SOA and MDSD –

Why SOA is only really useful in combination with MDSD

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

About me

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

- Independent Consultant
- Based out of Heidenheim, Germany
- Focus on
  - Model-Driven Software Development
  - Software Architecture
  - Middleware

CONTENTS

SOA and MDSD

CONTENTS

Overview

- SOA has become *the* hype topic.
- Several of my customers are currently in the process of establishing a SOA – however, all do **something different** ☺
- Thus, SOA is not a sharpy defined term
- In this session I want to convey a number of **best practices** when building SOAs with a special focus on MDSD.

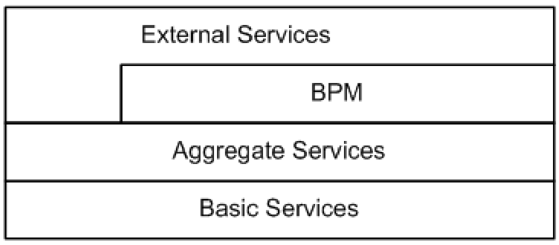
CONTENTS

SOA Perspectives

- **SOA == CBD**, i.e. SOA is components done right: building blocks with a well-defined responsibility that provide and use formally defined services.
- **SOA == EAI:** focuses on asynchronous, loosely coupled (message based) communication. Data structures have to be routed, filtered and mapped.
- **SOA == BPM**, i.e. emphasizes the potentials for the business department, the term „business driven“ is often used here. The definition and management of business processes is important.
- All views agree that SOA is important for **large and complex enterprise systems** – or groups of such systems.

SOA Perspectives II

- These views **fit together** quite well:
  - Components form the base layer
  - On top of them you can orchestrate business processes
  - Using legacy adapter, filter and mapping components you can use it for EAI
- It is also useful to distinguish **public services** (those used by external clients) as a separate layer on top.



SOA Perspectives III

- Two more points that are considered advantages of SOAs:
  - **Separation of technical and business concerns:** Service interfaces only expose business data/operations, and service implementation does not need to care about technical concerns ((Security, Persistence, Failover, ...)
  - **Managability:** You can manage components & services, version them, install them, and you know, which components are used (by whom), which business rocess are running
- However, while both of these are important, these things have been around for a while and **are nothing new** (specifically in component infrastructures)
- **In short:** if you talk about SOAs, you need to talk about components.

CONTENTS

Requirements to SOAs

- To maintain complex systems (such as SOAs) over a long period of time, you need to make sure that:
  - When implementing business logic, you don't want to care about runtime platform artifacts or transfer formats. Implementation must be **technology-independent** (not necessary language independent!) to keep business logic implementation efficient
  - Application logic needs to remain **testable**, i.e. testable without complex infrastructure. Otherwise developers will not adopt regular unit testing.
  - You need some level of technology independence, since technology changes faster than your architecture. You want to be able to **adapt to new technologies**.
- There is more...

Requirements to SOAs II

- The involved parties must be able to **communicate effectively** about the SOA – thus, you need a common language and formal definition of concepts.
- You need to stay **agile wrt. changing** service definitions, data structures and business logic. It is unacceptable that it takes weeks to add a new attribute to a data structure.
- You need to consider certain **organizational realities**: for example, business departments (and their IT projects) might not be able to willing to stick to centrally defined rules, tools or processes.

CONTENTS

Abstraction is Key

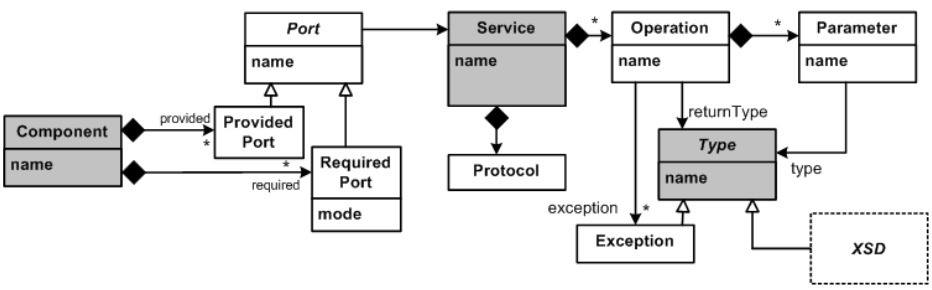
- **Formal models** are a good way to attack many of these issues. Defining such models (and the associated metamodels and DSLs) is quite essential for building an SOA.
- To use MDA terms: you need to build an **Architecture-PIM**. In this PIM you will find the central building blocks of your SOA, such as
  - Services,
  - Componnets,
  - QoS Constraints
  - Deployment information.
- This PIM is **independent of the concrete deployment platform** (web services, JBI, SCA). Automatic mappings (transformations, generation) produce the runtime infrastructure.

Metamodels

- To be able to „draw“ the above Architecture PIM you need a suitable modelling language – it, in turn builds on a **metamodel representing your architecture**.
- A metamodel defines the **language elements** („words“) that you can use to build models, as well as how they can be combined (how „sentences“ can be built)
- In our case, the metamodel thus contains all the relevant „kinds of things“ you might need to describe your SOA (services, components, networks, etc.)
- To be able to describe the lowest layer of an SOA (the component layer) we need **three viewpoints**:
  - Type models
  - Composition models
  - And Deployment models
- We will take a look at the metamodels for each of these in turn.

The Type Model

- The type model defines
  - **service interfaces**
  - **Components**
  - **data types**.

