Spring, Mule, Maven: Lightweight SOA with Java (2)

Andrew Cooke*, Alvaro Egaña

April 2007

Abstract

We show how Spring[2] and Mule[1] can help implement services within a Java–based SOA (Service Oriented Architecture). The presentation has a practical emphasis, based on our own experience.

The approach is minimal and incremental, building around 'core' business logic implemented in 'plain' objects (POJOs). Using careful design and choice of technology we extend this, focusing on messaging.

This is an updated version of our earlier (September 2006) paper, including details of the POrqi[3] library.

Contents

*andrew@acooke.org

1 Introduction

1.1 Aims

This paper introduces an 'implementation recipe', summarised in section 5.1. We are implementing a SOA system; this is the paper we wish we had read before we started.

Our approach is minimal and incremental. At its core is the service's business logic[1], implemented in 'plain' Java (POJOs), with an associated interface.

Some services depend on others. This is inevitable. Rather than obscure this dependency, we make it explicit; dependency is expressed through a direct call to the sub–service's business interface. This allows

[1] The ideas that the service embodies, no matter what supporting technology is involved.

easy verification of the system, via both compilation and 'integration tests'[2].

With careful design and choice of technology we can then add further functionality *without influencing the core*. This paper focuses particularly on messaging, but we believe that the approach is general; that one should start with the simplest system necessary, understand it, and then incrementally extend it.

1.2 Scope

Our ideas rely on the language having a (static) type system. In section 3 we recommend Java technologies and we will use Java language terminology throughout the paper, but a similar approach should be possible with C++ or C#.

1.3 Our Background

We[3] have been working with Mule[1] and Spring[2] at CTIO (Cerro–Tololo Inter–American Observatory), helping develop the DMaSS (Data Management and Science Solutions) platform (which includes the NSA (New Science Archive)). This is a (incomplete) Java–based set of services that can be assembled to create an archive for astronomical data[4].

Before this, we have worked with various J2EE server–based systems (WebLogic, WebSphere, JOnAS and JBoss). In comparison, Spring and Mule applications 'feel' much more flexible; the frameworks are less intrusive and the separation between logic and support services is much clearer. However, we can also 'retro–fit' the lessons we have learnt back into a J2EE server environment: we have written many services that can be deployed in either.

1.4 Road–map

We present the basic ideas behind our approach in section 2; section 2.1 motivates all the subsequent work. In section 3 we explain how certain technologies can simplify the approach. Section 4 is an extended example that shows how a single service can be deployed in a variety of different scenarios. Our 'recipe' is summarised in section 5, where we also discuss future work.

2 Interfaces

2.1 Introduction

'Programming to an interface' is a standard idiom within the Java community; interfaces[5] are used to define a framework which is then implemented with set of classes.

We use interfaces to:

— *Document* the components.

— *Verify* that components are compatible.

— *Delay* the choice of implementation.

Verification is implemented by the compiler, which checks that the code is consistent. This is impossible in untyped languages and difficult when components are completely isolated.

While static verification is important, it is not sufficient for a reliable system; tests are important too. We will show how the consistent use of interfaces helps improve testing.

Delaying implementation choices until assembly is the 'dependency injection pattern' (a.k.a. 'inversion of control') popularised by Spring and adopted by EJB3[4].

These ideas are complementary: delaying implementation helps manage inter–dependencies between packages which, in turn, enables verification; verification assures that delayed choices are safe choices.

The rest of this paper shows how these ideas can be applied to SOA.

2.2 Services

Verification requires that each service 'know' the interface provided by other services; but we must avoid

[2] In–memory tests of multiple services without messaging

[3] AC has since left the project.

[4] While we use DMaSS for our examples, and owe much to discussions with our colleagues at NOAO, this paper is a personal project; it is not an official NOAO publication and does not necessarily reflect how NOAO implements software.

[5] The general software engineering idea of an interface is represented in Java by an 'interface' construct, which is equivalent to a purely abstract class.

circular chains of references, or services cannot be compiled separately. There are several solutions to this problem: restrict communication between services to avoid cycles; use a hub/spoke architecture; separate interface and implementation; forgo verification.

For simple systems, the first of these may be sufficient. Otherwise, separating interface and implementation is preferred, since it allows verification without restricting topology.

