CTIO / DPPS / Archive — 2006

Andrew Cooke[1] Alvaro Egaña[2]

September 2006

[1]andrew@acooke.org
[2]alvaro@egana.net

Abstract

This is a collection of papers that illustrate our work over the last year.

The first — *Spring, Mule, Maven: Lightweight SOA with Java* — is a 'recipe' for implementing services. Our aim is flexible, adaptable software, exploiting third party solutions only when we understand why and how they will be used, keeping external interfaces separate from our core 'business logic'. It's not that we want to re–invent every wheel, but we are tired of being bound to wheels that we don't understand, rolling in directions we didn't choose, at speeds we cannot control[1].

Next comes a draft of our ADASS paper — *FITS Files and Regular Grammars: A DMaSS Design Case Study* — largely an exercise in stating the obvious. Which is not necessarily a bad thing; sometimes it's worth taking the time to understand what is intuitive. For us, this paper is important because we are trying to find a balance between the 'craft' of programming and its theory.

A Tiny Workflow in Spring and *A Simple, Lazy, Expression Evaluator* are shorter papers, describing two small libraries we created to help simplify our work[2]. The workflow helps structure our services; this description should throw some light on the ideas in the first paper. The evaluator was a 'bit of fun', thrown together in a couple of days; the ease with which something like that can be implemented in (recent) Java is worth noting.

Working in Chile, for a department still struggling to position itself within the VO, we sometimes feel that both our colleagues and the community as a whole have little idea of what we do. This is one small attempt to change that. It was written (mostly) in our own time which, we felt, gave us the liberty to express our own views, even when they differ from those of our co–workers.

Andrew & Alvaro

P.S. If there's a common style or attitude across these papers, and if it strikes a chord with anyone else, and if you are that someone else, and if, in addition, you have some idea of how we could build a community around it, or have ideas or experiences[3] you'd like to share, or, even, a couple of jobs you'd like to offer, then, please, drop us a line. Cheers.

[1] We try to avoid metaphors like this in the paper itself.

[2] Please don't think that's all the code we've written this last year!

[3] Or advice on proper use of the comma.

Contents

1 Spring, Mule, Maven: Lightweight SOA with Java **5**
1.1 Introduction . . . 5
1.1.1 Aims . . . 5
1.1.2 Scope . . . 5
1.1.3 Our Background . . . 5
1.1.4 Road–map . . . 5
1.2 Interfaces . . . 6
1.2.1 Introduction . . . 6
1.2.2 Services . . . 6
1.2.3 Communication . . . 6
1.2.4 Testing . . . 7
1.3 Technologies . . . 8
1.3.1 Java . . . 8
1.3.2 Spring . . . 8
1.3.3 Mule . . . 8
1.3.4 Maven . . . 8
1.4 Example . . . 10
1.4.1 Recapitulation . . . 10
1.4.2 Naming Service . . . 10
1.4.3 Structure . . . 10
1.4.4 Commentary . . . 10
1.4.5 Deployment . . . 11
1.4.6 Pattern: Nesting Facades . . . 12
1.5 Conclusions . . . 12
1.5.1 Recipe . . . 12
1.5.2 Simplicity . . . 13
1.5.3 Future Directions . . . 13
1.5.4 Philosophical Purity . . . 13
1.6 Acknowledgements . . . 13

2 FITS Files and Regular Grammars: A DMaSS Design Case Study **15**
2.1 The Problem . . . 15
2.1.1 FITS Headers . . . 15
2.1.2 NOAO DMaSS . . . 15
2.1.3 Vocabularies . . . 15
2.1.4 Types and Values . . . 16
2.2 The Analysis . . . 16
2.2.1 Formal Languages . . . 16
2.2.2 Simple Model . . . 16
2.2.3 Reintroducing Values . . . 16
2.2.4 Graph Structure . . . 17
2.2.5 Semantics, Ontology . . . 17

2.3 The Solution . . . 17
2.3.1 Recognition Trees . . . 17
2.3.2 Reality Check: Workflow . . . 18
2.3.3 Categories, Orthogonal Vocabularies . . . 18
2.3.4 Separation of Concerns . . . 18
2.3.5 Implementation . . . 18
2.3.6 Reality Check: Field Map . . . 20
2.4 Conclusions . . . 20
2.4.1 Vocabularies . . . 20
2.4.2 Loading and Remediation . . . 20
2.4.3 Duplicate Data . . . 20
2.4.4 Theory . . . 20
2.4.5 Problems . . . 21
2.5 Addendum . . . 21
2.5.1 Non-Unique Fields . . . 21
2.5.2 Hierarchical Headers . . . 21

3 A Tiny Workflow in Spring 23
3.1 Motivation . . . 23
3.2 Design . . . 23
3.3 Examples . . . 23
3.3.1 Pseudo–Code . . . 23
3.3.2 DB Loader . . . 25
3.4 Experience . . . 26
3.4.1 XML Tests . . . 26
3.4.2 Verbosity . . . 26
3.4.3 Lack of Features . . . 26
3.4.4 Persistent State . . . 26
3.4.5 Pluggability . . . 26

4 A Simple, Lazy, Expression Evaluator 29
4.1 The Problem . . . 29
4.1.1 The Remediation Service . . . 29
4.1.2 Remediation Actions . . . 29
4.1.3 Configuration via Programs . . . 29
4.2 The Solution . . . 30
4.2.1 Domain Specific Languages . . . 30
4.2.2 Design Choices . . . 30
4.2.3 Parsing . . . 30
4.2.4 Evaluation . . . 30
4.2.5 Generics, Reflection . . . 31
4.2.6 Examples . . . 31
4.3 Conclusions . . . 33
4.3.1 Experience . . . 33
4.3.2 Optimisation . . . 33
4.3.3 Philosophy . . . 34

A CTIO / DPPS / Archive Is... 35
A.1 Andrew Cooke . . . 35
A.1.1 Contact . . . 35
A.1.2 Interests . . . 35
A.2 Alvaro Egaña . . . 35
A.2.1 Contact . . . 35
A.2.2 Interests . . . 35

Chapter 1

Spring, Mule, Maven: Lightweight SOA with Java

Authors. Andrew Cooke, Alvaro Egaña.

Abstract. We show how Spring and Mule 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.

1.1 Introduction

1.1.1 Aims

This paper introduces an 'implementation recipe', summarised in section 1.5.1. 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 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.

1.1.2 Scope

Our ideas rely on the language having a (static) type system. In section 1.3 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#.

1.1.3 Our Background

We have been working with Mule and Spring 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[3].

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.

1.1.4 Road–map

We present the basic ideas behind our approach in section 1.2; section 1.2.1 motivates all the subse-

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

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

[3] 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.

quent work. In section 1.3 we explain how certain technologies can simplify the approach. Section 1.4 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 1.5, where we also discuss future work.

1.2 Interfaces

1.2.1 Introduction

'Programming to an interface' is a standard idiom within the Java community; interfaces[4] 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[1] and adopted by EJB3[2].

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.

1.2.2 Services

Verification requires that each service 'know' the interface provided by other services; but we must avoid 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'[5] that depends only on basic Java classes and stand–alone libraries[6]. Service business implementations then depend only on business interfaces to services (their own, and any services they depend on).

Maven (section 1.3.4) 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.

1.2.3 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.

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.

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.

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

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

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

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

So, in summary, each service should have a 'communications' package that provides client–server functionality. 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.

Note that the service's communications package does not depend on the business implementation during compilation (or vice–versa, of course).

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[7] 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.

Common Messaging 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.

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.

[7]The user of 'server' may be misleading here. The responsibility for listening to ports, queues, etc, is left to whatever messaging solution is used (in section 1.3.3 we recommend Mule). Our 'server' receives a 'message' when a method that takes a serialisable Java object is called.

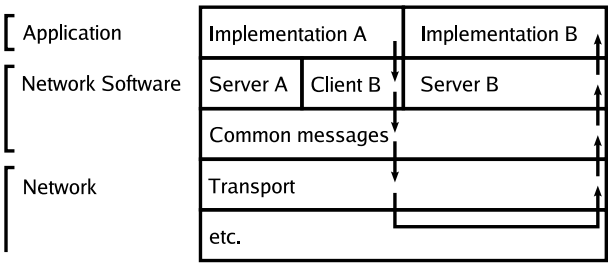

Figure 1.1: The communications stack: service A sends a message to service B.

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[8]. 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.

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 1.1. 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.

1.2.4 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.

[8]In a later paper we will address the automatic generation of the entire communications package.

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 implementations of sub–components can be written and injected as needed.

1.3 Technologies

1.3.1 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[3], which compiles to the JVM (Java Virtual Machine) and so can interoperate with Java–based tools.

1.3.2 Spring

Spring 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'.

1.3.3 Mule

Mule[4] 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[5].

1.3.4 Maven

We use Maven v2[6] 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.

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).

[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.