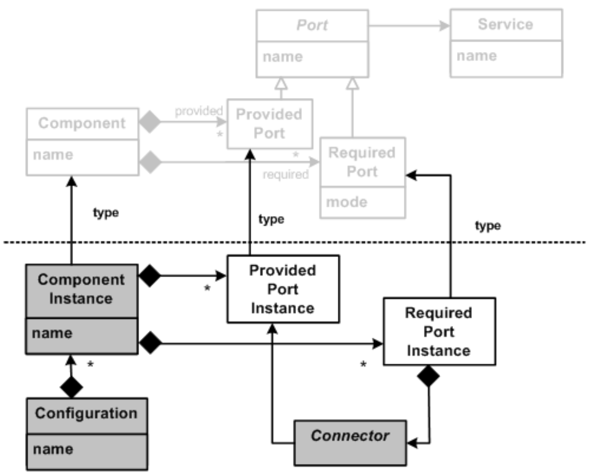


The Type Model II

- Core building block is the **Service**. **Services** are „interaction contracts“.
- A service has a number of **operations**.
- These use **data types** in their signatures. Types are often defined using (simplified) XML Schema.
- Often, Services also define **protocols** of how to use the operations (often a protocol state machine)
- **Components** provide services through **Provided Ports** and connect to services consumed by the component using **Required Ports. Components** realize interaction contracts (defined by services)

The Composition Model

- The composition model declares **component instances** and shows how they are **logically connected**.

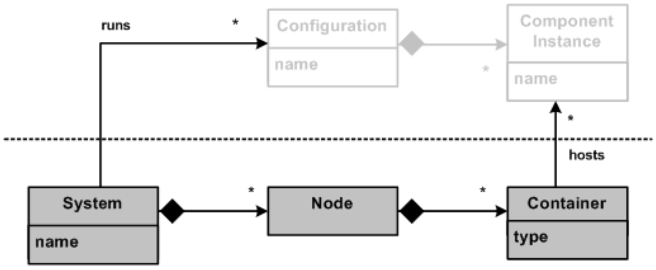


The Composition Model II

- **Connectors** connect a provided port with (one or more) Required Ports.
- **Additional constraints** have to be considered, such as: you can only connect ports that provide/require the same (or a compatible) service.
- Although it looks like **static (modelling time) wiring**, this approach works also in more dynamic environments: Instead of specifying the target port directly, **you specify a number of search criteria** for the to-be-connected port that are then evaluated at runtime using some kind of naming or trading service.

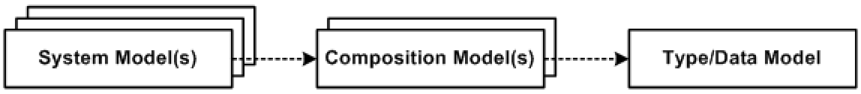
The Deployment Model

- The deployment model associates component instances with
  - **Hardware**
  - **Application server / processes**
  - **Communication middleware**



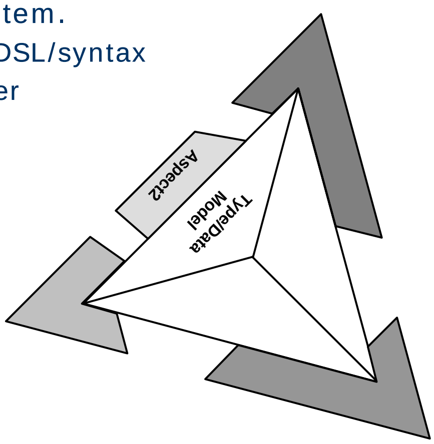
Model Dependencies

- **Dependencies** between the models (and metamodels, respectively) are important.
- You have to make sure that
  - you can deploy the same compositions **on different systems** (e.g. for testing)
  - You want to use the same components in **many compositions**

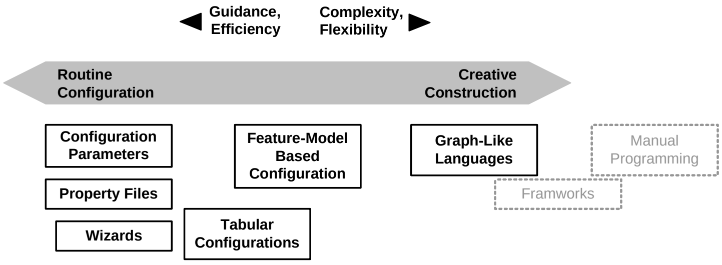


Aspect Models

- Often, the described three viewpoints are not enough, **additional aspects** need to be described.
- These go into **separate aspect models**, each describing a well-defined aspect of the system.
  - Each of them uses a suitable DSL/syntax
  - The generator acts as a weaver
- Typical **Examples** are
  - Persistence
  - Security
  - Forms, Layout, Pageflow
  - Timing, QoS in General
  - Packaging and Deployment
  - Diagnostics and Monitoring



Rountine Configuration vs. Creative Contruction



- This slide (adopted from K. Czarnecki) is **important for the selection of DSLs** in the context of MDSD **in general**:
  - The more you can move your DSL „form" to the configuration side, the simpler it typically gets.
  - We will see why this is especially important for behavior modelling.

C O N T E N T S

How to program with these things II

- You start by **defining the component model**; you define components in a model.
- Here we use a textual model for this.

```
serviceinterface IDatabase {
  readData(…);
  writeData(…);
}

serviceinterface IScripting {
  executeScript( String script );
}

component Copier {
  provides script: IScripting;
  requires srcDB: IDatabase;
  requires targetDB: IDatabase;
}
```


SOA and MDSD

How to program with these things II

- You can now generate an **implementation skeleton** that helps you implementing stuff.
- For example, here we generate a base class:

```
public abstract class CopierImplementationBase
       implements IScripting_script {

   public void ctx_setSrcDB( IDatabase db ) {
      sourceBD = db;
   }

   public void ctx_setTargetDB( IDatabase db ) {
      targetDB = db;
   }

   public abstract void
     script_execScript( String script );
     // from the interface IScripting
     // provided by the script port
}
```