So we require each service to have a clear, simple 'business interface'[6] that depends only on basic Java classes and stand–alone libraries[7]. Service business implementations then depend only on business interfaces to services (their own, and any services they depend on).

Maven (section 3.4) will automatically order the compilation if packages are structured in this way.

Note that we are relying on the ability to *delay* the choice of implementations. If Service A calls Service B, it is compiled against B's interface, alone. B's implementation is compiled separately and injected later, at run–time.

2.3 Communication

So far we have not addressed messaging. This is not an oversight: messaging should influence neither the business interface nor the core implementation.

Requiring that business interfaces and implementations be independent of messaging brings several advantages: it makes it easy to test multiple services; allows different messaging technologies to be used; and leads to an implementation in which additional functionality, like caching, is cleanly separated from the main business logic.

However, if we are building a distributed system, we obviously cannot ignore messaging completely. We must be able to send, route, and receive messages.

[6]In the next section we will introduce a distinct 'communications interface'.

[7]Some libraries may depend on others, but none must depend on any service.

Sending Messages Messaging is sent via a client. The client is a facade that implements the standard business service interface, but delegates implementation, via messaging, to a remote server.

The POrqi[3] library can automatically generate a client, given the service interface, via a Spring factory. This facade is injected into the calling component in the normal way; the original code (business logic) remains unaltered.

Receiving Messages A server is responsible for receiving messages, unpacking them, and calling an embedded (injected) service implementation. It must also return the result back to the client.

The server implements the 'communications interface'. This interface is exposed by the service via the communications system. Some callers may call address this interface directly, rather than using the client.

The POrqi library provides a generic server that can expose a service to messaging. It works in tandem with the client facade described above and, similarly, does not alter the original code.

Routing Messages We separate message routing from the client and server. The messaging technology must be capable of separate configuration.

So, in summary, the client is a facade that implements the service's business interface, packages arguments into messages, dispatches them to the messaging solution, and receives and unbundles the response. The server performs the reverse functions; receiving messages, unbundling them, calling an embedded implementation, receiving, packaging and returning the result.

We can then inject, into any user of a service, either the direct business implementation, or the communications client. In this way we can choose, at deploy time, whether or not the two services are separated by messaging.

The previous version of this paper placed this functionality within a 'communications package' — this package is not needed when the POrqi library is used since all the infrastructure is generated automatically.

Technology Independence As much as possible, neither client nor server should assume a particular technology. The client accepts a simple interface (injected at run–time) for sending a serialisable Java object. The server is a simple class[8] that accepts and returns a serialisable object; the main processing is handled by an embedded service implementation which, again, is injected at run–time.

Both servers and clients have a simple, regular structure that allows easy generalisation of related functionality, like caching.

The POrqi library is non–intrusive (it is not referenced by the business logic) and technology–agnostic. It includes support for messaging via Mule (which supports many messaging protocols and can itself be extended further), but this is only one implementation of a 'channel' abstraction; other implementations can be added.

Common Messaging The previous version of this paper noted that *common patterns in messaging can be abstracted to a separate library. This should be restricted to have no dependencies on other packages (except libraries). Typical message classes include carriers for a 'payload', an empty 'acknowledge', and carriers that include either an exception or a payload.*

Following this advice led to the creation of the POrqi library.

Note — Any object serialised by messaging (not just the generic messaging classes, but also payloads) must have an appropriate class in both client and server. Following the guidelines above guarantees this, but it is important to understand that the constraint also applies to chained exceptions.

Two Interfaces? Services have two interfaces: what we have called the business and communications interfaces. The two are closely related and the question naturally arises: which is more fundamental?

We are most concerned here with relatively closed systems where many inter–operating services are written in Java. In such a situation, the business interface dominates. However, the communications interface should not be ignored, as it documents an important logical component of the system which will likely become more important as the system matures, growing stronger connections with external processes.

[8]The user of 'server' may be misleading here. The responsibility for listening to ports, queues, etc, is left to whatever messaging solution is used. Our 'server' receives a 'message' when a method that takes a serialisable Java object is called.

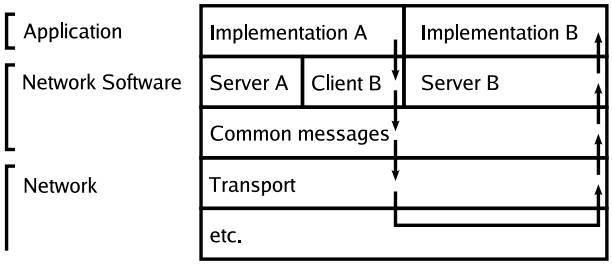

Figure 1: The communications stack: service A sends a message to service B.

