How To Implement Lightweight ESOA with Java

Andrew Cooke*

December 2006

Abstract

La arquitectura SOA utiliza servicios para integrar sistemas heterogéneos. Enterprise SOA (ESOA) extiende este enfoque a la estructura interna de sistemas, lo que ha introducido nuevos desafios. La industria de software ha respondido con los ESBs — herramientas bastante complejas. Al mismo tiempo ha habido un movimiento hacia el desarrollo 'lightweight', con su énfasis en objetos simples de Java (POJOs) y la injección de dependencias.

Este paper presenta un enfoque 'lightweight' al desarollo con ESBs.

Service Oriented Architecture (SOA) integrates heterogneous systems via services. Enterprise SOA (ESOA) extends this approach to internal development. This has introduced new challenges and vendors have responded with complex Enterprise Service Bus (ESB) technologies. At the same time, there has been a movement towards 'lightweight' development, with an emphasis on Plain Old Java Objects (POJOs) and Dependency Injection.

This paper brings a 'lightweight' approach to development with ESBs.

Contents

1 Introduction 1

2 Short Historical Sketch 2
- 2.1 Generation 1: SOA . . . 2
- 2.2 Generation 2: ESOA . . . 2
- 2.3 Complexity v POJOs . . . 2

3 The Approach 2
- 3.1 An Example . . . 2
- 3.2 Lightweight . . . 4
- 3.3 Service Interface . . . 4
- 3.4 Business Bean . . . 4
- 3.5 Late Binding . . . 4
- 3.6 Mediation . . . 4
- 3.7 Composite Services . . . 6
- 3.8 Summary . . . 6

4 Mule 6
- 4.1 Why Mule? . . . 6
- 4.2 Using Mule . . . 6
- 4.3 Deployment Options . . . 7
- 4.4 Support Library . . . 7

5 Grab–Bag 7
- 5.1 Team Structure . . . 7
- 5.2 Process . . . 7
- 5.3 Testing . . . 9
- 5.4 Implement Multiple Interfaces . . . 9
- 5.5 Heavyweight Containers . . . 9
- 5.6 Aspects . . . 9

6 Conclusion 9

7 Glossary 10

8 Acknowledgements 10

1 Introduction

Section 2, the 'Short Historical Sketch', introduces our problem: how to implement an Enterprise system using Service Oriented principles without being

*andrew@acooke.org

DRAFT

overwhelmed by (or irrevocably comitted to) complex technologies.

The solution, described in section 3, continues the 'lightweight' approach that first appeared as a response to the complexity of complex J2EE containers: place the business logic in a POJO and then connect that to the support technology.

One particular technology, the Mule ESB, is discussed in in section 4. A variety of related details and lessons learnt are collected in section 5.

2 Short Historical Sketch

2.1 Generation 1: SOA

Service Oriented Architecture (SOA) emphasised:

Business processes — each service implements a particular business task.

Decoupling — services are interchangeable, defined only by interface (syntax) and contract (semantics).

Multiple ownership — the architecture can integrate services belonging to several organisations.

SOA took principles familiar to software design and applied them within a wider, more business–oriented context. The emphasis was on cross–system, *inter*–organization integration.

Common features of this 'first generation' SOA can be traced back to these key aspects: standard communication protocols are necessary for interoperability; an emphasis on interfaces, contracts and resource discovery allows dynamic service selection; persistent state is avoided (hence asynchronous communication, which is also more efficient); orchestration and composition of processes occurs at a high level.

2.2 Generation 2: ESOA

The success of SOA encouraged people to apply the same principles to other areas; particularly, *within* organisations. Enterprise SOA (ESOA) introduced a shift in emphasis towards *intra*–organisation systems:

Common Data — a collection of services may implement distinct business processes that operate on common data.

New Code — new systems are designed 'in an SOA way', rather than using SOA to assemble existing business processes.

As SOA moves to smaller scales it has to deal with a wider range of technical problems. For example, shared state suggests a need for transactions. At the same time, development of completely new systems is often within a more homogenous environment.

