A PRAGMATIC AND VERSATILE ARCHITECTURE FOR LHC CONTROLS SOFTWARE

L. Mestre & Al
CERN, Geneva, Switzerland

ABSTRACT

The LHC Software Application (LSA) team aims at providing homogenous application software to operate the SPS, its transfer lines and the LHC [1]. In that frame, we have put in place a sound architecture based on the Java platform using novel concepts such as lightweight containers and dependency injection. This approach is addressing the shortcomings seen when using Enterprise JavaBeans (EJB) while preserving the benefits of using an application server compliant with the Java 2 Enterprise Edition (J2EE). In addition, it allowed us to develop using plain Java objects (instead of EJB Beans) and to unit test and debug the key parts of the system without the need to deploy to a J2EE container. In fact, all of our applications ran seamlessly in a 2-tier setup on a local machine or in a 3-tier setup using a J2EE server. In other words, the choice of using a J2EE container became a deployment issue rather than a design decision. The overall system was focused on modularity, extensibility, testability, simplicity and easiness to develop and deploy. We successfully used this software stack to operate the CNGS extraction and the TI8 LHC injection line tests in October and November 2004. It is now being used to implement a control system for the new Low Energy Ion Ring accelerator (LEIR) at CERN.

INTRODUCTION

When the LSA project started back in 2001, it was chosen to develop using modern object oriented methodology and language. All software layers above the equipment access (done through a CORBA based middleware) [2] were to be written in Java. Our aim was to leverage the knowledge and the tools available in the Java community and particularly in the open source community. It was to be pragmatic, to use appropriate technology and to prefer simple solutions whenever possible. We also chose to use standards when available. We wanted to set-up a solid base on which all future developments, and in particular those for the LHC, could be based upon.

The LSA software stack is organized in 3 logical tiers: the resource tier is made of the database and the accelerator equipment; the middle-tier is made up of the business logic of the system; and thirdly, the client tier represents the graphical application that a user will interact with. Although we don't have Web based clients yet, that possibility is supported in our design. The web client will perfectly fit as another type of client of the middle-tier with the addition of a thin web layer.

3-TIER ARCHITECTURE

The obvious reason to have 3 logical tiers is to be able to deploy the application on 3 physical tiers, as the so called a 3-tier architecture. 3-tier deployments address the many problems and limitations encountered when deploying and running 2-tier systems, as it was the case with previous control systems at CERN. Amongst those problems are:

- the complete development accumulates on the client machine that has to process and present information, leading to monolithic applications that are expensive to maintain,
- the network load is increased since the actual processing of the data takes place on the remote client,
- transactions are controlled by the client,
- the stress of the client machines is greatly increased,
- the stress on the database is greatly increased as each client has to keep opened connections,
- the stress on the equipment is greatly increased as each client has to perform access,
- the application logic cannot be reused or shared because it is bound to an individual client machine.

If 3-tier systems address those problems, they also introduce a new class of problems. In particular, programming, testing and debugging a 3-tier system is much more complex than a traditional 2-tier

one. It also usually introduces a new programming model (forcing one to use a container) that is not familiar to regular Java programmers. In the Java world, the standard way to develop 3-tier applications is to use the Java 2 Enterprise Edition (J2EE) platform together with the Enterprise JavaBeans (EJB).

We had experience in the Controls group in using both J2EE and EJB [3]. That experience was not completely satisfactory. In particular, the added complexity introduced by the EJB programming model was worrisome. EJB imposes a change in the programming model such as the objects created by the programmer are not anymore plain Java objects but managed entities that can only be used within an EJB container. This makes EJB very intrusive and also makes debugging and testing tricky as it can only be performed on entities already deployed in the EJB container. For the programmer, it means “program => deploy => test” while what he or she really wants is “program => test => deploy”. That deployment step gets in the way and slows down the speed of development. In addition, the EJB solution for object-relational mapping is both complex and inefficient and we already tried alternatives such as TopLink [4] or Hibernate [5] that look simpler are more promising.