How to program with these things III

- … from which we can inherit to actually implement our component:

```
public class CopierImplementation
    extends CopierImplementationBase {

   public void script_execScript( String script ) {
     // interpret the script.... assume it
     // contains some commands that require copying
     // data from sourceDB to targetDB
     data = sourceDB.readData(…);
     targetDB.writeData(data);
     // here you can see how the "port proxies"
     // sourceDB and targetDB are used.
   }

}
```


How to program with these things IV

- We can also use asynchronous communication. Here is the declaration in the component.

```
component CustomerRater {
  requires poll schufa: ISchufaService;
}
```

- The implementation could look as follows:

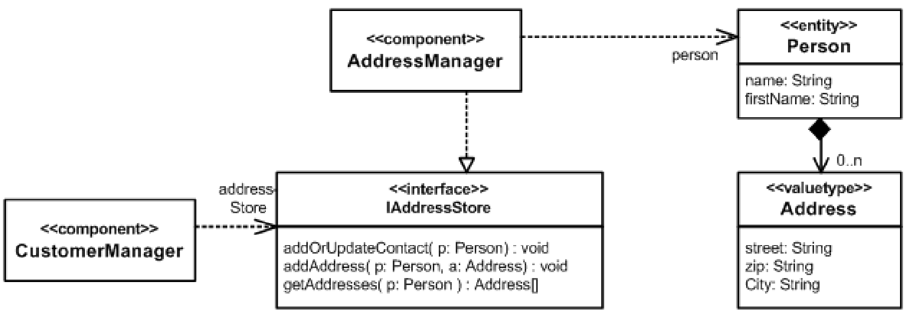
```
public class CustomerRaterImplementation {

   public void someMethode() {
      GetSchufaRatingPO poll = schufa.getSchufaRating(kundenID);
      // now we can do all kinds of things ....
      if ( poll.hasResult() ) return handleResult(poll.getResult());
      // do some more stuff, now we wait, blocking, until result comes in,
      // then we handle the result
      return handleResult(poll.getResultBlocking());
   }
   private boolean handleResult(SchufaReport r )  {
      // do something with it. ....
      return .... true if Schufa is good, otherwise false....
   }
}
```

Where does this get us to?

- So, we can now write **component implementations**
  - **Without** a technology dependency
  - **Without** deployment information
  - **Without** knowing with whom we actually interact
  - **Without** knowing on which platform we will run.
- We can now describe and implement component based software.
- We can add **additional models** (e.g. based on XML) that describe composition and deployment and generate all the necessary
  - Adapters
  - Glue code
  - Build scripts
  - Deployment scripts

Another Example – Type Model

- Here we use UML to define type-level artefacts



Another Example – Composition

- ... using XML

```
<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>
```


Another Example – Deployment

- ... using XML again

```
<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>
```

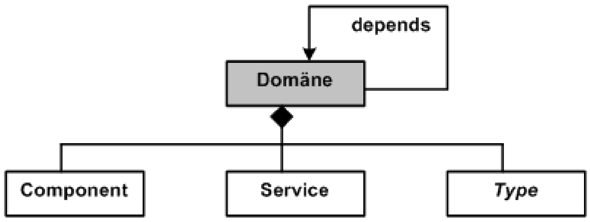
CONTENTS

Data Type Ownership

- To stay agile, an important question is: **who owns data types?**
- If you try to agree on a central **business object model** in a large enterprise, you typically will never reach an agreement – and if you do, there are the following consequences:
  - You will have a **hard time changing** the data structures if necessary because everybody else wants them to remain unchanged.
  - Also, the data structures will be **large, bloated and complex** because they have to fulfil everybody‘s needs.
- Working with such global data structures is thus **tedious and not very agile.** The BOM approach obviously does not work.

Data Type Ownership II

- An extreme solution of that problem is to define data structures **strictly local to a service**.
  - Only the service provides and users can use the data.
  - No sharing of data structures is possible.
  - Data structures must only be agreed among the service stakeholders.
- However, this will result in similar data structure remodeled again and again, for each service using it.



Data Type Ownership III

- Basically the visibility of a data structure is **restricted to the domain in which it is defined**.
- Services and components in the same domain can use the data structures.
- If you're in domain B, you can only use data structures defined in domain A if you **declare a dependency** on domain A and explicitly **import the data structure**.
- Consequently, dependencies on data structures are **explicitly modelled** and can be cautiously managed.

CONTENTS

Typing

- To further simplify working with data, make sure the **data structures are interpreted** by the components.
- These allows you to more easily **migrate and evolve that data structures over time** without have to redeploy the whole infrastructure
  - as opposed to changing the IDL definition of a CORBA struct. You need to recompile, redeploy, ..
- In an interpreted scenario you can
  - Ignore unknown attributes
  - Automatically add defaults
  - Use different (versions of) the defining XML schema to verify the data structure in different components.
- Note that interpreting data **does not relieve you** from defining data structures and coordinating them with stakeholders, but it simplifies the technical aspects of dependencies and deployment.
- End users of a data structure should always verify it (e.g. using schemas, but the **intermediary infrastructure should not**!)