Layered Architecture The description above describes each service as though it has a client–server structure. However, the same components fit within a layered framework, as shown in figure 1. Our client–server distinction is then a natural way to separate the implementation of the network software. The common messaging classes describe a common protocol shared by all services.

This could be emphasised by splitting the communications package into four: interface; client; server; messages. This minimises the network software required for any particular service deployment.

2.4 Testing

Delayed choice of implementation allows services to be assembled in–memory for testing, even if they are deployed separately; instead of injecting the communications client, the implementation is used directly.

This allows 'integration tests' to be run on developer's machines, without the need for complex deployment and test harnesses.

Acceptance tests, using a full deployment with messaging, are still necessary, but the simpler integration tests allow most cross–service bugs to be diagnosed and corrected more quickly.

Another advantage of the emphasis on interfaces is that dummy ('mock') implementations of sub–components can be written and injected as needed. Since these are typically both lightweight and useful, we suggest keeping them with the main service implementation, rather than restricting them to Maven's `test` directories.

3 Technologies

3.1 Java

The approach outlined here was developed with Java.

A similar approach might be possible with C++, but we can see little justification for doing so: Java includes memory management, a safer type system, and a wider choice of tools for this type of application; the advantages of C++ apply more to applications with restrictive hardware requirements.

Microsoft's .net platform is a possible alternative, but then it would be wiser to follow the implementation route assumed by their proprietary tools.

Ruby, Python and Perl require a different approach since their type systems do not support the compile–time verification that motivates many of our recommendations.

More modern, statically typed languages (ML, OCaml, Haskell) are another option, but we are not convinced that they have sufficient third–party support (particularly for messaging). One promising candidate is Scala[5], which compiles to the JVM (Java Virtual Machine) and so can interoperate with Java–based tools.

3.2 Spring

Spring[2] is a rather large application framework. Here we are concerned only with the inversion of control container it provides (although it may also be useful later for transaction management, ORM, and the presentation layer).

The Spring container allows us to assemble applications from the classes in the implementation package. It largely eliminates the need for factories and singletons in the code. Instead, at run–time, the Spring configuration defines how instances are constructed. This works extremely well with the interface–based approach we have described, since much of the boilerplate code is provided 'for free'.

3.3 Mule

Mule[1] can be considered as 'Spring for messaging'. In as non–invasive a manner as possible it connects Java objects with messaging technologies.

In particular, it can create Java objects from a Spring description and, when an appropriate message is received, it can call an instance method, passing the message as an argument. Similarly, it can take the value returned by the method and treat it as a response method. This is why our 'server' code can be plain Java objects (POJOs).

It also provides a standard, generic interface to a wide variety of other messaging systems. So, for example, it can interoperate with REST (or even email). It can also build synchronous messaging from an asynchronous transport like ActiveMQ[6].

3.4 Maven

We use Maven v2[7] to manage the build process. Its rigid approach makes it easy to track dependencies between packages.

We suggest using a flat 'logical' structure, with all packages being children on a single top–level parent[9]. Each package is then configured separately, the parent is used only to set a few global parameters and allow whole–project compilation, testing, etc.

This does not mean that the packages themselves need to be in a flat directory structure. Instead, we suggest grouping by service. So the parent directory contains a sub–directory for each project; each of those contains further directories for each package (interface, implementation, communication, deploy, etc). A further child of the parent directory contains all the libraries, etc.

[9] If the auto–generated website is important then a two level system that groups packages by service may be preferable; even then it is simplest to keep each package as independent as possible.