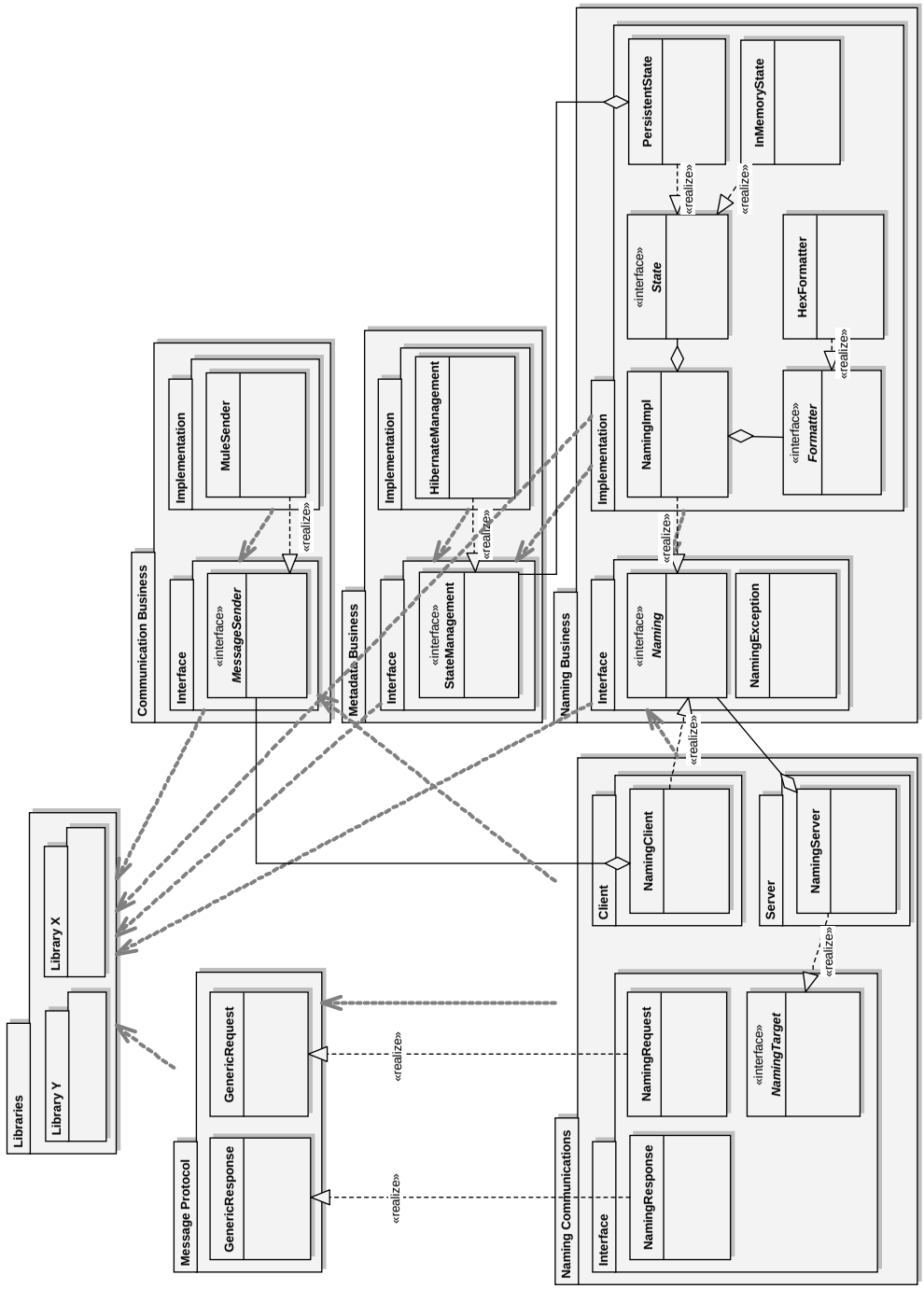

Figure 1.2: Naming Service structure (with related packages, including messaging and a possible dependency on Metadata Service to persist state). Dashed, grey, thick lines are compile–time dependencies.

- 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.

1.4 Example

1.4.1 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 a separate package (whose components might be separated) containing:

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.

1.4.2 Naming Service

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

1.4.3 Structure

The Naming Service and its main dependencies are sketched in figure 1.2.

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: `Source` 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 a `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 interfaces* package contains two serialisable message objects: `NamingRequest` is empty; `NamingResponse` contains either a `NamingException` or a `String` (the name). The *communications client* takes an interface, `MessageSender` to which it sends the request and receives a response. The *communications server* takes a `Naming` instance which it calls when it receives the request.

1.4.4 Commentary

Figure 1.2 illustrates how our approach leads to a system with the following properties:

- Anything can depend on a library.

- Anything except a library, or the Communications Service and Message Protocol Library, can depend on any other service's interface.

- Nothing can depend on a service's business implementation package.

- Nothing can depend on a service's communication package.

- Only the communication packages depend on the packages related to the Communications Service.

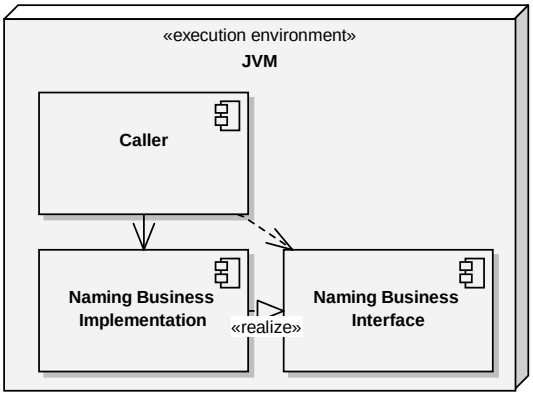

Figure 1.3: Local deploy of the Naming Service showing call and dependency.

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 communications package bundles together interface (including associated messages), client, and server. Both client and server code are deployed when only one is required; the client and server are both very simple, 'thin' classes, but they could be separated if required.

It is important to understand that *server* and *service* are not synonyms. The *service* is the logical core; the *server* is a practical detail necessary for the messaging technology we have chosen. Other technologies might (but not necessarily; the code is reasonably general) require different servers, but the service itself would remain unchanged.

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.

1.4.5 Deployment

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

- As part of a composite service, running within the same JVM as the caller.
- As a remote server, called from Java.

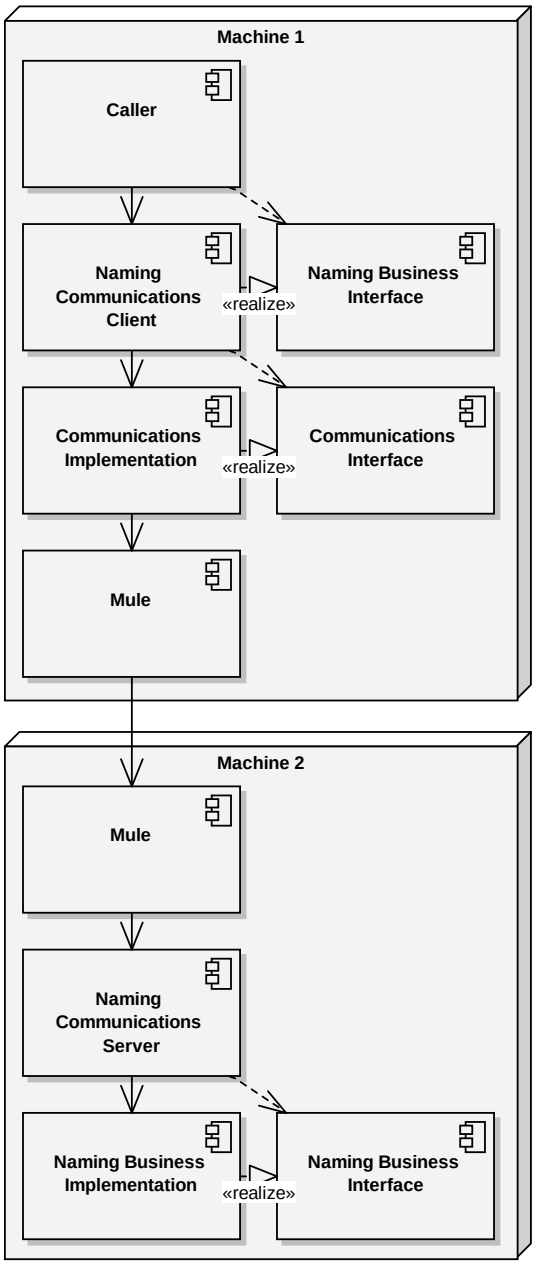

Figure 1.4: Remote deploy of the Naming Service showing remote call and dependencies.

- 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.

Composite Service in JVM. For the Java–based examples here, the user code is always compiled against the Business Interface Package.

For a composite service, the caller will expect an implementation of `Naming`, which is injected. The business interface and implementation packages are used. This is shown in figure 1.3.

Remote Server, Java. The same user code as in the previous section can be configured to call a remote service by injecting the `NamingClient`. This will, in turn, require a suitable `MessageSender` implementation (typically `MuleSender`) configured with the correct Mule endpoint. Mule must be configured to route the endpoint correctly to the server.

The Naming Service server process is a `NamingServer` (from the communications Package) which is injected with a `Naming` implementation. See figure 1.4.

