Best Practices for Developing Components for Shared Services

Hari Rajagopal
Galileo International
Agenda

- Definitions
- Background
- Best practices
  - Service granularity
  - Service composition
  - Service presentation
  - Common Business vocabulary
  - Automation of process
What This Is…

- A set of principles that allows efficient construction of services that may be deployed within a Service Oriented Enterprise

- Applicable to web services, services under a SOA and enterprise services that are standalone
Some definitions

- Component
- Service
- Process
Component

- Reusable unit of code created in a silo’d environment

- Typically a POJO (plain old Java object) but well packaged and documented

- Deals with the basic business objects
What Is a Service?

- A unit of software that is
  - Reusable
  - Confirms to a well defined interface
  - Is complete in itself
Classic View of a Service

- Provided by a ‘provider’
- Exposed via a ‘directory’
- Looked up and located by a ‘consumer’
View of a Service

1. find service
2. invoke
3. receive
4. reply

Service Lookup
<<directory>>

<<consumer>>

<<provider>>
Service Categories

- Component services
  - Finer grained, atomic
  - Do not depend on others
  - Typically not directly accessed by external client

- Composite services
  - Built using component services
Process

- A **business process** is a collection of related structural activities that produce a specific outcome for a particular customer.

- A business process can be part of a larger, encompassing process and can include other business processes that have to be included in its method.
Hierarchy

- Components aggregate and wrap classes
- Services do the same for components
- Processes *compose* workflows using services

How do all these relate to SOA?
Service Oriented Architecture

Implications for service design
Service Oriented Architecture

- SOA is a pattern that allows construction of applications by composing them using loosely coupled independent services.
- It is a flexible and resilient pattern that accommodates change by allowing a system to be reconfigurable at the assembly level, rather than by using code.
SOA – Goals

- Move components out from silos within an enterprise to be more generically accessible
- Moves focus from integration efforts to process workflow definition
- Loosely couple the services deployed so that change is easier to accommodate
- Do this in a standards based environment
Phased Approach to Adoption

- Big bang is rarely (if ever) an option

- Pick a small application that has a good chance of success

- Reuse legacy assets rather than doing it from scratch
  - Use service adapters, façades to wrap them
Design Time Issues to Consider

- Do not overlook issues such as:
  - Billing
  - Security
  - Logging
  - Auditing
  - SLA management
Billing Issues

- Most services are for profit

- How do you track who owes how much?
  - Keep track of service call chain
  - Persist this to a repository
  - Ideally this is done in an async fashion
    - Push it to a billing queue in a fire and forget fashion
Security

- A presentation by itself, basically security can be done at the:
  - Transport layer
    - HTTP header
  - SOAP layer
    - SOAP header element
  - Payload (in-band)
    - Attribute in the message body
Service Level Agreements

- Sooner or later someone is going to dispute a bill or payment
- Proof is needed
  - Make sure there is enough data persisted so that audits can verify it
- Runtime correction – dynamic load balancing of services
SOA Best Practices

- Use *service Façade* pattern

- Use a common data model

- Use a multi-grained interface to maximize reuse
  - Create focused interfaces
Payload Format

- By and large, the web services seen so far have been an ad-hoc mix of SOAP encoding and document literal
- Tooling is driving the standards toward doc-literal payloads
- Doc literal format has the advantage of inherent scalability
  - Marshalling/unmarshalling step is skipped until needed
Service Design
What Makes a Service Reusable?

- Easy to use (well defined interface)
  - Few operations
  - Meaningful parameter (Doc literal)

- Easy to maintain and version
  - Documentation (tools)
  - Consistent schema definitions

- Correct level of granularity
What Makes a Service Reusable? (Continued)

- Design to an interface
  - Build a model
  - Generate schemas
  - Derive and compose request/response types based on the model
Well Defined Interface

- Typically a single service addresses a specific business need

- If you see the operations within a service grow – question yourself whether they are really necessary there

- Keep them stateless (easier to compose)
Well Defined Interface (Continued)

- Make the service accept multiple input formats and increase its reuse

  - Does not mean the interface is different, simply means the service can be deployed within different contexts (object 2 object in a JVM and XML-XML across JVMs)
What NOT to Expose as a Service?