CONTENTS

Service Reuse

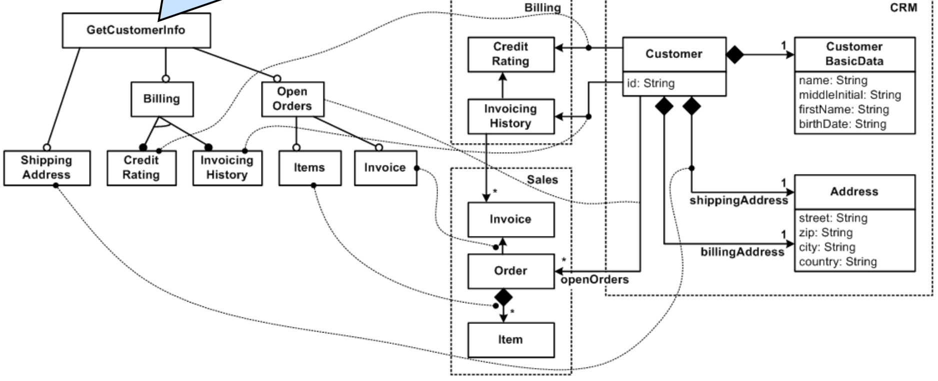
- Building a SOA often goes along with the idea of **standardizing and harmonizing** things.
- This is **very useful on the meta level** (i.e. standardizing on metamodels).
- But on the **concrete level** this is not that easy.
- Assume you want to agree on a service that returns customer information for a customer ID.
- You will first have the problems of harmonizing data structures – as just discussed.
- The second problem: various clients have **different QoS requirements:**
  - The call-center requires the data **very quickly**, but **only few data** items are required initially. The rest is lazily loaded if required.
  - Other clients require **more data** all the time (i.e. in one call) and are willing to wait a bit longer upon the first call.

Service Reuse II

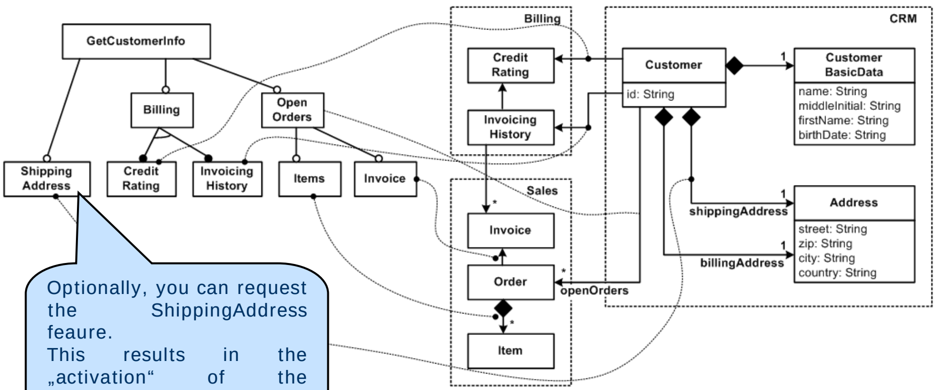
- This shows that the **harmonization** of services (interfaces, data structures, etc) **will not work** in practice.
- … if only because the services develop over time (versioning).
- To address this topic systematically, you should view the various services as a **product line** and manage variants and version explicitly.
- This can be achieved, for example, **using feature modelling.**
  - Specifically, you can systematically describe the variations in the data structures.
  - Using code generation you can then generate all kinds of dependent artifacts automatically (e.g. schemas).

Service Reuse III

GetCustomerInfo returns only the customer identity, basic data as well as the billing address



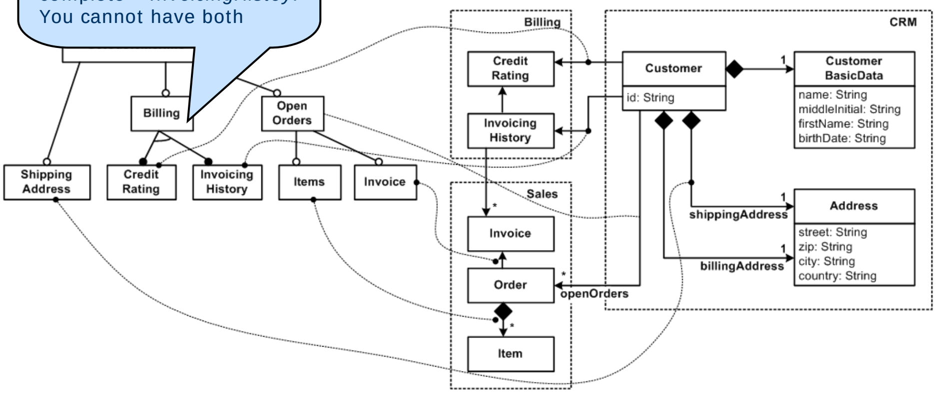
Service Reuse III



Optionally, you can request the ShippingAddress feaure.
This results in the „activation" of the association to the shipping address.

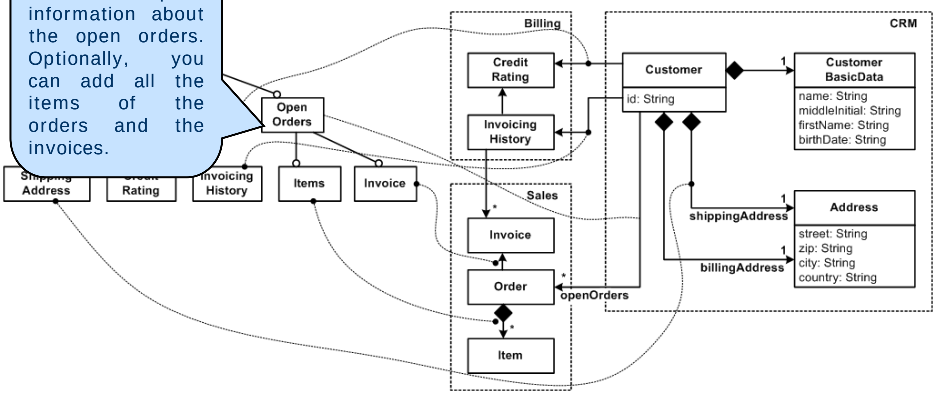
Service Reuse III

Another option is *Billing*. After selecting it, you have to decide if you want only the *CreditRating* or the complete *InvoicingHistoy*. You cannot have both



Service Reuse III

In addition, you can request information about the open orders. Optionally, you can add all the items of the orders and the invoices.