Note that the Naming business implementation package is not needed on the caller machine.

Remote Server, Non–Java. In the previous section the Naming service server process was called by Java code via `NamingClient`. 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 server (more exactly, the `NamingTarget` interface) using any messaging implementation that inter–operates with Mule.

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[10].

[10]This extension has been submitted to Mule and, hopefully, will be included in the next official release.

1.4.6 Pattern: Nesting Facades

The variety of deployment options described above is possible because several different implementations of the same interface are available, most of which can be injected with a further instance *of the same interface*. Injection gives two advantages: first, compilation depends only on interfaces (no circular dependencies); second, deployment decisions need not appear in the source code (the same jars can be deployed in different scenarios without recompiling).

A surprising amount of functionality can be composed in this way, especially if the interface is simple. It is important to note that each facade implementation can have a more complex interface, which can be configured by Swing (so various parameters can be 'fine–tuned' during deploy); only the common service interface needs to be kept simple.

1.5 Conclusions

1.5.1 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.
- Place communications classes in a new package, with separate client and server functionality.
- Make the communications client implement the same interface as the basic service.
- 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 the messaging server directly to handle 'external' connections.
- Test deployed services (with messaging) as much as possible.
- Java, Spring, Mule and Maven make this approach easy.

1.5.2 Simplicity

The implementation package contains the business logic — and nothing little 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.

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. Mule adds 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.

1.5.3 Future Directions

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

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[11] eventually receive a reply; if A has restarted in the meantime, however, it will not contain the correct state to process the response.

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. This could be added to our Tiny workflow[7], but it would probably make more sense to move to a more complex third party workflow / framework.

This conclusion — that we may need to move to a more sophisticated technology in the future — does not alarm us. We do not claim to have solved every problem. More importantly, the approach described here is so simple, and commits so little to any one technology, that the transition should be easy[12], particularly if existing services use the Tiny workflow[7] to keep their structure simple.

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 (our 'client' matches 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)[8] uses SOAP and is supported by recent versions of Mule. This is another possible route for future development.

1.5.4 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.

1.6 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

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

[12] With the possible introduction of continuations in the next release of Java we may find a more lightweight solution.

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.

Chapter 2

FITS Files and Regular Grammars: A DMaSS Design Case Study

Authors. Andrew Cooke, Alvaro Egaña, Sonya Lowry.

Abstract. The NOAO DMaSS (Data Management and Science Solutions) Platform has already passed through several development iterations and includes a Database (DB) Loader service. Experience using that service has been mixed.

Recent discussion of the proposed Metadata service focused on 'vocabularies'. We realised that we could treat FITS headers as a formal language and that this language was described by a regular grammar. It then became clear that part of the DB Loader's configuration was, in fact, a regular expression (expressed in several files of rather verbose Spring XML!) Furthermore, a separate (and equally confusing) configuration was now clearly related. And we could extend things further: the same data structures could help detect 'almost duplicate' data reliably.

Reviewing our results in terms of the system architecture, we suggest moving newly identified responsibilities to appropriate services. The end result is a cleaner design, better integrated within our architecture, and in which we have much more confidence. What started as an idle theoretical curiousity yielded very practical results.

2.1 The Problem

2.1.1 FITS Headers

Scientific data often consist of two components: a collection of 'measurements' ('science data'; e.g. an image) and a 'description' of those values ('science metadata').

In the most common file format, FITS (Flexible Image Transport System[9]), the science metadata are stored as name/value pairs (grouped in 'headers'). Apart from multi–extension (MEF) files (which we do not believe significantly complicate the problem, and which we will ignore here), the metadata have very little structure: the names form a simple set, with no repetition or nesting, and almost no ordering.

2.1.2 NOAO DMaSS

The NOAO DMaSS[10] is a set of services that can be deployed as an archive for astronomical data. Early iterations already include a DB Loader service, responsible for loading science metadata into an SQL database.

Experience with the DB Loader has identified problems with the configuration, which is distributed across several different sets of files: a workflow[7], which identifies particular data products (for example: images observed on a certain camera/telescope); a mapping of metadata names to database fields; and a description of the database. Only the last of these is generated automatically, the others must be carefully constructed by hand[1].

2.1.3 Vocabularies

Recent discussion within NOAO has focused on the concept of 'vocabularies' — the *types* of science metadata expected for different data products. NOAO manages many instruments/telescopes, and

[1]This service currently loads database fields (SQL table columns) directly. Changing the system to bind values in a set of objects, managed by the Metadata service, is anticipated in the design, but would not simplify the problem addressed here.

the DMaSS is intended as a general system, so we must be able to manage a wide variety of vocabularies.

Since science data enter the system as FITS files the range of vocabularies reflects the range of headers we must handle. Further complexity is added by the evolution of vocabularies over time and modifications/additions to the metadata by various data–handling systems.

2.1.4 Types and Values

Faced with a wide range of vocabularies, we needed a way to structure our problem. One source of complexity was the interplay between types and values: the set of names that a FITS header might contain depended on the values associated with some of the names.

While this is completely obvious to anyone who has worked with FITS data, it is easy to see how it can become difficult to manage in a software system. The same dependency within a naive OO design would give objects whose class changes depending on the values contained. While this is possible with predicate classes[11] it cannot be expressed directly in a language with a fairly simple, static type system like Java or C++[2].

On the other hand, this didn't seem like a 'hard' problem. After all, lots of people write programs that manage FITS data! We were tempted to ignore the issue and simply 'do the work', but then stumbled across an argument that led, in a sequence of simple steps, to a solution that combines decent engineering with a consistent logical framework.

2.2 The Analysis

2.2.1 Formal Languages

In the following sections we use a few ideas from Formal Languages[12], but we do not rigorously prove any results; our aim is to use the theory to help illuminate a practical design. While we do not expect 'real life' to behave in such an idealised manner, we do hope that the argument will help us understand how best to deal with exceptions and irregularities.

This theory is not particularly obscure; it is probably familiar as the basis for parsing, regular expressions, etc. We will use the following ideas: that a *language*[3] is composed of an *alphabet* (a list of *words*) and a *grammar*; that *grammars* define how *words* are combined to produce valid *sentences*; that *grammars* can be categorised in various ways; and that perhaps the simplest and most common of these is a *regular grammar*, whose *sentences* can be described by regular expressions.

2.2.2 Simple Model

The metadata in a FITS header are, for the most part, unordered. We start by imposing an arbitrary order (e.g. sorting their names alphabetically).

To further simplify our initial model we will associate each metadata name with a different letter and ignore the associated values. This gives us our *alphabet*. Valid headers, sorted and represented in this manner, form *sentences*.

With these assumptions, example FITS headers might be `abcde`, `abpqrs`, or `xyz`.

We can match those examples using the regular expression `(ab(cde|pqrs)|xyz)`. Since we could write a similar expression for any header (*sentence*) in this model we have shown that they are described by a *regular grammar*.

This result is not very surprising. The next level of complexity is a *context free grammar*. These are characterised by nested structures, but FITS headers are not that complex (they lack any idea of 'namespaces' for example).

2.2.3 Reintroducing Values

We assumed that the we could represent metadata as letters. This is reasonable if we do not care about the value. However, there are at least two cases which require more care.

First, as mentioned in section 2.1.4, there are some metadata whose values affect the choice of vocabulary. For example, the instrument type. In such cases we must associate a different *word* in the *alphabet* with each distinct option. If originally we associated the name `INSTR` with the *word* `a` we must now introduce distinct *words* for each type of instrument. `INSTR='MOSAIC'` might be mapped to $\mathtt{a}_1$, `INSTR='ODI'` to $\mathtt{a}_2$, etc.

Second, we might want to validate the values. It is trivial to extend the argument to include regular expressions for each value, but perhaps validation requires a more complex function. Technically

[2] This mixing of types and values also appears to be related to Russell's paradox in set theory.

[3] We use *italicised* text to indicate that the word is being used in a formal manner, as the meanings do not coincide exactly with common use.

a *grammar* is *generative* — it can be used to construct valid sentences — but in our case we are using the *grammar* only to recognise (not construct) valid sentences. In such cases, we can use an arbitrary predicate as a *word*, providing we do not use any information from 'outside' the value in question.

The second case may also apply to the first. There might be some metadata that affected the choice of vocabulary via a complex function. Provided this function takes only the single metadata value as an argument we can extend our *alphabet* with a suitable predicate[4].

2.2.4 Graph Structure

The regular expression `(ab(cde|pqrs)|xyz)` from section 2.2.2 was constructed by grouping common prefixes. Matching a FITS header against the expression involves identifying a path through a tree, starting at the root (leftmost node) and ending at one of the leaves (`e`, `s` or `z`). We can consider the regular expression as a 'decision tree' whose leaf nodes classify FITS headers.

This tree–like structure is intuitive and easy to represent in software. And the leaf nodes suggest a process for identifying vocabularies. In general, however, regular expressions can be more complex — they are equivalent to finite state machines. So we would like to understand whether this tree structure is (1) sufficient and (2) practical.