The result was the introduction of Enterprise Service Buses (ESBs), which extended SOA's communication protocols to handle more complex problems and tried to combine wide inter–operability with ease of use (when deployed with specific — often vendor–specific — technologies).

2.3 Complexity v POJOs

So we can see how moving inter–organisation SOA technologies (typically XML based Web Services) to intra–organisation applications (typically using heavyweight J2EE containers) lead to complex ESB technologies.

But at the same time many people were starting to question the utility of heavyweight J2EE containers and moving towards lighter–weight, dependency injection solutions with a stronger emphasis on Plain Old Java Objects (POJOs).

This paper tries to resolve these two apparently opposing movements: it applies the lessons learnt from 'lightweight' development to the challenges introduced by ESOA.

3 The Approach

3.1 An Example

Figure 1 shows a typical ESOA problem. We must implement a new system, with a Service Oriented architecture, which inter–operates with existing services both inside and outside our organisation.

DRAFT

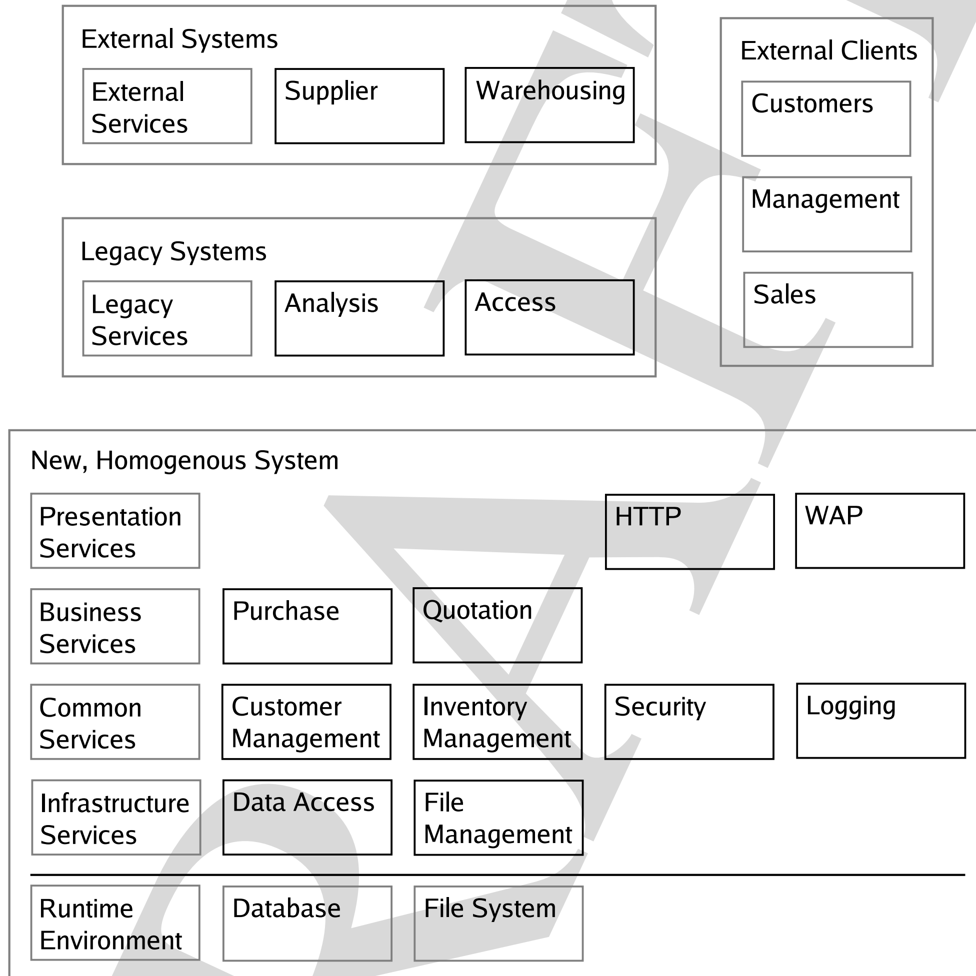

Figure 1: An example ESOA problem — how to develop a new, relatively homogenous system that integrates external/legacy systems from *outside* while exploiting the advantages of developing within a relatively homogenous environment *inside*. Services are shown with dark borders; note the layered architecture within the new system.

```
public interface Example {
  public void voidMethod() throws ExampleException;
  public String multiMethod(int n, String text) throws ExampleException;
}
```

Figure 2: The interface for the Example service.