- Infrastructure APIs

- Rather than build a transcoding layer – use the far more efficient transforms that an ESB provides

- Rather than roll your own SAML token service, wrap an existing API
Interface Design

- Interface design should be driven by business needs
- If a business need is not met the service has little if any reuse potential
- For example: In the travel industry, writing a web service to return a cryptic dataset from a mainframe GDS (Global Distribution System) has no relevance to a web site based travel aggregator
Interface Design (Continued)

- Instead a ‘HotelShopper’ service with an interface that takes the destination, duration of stay and price range and returns a list of properties with pictures attached is VERY useful.

- [http://www.orbitz.com](http://www.orbitz.com)
- [http://www.octopustravel.com](http://www.octopustravel.com)
What Does an Interface Mean?

- In the world of SOA an interface is usually rendered (I use this term deliberately) as a network accessible WSDL document.
- The WSDL (Web Services Description Language) document describes the interface and its allowed operations – down to the data type of the input(s) and output(s).
Granularity

- The level of detail that an interface presents determines the reuse
  - An interface that is very fine grained is more usable within the enterprise at the app level
  - An interface that is coarse grained has more business tier applicability for clients

- There are always exceptions – go multi grained in that case
Favor Composition over Inheritance

- Use inheritance ONLY where necessary

- Design services so that they are easily used in process flow engines (BPEL)
Using Our Travel Domain Case

- Usually a travel website deals with a GDS on the level of availabilities, bookings and cancellations
- Interfaces are coarse grained and resemble the one shown:
  ```xml
  <PlanTrip name="joe" startDt="20040812" endDt="20040815" DEP="LAX" DEST="DEN"/>
  [abbreviated for clarity 😊 ]
  ```
Using Our Travel Domain Case

(Continued)

- The example on the prior slide is actually a façade that uses a number of finer grained services.
- The output from each of these services is sliced and diced to return the end result that the client of the ‘trip planner’ service expects.
Sometimes ...

- However, on occasion – when a client requests that an additional person be added to his tour then a search by the PNR (Passenger Name Record) needs to be done to retrieve the trip.
- This is a very host specific call that represents a fine grained case.
Fine Grained Call
Multi-grained Approach?

- In order to maximize reuse potential, use the multi-grained approach.
  - Expose both fine grained as well as coarse grained interfaces
  - If you cannot – err on the side of excess, rather than increase the number of network hops
In General Though ...
Effect on Performance

- In the case of very coarse grained interfaces overall network time is reduced (chunks of data returned in a single call)
- However, often a large server side collation of data may prove to be expensive as well (leave XML processing to the client?)
- The answer: choose wisely depending on the circumstance (intranet roundtrip vs. web times)
Service Façade Pattern

- Coarse grained access to finer grained atomic services
- Less network time
- Handles the process flow

- This can be accomplished in 2 ways
Service Façade Implementations

- In Java code
  - Basically build your own process flow
  - Unless you create a generic rules-engine, it’s a task per application

- Use a standards based Business Process engine to choreograph the flow
Service Façade Implementations

(Continued)

- Take the example of a data access service (DAS)
  - Obviously SQL via SOAP is NOT the answer
  - Neither is allowing access to CRUD

- What then is the level of access?
  - Façade the lower level CRUD with process based services (clients don’t do DB access)
Allow for Long Running Processes

- Design services such that a result is not black or white
- Accommodate partial results
- ACID principles do not always apply in the case of web services
- WS- transaction has options
  - Relaxing one or more stringencies
  - Take this into account when designing messages
Allow for Long Running Processes

(Continued)

<results>
  <serviceResults>
    <serviceResult/>
    <error/>
    <serviceResult/>
    <error/>
    
    ........
    
    ........
  </serviceResults>
  <error/>
</results>
Handlers

- Split your information between the application payload and the out-of-band data

- Example:
  - Information such as credentials, TP context need not become intermingled with the business data
<soap:env>
  <soap:header xmlns:ns1="mynamespace">
    <ns1:tp_info context="45ff"/>
    <ns1:myCreds uid="kdjkdj" pwd="#$#$"/>
  </soap:header>
  <soap:body>
    ....
    ...
  </soap:body>
</soap:env>
Is There a Magic Mantra?