A state machine can be represented as a directed graph (DG) whose edges are transitions from one state to another. Each edge is associated with matching a *word* in the *language*. A DG may contain cycles, but we have already noted that *words* in our *sentences* do not repeat (every name in a FITS header is unique; there is a single namespace). So cycles (repetition) do not seem to be necessary.

Without cycles, we have a directed acyclic graph (DAG). This is still more general than a tree; the branches of a tree cannot 're–join'. How significant is this lack of expressivity for trees? For finite length sentences we can enumerate all possible paths through a graph, giving a tree. So for any DG we can always construct a tree–like regular expression that identifies the *language* (at the expense of duplication of information). Since DAGs are a subclass of DGs the argument also applies there. A tree is sufficient, but not optimal.

We will return to this later when we discuss 'orthogonal vocabularies'.

[4]In extreme cases we might even group several metadata together and treat them as a single value (e.g. the WCS).

2.2.5 Semantics, Ontology

Our initial analysis assumed (section 2.2.2) an alphabetical ordering of names. Optional metadata names near the start of the alphabet will introduce structure in our tree–like regular expression (they will add unnecessary branching near the root). Our *grammar* is very sensitive to 'noise' in the FITS header.

We have not relied on the type of ordering in any of our arguments above and are therefore free to choose one to address this problem. We will impose an order based on our intuition about 'importance'. The introduction of a semantic ordering will give a more compact representation[5].

This is (again) an argument that sounds opaque within our semi–formal approach, but which is 'obvious' in practice. We are suggesting that the regular expression should start with metadata common to all valid FITS files[6]. Next should be 'important' fields (`INSTR`, for example, and then perhaps `OBSTYPE`) that identify different 'types' of FITS files.

In this way, our decision tree reflects the meaning within the metadata; the leaf nodes classify FITS headers within some ontology.

2.3 The Solution

2.3.1 Recognition Trees

We call the semantic decision tree described above a 'recognition tree'. Recognition trees are a tool to organise our knowledge of the structure of FITS headers and classify science metadata.

The application of a recognition trees to a FITS header mirrors the use of a regular expression to match a *sentence*. Starting at the tree root we repeatedly select a child node based on matching the science metadata. Eventually we either fail to match — the FITS header is malformed or we have an incomplete tree — or we arrive at a leaf node.

Each edge in the tree corresponds to a predicate that tests science metadata. Each predicate specifies a single name and an optional test on the associated value. The test need not be a regular expression, but must base its decision only on the value given.

The structure of a recognition tree is influenced by our knowledge of the semantics associated with each predicate. This allows us to associate leaf nodes with ontological concepts (e.g. to classify the FITS file as a certain data product).

[5]This is, in some sense, the definition of a 'useful' ontology.

[6]Later (section 2.3.4) we will restrict the use of this tree to identifying vocabularies; metadata common to all can then be discarded.

2.3.2 Reality Check: Workflow

The workflow in the current DB Loader is, in retrospect, a simple recognition tree. The first edges are tests on the `INSTR` metadata value and the leaf nodes identify different data products (Mosaic sky flats, generic bias frames, etc).

This workflow is a hand–written, somewhat ad–hoc, Spring XML configuration. Our analysis shows that we can safely (ie without risk of using too restrictive a representation) move this definition to a tree–like structure in the database. The emphasis will change from 'DB Loader configuration' to a more specific 'Vocabulary Service'.

2.3.3 Categories, Orthogonal Vocabularies

The introduction of a separate service, any associated improvement in the user interface, and a better understanding of the role of this information, will all lead to a more detailed recognition tree with a finer granularity in the ontology. Rather than a simple 'Mosaic sky flat', we will have categories like 'Mosaic sky flat using WCS projection XXX with additional information from v1.2 of the Kitt Peak Data Handling System'.

For many applications this is 'too much information'. We must provide broader classes that group related leaf nodes. A practical system will probably support many different, overlapping categories. These different categories will probably correspond to different parts of our data model.

This fine level of detail in the leaf nodes is also driven partly by the use of a tree. We noted in section 2.2.4 that a DAG allowed branches to re–join. The structure we have chosen leads to a proliferation of branches.

An alternative solution to this problem might be to separate orthogonal issues into separate trees. For example, consider WCS–related header properties. We may want to detect different WCS representations, even though these variations have little bearing on the classification into different data products. This could be separated into a second, WCS–specific decision tree.

This separation appears to be possible even when there is some overlap of concerns. For example, we might include WCS coordinate type in both the WCS vocabulary and the science data product vocabulary (where it would be checked for consistency with imaging or spectral data).

Or perhaps all three approaches (categories, orthogonal trees and DAGs) are complementary. This is still an open problem — we will probably implement a single tree with categories initially and then revisit the issue in a later development iteration.

2.3.4 Separation of Concerns

There is a clear parallel between the Vocabulary service and a parser. Both use grammars to recognise and validate data. A parser also returns values. This raises the questions 'how much validation should the Vocabulary service do?' and 'to what extent should the Vocabulary service be responsible for extracting values from data?'

To answer these questions we need to consider the Vocabulary service within the context of the system architecture. Our architectural definition is:

A Vocabulary is a translation between the system metadata (as surfaced by the appropriate service) and an external syntax. The translation may be in either or both directions.

It is easy to imagine examples (e.g. when the external syntax is conversational English, and the vocabulary service is used to provide 'helpful names' for system metadata) where the Vocabulary service is little more than a table that associates the appropriate syntax with system metadata fields.

So a possible separation of concerns would be to use the recognition tree to select a vocabulary[7] for a given FITS file and then provide translation through appropriate cross–references.

These 'translations' are a natural point at which to store additional functionality that is specific to a metadata value: validation instructions; Java type used for encapsulation, etc.

The Vocabulary service is then used in two stages. First, the relevant vocabulary is recognised. Second, the appropriate vocabulary is provided, with appropriate translation information provided for each element in the data model[8]

This reduces the amount of detail needed in the recognition tree — only properties at branch nodes are required — and will reduce the complexity discussed in the previous section.

2.3.5 Implementation

Figure 2.1 sketches a possible implementation of the ideas described above. The recognition tree is on

[7] We group vocabularies that are related by a single recogniser within a 'vocabulary family'.

[8] One limitation of this analysis is that it only handles data with a name/value structure; however, this is itself a broad class.

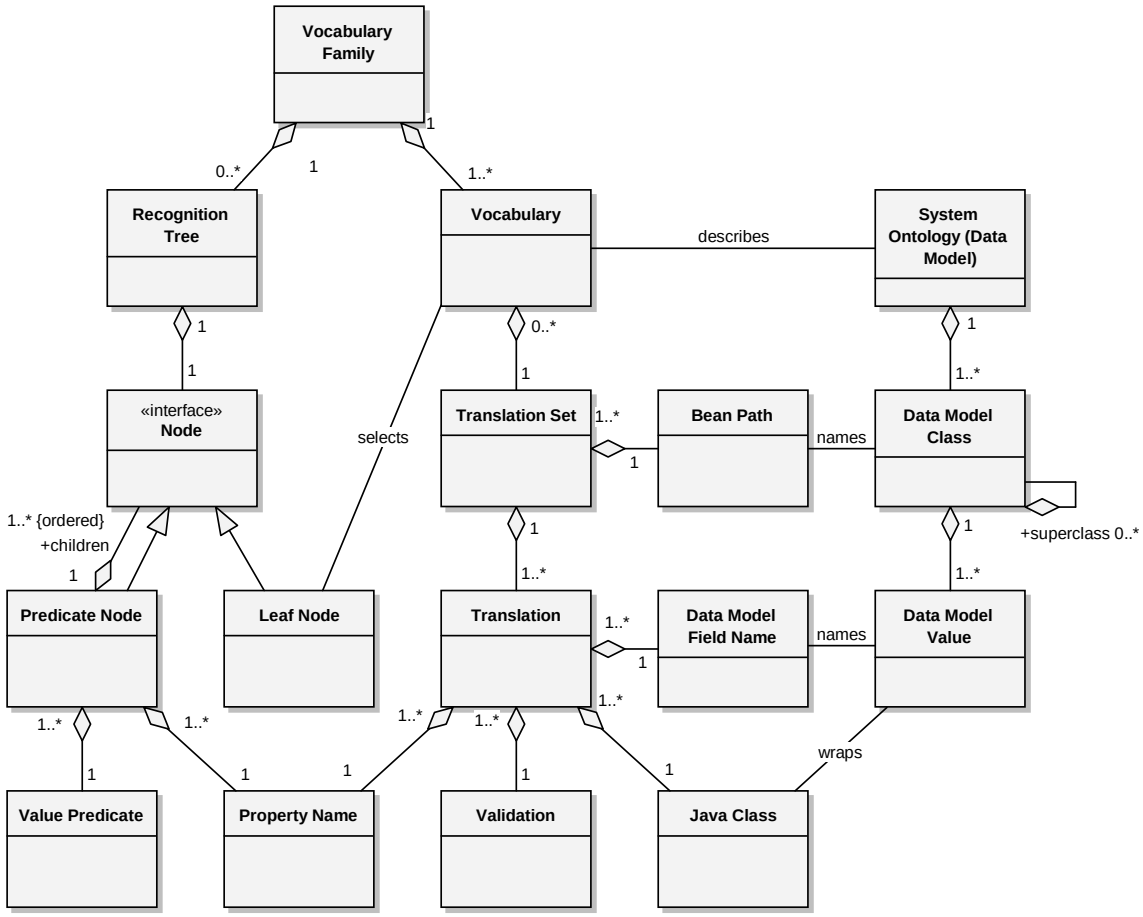

Figure 2.1: A possible implementation of the ideas discussed here. Details of the data model are intentionally vague.

```
HIERARCH ESO DET WIN1 STRX =   3 / Lower left pixel in X
HIERARCH ESO INS FILT1 NAME = 'OIII/3000' / Filter name
HIERARCH ESO OBS NAME =  'NGC 1275' / Observation block name
```

Figure 2.2: Example ESO "hierarchical" headers.

the left; the vocabulary in the middle; and the data model (part of the system metadata) on the right.