DRAFT

The new system uses a best–practices layered architecture — there is no need to throw away lessons we have learnt from earlier approaches — and, typically, all the services within this system will be implemented using a single technology.

So we want to simplify the *inside*, intra–organisation SOA applications, using a POJO–based approach, without sacrificing the flexibility of the *outside* inter–organisation communication. We must also preserve the ability to expose *any* service, even those deep *inside* our new system.

3.2 Lightweight

We have already noted (section 2.1) that SOA is the application of standard software practices in a cross–business context. If we make this comparison explicit we find:

Service ⇔ Interface
Message ⇔ Method call
Discovery ⇔ Late binding
Orchestration ⇔ Mediator pattern

I take 'lightweight' to imply that the impact of support technologies on the business logic is minimal; a 'lightweight' implementation will aim to reflect the correspondences. So a service *is* an interface. When one service calls another, it will appear in the code as a simple method call to the corresponding interface.

3.3 Service Interface

Each service has a corresponding Java interface. This interface is written in normal, idiomatic Java: there is no restriction on the number of method parameters; exceptions can be thrown; etc.

However, since services are only weakly coupled, we expect that any classes that appear in a service interface are themselves either interfaces or defined in a small library. It must be possible for a consumer of that service to use the interface without linking to the entire service implementation.

The service interface is named to reflect the service. Typical examples might be 'User', 'Orders', or, as used later in this paper, 'Example'.

The service interface and any additional support classes form the 'interface library' (a jar file).

3.4 Business Bean

The business logic for each new service is implemented as a 'bean' — a POJO that implements the service interface. The logic refers to other services directly, by calling their interfaces. No infrastructure APIs (or annotations) are present in the code.

The business bean's name is the service interface name, with 'Bean' appended. So 'ExampleBean' is the implementation of the 'Example' service interface.

If the service requires access to other services then the code calls the appropriate service interface. The appropriate instance will be injected into the bean during deployment. Figure 3 gives a simple example: ConsumerBean implements the Consumer service interface and calls the Example service (defined in figure 2).

The business bean and its support classes form the 'business library' (a jar file). This will depend on the same service's interface library and also on the interface libraries of any client services. It will *not* depend on other services' business libraries for compilation.

3.5 Late Binding

There are different degrees of lateness. In some cases (eg. the *inside* of Figure 1) binding on deployment is sufficiently late. When later binding is required (eg. for *outside* services) it can often be achieved via the mediator (eg. the messaging system). Another option is a 'late binding facade', which implements the service interface but postpones binding as required[1].

The approach described here allows the correct degree of lateness of binding to be selected during deployment. It is independent of the business logic.

3.6 Mediation

Above I describe how services are injected as required. This is not directly consistent with the medi-

[1] Java's dynamic proxy support would allow this functionality to be implemented in a generic, service–neutral manner.

DRAFT

DRAFT

```
public class ConsumerBean implements Consumer {
  private Example example;
  public void setExample(Example example) {this.example = example;}
  public Example getExample() {
    return MissingServiceException.nonNull(example, getClass());
  }
  public Result someBusinessAction(Arguments args) {
    ...
    // call the Example service
    ... = getExample().singleMethod(...);
    ...
  }
}

public class MissingServiceException extends NullPointerException {
  static <Type> Type nonNull(Type client, Class consumer) {
    if (null == client) throw new MissingServiceException(...);
    return client;
  }
}
```

Figure 3: Consumer service implementation calls Example service. Only the Example service interface — not the ExampleBean business bean — appears in the code for ConsumerBean. The MissingServiceException class generates an appropriate error message if no service has been injected during deployment..