CONTENTS

- Overview
  - SOA Perspectives
  - Requirements to SOAs
- Models
- Programming
- Issues
  - Data Ownership
  - Typing
  - Service Reuse
  - **Process Issues**
  - Infrastructure vs. App Development
  - The spaghetti misunderstanding
- BPM
- Technologies
- Summary & Recommendations
- Appendix: An SOA Metamodel

Process Issues

- The goal of harmonization and centralization often also has other consequences:
  - **centralized service repository**
  - **Heavy-weight, centralized processes**
- Consequently,
  - developers have to be **online** all the time to access the repository when developing services,
  - They have to **coordinate** „with the whole company“ to devlop a (possibly simple) service
- This kills productivity and makes development unagile.

Process Issues II

- To avoid this, make sure that
  - service definition, implementation and test can be done **locally** withoug access to the central repository
  - The repository uses a **checkin/checkout methaphor** to support offline work (just like CVS)
  - **Coordination with central processes** becomes necessary only when the service becomes „public“
- You need to establish a status model:
  - **Developer-local:** you can do everything that is technically possible with the SOA, no access to enterprise service bus
  - **Repositoy global**: service has to conform to enterprise-wide standards
  - **Staging:** Only bugfixes possible
  - **Production:** no changes to service possible, need to define new versions, etc.

CONTENTS

Infrastructure vs. Application Development

- Often, the introduction of an SOA is **driven by a central** IT department
- Goal: **standardization and harmonization** of the IT infrastructure to simplify deployment and management
- Consequence: a **focus on middleware and technologies**
- However, application developers have different goals:
  - To get the to-be-developed application out of the door ASAP
  - Satisfy business requirements of their stakeholders
- **Conflict of interest:**
  - Application developers don't see benefits when using the SOA
  - Their life becomes often more complicated
  - Slow Adoption

Infrastructure vs. Application Development II

- To change this, make sure that
  - The SOA has **advantages for the application developers**
  - Make developing „correct“ applications **as simple as possible**
  - **Hide the SOA technology** (WS-*) as much as possible

- **Provide good tooling** for app developers from the start!

- In a model-driven world, this is quite easy:
  - Building an IDE (plugin) that **generates skeleton code** based on the models is not too much work
  - Glue code, that „connects“ application code with the SOA can be automatically generated
  - Support deployment and testing based on the models is also feasible

Infrastructure vs. Application Development III

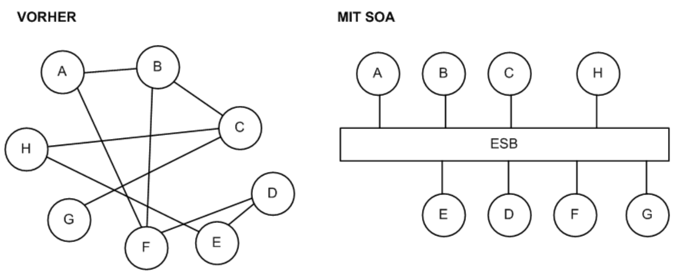
- This approach is especially useful for new services but can also be **used for legacy code**:

  - You can define the service interfaces using the above models; you can then generate the usual glue code. Accessing the legacy system is considered an **implementation detail**, i.e. it is done manually and not supported by the tooling.

  - The other approach is based on **automatically generating models** and implementation code for the components from the interfaces of the legacy systems (assuming they are somehow formally defined, e.g. source code).

CONTENTS

The Spaghetti Misunderstanding



- You probably know these kinds of drawings:
  An SOA **solves the point-to-point communication issue** and attached all components/services to an enterprise bus.
- However, that's not that easy in practice.
- And by the way: this same picture has been used by CORBA 10 years ago....

The Spaghetti Misunderstanding II

- One of the problems is that every ESB vendors has a **different idea of what an ESB is.**
- It is also not very useful to run everything over the same middleware, since
  - You might want to have different **organizational partitions**
  - Different systems need **different QoS**: External Services need to be interoperable. Internal Services have to be fast.
- So it is not important that everything uses the same technology, but rather that you can **potentially let everybody talk to everybody** (using a limited number of middlewares, but not just one!).
- Thus it is essential that **services are defined in a technology independent manner** – in models – so that you can generate mappings to the various middlewares used in the enterprise – based on the required QoS.
- This approach specifically allows the „Null-Middleware", i.e. running everything in one process to support testing.

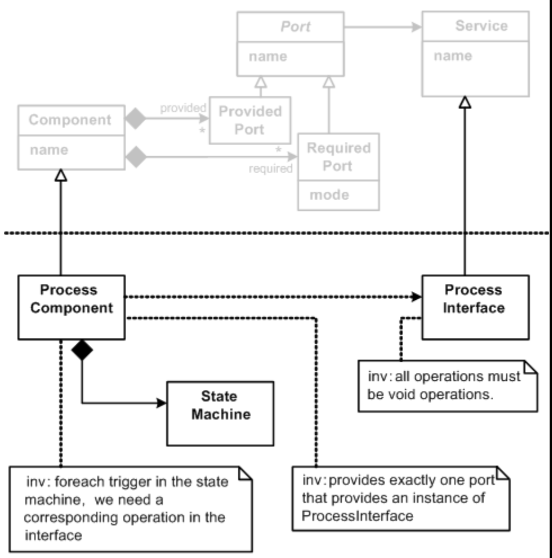
CONTENTS

BPM

- Business Processes typically **run over a long time** (hours, days, weeks, months).
- Executing a BP involves **access to various services** as defined the SOA.
- You can **describe services** in different forms.
  - State charts
  - Activity diagrams
  - BPMN
- To keep the definition of BP flexible, it is often useful to **interpret** BP definitions at runtime.
- There are **two ways of integrating** BPM into an SOA:
  - Process Components
  - External Engines