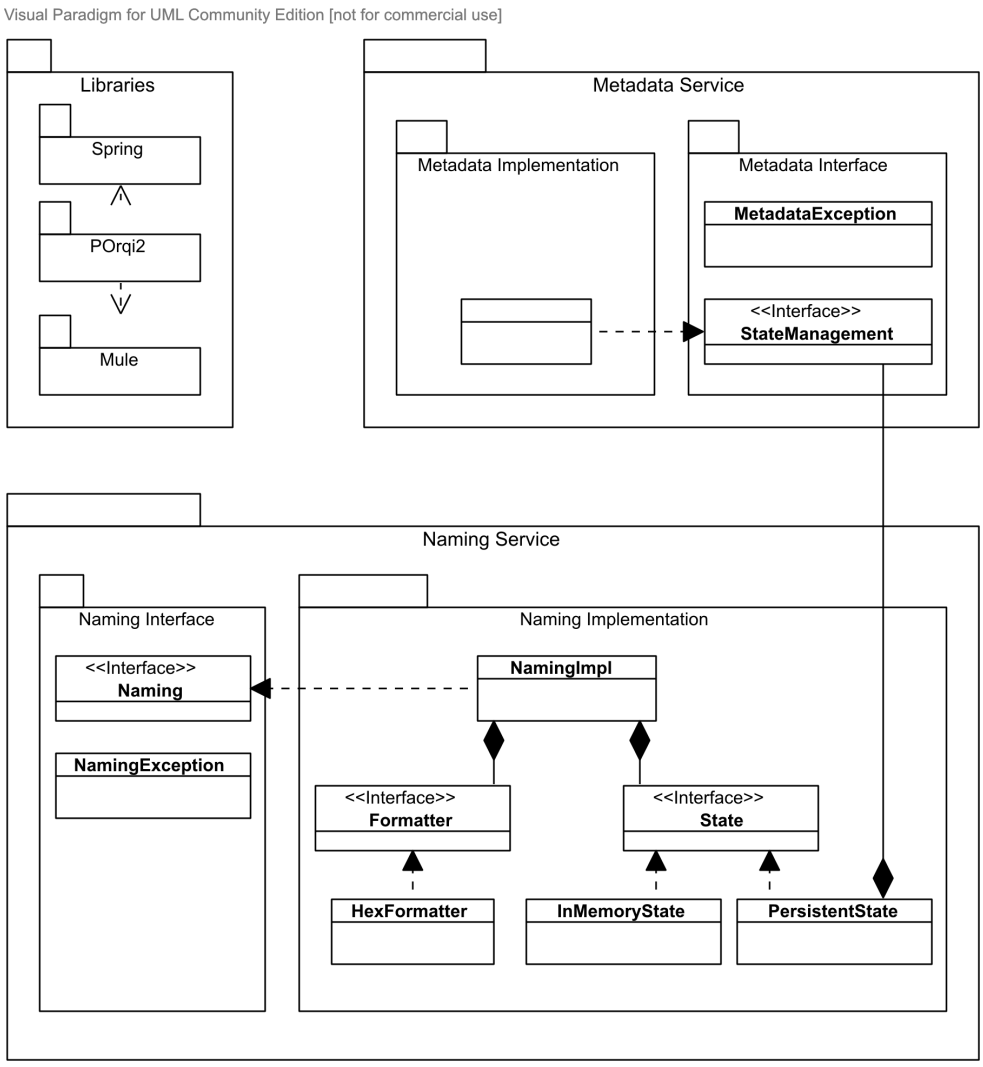

Figure 2: Naming Service structure (including a possible dependency on Metadata Service to persist state). There are no compile-time dependencies on infrastructure libraries (POrqi, Spring, Mule).

Guidelines Maven works just fine, as long as you follow these simple rules:

- Within a project, use the standard directory structure.
- Use many, many projects (Maven does not provide functionality at a resolution lower than projects; you can easily use Maven to combine them later if you need to).
- The projects can be located wherever you like (so you can group related projects in a single directory).
- Use a simple bash script to create the top level POM (Project Object Model; a Maven configuration file) from those living in any sub–directory.
- Trust Maven. If something is hard, you are either doing it the wrong way (try splitting your work into more projects) or your code's high–level structure is poor.

3.5 POrqi

The POrqi[3] library provides support for the ideas described here. It is intended for use with Spring and Mule, but can alo be used with other inversion of control and messaging frameworks.

POrqi will also address the problem of Asynchronous Communication described in section 5.3.

4 Example

4.1 Recapitulation

In the previous sections we advocated splitting each service's business code into two separate packages:

Business Interface: A description of the service, in terms of simple Java language and library objects.

Business Implementation: A direct implementation of the business logic that conforms to the business interface and refers to other services via their own business interfaces. Spring can be used to inject appropriate instances at run–time.