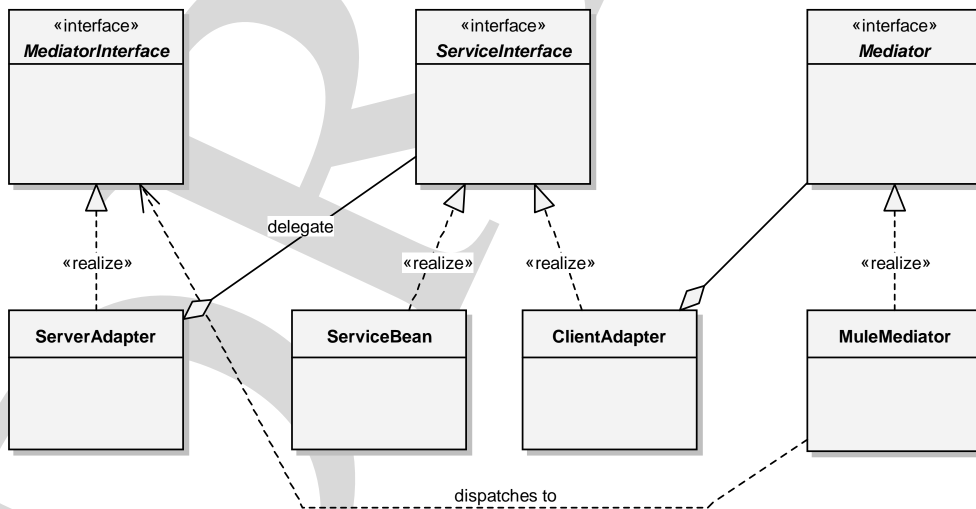

Figure 4: The relationship between Service, Adapters and Mediator. Here Mule is used as mediator and MediatorInterface conforms to Mule's requirements.

ator pattern, which is equivalent to injecting a single interface (the mediator).

Each service therefore requires an adapter, called the 'client adapter', which implements the service interface and transfers requests to the mediator. If several mediators are used then more than one adapter may be required (the relationship need not be one–to–one since, for example, many messaging systems may conform to a single generic 'send message' interface).

In addition, the technology used within the mediator may not be able to access each service directly, but instead requires an adapater. So each service may also have a 'server adapter' and, again, several mediators may mean that more than one adapter is required. This adapter will implement an interface specific to the mediator.

The relevant classes and interfaces are shown in figure 4.

Section 4 describes how the Mule ESB can be used as a mediator and how the generation of these adapters can be automated.

3.7 Composite Services

A composite service is one that choreographs the interaction of several client services (and, typically, contains little separate functionality — it is a workflow within a service).

If we have services that are implemented as decribed above then a composite service receives its clients through injection. In a 'full' deployment these will be adapters that interface to the messaging mediator. However, it is sometimes sufficient (for some tests, or when *inside*, as in figure 1) to inject the service business bean directly and so avoid the mediator. Obviously the advantages and disadvantages depend on the details of the mediator but, again, the choice can be made during deployment.

3.8 Summary

Problem: How to exploit new technologies without becoming 'locked in' to a particular vendor oroverwhelmed by irrelevant technical details?

Solution: Keep business logic in Plain Old Java Objects (POJOs); connect them to ESBs with standard patterns; use appropriate language features to simplify the boilerplate.

The result is a clean separation between business code and the support platform. Deployment decisions do not influence the code and cross–service testing is simplified.

4 Mule

4.1 Why Mule?

Until now, this paper has carefully avoided specific technologies. The emphasis has been strictly on keeping business logic separate from supporting platform technologies.

Any real project, however, will have to choose a messaging technology and I have found Mule to be a useful tool — it is a simple, unintrusive layer that simplifies Java–based development, supports a wide range of transports, and supports more complex technologies like JBI, transactions, etc, when needed.

Mule adapts to the distinction made in figure 1: it provides a simple solution *inside* and adapts to standard technologies *outside* (and the mediator approach allows us to chose whether a service is *inside* or *outside* at deployment time).

In addition, as a safety net, Mule is open source with a clean, extensible, well-designed architecture.

4.2 Using Mule

Mule allows a set of Beans to be assembled using Spring. It can then expose some of those beans as services, receiving external messages from a variety of transports and delivering them to the services as Java objects.