- Design your domain model
- Derive your business objects from the business model artifacts
- Compose granular components and services using these fine-grained business objects
- Compose coarser grained services using a mix of these components and services
The Model Driven Approach

- Design your domain model using a language neutral format such as UML

- Use tools that allow easy movement between logical model and physical artifact

  [generate XML schemas from the UML diagrams]
Model to Implementation

- Using tools such as PowerDesigner, we can generate the business model as a set of XML schemas

- Following best practices for using namespaces in these schema documents, we get a set of schemas that represent the business domain model
Sample Schema

```xml
<schema>
  <import namespace="personNs"
    schemaLocation="./person.xsd"/>
  <complexType name="traveller">
    <complexContent>
      <extension base="person"/>
    </complexContent>
  </complexType>
</schema>
```
Compose a Higher Level Schema

```xml
<schema>
  <import namespace="ns1" schemaLocation="traveller"/>
  <import namespace="ns2" schemaLocation="address"/>
  <complexType name="profile">
    <element ref="ns1:traveller"/>
    <element ref="ns2:address"/>
  </complexType>
</schema>
```
And Use That in Your Request

```xml
<schema>
  <import namespace="n1"
    schemaLocation="profile.xsd"/>
  <import namespace="n2"
    schemaLocation="preferences.xsd"/>

  <complexType name="AirlineRequest">
    <element ref="n1:profile"/>
    <element ref="n2:preferences"/>
  </complexType>
</schema>
```
Putting It All Together

<definitions>
    <types>
        <element name="req" type="AirlineRequest"/>
        <element name="resp" type="AirlineResponse"/>
    </types>
    ....
    <message name="xmlIn" part="req"/>
</message>
Namespaces for Schemas

- Remember to qualify namespaces

- As we move to more and more services there needs to be a way to uniquely identify the language
  - *i.e.* – make sure namespaces distinguish intent

- RDF may be one way
Advantages

- When a business object changes, the client automatically picks up the change by a refresh of the WSDL

- The server side objects and client code are kept in sync with minimal effort

- As long as the interfaces are invariant, there is no ripple effect
Importance of WSDL Doc

- Almost every tool on the market uses them.

- Make sure the WSDL accurately reflects the message content.
  - Tools such as .NET VStudio won’t be able to use it otherwise (interoperability is a concern).
  - Namespaces are a concern as well (standards differ between how .NET and AXIS handle them).
Loosely Coupled Interfaces

- Only two things need be known for a client to use a service:
  - Endpoint URL (available from the WSDL)
  - Input and output message types (-ditto-)
While on the Server Side ...

- The service APIs and the business schemas are processed using an XML binding tool such as CASTOR or JAXB
- A common business domain model jar is then created and used throughout the domain
- All these changes are done at compile time
Business Process Integration

- Done using tools that allow workflow design
- Visually compose the flow and inspect the generated WSDL

- A process is itself a ‘Service’ – hence the meta-service paradigm still holds
- Standards compliant BP Managers are the way to go
Business Process Managers

- Run on app servers (Tomcat)

- Typically expose the process as yet another web service that can be invoked by a client

- Have tools that integrate with IDEs such as WSAD and eclipse (generate XML)
Design Your Process

<definitions name="EmpService" ...
   xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/
   xmlns:soap="http://schemas.xmlsoap.org/wsdI/soap/">
...

<portType name="EmpInterface">
   <operation name="getEmpSalary">
      <input message="tns:EmpInterface_getEmpSalary"/>
      <output message="tns:EmpInterface_getEmpSalaryResponse"/>
   </operation>
</portType>
...

<plnk:partnerLinkType name="EmpServiceLink">
   <plnk:role name="EmpServiceProvider">
      <plnk:portType name="tns:EmpInterface"/>
   </plnk:role>
</plnk:partnerLinkType>
</definitions>
Instrument Your Services

- SOA framework should have this at all touchpoints

- This provides out of the box monitoring information

- In addition provide hooks for service writers to implement custom monitoring
Example

- Client calls a service
  - The SOA framework logs transit times between a WS Gateway and Broker
  - When a service makes a network hop to another remote node, the transit time there is logged by that instance of the broker

- How is transit time spent in a legacy or 3rd party service available?
  - Case for use of the custom hooks
Location and Interface Neutral Code

- Get URLs from JNDI, nothing is hardcoded

- Interface ubiquity – all the services on all nodes are accessed using identical interfaces
  - The payload determines the processing

- Relocate responsibilities for infrastructure tasks (security, logging, monitoring) to the framework
Don’t Reinvent the Wheel

- Use off the shelf component (within budgetary constraints)

- Rewriting transport adapters and XSLT transforms for every new project is a waste
Contact Info

- Grimgaunt@yahoo.com
- Hari.rajagopal@galileo.com