In addition, for communication, we recommended using:

Communication Interface: The interface exposed directly through the communications service. This includes the messages, based on a common protocol library.

Communication Client: A facade that implements the business interface but sends messages to an implementation of the communications interface (the server).

Communication Server: An adapter that converts the business interface to the 'message friendly' communication interface. Spring can be used to inject a business implementation; Mule can transfer messages between client and server.

The POrqi library can provide, or create (via factories used during deployment), these communication components.

4.2 Naming Service

This example describes a Naming Service which provides unique, system–wide identifiers. Its interface is a single method, `String getNewName()`, that returns a new value on each call.

4.3 Structure

The Naming Service and its main dependencies are sketched in figure 2.

The *business interface* package contains the `Naming` interface described above and `NamingException`, which is the exception thrown by the interface on error.

The main class in the *business implementation* takes two interfaces: `State` which provides a series of `long` values; `Formatter` which converts `long` to `String`.

Different implementations of these interfaces can be deployed for different behaviours. `InMemoryState`

is a class with an `AtomicLong` instance field, whose value is incremented and returned (obviously this functionality is too simple for a reliable, distributed system, but it will serve for testing). `HexFormatter` formats the number in hexadecimal.

`PersistentState` is a second, more sophisticated implementation of `State` that uses the Metadata Service to persist values. This service is not shown in detail in the figure, but illustrates how one service can depend on another.

The *communications* packages are not shown because they will be generated by the POrqi library (the previous version of this paper required the user to write the mesagig related classes).

4.4 Commentary

Figure 2 illustrates how our approach leads to a system with the following compile–time relationships:

- Anything can depend on a library.
- Anything except a library can depend on any other service's interface.
- Nothing can depend on a service's business implementation package.
- Nothing depends on the infrastucture libraries.

Together, these guarantee that the core business logic of each service remains isolated from technology choices, from other service implementations, and from the majority of implementation details.

The description of the Naming Service business implementation and interface above (one exception, a handful of interfaces and a similar number of classes) is practically complete. It is difficult to convey quite how simple this code is. It contains no references to J2EE, container, or messaging technology; nor does it contain any singletons or factories (despite the clear need for a singleton `State`). Yet it can be deployed to give unique names to several different callers distributed across a network; this is addressed in the next section.

4.5 Deployment

The service can be deployed in a number of ways, relative to some calling service:

- As part of a monolithic system, running within the same JVM as the caller.
- As a remote server, called from Java.
- As a remote server, called by a non–Java service.
- Embedded within a J2EE server.

The choice of deployment method should depend on the particular circumstances; typically depending on the underlying communications service being used, the degree of reliability required, etc.

Monolithic System in JVM. The caller will expect an implementation of `Naming`, which is injected. The business interface and implementation packages are used. This is shown in figure 3 (left).

Remote Server, Java. The same user code as in the previous section can be configured to call a remote service by using the POrqi library. This provides `SynchClientFactory` which can be configured with Spring / Mule to generate a proxy client which implements the `Naming` interface.

The Naming Service implementation is deployed on a second machine, embedded in `SynchServer` (also provided by POrqi). The client and server can be configured to use any messaging system supported by Mule (or any other communications channel written to POrqi's `SynchChannel` interface).

Note that the Naming implementation package is not needed on the caller machine and that the business logic (the code in the Naming Service) is unaltered.

We chose, in this example, to deploy the entire Naming Service remotely. However we could just as easily have kept most of the service local to the caller, extracting just the `State` interface. In effect, we are considering the Naming Service to be itself composed of sub–services which can be deployed as we require. This is possible because the only requirement on a

