ this component and define instances of it. We couldn't have done that if we'd directly generated code from the entities

```
<configurations>

  <configuration name="addressStuff">
    <instance name="am" type="AddressManager">
      <wire name="personDAO" target="personDAO"/>
    </instance>
    <instance name="personDAO" type="PersonDAO"/>
  </configuration>

  <configuration name="customerStuff">
    <instance name="cm" type="CustomerManager">
      <wire name="addressStore"
        target=":addressStuff:am"/>
    </instance>
  </configuration>

  <configuration name="test"
    includes="addressStuff, customerStuff"/>

</configurations>
```

Our third model describes the system(s) onto which we deploy the configurations defined in the composition model.

```
<systems>

  <system name="production">
    <node name="server" type="spring"
      configuration="addressStuff"/>
    <node name="client" type="eclipse"
      configuration="customerStuff"/>
  </system>

  <system name="test">
    <node name="test" type="spring"
      configuration="test"/>
  </system>

</systems>
```

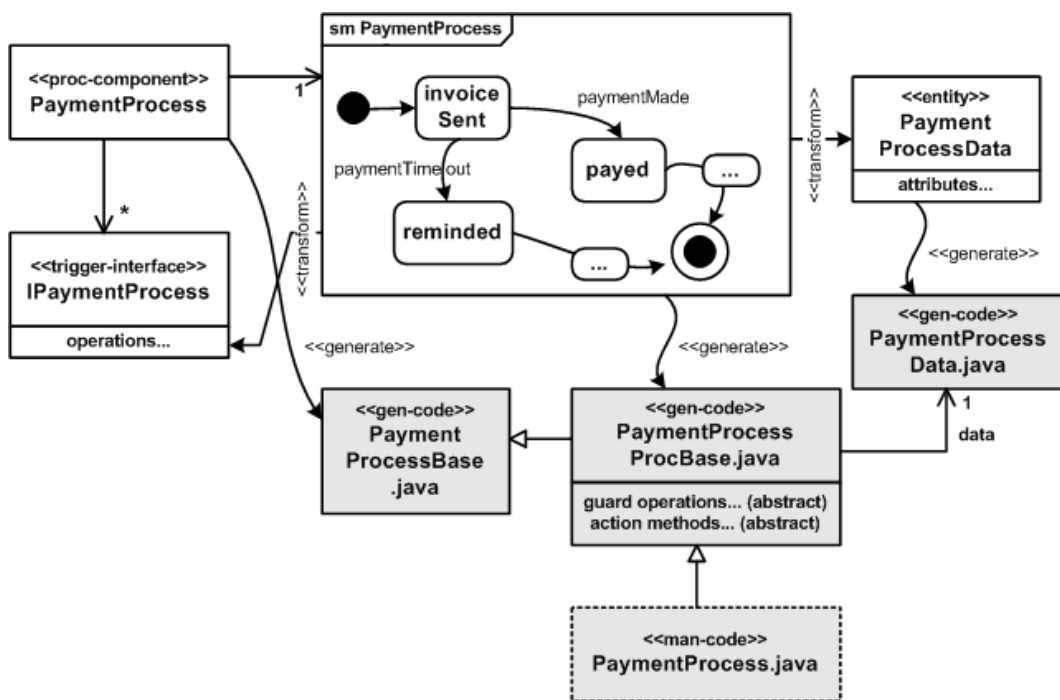
Here we define a system called production that consists of two nodes; one plays the role of the server, the other one plays the role of the client. The server hosts the addressStuff configuration, the client hosts the customerStuff. Note that we also define the types of the respective nodes (spring for the server, eclipse for the client). From these two models, we can generate

- a Spring configuration file for the server
- the plugins needed for the client
- the build files that create the necessary deployment artefacts
- the remote communication infrastructure (CORBA, web services)
- build files that assemble the necessary deployment artefacts

If we actually generate the test system, then we get only a single Spring node onto which all component instances are deployed for unit testing. No remotng infrastructure will be generated

To complete the picture let us look at how we work with process components and their state machines.

In the model we will create a process component called *AProcess*. This component provides an interface *IAProcess*. We do not model any operations in that interface - it's empty. We also associate the process component's state machine (*smAProcess*) with the component. This state chart contains states, transitions, triggers, actions and guards just like any other state chart.



The following process kicks in when the generator is run:

- from the triggers in the state chart, we add the necessary trigger operations into the empty interface using a model-to-model transformation.
- using another model-to-model transformation, we create an entity that contains all the data necessary to describe the a process instance described by the respective state chart.
- now the mechanics that handle entities „grab“ that entity and create the DAO component, the its interface, the Java bean as well as the Hibernate mapping file. This is the identical process that has been defined in the section on handling entities. No specific transformation or template has to be written.
- the interface for the process component is handled just as any other interface - a Java interface is generated from it.
- the component is andled like any other component: an implementation skeleton class is generated.
- now we need to provide an additional template that is specific to process components. Just as the implementation for the DAO components can be generated automatically, we can now generate an implementation of the process component (*AProcessProcBase.java*) that executes the process' state machine (we use a big *switch* statement for this). As the rules prescribe, this class extends the generated implementation skeleton class. However, since we have to add business logic to the action methods as well as to the guard operation, this generated class *is still not complete*: developers have to extend it once more and overwrite the guard and action methods. Again, the recipe framework is used to guide developers.

Again we have made heavy use of model-to-model transformations. While, at first, this approach seems quite intricate and complex, it proves to be very useful because very little transformation code has to be developed. And in case we change the persistence mechanism from Hibernate to something else, the persistent process implementations are changed automatically, too.