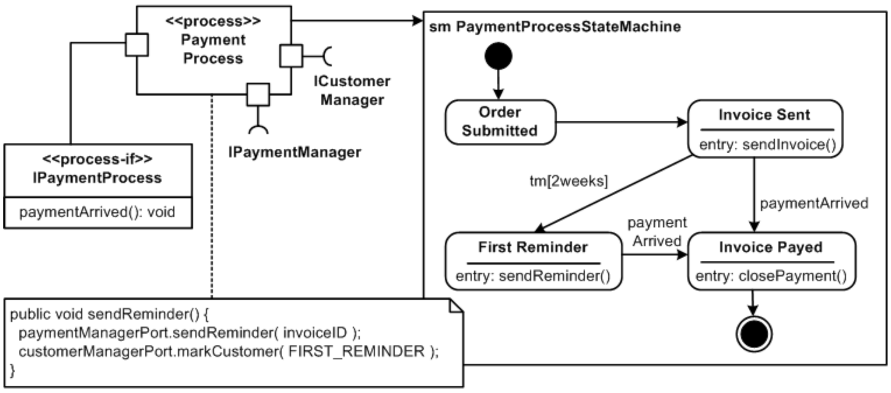
BPM: Process Components

- In this case we introduce a **special kind** (sub-metatype) **of component** the process component.
- These are **ordinary com-ponents**, i.e. they have required and provided ports, they can be wired and deployed.
- Their provided interface has an **operation per statemachine trigger**. These must be void ops, since they're typically called asynchronously



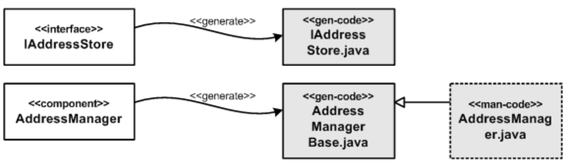
BPM: Process Components II

- The following is an example where the components, services and the processes are **modelled using UML.**
- Using other modelling notations will require different means of tool integration.



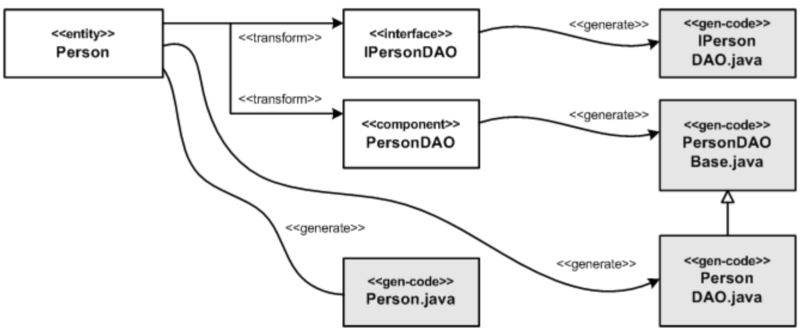
BPM: Process Components III

- Integration into the code generation infrastructure: **Cascading**
- **Basic** Transformation



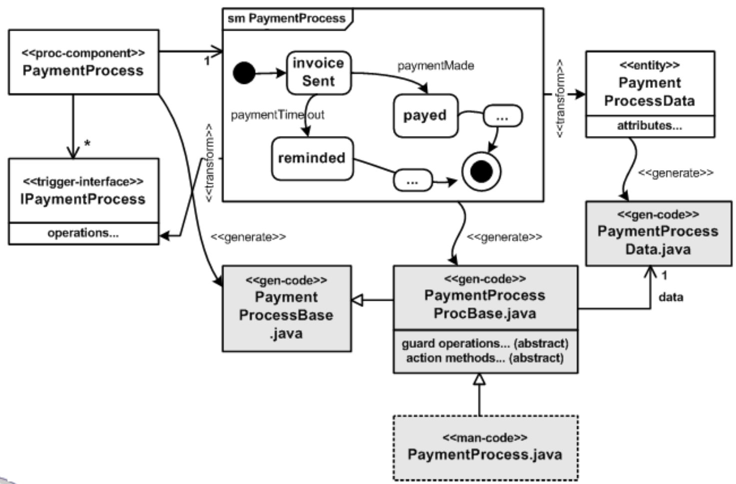
BPM: Process Components III

- Integration into the code generation infrastructure: **Cascading**
- **Persistence** Transformation



BPM: Process Components III

- Integration into the code generation infrastructure: **Cascading**
- **Processes** Transformation



BPM: External Engine

- You can also run the business processes by an **external BPM engine:**
  - Such tools often provide convenient process modelling IDEs (using BPMN, for example).
  - Adapters for accessing services using all kinds of technologies are available. Often, WSDL is the basis for service access.
- Here, too, we use models to access the services and define the processes.
- Reasons for using such a tool
  - It is (seems to be) **easier to use** by the business people (remember: „business driven“!)
  - **Services can be changed more often** and more easily while the services serve as the „solid base“ on which the services reside.

CONTENTS

Data Definition

- For simple data structures, **„nested structs“** are enough. These can be serialized using language serialization or YAML.
- More complex data should be represented using **XML**
  - Performance issues (might want to use binary XML)
  - Use **XML Schema** for data type definition
  - More comfortable access can be provided by **generated binding classes** (Attention: interpretation advantage is lost!)
  - Make sure you **restrict the power of XML schema**! Otherwise,
    - It will be hard to manage dependencies
    - It will not be interoperable (redefines, import/include, ...)
    - Don't go too far into details (don't use schema to define the semantics of an ISBN numer!)
    - You might want to use UML to define the schemas in a restricted way
- Make sure you actually **validate the data "at both ends",** but make sure the middleware does not care!