As we said, we wanted to have a 3-tier architecture without compromising on the simplicity and the efficiency of the solution. One important condition was to be able to run an application seamlessly in a 3-tier setup with a server or in a 2-tier setup with 2 layers on the client machine for testing and debugging. Therefore, we were not satisfied with the solution proposed by the EJB model while we deeply agreed on the necessity to maintain the benefits of the EJBs such as the transaction management, the object-relational mapping, the remoting middleware, the wiring of services, and thus the ability to focus our developments on the accelerator controls domain and not on the infrastructure.

LIGHTWEIGHT CONTAINERS

We were not the only ones to have those thoughts and around the same time (summer 2003), a new breed of solutions for 3-tier systems came out. It was labelled lightweight containers in opposition to the EJB containers that are rather heavy given the thick specification they have to implement. The basic idea was to provide all services provided by the EJBs (transaction, remote access, service discovery, JMS and database mapping) without their complexity and without introducing a new programming model requiring a container to be used. The two innovations that allowed the lightweight containers to exist were the Inversion of Control (IoC) used mainly in the form of the dependency injection and the Aspects Oriented Programming (AOP). The lightweight container we used in the frame of the LSA project was the Spring Framework [6] but the concepts presented here are not limited to this particular one.

Inversion of Control

Inversion of control is the idea that you give control to an external entity that calls part of your application to perform a given task rather than your application calling the services of an external entity to perform the same task. For instance, an IoC database access will be based on a template that the application fills with the specific code that has to be performed upon retrieval of data. The template will be given to the external entity that will do everything else and only invoke the template when needed.

Dependency Injection

Dependency injection is a particular application of IoC. The basic idea of dependency injection is to have a separate object, an assembler, which populates the dependencies of a given object [7]. This solves the well known problem of service discovery in Java.

Suppose we have a service interface called *SettingFinder* allowing us to retrieve settings in the database. An implementation of that service needs a data source that gives access to the database. Suppose we have another service interface called *TrimController* that needs a setting finder to find settings. The question is how the implementation of *SettingFinder* will get a reference on the data source it needs and how the implementation of *TrimController* will get a reference on the implementation of *SettingFinder* it needs.

The two traditional ways to do that in Java are to use a factory that an object can statically access to get an instance of some other objects or to use a registry that an object can statically access to lookup some other object. The factory way really works at small scale but cannot be used to wire up a

complex application with lot of dependencies. The registry way works well and is used in the EJB model. We see on figure 1 the way the wiring occurs.

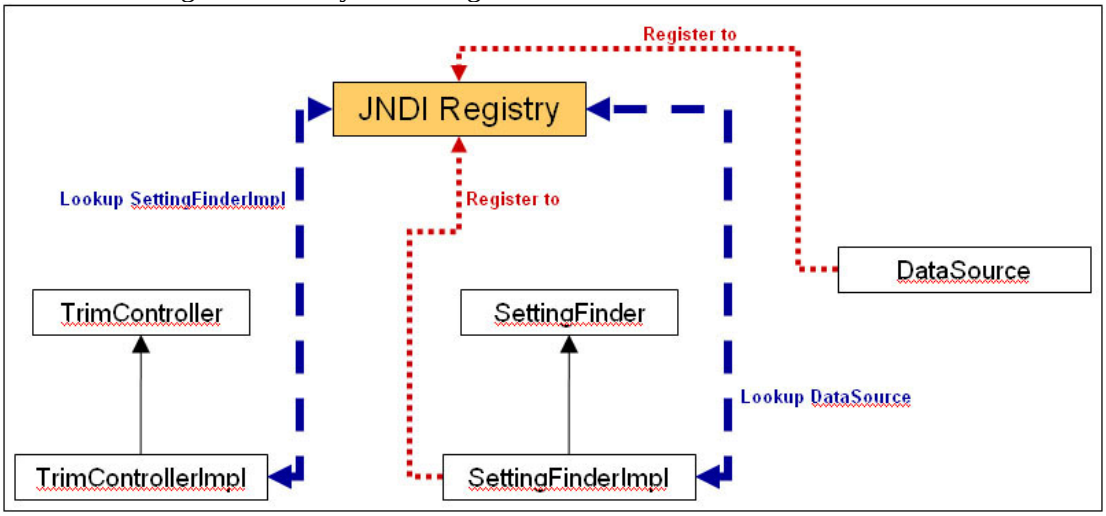

Figure 1: The lookup mechanism when using a registry

The various implementations needs to first register to the registry. This can be done programmatically or by declaration using a configuration file. Then, inside of each implementation, it is necessary to lookup the needed service. This has two drawbacks. First it places a lot of boilerplate code in each implementation to perform the lookup and second, it makes each implementation dependents on the API offered by the registry.

