JBoss architecture evaluation

Bertjan Broeksema (s1554697) Bram Noordzij (s1741047)
Cherian Mathew (s1623389) Lazaro Luhusa (s1818473)
Simon Takens (s1675478) Uwe van Heesch

January 25, 2009

Revision History

Date	Author	Description
15-12-2008	Bertjan	Initial setup of the document.
15-12-2008	Simon	Proxy and Plugin patterns added.
16-12-2008	Cherian	JBoss Remote Framework patterns added.
16-12-2008	Uwe	Added Microkernel pattern
16-12-2008	Cherian	Integrated Uwe's pattern.
16-12-2008	Bertjan + Bram	Added context (from Lazaro) and Stakeholder concerns.
18-12-2008	Cherian	Separated patterns into individual files. Removed JBoss Remoting description.
05-01-2009	Cherian	Improved pattern description (Broker, Factory).
05-01-2009	Simon	Implemented feedback and Improved patterns (Proxy, Plugin).
06-01-2009	Uwe	Improved Microkernel pattern. Added Interceptor pattern. Integrated Logical view and process view of core architecture.
06-01-2009	Cherian	Added Active Repository details. Improved pattern tables for quality attributes
06-01-2009	Bertjan	Added first version of the introduction section.
06-01-2009	Bram	Updated stakeholders.
08-01-2009	Cherian + Bertjan	Added description in the QA Evaulation section
11-01-2009	Uwe	Described overall architecture. Provided explaining images for microkernel, interceptor and dynamic proxy patterns.
11-01-2009	Cherian + Bertjan	Added details for the last two sections (step 5,6 of PBAR).
20-01-2009	Lazaro + Simon	Mapping Quality Attributes to stakeholder concerns.
23-01-2009	Lazaro + Simon + Bram	Final Revision, Final version
25-01-2009	Cherian	Last Minute Changes, Final version

Contents

0 Glossary

JBoss	Java Bean Open Source Software
JBoss AS	JBoss Application Server
J2SE	Java 2 Standard Edition
J2EE	Java 2 Enterprise Edition
PBAR	Pattern Based Architecture Review
UML	Unified Modelling Language
QA	Quality Attributes
JMX	Java Management Extension
MBeans	Managed Beans
AOP	Aspect Oriented Programming
JNDI	Java Naming and Directory Interface
POJO	Plain Old Java Object
EJB	Enterprise Java Bean
IDE	Integrated Development Environment
EIS	Enterprise Information System
API	Application Programming Interface
RMI	Remote Method Invocation

1 Introduction

This document is the result of the pattern-based recovery and evaluation assignment, as applied by group B, for the course Software Patterns 2009. For this assignment, we have chosen to recover and evaluate the architecture of the JBoss application server version 5.0.

The goal of this document is to give an overview of the most important architectural patterns that represent the core structure of the JBoss application server. Further more we want to evaluate the found patterns using the Pattern Based Architectural Review method (PBAR). [5].

The patterns were recovered using various methods. First of all, we looked into the documentation which can be found on various parts of the JBoss website. We extracted patterns from written text as well as from the images. Next, we investigated by reading sections of the source code in order to try to find the described patterns. Furthermore, we tried also to reverse engineer by generating UML diagrams from the code. The structure of the code directories however, was so complex and large that the tools we used where not able to produce useful results. Lastly, we also have some experience in building applications on top of JBoss as well as on top of other application servers.

The document is structured in such a way to reflect the steps of PBAR. Chapter 2 and 3 describes the context, the stakeholders of the system and the concerns of the stakeholders which together lead to the most important Quality Attributes which should drive the development of the system. The chapters 4 and 5 give an overview of the patterns we have discovered and how those patterns are related to each other. For each of the patterns we describe the problem they aim to solve, what the impact of the pattern on the architecture is and what variant (if applicable) is used. In chapter 6 we give an evaluation of how well the quality attributes identified are addressed. Finally, in chapter 7 we give some recommendation of where additional assessment of the architecture might be needed.

One last remark to make here is that for the quality attributes we use the definitions as given in the quality model described in the ISO-9126 standard.