Communication Middleware

- You can use all kinds of middlewares for the communication aspect.
- The default choice is **Web Services** (WS-I Basic Profile 1.1, typically), but it is only required (and often only suitable) for external services
  - Note that **WSDL 1.1 contents are not enough** to build an SOA
  - In WSDL 2.0 things will get a little bit better (notion of „component“)
  - Potential performance issues because of XML/Web Services
- Other **infrastructures** are also ok,
  - RPC-style: CORBA, RMI, .NET Remoting, HTTP/Rest
  - Messaging-style: JMS, MQSeries, MSMQ, Tibco's products
- **Decision** should be based on
  - What's already there
  - Non-functional requirements

Component Runtime Platforms (Containers)

- You can use all **the well-known component runtime platforms** in an SOA. Examples include
  - J2EE (servlets, EJBs, MDBs, WS)
  - Spring
  - OSGi/Eclipse
  - WCF/Indigo
  - CCM
  - COM+
- Again, the choice should be based on **experience and non-functional requirements**.
- A new breed of **SOA component platforms** is emerging:
  - Java Business Integration (JBI)
  - Service Component Architecture (SCA)
- Both approaches **leverage existing component infrastructures** by integrating (at least some of) them.
- Note that both of these are still „bleeding edge“

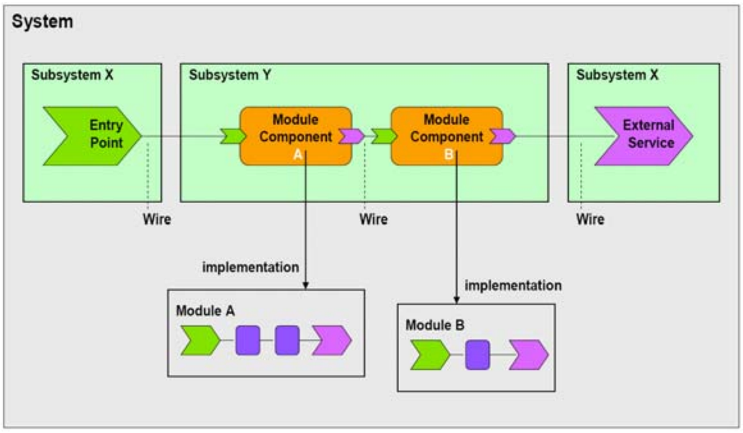
JBI

- JBI is a **Middleware Middleware,** specific to Java
  - it provides a unified view on various middlware systems,
  - maps communication to a standardized message format (the **Normalized Messages**)
  - And routes the messages among the various components in a JBI container (using the NM Router)
- JBI Components come in one of two flavors:
  - **Services Engines**: implementing business logic or transformations
  - **Binding Components**: those serve as communication adapters to communicate with „outside“ middleware
- Services are described in **WSDL** (more specifically: using the Abstract Message Definitions from WSDL 2)
- Distributed JBI implemenations will become available
- **Personal Opinion:** Sceptical, I am specifically missing the „system view“, i.e. the stuff described in the composition and deployment models.

SCA

- SCA is an **upcoming standard** developed by IBM, SAP, Oracle, BEA, Sybase, Iona, and Siebel. It is **language independent**.
- SCA encourages an SOA organization of business application code based on **components that implement business logic**, which offer their capabilities through **service-oriented interfaces** and which consume functions offered by other components through service-oriented interfaces, called **service references**. SCA divides up the steps in building a service-oriented application into two major parts:
  - The **implementation of components** which provide services and consume other services
  - The **assembly of sets of components** to build business applications, through the wiring of service references to services.
- SCA emphasizes the decoupling of **service implementation** and of service assembly from the **details of infrastructure capabilities** and from the details of the access methods used to invoke services.
- **Personal opinion:** looks interesting, since it considers the whole system (i.e. including composition & deployment)

SCA II



- System definition is based on XML
- Programming Model based on Annotations
- Two prototype **implementations**:
  Apache Tuscany and Eclipse SOA Tools Project

CONTENTS

Summary & Recommendations

- SOA is not about technology. And SOA is not about business.
  SOA is first & foremost about architecture.
- Keep all **important information in models** – separated by different concerns and viewpoints.
- Define **your own metamodel** so that it suits your needs. This is the strategic architecture repository that should be under your control.
- Consider **technology an implementation detail** – keep the models and the (business logic) development process free from it.
- **Do not build your own** communication **middleware** or execution platform. Select 3rd party tooling based on your non-functional requirements.
  - **don't start** with the technology!
  - Limit yourself to a small number of middleware technologies

Summary & Recommendations II

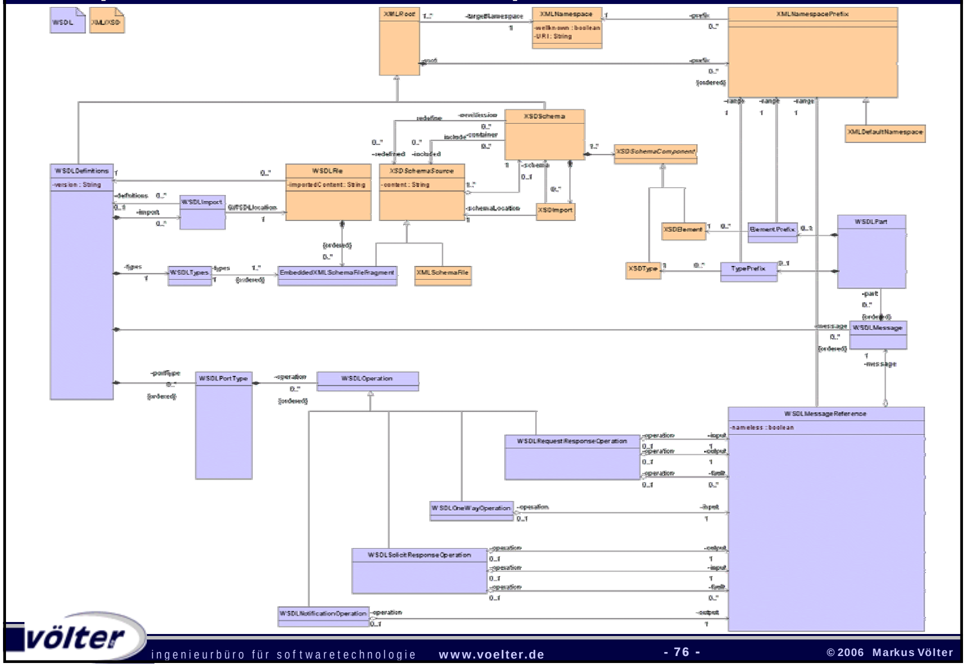
- Consider **application developers the primary user group** of your SOA – provide tooling to simplify their life.
- Make sure **service implementations remain testable** and consider (developer and integration) testing an important aspect of an SOA.
- Consider **deployment, operations and monitoring** another important stakeholder – support these folks by generating deployment/monitoring relevant artefacts for them.
- On the concrete level, **harmonize only where absolutely necessary** – do it with refactorings, don't slow down application development because of „global coordination"
- Integrate **BPM on top of** a well-defined component/service architecture, don't start with BPM!