On the other hand, the dependency injection relies on an Assembler that creates and initializes the objects. As with the registry, it is necessary to declare the objects to the Assembler and this is usually done through an XML configuration file (Figure 2).

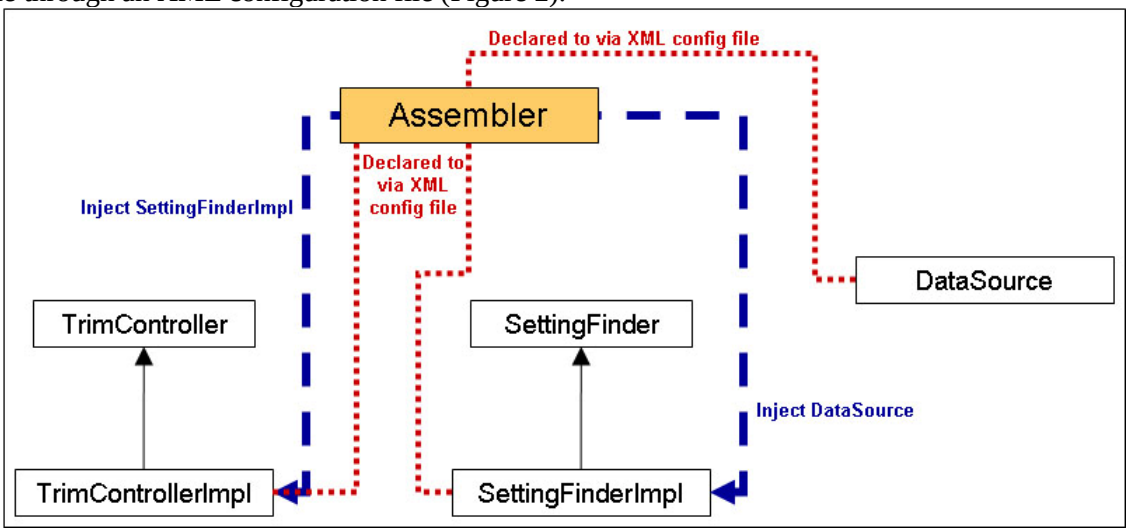

Figure 2: The dependency injection mechanism using an assembler.

Then when the application needs the *TrimController* instance it asks the assembler for it. The assembler will automatically perform the initialization of the instance with the dependent objects (here the instance of *SettingFinder*) and perform at the same time the initialization of the instance of *SettingFinder* with the instance of the *DataSource*, based on the dependencies declared in the XML description. Note that there are two ways for the assembler to perform this initialization. It can pass the dependencies to the constructor of the object (constructor based dependency injection), or it can build the object with an empty constructor and use setters (setter based dependency injection) to set each dependency one after the other. The two methods are supported by the Spring Framework and we chose to use the one based on setters which is more convenient when the number of dependencies is important.

We give below an example of the XML description in the Spring Framework format:

