Software Architecture

A pattern language for building sustainable software architectures

Examples - only

Version 1.1, Mar 10, 2005

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

Phase 1 – Elaborate!

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

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.

■ Programming Model

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.

■ Technology Mapping

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.

■ Mock Platform

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.

■ Vertical Prototype

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.

Phase 2 – Iterate!

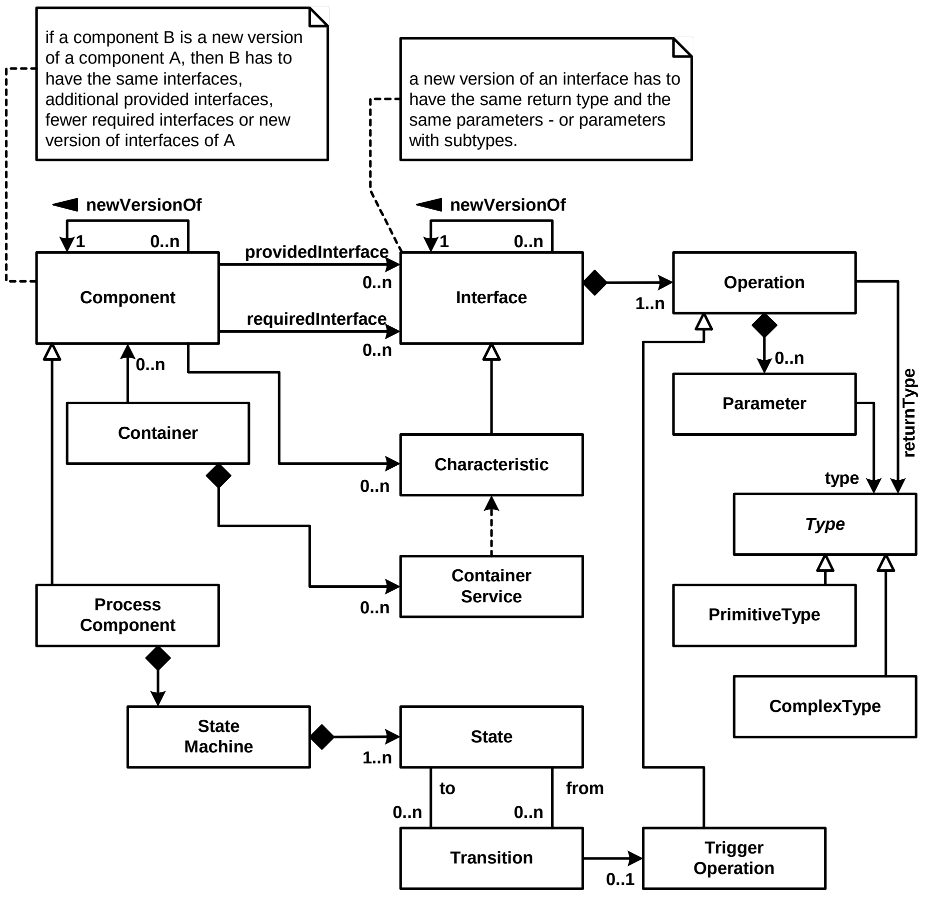
There was the idea to use Spring not just as the MOCK PLATFORM, but also for the production environment. However, as a consequence of new requirements, this has become infeasible. Spring does not support two important features: Dynamic installation/de-installation of components, and isolations of components 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!

■ Architecture Metamodel

The metamodel for the system is shown below, it is rendered as a MOF model[1]. 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.


if a component B is a new version of a component A, then B has to have the same interfaces, additional provided interfaces, fewer required interfaces or new version of interfaces of A
a new version of an interface has to have the same return type and the same parameters - or parameters with subtypes.
newVersionOf
1
0..n
newVersionOf
1
0..n
Component
providedInterface
0..n
requiredInterface
0..n
Interface
1..n
Operation
0..n
Parameter
returnType
type
Type
PrimitiveType
ComplexType
0..n
Container
Characteristic
0..n
Container Service
0..n
Process Component
State Machine
1..n
State
to
from
0..n
0..n
Transition
0..1
Trigger Operation

[1] 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.