Note that this cascading of several levels of model-to-model transformations on top of each other allowed us to reach a DSL for modelling business processes that was quite appealing for the business analysts that would define the processes. The mechanics of integrating these people was as follows:

- The analysts initially created a state chart that described the business process intuitively, just like the analysts were used to using state charts.
- Then the analysts were joined by a developer. Together they marked up the intuitive state chart into one that was formal enough to serve as an input for code completion. This involved the application of stereotypes, formulating guards in a structured manner as well as checking the chart for completeness. The code generator's verification facilities were used to check the state chart for completeness with regards to code generation.
- Further changes on the state chart were made mostly by the analysts by working directly on the formalized version. In some cases a developer was involved, too.

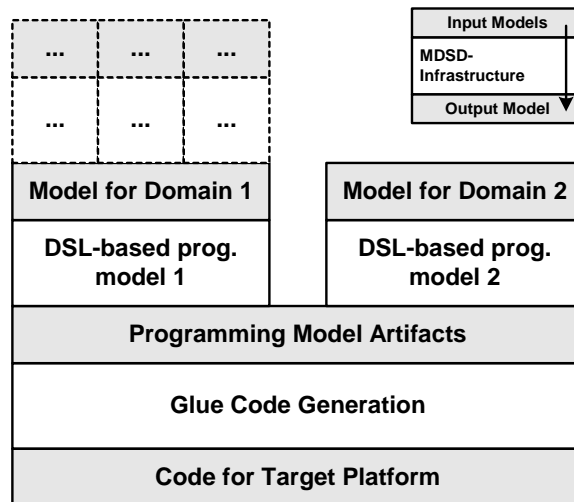
This approach resulted in a much improved integration of analysts and developers - making the process of analysis a big step more „tangible“ than before.

---

## 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 a generator that 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 its 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 of 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

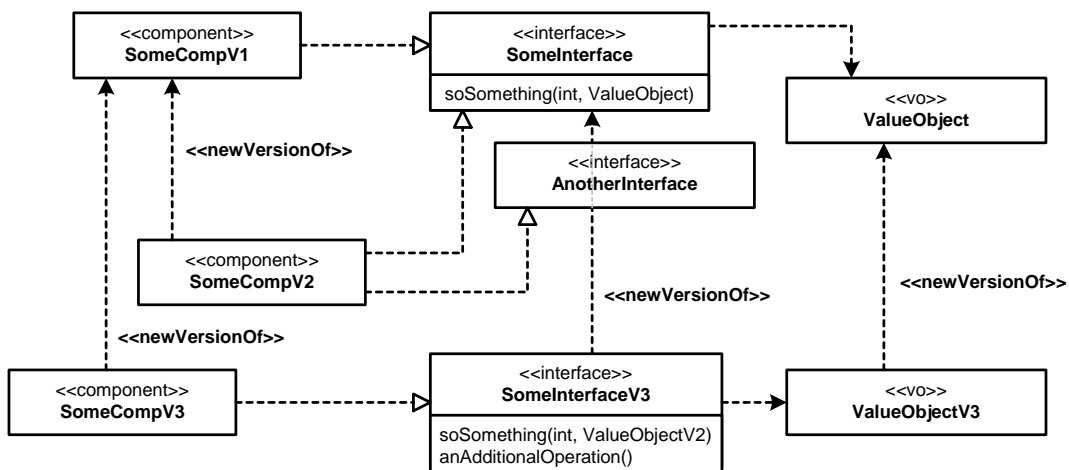
Make sure critical architectural things are either specified as part of the DSL-BASED PROGRAMMING MODEL, or the developers are restricted in what they can do to the generated skeleton, into which they add their 3GL code. Architectural

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:



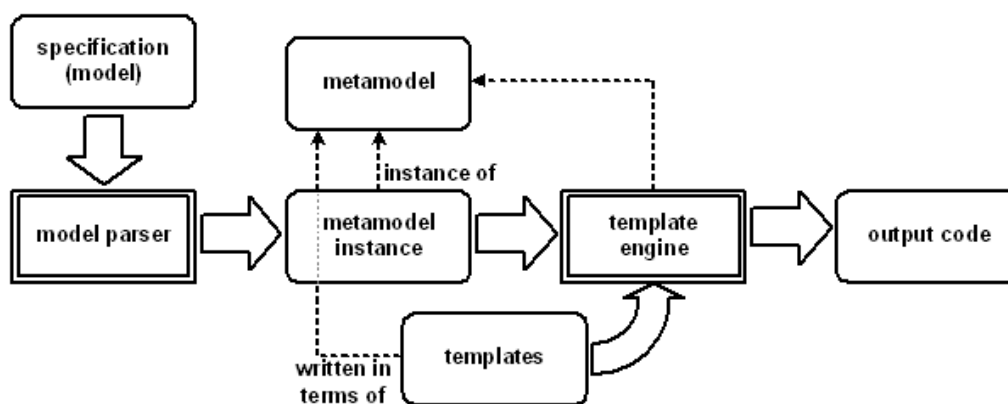
Note how this diagram makes new versions of things explicit! This is essential to check and enforce compatibility rules that make sure that a client that expects *SomeInterface* can also deal with a new version, i.e. *SomeInterfaceV3*. The generated implementation of *SomeInterfaceV3* inherits from *SomeInterface*. 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.

---

## Acknowledgement

First and foremost I'd like to thank my EuroPLoP 2006 shepherd Neil Harrison for his very useful and insightful comments. They improved the paper quite a bit.

I'd also like to thank Gregor Hohpe, Michael Kircher, Frank Westphal and Eberhard Wolff for their interesting (and sometimes very critical) Feedback.

---

## 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>
- CH04 Jim Coplien, Neil Harrison, *Organizational Patterns*, Prentice Hall, 2004
- 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