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName"><value>${jdbc.driverClassName}</value></property>
  <property name="url"><value>${jdbc.url}</value></property>
  <property name="username"><value>${jdbc.username}</value></property>
  <property name="password"><value>${jdbc.password}</value></property>
</bean>
<bean id="settingFinder" class="cern.accsoft.settings.spi.SettingFinderImpl">
  <property name="dataSource"><ref bean="dataSource"/></property>
</bean>
<bean id="trimController" class="cern.accsoft.trim.spi.TrimControllerImpl">
  <property name="settingFinder"><ref bean="settingFinder"/></property>
</bean>
```

Aspect Oriented Programming (AOP)

AOP tries to factor out common behaviour in a reusable and generic way so that it can be reused across objects without affecting the actual code of the objects. The common behaviour is called a crosscutting concern: something that cuts across the typical divisions of responsibility such as logging, security or transaction management. The core construct of AOP is the aspect, which encapsulates the common behaviour affecting multiple classes into a reusable module.
There are two main methods to inject the additional behaviour defined by the aspect into the class. The first method is to generate a modified version of the class at compile time that captures the method calls and modify them according to the aspect. This method is used by AspectJ [8] for instance. The second method is to provide the client a proxy generated on the fly at runtime that implements the same methods as the target object but that intercept them to inject the additional behaviour. This is the method used by the Spring Framework for instance as illustrated on Figure 3.

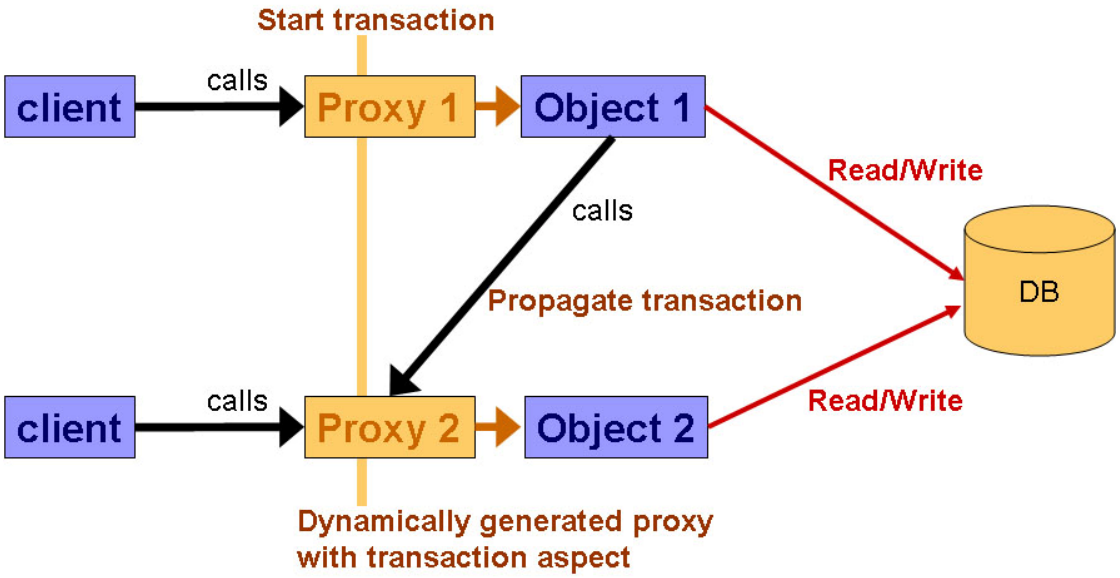

Figure 3: The client calls methods on a proxy automatically generated at runtime by the AOP framework. The proxy adds the transaction aspect that starts or propagates the transaction.

We see below the associated simplified configuration to give to the Spring Framework to add the transaction aspect. The target object *settingFinderTarget* is no more directly accessed by the client. Instead, it will be given a generated proxy *SettingFinder* that delegates to the real object while adding the transaction aspect.