A Vocabulary is composed of 'translation sets' that correspond to classes in the data model; a bean path[9] identifies the appropriate class and, together with the field name for a particular translation, would allow dynamic access to values.

This solution will probably evolve further. For example, it is probably incorrect to assume that there is always a 1–to–1 connection between the metadata items in the external syntax and those in the internal model.

2.3.6 Reality Check: Field Map

The 'translations' correspond to the mapping from science metadata to database (or object) fields that is configured in the DB Loader (section 2.1.2).

Our experience with the DB Loader suggests that many translation sets will be shared between several vocabularies.

2.4 Conclusions

2.4.1 Vocabularies

We started with very little understanding of how to apply the concept of vocabularies to FITS header properties. We now see that each distinct combination of 'interesting' metadata names identifies a separate vocabulary. Since FITS files vary widely we have many vocabularies and need an efficient approach to managing them. This work separated into two related steps.

First, we will use a recognition tree to identify the vocabulary in question. In practice we may split vocabularies into several orthogonal (sub-)vocabularies; we can use separate recognition trees for each, or we can use a single tree whose leaf nodes are categorised by vocabulary.

Second, we structure vocabularies to match our data model. For each field in the data model, we store appropriate information in a 'translation'. These are grouped in 'translation sets' that mirror the classes in the data model.

2.4.2 Loading and Remediation

In retrospect, the code we developed in an earlier iteration to remediate data and load the database, already contained the structures we describe above. However, we had a poor understanding of the logic that fixes the underlying relationships.

For example, we constructed (the equivalent information to) vocabularies with a text–based configuration that 'imported' related files. The import mechanism was general and poorly constrained. We now understand that by restricting the configuration to reflect the structure in the data model (and moving this information into the database) we can make the relationship between vocabulary and data model explicit, avoid inconsistencies, and simplify the management of these relationships.

2.4.3 Duplicate Data

An interesting additional use for the ideas identified above is in the detection of 'practically duplicate' data: we would like to be able to identify duplicate copies of the 'same' data, even when some header values change. To do this reliably we must distinguish between metadata which are meaningful, and those which are 'noise' (e.g. time of entry into a data transport system).

One solution to this would problem would be to mark certain translations as 'meaningful' and use only the science metadata that they select. Furthermore, one way of defining 'meaningful' is to use metadata names that appear in the recognition tree.

2.4.4 Theory

We are interested in how best to use ideas from 'theoretical' computer science in practical programming[10]. While we see many interesting ideas we are also acutely aware that we must avoid self–indulgent hand–waving that gives no practical results. Since we feel that this analysis was productive we are somewhat encouraged to try a similar approach in the future.

One reason for the success here, we believe, was an iterative approach to development. Earlier code helped identify problems and motivate our work. It was also reassuring to find concrete support for our new ideas — albeit in a rather rough form — already implemented in our software.

[9] This discussion assumes that the data model is instantiated as JavaBeans[13]. A bean path is a dotted sequence of names that select a path through linked objects (typically translated to a sequence of calls to 'getter' methods).

[10] As is probably obvious, we are somewhat sensitive about our approach, feeling trapped between theoreticians, who will see very little theory here, and those programmers who believe every problem can be solved by using someone else's toolkit.

2.4.5 Problems

A further advantage of this analysis is that we can identify possible problems by checking our assumptions against reality. One obvious difficulty is the handling of FITS properties which are 'sequential fragments'. This occurs with WCS[9] encodings, for example, which exceed the maximum length for a single header field[11]. The solution in this case is probably a pre–processing step that generates a single, concatenated value when parsing incoming metadata (and a corresponding fragmentation step when generating a header).

2.5 Addendum

This section was added after my ADASS talk in October 2006. It addresses questions raised by the audience.

2.5.1 Non-Unique Fields

Rob Seaman pointed out that some files contain duplicate fields; it seems that these are badly formed files from non-compliant software/bugs. This raises an issue I did not address — verification.

The information stored in the vocabulary service can help do three things: identify headers; verify headers; extract information from headers. Only the first of these is covered in any detail here. Rob's example shows that verification should occur before identification (and implies that some remediation may also be necessary, if these files are to be ingested).

2.5.2 Hierarchical Headers

Dick Shaw raised the (non–standard) use by ESO of "Hierarchical Keywords". The ESO documentation[14] gives the example shown in figure 2.2.

While it is true that a regular grammar cannot handle indefinite hierarchies, the scheme chosen by ESO is sufficiently restricted that it would work within the framework described here. It is only necessary to treat the whole name (including spaces) as a single string.

[11] Based on punched cards

Chapter 3

A Tiny Workflow in Spring

Author. Andrew Cooke.

Abstract. I define a simple workflow within Spring, give some examples, and discuss its use. While is works well for prototyping and structuring simple flow–based processes, it doesn't scale well if the control becomes complex.

3.1 Motivation

A workflow is a software tool that separates 'control', 'state' and 'process'. A workflow engine is often a 'composition language' used to express the high level functionality of a system.

There are many complex, powerful workflow engines available, but their associated cost makes them attractive only at the higher levels of a system. Since I have just defined workflows in terms of 'high level functionality' this might seem reasonable.

However, 'high level' is a relative judgement (and the definition of 'business logic' is more flexible than many people pretend). I felt a lightweight workflow would be a useful tool that could help clarify the structure of many systems[1].

3.2 Design

Figure 3.1 is a class diagram for the Tiny Workflow library.

The design exploits the 'nesting facade' pattern[18]: a workflow contains a `Flow`; many implementation of that interface can contain further embedded `Flow` instances.

The most abstract view of a workflow presents two separate ideas: decisions and processes. These suggest `Test` and `Action` interfaces. To reconcile this with the pattern, I make `Flow` inherit from both and give appropriate semantics: the `Action` in a `Flow` is called if the `Test` returns true[2].

A useful practical class is `Pair` which binds a `Test` with an `Action`. I can then develop separate decisions and process, combining them with `Pair` as needed. Alternatively, instead of aggregation, I can use inheritance: `Null` is a `Flow` that can be subclassed as required (by default the test is `Always` true and the `Action` does nothing).

Implicit in the discussion so far is the idea of `State`. This single interface is used both as the basis for decisions, and as the data to be processed. This is not an interface, but a generic parameter to almost all classes. In this way, the workflow is type–safe, but not intrusive. `Test` interface's main method to take a `State` and returns a `boolean`; `Action` takes and returns `State`.

I can also define `Test` instances that are injected with other tests (nesting facades again). The library contains the obvious `Or` and `And`, as well as `Always` and `Never`.

The library also contains a set of composite `Flow` instances. `Sequence` takes a list of `Flow` instances and calls them in turn. `First` also takes a list, but stops running through the contents as soon as a `Flow` whose `Test` returns true is found (and the corresponding `Action` invoked).

Finally, the workflow includes some support for converting incoming data to the internal `State` representation: `Normalizer` and `InputFilter`.

3.3 Examples

3.3.1 Pseudo–Code

Figures 3.2 and 3.3 show the same ideas expresed in a Java–like syntax and as Tiny workflow compo-

[1] Ad-hoc 'workflow–like' classes emerge naturally in Spring–based systems. All I do here is provide a little more structure.

[2] Assuming that the workflow has 'reached' the flow in question.