Summary & Recommendations III

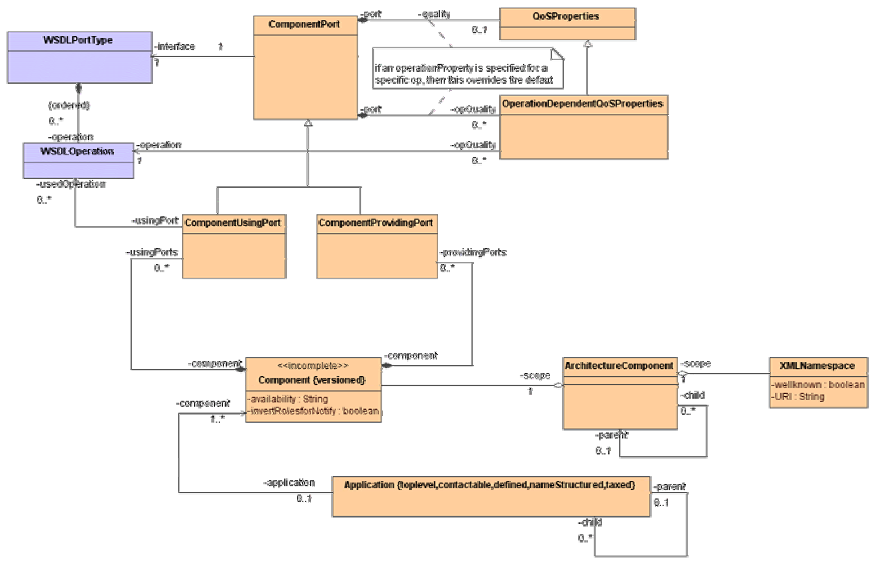
- And don't forget: There are **many more challenges** to establishing an enterprise-wide SOA that I consciously ignored, such as:
  - Required organizational changes
  - Different compensation schemes
  - A lesser focus on technology,

CONTENTS

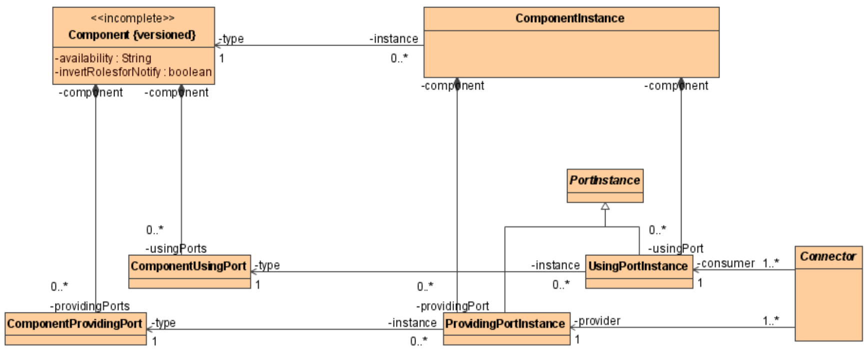
Enterprise SOA Metamodel Example: Services & Data



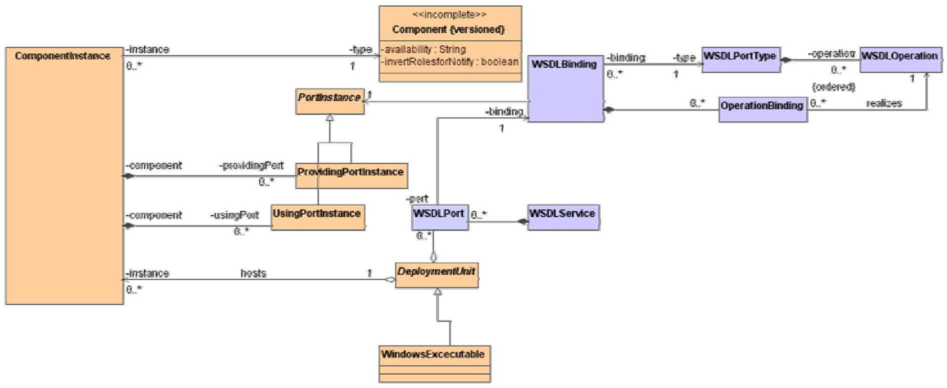
Enterprise SOA Metamodel Example II: Components



Enterprise SOA Metamodel Example III: Composition



Enterprise SOA Metamodel Example IV: Deployment



CONTENTS

THE END.

Some advertisement ☺

- For those, who speak (or rather, read) german:

  Völter, Stahl:

  Modellgetriebene Softwareentwicklung
  Technik, Engineering, Management

  dPunkt, 2005

  www.mdsd-buch.de

- A **very much updated** translation is under way:
  Model-Driven Software Development,
  Wiley, Q2 2006

  www.mdsd-book.org