```
<bean id="settingFinderTarget" class="cern.accsoft.settings.spi.SettingFinderImpl">
  <property name="dataSource"><ref bean="dataSource"/></property>
</bean>
<bean id="settingFinder" class="org.springframework.transaction.interceptor.TransactionBean">
  <property name="target"><ref bean="settingFinderTarget"/></property>
  <property name="transactionAttributes">
    <props><prop key="*">PROPAGATION_REQUIRED,readOnly</prop></props>
  </property>
</bean>
```

Lightweight containers were a very innovative concept in 2003 and although they did not create the idea of IoC or AOP they succeeded to used them coherently in a non intrusive efficient way. The value proposition of lightweight containers was very compelling and their exponential growth in the past 2 years demonstrated that they filled a real need. Our choice to based our architecture on them has been validated both by the success we had with them and by the generalization of their ideas in the computing industry. The forthcoming EJB3 specifications take into account most of the innovations found in lightweight containers.

LSA ARCHITECTURE

As we said before the LSA software is based on 3 logical tiers. For testing and debugging, the software ran on the developer computer on 2 physical tiers. For operations, we deployed the core part on a middle tier based on J2EE and only the graphical part of the applications was running on the operator consoles.

This was possible thanks to the use of the Spring Framework and could have been possible with the use of any other lightweight container providing a similar set of functionality. The lightweight container acts more or less like a library that provides services and that can run inside or outside a J2EE server. With the IoC, many of the services provided by the lightweight container are transparent for the application.

Figure 4 below shows the complete architecture of the LSA stack.

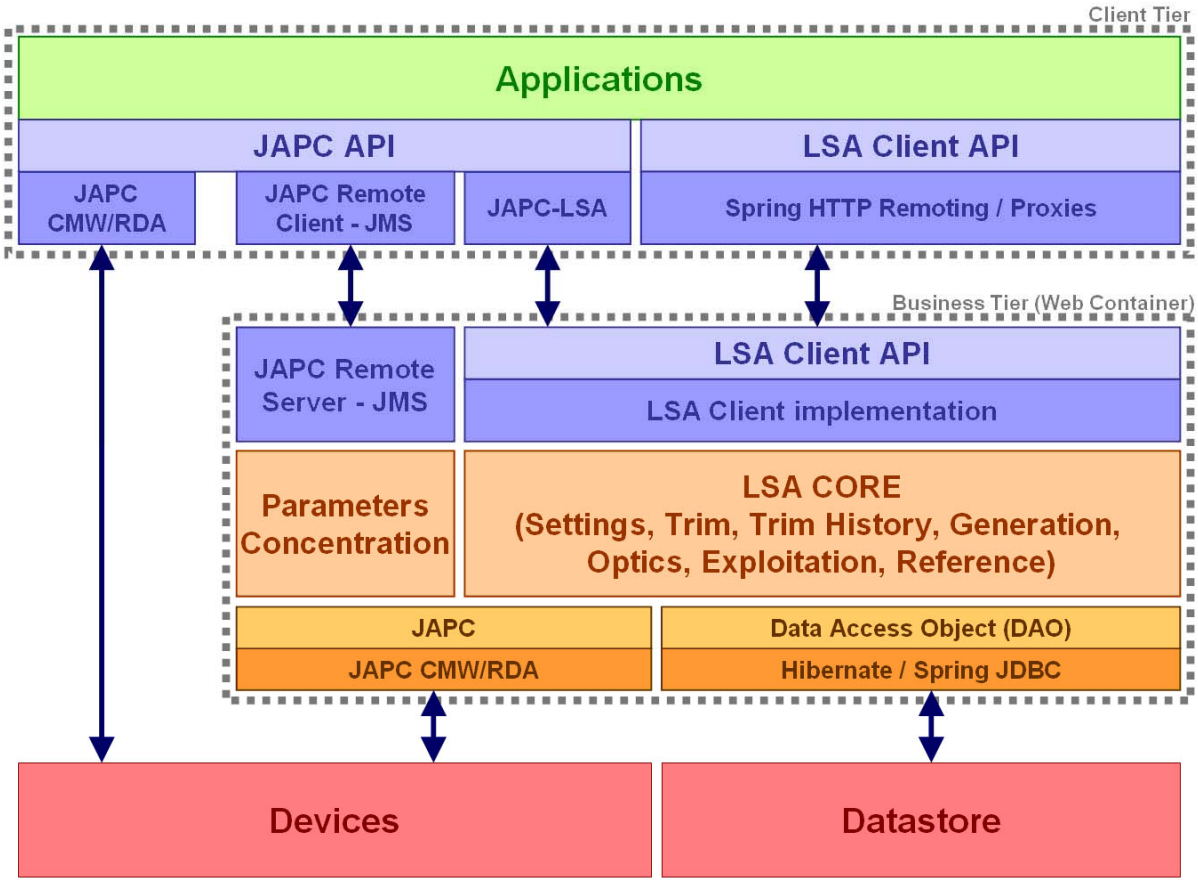

Figure 4: The LSA software stack architecture.

Clients were provided with two APIs. A narrow API called JAPC (Java API for Parameter Control) used for equipment access and more generally for read/write/monitor access of everything assimilated to a parameter [9], and a wide, used case based client API, for accessing the various services found in LSA Core [1]. LSA Core itself was a modular and layered set of interdependent services that clients could only access through the façade client API.

All the wiring of the application was done though the XML based application configuration provided by the Spring Framework. Transactions were managed by the Spring AOP based transaction

abstraction. Transactions were declared in the same XML configuration file in the format describe above. Database access was performed through the Hibernate object-relational mapping library that integrates directly into Spring and through the Spring IoC based JDBC abstraction that is simple and efficient to use.

Synchronous communication between clients and server was handled by the Spring based HTTP remoting. Asynchronous one was handled by Java Messaging Service (JMS). Spring JMS integration was not used at the time because not yet available but the recent version now integrates it. We did not have Web based clients but that eventuality is considered and we would probably use the Web framework provided by Spring for that.