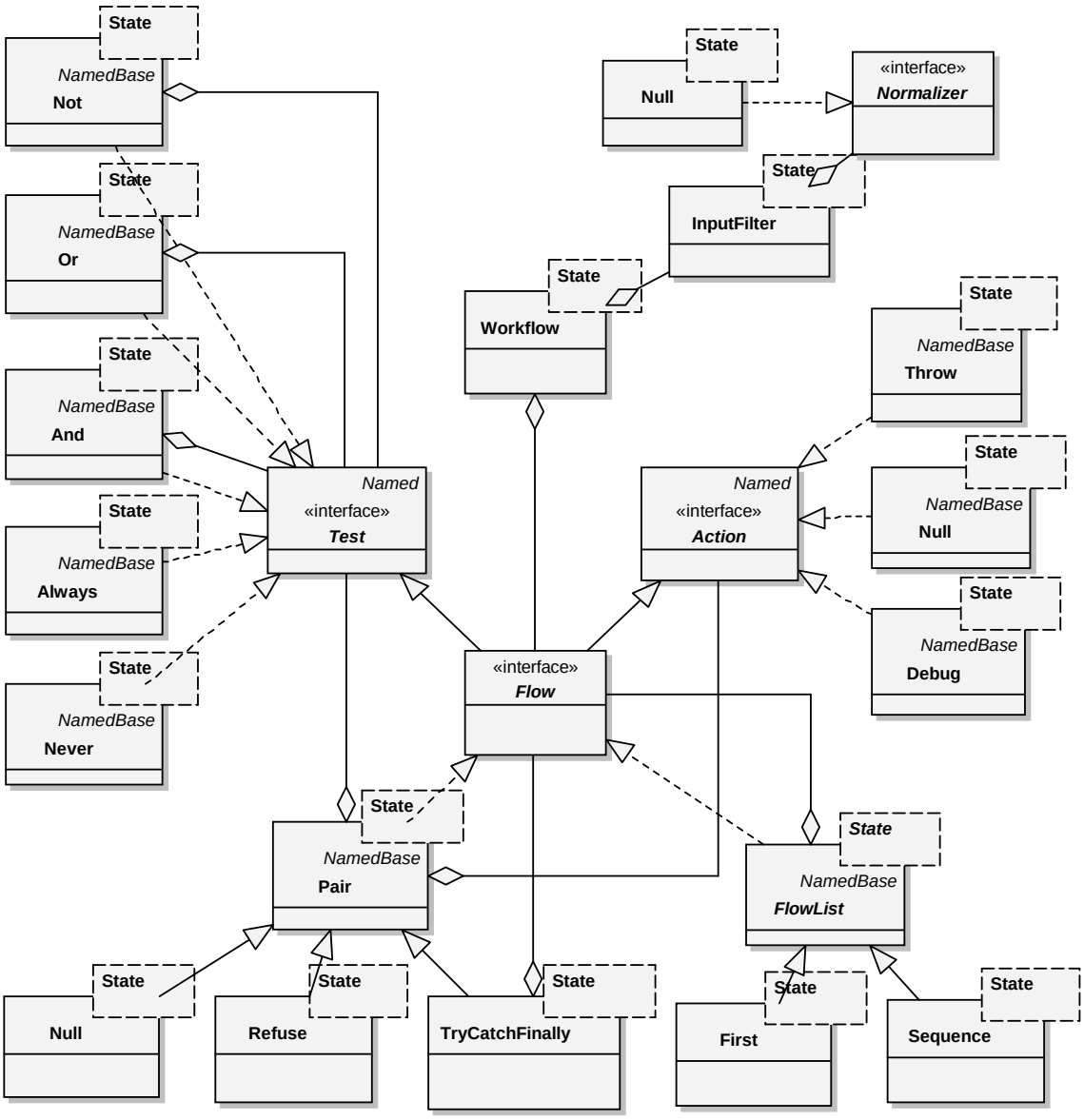

Figure 3.1: The basic structure of the Tiny Workflow.

```
if (testA(state)) {
  try {
    if (! testB(state)) {
      state = processB(state)
    } else if (testC(state)) {
      state = processC1(state)
      state = processC2(state)
      state = processC3(state)
    } else if (testD(state)) {
      state = processD(state)
    } else {
      throw
    }
  } catch {
    state = processE(state)
  }
}
```

Figure 3.2: A fragment of pseudo–code; see figure 3.3 for the equivalent workflow structure.

nents (for clarity some components have been omitted, typically where default values are used).

The advantages of structuring code like this include:

- The ability to restructure the logic without recompiling the software.
- Explicit state allows serialisation (see dicussion in [18]).
- Clear separation of decisions and process (often confused in OO code).
- Use of a standard, simple interface allows composition of generic components.

3.3.2 DB Loader

Figure 3.4 shows part of a fairly complex flow (further loaders have been omitted for clarity). The `First` element will select the first sub–flow whose test is successful. By ordering the loaders from most to least specific the most complex that is consistent with the data will be chosen. If none are chose an exception is thrown; this, or any other exception thrown by a failing loader, is then caught and a generic loader called.

Figure 3.5 is part of the test for `mosaicSkyFlatImageDataProductLoader`. Three different tests are combined with `Or`.

(This system used the XML–based representation of state discussed in section 3.4.1, hence the `tiny.xml` namespace.)

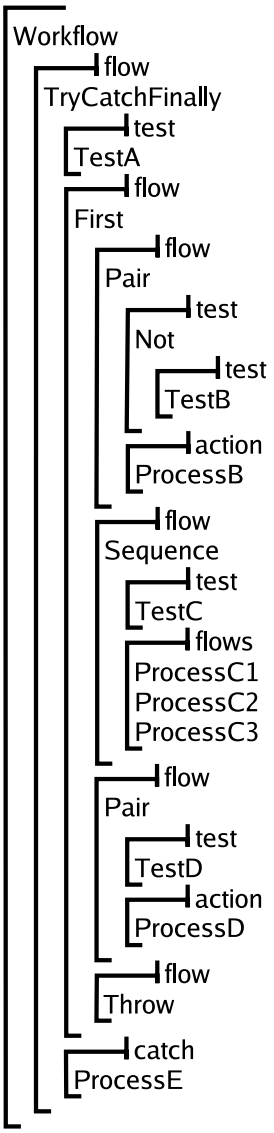

Figure 3.3: A workflow structure that corresponds to the pseudo–code in figure 3.2.

3.4 Experience

In general, my experience with the Tiny Workflow has been positive. It was simple to develop, is easy to understand, and has proved to be very flexible. However, a number of issues restrict its use. It is not a replacement for a 'real' workflow, but is a useful additional tool.

3.4.1 XML Tests

An early iteration forced the state to conform to a specific interface, which provided an XML 'view' of the contents. This had the advantage that tests could be written using XPath, and an XPath–based `Test` could be used in all workflows. Often, no additional, state–specific tests were needed.

However, this approach had some drawbacks. Implementation details were exposed in the workflow configuration (this may have been due to poor design of my XML). Maintaining consistency between the state and the XML was expensive: either the state was XML, in which case actions were inefficient, or the state was Java objects, in which case test were inefficient. Also, XPath is often not very compact or readable; I found myself writing state–specific tests to provide a more friendly interface to the user.

I eventually refactored the code so that the state was generic; the XML–aware interfaces were then (trivially) re–implemented as more specific versions of the generic base.

3.4.2 Verbosity

XML is not very compact. Spring configuration is not very compact, even for XML. Writing complex control flow logic in Spring XML soon becomes rather messy. To some extent this can be managed by standard programming practices (in particular, separating the workflow into a hierarchy of processes and placing each level in a different file), but even that soon becomes inadequate.

Tiny was used successfully in a variety of services, but when it was most useful — when the flexibility was really necessary — I switched to a different configuration mechanism once I understood the issues involved[15]. This was not really a failure, since the library had proved to be very useful, but did show that the logic scales poorly.

3.4.3 Lack of Features

This workflow provides a way to structure related processes, but it does not have any other workflow feature. In particular, it does not provide a mechanism for persisting state. Sub–flows cannot be stopped/started in response to reliable asynchronous messaging, or to provide continuity across system failures.

3.4.4 Persistent State

If this workflow were able to persist state it would simplify the interface with asynchronous messaging[18].

The next release of Java is rumoured to contain support for continuations, but the Tiny is so simple that this would be possible to add by hand; `State` and the call stack would need to be serialisable[3]. Extending `State` to track the call stack would be one approach.

In addition, some mechanism would be needed to interface this functionality with the messaging user, but again, at first glance, this does not seem to be a hard problem.

3.4.5 Pluggability

Where Tiny really shines is in assembling simple, composite services. Combined with the approach described in [18], the `Flow` interface makes building modular, composite services, whose sub–services can be configured independently, trivial. Sometimes it is not even necessary/appropriate to expose the workflow externally; the internal structure works well and a wrapper Java class can simplify configuration.