Mule also allows messages to be sent from Java and can construct synchronous messaging over asynchronous transports.

The work needed to integrate Mule with the approach in this paper focuses mainly on marshalling message parameters, return values and exceptions

DRAFT

within serializable Java ‘messages’. This can largely be automated (see section 4.4), but amounts to:

- Defining a set of message objects.
- Writing a client adapter that packages parameters into request messages and dispatches them to Mule, waits, then unpackages the response and throws an exception or returns the result.
- Writing a server adapter that receives the request messages, calls a delegate service implementation (typically a service bean), and then returning the result (or exception) in a response message.

This work could be avoided by writing service directly in ‘Mule style’: taking a single message parameter in arguments; either returning a response that encapsulates exception or result, or configuring Mule to propagate exceptions. However, my experience has been that this complicates the service interfaces, making the business logic harder to read, and tends to tie the business logic to a certain context (even when using Mule there are occasions when it can be useful to have several adapters, which function as a simple, pluggable ‘presentation layer’ for interacting with different consumers/clients). This is particularly true when the cost of writing adapters for Mule can be largely automated (section 4.4).

4.3 Deployment Options

Atypical ESOA system using Mule can group services at three levels:

- Services can co–exist in a single JVM. They can be assembled using Spring and communicate directly with Java method calls.
- Services can exist in several different Mule instances, within the organisation, each in different JVM instances. Messages are sent from one Service to another through the chain: calling service; client adapter; Mule mediator; service adapter; business bean.
- Services can exist in different organisations. They communicate with standard SOA protocols. Within the local organisation, Mule provides the interface to these services and calls the appropriate service adapter. Mule also provides the adapters necessary to interface with the standard protocols used externally.

4.4 Support Library

I have developed a simple library that automates the creation of client and server adapters for Mule. This code is part of DMaSS[2] and should soon be released into the public domain (the package is called ‘genericessages’[2]).

The package provides standard base classes for messages and factories that dynamically generate the adapters, given appropriate Java interfaces. Figure 5 shows example code; the Spring configuration used to dynamically generate the adapters is shown in figure 6.

5 Grab–Bag

5.1 Team Structure

The business logic is implemented as normal Java code; the infrastructure uses standard, open, third-party toolkits. Much of the system can be implemented by a team of programmers with no specialist knowledge. In small projects the remaining work can be handled by the architect / technical lead.

5.2 Process

Developers can concentrate on business logic while the final system can be adapted during deployment to meet a variety of different operational requirements: the methods described here accomodate and support an Agile culture.

[2]Due to a Java bug (id 6199662) the full library only compiles with JDK 1.6 and may not be part of the release planned for late 2006.

DRAFT

```
public interface MuleServerIface {
  public ExceptionResponseMessage<ExampleException>
    voidMethod(ExampleVoidRequest request);
  public PayloadExceptionResponseMessage<ExampleException, String>
    multiMethod(ExampleMultiRequest request);
}

public class ExampleVoidRequest
extends RequestMessage<ExampleException,
                       ExceptionResponseMessage<ExampleException>> {
  private static final long serialVersionUID = 1L;
  public ExampleVoidRequest() {super(ExampleException.class);}
}

public class ExampleMultiRequest
extends PayloadRequestMessage<ExampleException,
                              ExceptionResponseMessage<ExampleException>> {
  private static final long serialVersionUID = 1L;
  public ExampleMultiRequest() {super(ExampleException.class);}
  public ExampleMultiRequest(int n, String text) {
    super(ExampleException.class, n, text);
  }
}
```

Figure 5: The Java code needed to interface the Example service to Mule. The server adapter interface (MuleServerIface) and relevant messages must be defined. The adapters themselves can be generated (and tested) dynamically. The generic type information in the messages documents return and exception types and provides restricted compile–time checks.