CONCLUSION

In 2004 this architecture was successfully used to provide the high level control software for the TT40 extraction tests, the CNGS extraction tests, the hardware commissioning of TI8 and the TI8 tests with beam. Looking forward to SPS consolidation, tests were performed that included steering of the SPS ring orbit and Q, Q' trims in the SPS and drive of the main power supplies.

Four factors helped to make the software successful during those tests. First the fact that all the development was done in two tiers running on the developer's computer made development much more efficient. We were able to focus entirely on the accelerator domain and we were much more productive than if we had to cope with the never ending deployment steps of a regular 3-tier system. In fact the proper 3-tier deployment was not even considered up to very late in development.

Second, developers didn't have to deal with any complex and unfamiliar APIs. IoC made sure that most of the services used from Spring were in fact completely transparent to them. They were dealing with regular Java classes and that made them more productive, more confident and limited the mistakes due to misunderstanding of APIs.

Third, testing was easy and encouraged. Regular unit tests could be written and could be ran regularly by each developer. Again, the fact that code was not bound to a container and didn't need to be deployed in order to be tested helped a lot. Despite that, we could have done more testing. Because almost everything is data driven we should have tried to have a dedicated test environment with dedicated test data. We did not have the time and manpower to do that. Now that the LSA software is being used with other accelerator we will gain a critical mass that we allow us to have that.

Last but not least, deployment was simple. A simple Web container was enough to provide the HTTP remote protocol needed for remote access. There was no need to deal with the heavy weight EJB container and less problem to deal with.

In summary, lightweight containers are not a panacea but they are definitely a step forward in the right direction. We believe in a simple and pragmatic approach of development and the use of a lightweight container allowed us to stay focused on what we need. Having an overall simpler system to deploy provides immediate benefits and will provide even more later by making maintenance easier.

REFERENCES

[1] M. Lamont, L. Mestre et al, "LHC Era Core Control Application Software", ICALEPCS'2005, Geneva, October 2005.

[2] K. Kostro et al, "The Controls Middleware (CMW) at CERN - Status and Usage", ICALEPCS'2003, Gyeongiu, Korea, October 2003.

[3] V. Baggiolini et Al, "The Cesar Project – Using J2ee For Accelerator Controls", ICALEPCS'2003, Gyeongiu, Korea, October 2003.

[4] Oracle TopLink : www.oracle.com/technology/products/ias/toplink

[5] Hibernate : www.hibernate.org

[6] The Spring Framework : www.springframework.org

[7] Martin Fowler, "Inversion of Control Containers and the Dependency Injection pattern", www.martinfowler.com/articles/injection.html, January 2004.

[8] AspectJ : www.eclipse.org/aspectj

[9] V. Baggiolini et Al, "JAPC - the Java API for Parameter Control", ICALEPCS'2005, Geneva, October 2005.