[3] The simple structure Tiny imposes on the code is what makes this so simple; there is no need to store any further information from the Java stack and heap.

```
<bean name="loaderFlow"
  class="edu.noao.nsa.util.workflow.tiny.xml.flow.TryCatchFinally">
  <property name="action">
    <bean class="edu.noao.nsa.util.workflow.tiny.xml.flow.First">
      <property name="name" value="loadFlow.action"/>
      <property name="flows"><list>
        <ref bean="mosaicDomeFlatImageDataProductLoader"/>
        <ref bean="mosaicSkyFlatImageDataProductLoader"/>
        <ref bean="mosaicBiasImageDataProductLoader"/>
        <ref bean="mosaicReducedObjectImageDataProductLoader"/>
        <ref bean="mosaicRawObjectImageDataProductLoader"/>
        ...
        <bean class="edu.noao.nsa.util.workflow.tiny.xml.flow.Pair">
          <property name="action">
            <bean class="edu.noao.nsa.util.workflow.tiny.xml.action.Throw">
              <property name="message" value="No match with specific loaders"/>
            </bean></property>
        </bean>
      </list></property>
    </bean></property>
  <property name="catch">
    <ref bean="genericDataProductLoader"/>
  </property>
</bean>
```

Figure 3.4: A flow for loading data; the sub–flows are ordered from most to least specific and only the first that succeeds is used.

```
<bean name="_obsTypeSkyFlat" class="edu.noao.nsa.util.workflow.tiny.xml.test.Or">
  <property name="name" value="_obsTypeSkyFlat"/>
  <property name="tests"><list>
    <bean class="edu.noao.nsa.util.properties.tiny.PropertyTest">
      <property name="property" value="OBSTYPE"/>
      <property name="value" value="SFLAT"/>
    </bean>
    <bean class="edu.noao.nsa.util.properties.tiny.PropertyTest">
      <property name="property" value="OBSTYPE"/>
      <property name="value" value="SKYFLAT"/>
    </bean>
    <bean class="edu.noao.nsa.util.properties.tiny.PropertyTest">
      <property name="property" value="OBSTYPE"/>
      <property name="value" value="SKY FLAT"/>
    </bean>
  </list></property>
</bean>
```

Figure 3.5: A composite test for FITS header values in a remediation pipeline.

Chapter 4

A Simple, Lazy, Expression Evaluator

Author. Andrew Cooke.

Abstract. I explain the motivation for, design of, and experience using a simple domain-specific language in Java.

4.1 The Problem

4.1.1 The Remediation Service

The NOAO DMaSS (Data Management and Science Support) platform includes a Remediation Service. Separate instances of this service can be individually configured to perform fairly complex 'processing' of textual data. For example, one might be used to construct values for a general science data model given data from instrument-specific FITS headers[1] for different instruments.

The Tiny Workflow[7] provides a suitable framework for configuring remediation. Continuing with the example above, it would allow different 'flows' to be defined for each instrument, selecting the appropriate flow depending on the science data received.

Data within the remediation workflow are represented as name/value pairs. A simple library (similar to Java's `Properties` class) supports this abstraction, adding immutable values (both per-value, to protect system data, and per-collection, to allow more efficient processing).

Given this framework, the work of implementing the remediation service was reduced to writing various 'actions' that, when appropriately configured, modified the workflow data.

4.1.2 Remediation Actions

Initial analysis suggested that a sufficiently flexible system could be constructed from a set of very simple actions:

- Require that a given set of properties (names) is present.
- Rename a set of properties.
- Provide default property values.
- Rewrite property values using regular expressions.

These different actions could be composed within the workflow (which included the ability to test for values by evaluating XPath expressions against an XML representation of the properties) to construct progressively more complex procedures.

Although these simple actions were easy to implement and configure individually, two problems soon became clear.

First, the Spring workflow configuration exploded into a mass of unsightly XML. I consider myself XML–tolerant, but felt that the verbosity involved in configuring non-trivial processes was excessive.

Second, I had forgotten the need to concatenate values. While this could be fixed this with another action, I was concerned that the set of actions was going to continue to expand, making configuration even more complex and opaque.

4.1.3 Configuration via Programs

One solution to the problems described above is to provide a single, more powerful 'general action' for processing data — ie. provide access to a programming language. The configuration for the action then becomes a small program in its own right.

Various languages are available to provide 'scripting' in Java (e.g: Groovy; Javascript in the next major JDK release). However, these might cause further problems.

[1] FITS headers contain astronomy image metadata in a simple name/value format.

Introducing a general language provides a lot of (unnecessary) functionality at one point in a layered architecture. There is a temptation to use it to solve unrelated problems. For easy maintenance/operation I would like to restrict the functionality available. For example, high–level control should remain in the workflow rather than being implemented in the same script used to manipulate values[2]

There is also a cost involved in using external packages. This is typically dismissed as unimportant compared to the 'obvious expense' of implementing a language from scratch. However, I felt otherwise.

4.2 The Solution

4.2.1 Domain Specific Languages

In recent years there has been an explosion of interest in Embedded / Little / Domain Specific Languages (DSLs), particularly in the functional language programming community. This 'movement' has recognised that:

- Implementing a simple language is not difficult or expensive.
- Languages targeted at particular problems make good user interfaces.
- Careful language design can make the interface easy to control (it can be as restrictive or extensible as needed).

Although much of the literature emphasises the (undeniable) advantages of functional programming languages in supporting this paradigm, I felt that a similar approach should also be possible in Java.

4.2.2 Design Choices

A typical DSL builds on existing language support (e.g. combinator libraries, macros) or uses a dedicated recursive descent parser. Unfortunately, neither of these approaches would work in this case: Java does not have simple runtime compilation or the kind of features (e.g. high order functions) necessary for embedding; a typical hand-written recursive descent parser would be tediously verbose.

I considered using a parser library (GI[16] is very nice) but instead decided that it would be simpler to use a prefix syntax (like Lisp / Scheme) whose parsing is trivial.

Initial requirements involved only function evaluation. Ignoring function and variable definition removed the need for scoping or mutable data. Nor did I need support for lists (at first this seems irrelevant, given the syntax, but it simplifies parsing by eliminating quotation).

However, some control flow was necessary. Learning from the behaviour of '?:' (taken from C language syntax) in some IRAF[17] tasks, which evaluates *both* sides of the branch, it seemed simplest to avoid such problems by using lazy evaluation (terms are calculated only as they are required).

I also chose untyped semantics (a.k.a. 'dynamic' or 'tagged' types), assumed arbitrary dynamic casts, and did not consider operator overloading.

Examples of the resulting syntax can be seen in section 4.2.6.

4.2.3 Parsing

The design described above is so simple it hardly warrants being called a 'language' - it is almost identical to the calculator example (terms and expressions) given in programming textbooks.

Program text is lexed using a simple state machine (a single Java class with a case statement over an `Enum` state). Single character look–ahead is sufficient; I used `nio.CharBuffer`'s 'mark' mechanism to provide this. There is a token for 'open', 'close', 'eof' and each atomic type (including 'name' for unquoted text).

The stream of tokens is assembled into an AST using a simple recursive descent parser that consumes tokens and returns AST nodes. This was implemented as a single class (the productions are private methods). Since the JVM does not have tail call optimisation I needed to worry about stack use; it is proportional to the maximum nesting depth in an expression which, for the intended use, is not a problem.

4.2.4 Evaluation

Evaluation is equivalent to a traversal of the AST, selecting (evaluating) only those nodes required by the semantics. This is not explicit in the implementation (there is no tree walker, for example). Instead, sub nodes are evaluated via the `Node.evaluate(Namespace)` method. Each node implementation is responsible for deciding which child nodes to evaluate in turn.

[2] I later realised that the workflow is closely related to information in the Vocabulary Service[15]; this separation helped us exploit that commonality.

Types are identified ('tagged') by the subclasses of the AST `Node` class. The common superclass guarantees that future extension to include first class functions and lists will be painless (both already exist as `Node` subclasses, but the current operational semantics do not expose their dynamic creation to the user).

Functions and variables are provided via a `Namespace` interface. Typically, variables are read from the `Properties` interface described earlier, while functions include a standard collection that provides basic functionality. There is also an adapter for eager functions (the user programs to an interface that is provided with pre-evaluated arguments) so that simple functions for specific applications can be added without understanding how lazy arguments are implemented.

4.2.5 Generics, Reflection

I was pleasantly surprised at the level of integration possible between the language being implemented and the Java host. Figure 4.1 shows almost all the code necessary to implement literal values, including the base classes, support for dynamic casts and type errors, and the implementation class for string literals. Generics and reflection help us write code that can be used for all (literal) types while also helping leverage the support for these types in the host language.

The dynamic cast method (`as`) is a good example[3]. It is used in the example in figure 4.2.

For those not familiar with Java, the important aspects of figure 4.1 are that (1) most of the logic for literal types (everything except type conversions) is in a single type–safe generic class, `Literal`; (2) the code for dynamic type conversions is statically checked and type–safe; (3) the runtime check for type errors (in the 'new' language, not Java) is made in a generic class (a static member of `TypeException`) using reflection. The ease with which a 'dynamic' language can be implemented in 'static' Java is surprising[4].

[3] Although this implementation makes the addition of later 'third party' types difficult, since casts are encoded in each source class; a 'from' approach would be better, but with an uglier syntax, if this kind of extension were an important requirement.

[4] I believe similar code would be possible in C#

4.2.6 Examples

The 'If' Function

Figure 4.2 shows the implementation of a function that is equivalent to an 'if statement' in eager languages. For example

```
(if false (/ 1 0) (+ 1 2))
```

will return `3` — there will be no division by zero error because the `if` function does not evaluate the second argument if the first is false (`true` and `false` are parsed as literal boolean values).