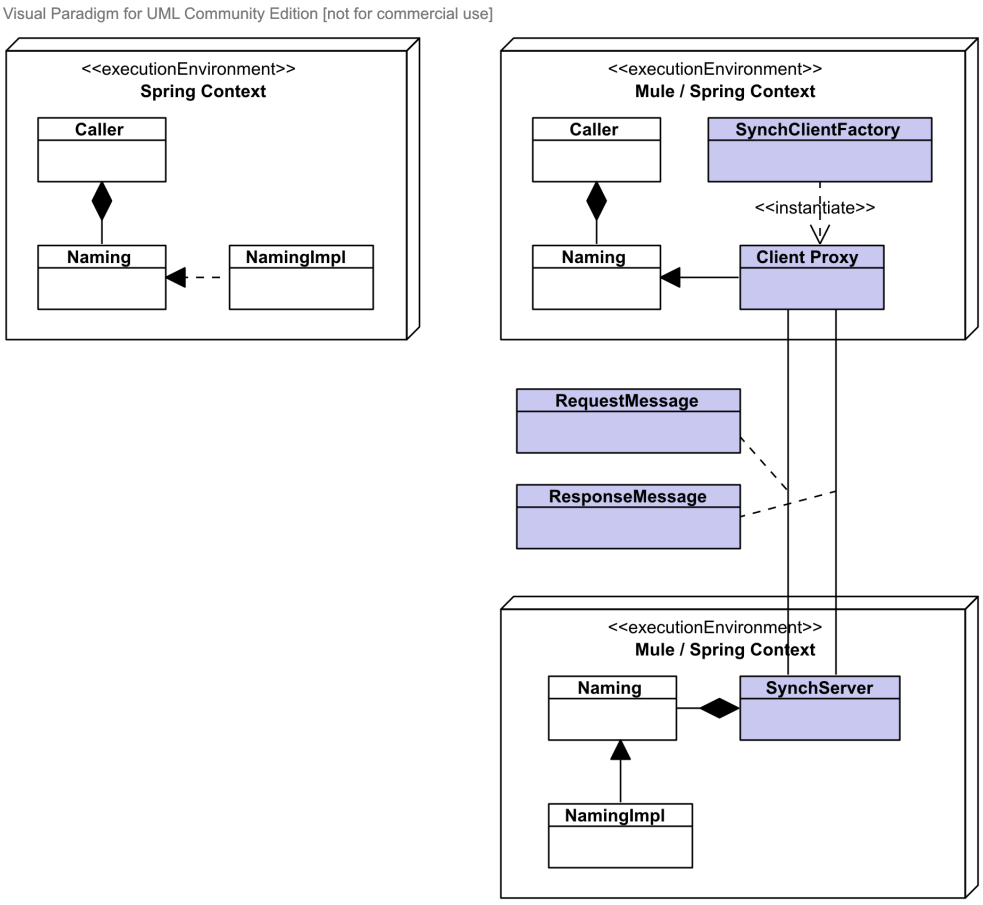

Figure 3: Local (left) and distributed (right) deployment of the Naming Service. Grey classes are provided by the POrqi library.

'Service' is that it be a POJO that implements an interface.

One advantage of a local Naming Service (with a remote `State`) would be the ability to cache names locally, providing a fast, reliable response even when communication to the remote server, which persists the state, is slow[10].

Remote Server, Non–Java. In the previous section the Naming service server process was called by Java code via POrqi's client proxy. The actual messaging work was done by Mule (possibly over an appropriately configured delegate transport layer). Non–Java code (or Java code written separately from our system) can also call the `SynchServer` using any messaging implementation that inter–operates with Mule.

However, in most cases, it will make more sense to configure a separate Mule endpoint for the target service (ie `NamingImpl`). Mule can deliver directly to POJOs, so the infrastructure remains non–invasive, and provides more flexibility in matching the caller's requirements.

J2EE Server. It is easy to write adapters that allow services to be deployed in J2EE servers as stateless session beans. We place the necessary facades in a separate package and use EJB3 annotations to inject the implementation.

We wrote a `MessageSender` implementation that can be deployed in a J2EE server and extended Mule so that it can call EJBs in the server[11].

5 Conclusions

5.1 Recipe

- Use interfaces to document and verify the code, and to delay the choice of implementation.
- Separate interface and core implementation into separate packages.
- Require that interfaces and core implementations ignore communications.
- Unit test.
- Auto–generate messaging client and server (eg. via POrqi).
- Choose the appropriate service implementation (direct or via the messaging client) at deploy time.
- Compose services in memory (without messaging) for further 'integration' testing.
- Use Mule directly to handle 'external' connections.
- Java, Spring, Mule and Maven make this approach easy.
- Don't forget to test deployed services (with messaging) too.

5.2 Simplicity

The implementation package contains the business logic — and nothing else. This makes the approach here future–proof: no matter what happens, the important 'logic' of the service is cleanly isolated and ready for re–use.