■ Glue Code Generation

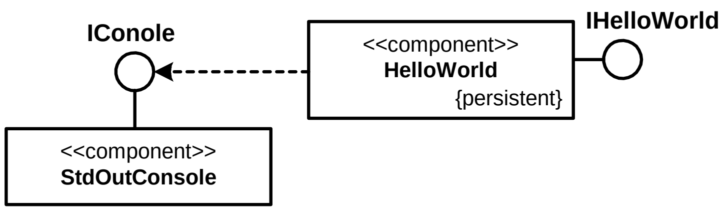
Our scenario has several useful locations for glue code generation.

- 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.

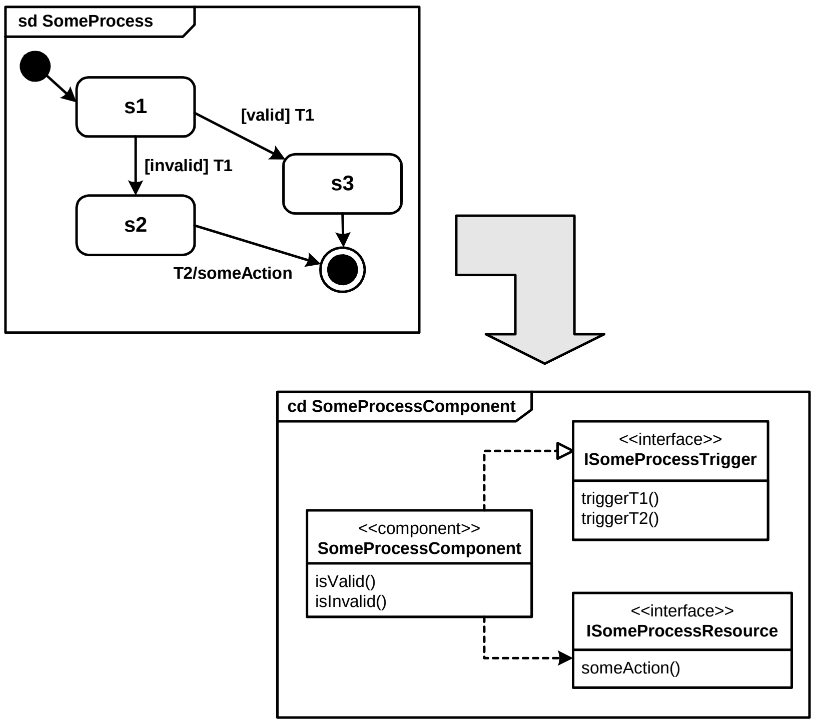
■ DSL-based Programming Model

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 generated 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.

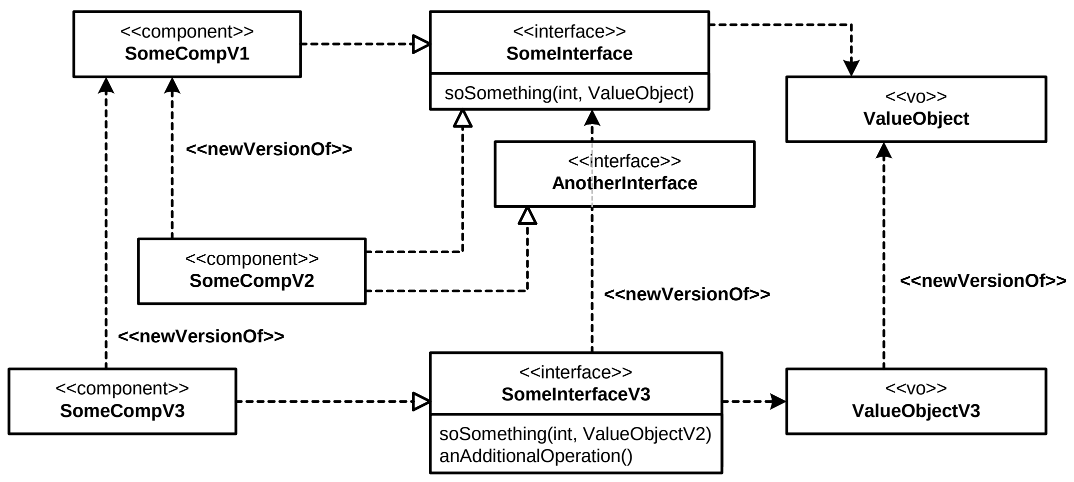


Verifying the consistency of these models and generating the necessary code is standard, and no particular problem with today’s tools.

■ Model-Based Architecture Verification

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.