DRAFT

DRAFT

```
<beans>
  <bean name="exampleServer" class="...ServerFactory">
    <property name="serviceIface" value="...Example"/>
    <property name="muleIface" value="...MuleServerIface"/>
    <property name="delegate" ref="exampleService"/>
  </bean>
  <bean name="exampleClient" class="...ClientFactory">
    <property name="serviceIface" value="...Example"/>
    <property name="muleIface" value="...MuleServerIface"/>
    <property name="muleMediator" ref="..."/>
  </bean>
  <bean name="exampleService" class="...ExampleBean"/>
</beans>
```

Figure 6: The Spring configuration used (within Mule) to instantiate server and client adapters for the example in figure 5İn practice any service only has one adapter (client or server) in any one Mule instance; the other will be remote.

5.3 Testing

This approach does not affect unit tests, which work as normal.

Integration, regression and acceptance testing, which all involve more than one service, are *significantly simplified*. A group of services can be assembled as a simple collection of objects, injected one into another, and tessted within the same framework as unit tests.

Mock services are equally simple — an IDE like Eclipse will automatically generate a skeleton service implementation from the interface definition.

5.4 Implement Multiple Interfaces

A clean architecture may introduce services which, particularly in early development iterations, do not justify separate implementations. In such cases a Business Bean may implement several service interfaces.

For example, a Context Service may be associated with a particular Metadata Service. A single bean may implement both. This restricts the ability to deploy the two services separately, but such services are typically deployed locally anyway, to reduce the costs of object marshalling.

5.5 Heavyweight Containers

If necessary, this approach can be extended to heavyweight J2EE containers. It is trivial to write EJB3 wrappers for Service Beans; Mule can invoke beans via JNI[3].

5.6 Aspects

Advanced functionality, like security and transactions, cross-cuts the business logic. We can use aspects to keep the appropriate technology APIs separate from the business code.

6 Conclusion

Careful implementation is critical in ESOA. A project can be structured to use complex technologies only when they are required. This allows the advantages of SOA and 'lightweight' approaches to be combined. The approach described in this paper cleanly separates business logic from support technologies. As a consequence, different technologoies and topologies can be chosen at deploy time.

[3]I contributed this code to the Mule project; it is available in the most recent (development) release.

7 Glossary

Adapter — A pattern from GoF[3]. A class that provides a new interface to a delegate.

Bean — A POJO with an empty constructor. It can be instantiated by an external framework (typically Spring[1]) and used to construct a larger system via *Dependency Injection*.

Business Bean — A *Bean* that implements the business logic for a service. It calls other services 'directly'; it receives service instances by *Dependency Injection* at deploy time.

Client Adapter — An *Adapter* that implements the *Service Interface*, and can be injected into a *Business Bean* as a service, but which transfers responsibility for providing the service to a *Mediator* (typically an *ESB*).

Dependency Injection — (aka. Inversion of Control) *Beans* are 'assembled' when the system is deployed by a framework (typically Spring[1]) that instantiates each object and then calls 'setter' methods, passing related services as method arguments.

ESB — Enterprise Service Bus. A standards–bsaed messaging system that enables the integration of business services.

Mediator Interface — The interface for a service, exposed via a particular mediator (messaging) technology. For SOAP this may be described by a WSDL; for Mule it is a Java interface whose methods match those of the Service interface, but which receives a single request message as an argument and returns a single response message as a result.

Mediator — A pattern from GoF[3]. A class that mediates communication between other classes (ie. other services).

POJO — Plain Old Java Object. An instance of a Java class with no dependency on the supporting technologies used in the larger system.

Server Adapter — An *Adapter* that implements a *Message Interface* and delegates requests to a service implementation (typically a *Service Bean*) received via *Dependency Injection*.

Service Interface — The Java interface that a service implements, expressed in a normal, idiomatic style (multiple parameters, exceptions, return values).

8 Acknowledgements

This paper depends heavily on techniques I learnt and developed while working on the NOAO DMaSS project. As such I am indebted to the contributions made by the entire DMaSS team: Sonya Lowry (Technical Lead); Evan Deaubl, Alvaro Egaña, and Phil Warner (Engineers, alphabetical order).

References

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

[2] NOAO Data Management and Science Support Solutions Platform
http://chive.tuc.noao.edu/noaodpp/DMaSS/

[3] Design Patterns: Elements of Reusable Object-Oriented Software
E. Gamma et al.

DRAFT