POrqi, Spring and Mule provide the extra structure necessary to convert the business logic into a practical service. Spring imposes the structure that would normally require singleton and factory boilerplate. POrqi and Mule add messaging.

It is hard to understand just how simple the resulting code is. Despite having used the techniques described here we still find ourselves writing code that is too complex; in a later iteration we will delete factories and singletons we had earlier written through habit.

5.3 Future Directions

Asynchronous Communication. The description above has focused on a system in which communication is synchronous.

[10] POrqi does not address reliability issues directly, but Mule provides configurable retry, error handling, etc.

[11] This extension has been submitted to Mule and, apparently, is included in the 1.4 release as the JNDI provider.

Mule can construct synchronous messaging over an asynchronous transport, but this does not address the more important problem of making the caller process reliable.

If service A calls service B via Mule using a reliable transport then *some instance* of A will[12] eventually receive a reply; if A has restarted in the meantime, however, it will not contain the correct state to process the response. The system as a whole, therefore, fails.

To solve this problem we must either persist the state of the caller while waiting for a response, or restructure the system into smaller steps which communicate asynchronously. The former of these can be seen as a way of automating the latter.

The standard approach to persisting state is to use a workflow which manages persistence and the interface with the communications system. An alternative approch, which is partially implemented in the current release (1.3; the 'asynch' packages) of POrqi, is to adapt classes so that state is automatically managed via continuations. If this is successful then POJOs will become as reliable as workflow engines.

SOAP. Separate from the discussion about asynchronous communication above, we want to comment briefly on SOAP and web services. The service structure outlined here is very similar to that used by web services (`ClientProxy` plays a similar role to the client code generated from the WSDL). And Mule can interface to external SOAP services.

However, our most painful experience so far has been trying to interface to an external web service. We think this was for the following reasons: poor communication between two different groups during development; lack of experience with SOAP on our part; shifting standards and incompatible implementations.

JBI. Java Business Integration (JBI)[9] uses SOAP and is supported by recent versions of Mule. This is another possible route for future development.

[12] In theory; in practice Mule will fail unless explicitly configured for asynchronous messaging, for the reasons described next.

Unforseen Changes. We cannot predict the future, but that does not alarm us. We do not claim to have solved every problem, but we do feel that we have a future–proof approach. Code developed following these guidelines is so simple, and commits so little to any one technology, that future transitions should be easy.

5.4 Philosophical Purity

And yet... The approach above seems to be a poor approximation to a *Service* Oriented Architecture: if the services need each other's interface to compile, then why not build a monolithic (but modular) system?

Perhaps the best defence of our approach is that we can build a 'monolithic' system in a way that is so modular that the transition to 'real' SOA is relatively painless. We (1) have described some approaches to this transition (interfacing with external / non–Java code via Mule; JBI; persistent state); (2) can often limit this transition only to isolated parts (often the 'edges') of our system; (3) believe that the advantages of this approach — stronger guarantees on consistency and simpler tools — are perfect for initial development and, with (1) and (2), appropriate for the 'core' of many mature systems.

If SOA is more than fashion, it embodies some insight about engineering systems. We hope to respect those insights without needing to wear an unnecessary hair shirt. This is the best compromise we have found.

6 Acknowledgements

The criticism of others on the NOAO NSA team has helped us improve these ideas and motivated us to write this paper: Sonya Lowry has tried very hard to explain SOA to us; Evan Deaubl was particularly helpful with Maven and packaging issues; without Phil Warner we would never have understood how these ideas can adapt to J2EE servers; Brian Thomas at UMD pushed the limits of XML and, with Ping Huang, helped us hone our arguments for (sometimes) re–inventing the wheel.

References

[1] Mule
http://mule.codehaus.org/

[2] Spring Application Framework
http://www.springframework.org/

[3] POrqi the POJO POrquestrator
http://www.acooke.org/jara/porqi/index.html

[4] JSR 220: Enterprise JavaBeans 3.0
http://jcp.org/en/jsr/detail?id=220

[5] The Scala Programming Language
http://scala.epfl.ch/

[6] ActiveMQ
http://www.activemq.org/site/home.html

[7] Apache Maven Project
http://maven.apache.org/

[8] A. Cooke 2006; A Tiny Workflow in Spring
http://www.acooke.org/andrew/papers

[9] JSR 208: Java Business Integration
http://jcp.org/en/jsr/detail?id=208