The first argument is evaluated, cast to a boolean, and used to select the first of second argument. The `evaluate` method would be called from the s–expression, which itself implements `Node.evaluate` by evaluating the list head (typically a `Name` whose `Node.evaluate` retrieves the function from the `Namespace`), casting to a `LazyFunction`, and passing the list tail as arguments.

Basic Actions

Section 4.1.2 listed basic remediation actions. Here I show how the function evaluator can reproduce that functionality.

Exceptions in the evaluator are implemented as Java exceptions in the implementation. The evaluator is structured so that the evaluator automatically inherits the correct semantics.

Required name

```
(require name1 name2 ...)
```

The `require` function throws an exception if there is no binding for any of its arguments. If all arguments are bound it returns `()` (the empty list).

Rename values

```
new-name = old-name
```

The simple evaluator does not bind values, so the syntax above, and the associated responsibility for assigning a new value, belong to the calling package.

Default values

```
name = (default name "value")
```

The `default` function evaluates and returns the first argument. If an exception is thrown during evaluation and the function has a further argument, the

```
public abstract class BaseNode implements Node {
    public abstract Node evaluate(Namespace namespace) throws Exception;
    public <Cast extends Node> Cast as(Class<Cast> clazz) throws TypeException {
        if (clazz.equals(StringLiteral.class)) {
            return TypeException.assertType(new StringLiteral(toString()), clazz);
        } else {
            return TypeException.assertType(this, clazz);
        }
    }
}

public abstract class Literal<Internal> extends BaseNode {
    private Internal value;
    public Literal(Internal value) {this.value = value;}
    public Node evaluate(Namespace namespace) throws Exception {return this;}
    public Internal getValue() {return value;}
    public String toString() {return getValue().toString();}
}

public class StringLiteral extends Literal<String> {
    public StringLiteral(String value) {super(value);}
    public <Cast extends Node> Cast as(Class<Cast> clazz) throws TypeException {
        if (clazz.equals(BooleanLiteral.class)) {
            return TypeException.assertType(new BooleanLiteral(getValue()), clazz);
        } else if ( ... ) {
            ...
        } else {
            return super.as(clazz);
        }
    }
}

public class TypeException extends EvaluationException {
    public TypeException(String msg) {super(msg);}
    public static <Type extends Node> Type assertType(Node node, Class<Type> clazz)
    throws TypeException {
        if (! clazz.isAssignableFrom(node.getClass())) {
            throw new TypeException("Expected " + clazz.getSimpleName()
                    + ", but got " + node.getClass().getSimpleName());
        }
        return clazz.cast(node);
    }
}
```

Figure 4.1: Generic annotations and reflection simplify integration between language and Java host.

```
public class If extends LazyFunction {
    public Node evaluate(NodeList arguments, Namespace namespace)
            throws Exception {
        Iterator<Node> args = arguments.iterator();
        if (! args.next().evaluate(namespace).as(BooleanLiteral.class).getValue()) {
            args.next();
        }
        return args.next().evaluate(namespace);
    }
}
```

Figure 4.2: Implementing 'if' as a lazy function.

exception is discarded and the next argument evaluated.

Note that only the expression enclosed by `()` evaluated by the library described here. The final assignment to a value is the responsibility of the caller.

Regular expressions

```
(regexp firstword "^\\s*(\\w+).*" "$1")
(regexp lastword ".*?(\\w+)\\s*$" "$1")
(regexp firstletters "(\w)\w*\s*" "$1,")
(regexp dropcommas "(.*),$" "$1")
firstname = (firstword DTPI)
lastname = (lastword DTPI)
initials = (dropcomma (firstletters DTPI))
```

Regular expressions are not supported by the evaluator, but they show how easy it is for the caller to add extensions. Here `regexp` is a function, provided by the caller, that defines a Perl 5 regular expression in the caller's state as a side-effect. The caller then supplies the compiled expression via `Namespace` for the next call to the evaluator.

As before, assignment is the responsibility of the caller. The caller is also free to schedule evaluation so that, for example, the regular expressions are defined only once, when the system starts.

4.3 Conclusions

4.3.1 Experience

The initial implementation of this 'language' (lexer; parser; interpreter; unit tests) took about 20 hours (two 10 hour days). As a result, the configuration for remediation was simplified significantly, using a tool whose power is limited, as required, but which could be easily extended in the future.

A careful choice of syntax and semantics allowed us to exploit techniques developed (or rediscovered) with significantly more powerful tools. Implementation time was short, allowing easy integration into our iterative (agile) development process. I believe that many of the approaches emerging from 'academic' functional programming (an emphasis on declarative approaches; little languages; a realisation that powerful modern languages make some heavyweight libraries obsolete) fit well with the agile approach.

Embedding a 'dynamic' language in 'static' Java was surprisingly easy (section 4.2.5). Similar ideas can be seen in the elegant generic form handling within the Spring MVC framework. In my opinion the power of generics and reflection are too often overlooked by Java programmers.

The main drawback to this approach is raw execution speed. A third party language that compiles to bytecode would be many orders of magnitude faster. For this application — evaluating simple expressions during remediation — this is not an issue; if it becomes important later then this solution is, of course, carefully encapsulated within a small number of interfaces, and easily replaced.

4.3.2 Optimisation

As an experiment, I modified the system to cache known results if they were constant. Since I make no assumptions about purity (in particular, functions could be implemented to have side effects and variables mutable) this was not completely trivial; however, it was simpler than I expected (one evening's work).

The modification consisted of: adding `Thunk` with the same interface as `Node`, but the ability to cache a value; extending `NodeList` to wrap `Node` instances in a `Thunk`; changing the return type of `evaluate` to return both a `Node` and an indication whether it could be cached (a signal to the surrounding `Thunk`);

extending functions to include the necessary logic for propagating 'constness'.

In addition, to complete the work, I would have needed to make a distinction between pure and impure functions, and fixed and mutable variables (not difficult; this distinction is already present in the Properties library, but the code there would need to be made more generic, to handle functions as well as strings).

This work was discarded, however, because it seemed (i) too complex for any expected gain (in particular, intermediate objects were generated and discarded for each evaluation) and (ii) I could not see how to make the changes either optional or cost–free when used in an impure context.

4.3.3 Philosophy

Finally, I also feel that there is a hidden value in work like this: it gives real pleasure in what is otherwise an increasingly mechanical and uninvolving profession[5]. Happy programmers are better programmers.

[5]Populated by technicians who use 'never re–invent the wheel' to justify over–complex solutions with fragile dependencies on rapidly evolving third–party systems; the consequent lack of knowledge about basic programming techniques makes their warning a self–fulfilling prophecy.

Appendix A

CTIO / DPPS / Archive Is...

A.1 Andrew Cooke

A.1.1 Contact

Email andrew@acooke.org
acooke@noao.edu
Phone +56 51 205 286

A.1.2 Interests

Functional Languages. Language research, particularly on declarative / functional languages, is a fertile source of ideas, many of which help illuminate both SOA and agile processes. For example, the importance of state and type systems influenced almost every paper in this collection.

Mid–Level / Lightweight Design. Somewhere between 'architecture' and 'algorithms' there's a region where important decisions on implementation must be made. The paper on SOA focuses on this area.

I'm interested in how decisions can be made in a progressive way, so that we can iteratively develop systems without heading down expensive one–way streets. By proceeding stepwise, we can understand the problems and the relevant solutions. Once understood, the right approach is often simple.

Remote Agility. I would love to understand how agile development can be made to work across continents.

A.2 Alvaro Egaña

A.2.1 Contact

Email alvaro@egana.net
aegana@noao.edu
Phone +56 51 205 325

A.2.2 Interests

Distributed Systems. Many arguments can be made in favour of distributed systems, but I think this phrase summarises most of them: don't put all your eggs in one basket. This was the motivation to, first, consider messaging–based designs and, later, to move our attention to SOA.

Bibliography

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

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

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

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

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

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

[7] A. Cooke 2006; A Tiny Workflow in Spring (chapter 3)
http://www.acooke.org/andrew/papers

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

[9] FITS Documentation
http://fits.gsfc.nasa.gov/fits_documentation.html

[10] S Lowry 2006 (ADASS, in preparation)

[11] C, Chambers 1993; Predicate Classes, Lecture Notes in Computer Science
http://citeseer.ist.psu.edu/chambers93predicate.html

[12] D. Grune, C. Jacobs 1990; Parsing Techniques — A Practical Guide
http://www.cs.vu.nl/ dick/PTAPG.html

[13] JavaBeans
http://java.sun.com/products/javabeans/

[14] The ESO Data Interface Definition
http://archive.eso.org/dicb/

[15] A. Cooke, A. Egaña, S. Lowry 2006; FITS Files and Regular Grammars: A DMaSS Design Case Study (chapter 2)
http://www.acooke.org/andrew/papers

[16] GI (Generic Interpreter):
http://www.csupomona.edu/ carich/gi/

[17] IRAF (Image Reduction and Analysis Facility):
http://iraf.net

[18] A. Cooke, A. Egaña 2006; Spring, Mule, Maven: Lightweight SOA with Java (chapter 1)
http://www.acooke.org/andrew/papers