2 System context

Java Bean Open Source Software (JBoss) is an open source component based framework for deploying web applications and services in a service oriented architecture. The JBoss project started in 1999 as a research project. It became a coorporation in 2004. In April 2006 it was bought by Red Hat, which is the current owner. The enterprises that want to deploy their distributed application using JBoss, can either buy support from Red Hat or at other companies which support JBoss.

The power of application servers is that it enables developers to create distributed applications without having to implement basic features for these kind of application (e.g. logging, transaction support, caching, load balancing, etc.) over and over again.

JBoss provides middleware services for data and code integrity, centralised configuration, security, performance, transactions and total cost of ownership. It is an implementation of the Java 2 Enterprise Edition (J2EE) standards, using Java SE and therefore is platform independent. JBoss connects the custom application build on top of it, with one or more databases. Figure 1 [1] gives an overview of how the JBoss (J2EE) platform connects with other components.

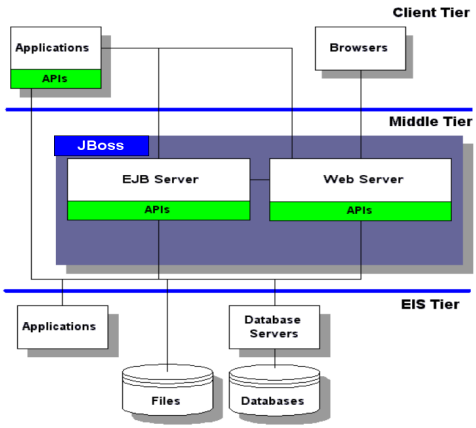

Figure 1: System context diagram

[1] retrieved from
http://www.service-architecture.com/application-servers/articles/application_server_definition.html

3 Stakeholders

From the system context we derive the most prominent stakeholders of the system. Table 1 lists these stake holders with their most significant concerns. The stakeholders are listed in order of importance.

3.1 Stakeholder Concerns

stakeholder	**concerns**	**Priority**
Red Hat Management	1.1- The product should integrate well with Red Hat Linux.	high
	1.2- The product should contribute to sales increase of Red Hat products.	medium
	1.3- The product should be very competitive on the application server market.	high
	1.4- It should be relatively simple to add new features to the product.	high
Red Hat Customers	2.1- The product should provide a stable and reliable environment for the business applications.	high
	2.2- The product should scale to the needs of the costumer.	high
	2.3- The product should provide means for high performing distributed applications.	high
	2.4- The product should perform well.	high
	2.5- The product should integrate well with existing business applications.	medium
Application Developers	3.1- The product should be well documented.	medium
	3.2- The product should provide means to easily integrate clustering, security, transaction support, caching, monitoring and persistence into a custom application.	high
Red Hat JBoss Developers	4.1- It should be easy to test if contributions break functionality or performance of the system.	medium
Community JBoss Developers	5.1- It should be easy to understand the internals of the JBoss product.	low

Table 1: Stakeholders and their concerns

3.2 Quality Attributes

From the stakeholders we derive the following Quality Attributes, or key drivers. The most significant first, the Quality Attributes are also mapped to the stakeholder concerns in Table 2:

1. Reliability - The application server should be an high perfoming framework which works as expected under specified circumstances.

2. Adaptability - The application server should easily integrate in various environments (e.g. It should easily adapt to various communication protocols and hardware platforms).

3. Availability - The application server should be able to avoid failures.

4. Changeability - The application server should be easy changable to adapt to the specific needs for the application which is run on top of it.

More QAs were identified, but they're not as important as the key drivers.

1. Integratability - It should be as simple as possible to integrate the JBoss platform with other technologies.

2. Reusability - It should allow reuse of parts of applications build in JBoss in other applications.

3. Usability - It should be as simple as possible to build and deploy applications on top of JBoss.

Presented in table form:

Quality Attributes	**stakeholder concerns**
Reliability	1.1, 1.2, 1.3, 2.1, 2.3, 2.4
Adaptability	1.4, 2.2, 3.2, 4.1
Availability	1.2, 1.3, 2.1, 2.4
Changeability	1.4, 2.2, 3.2
Integrability	1.1, 2.5, 3.2
Reusability	1.1, 2.5
Usability	2.3, 3.1, 3.2, 1.2, 5.1

Table 2: Mapping Stakeholders concerns to Quality Attributes

4 Architecture - Logical view and Process view

This section describes the overall architecture of the JBoss server, including the most prominent patterns.

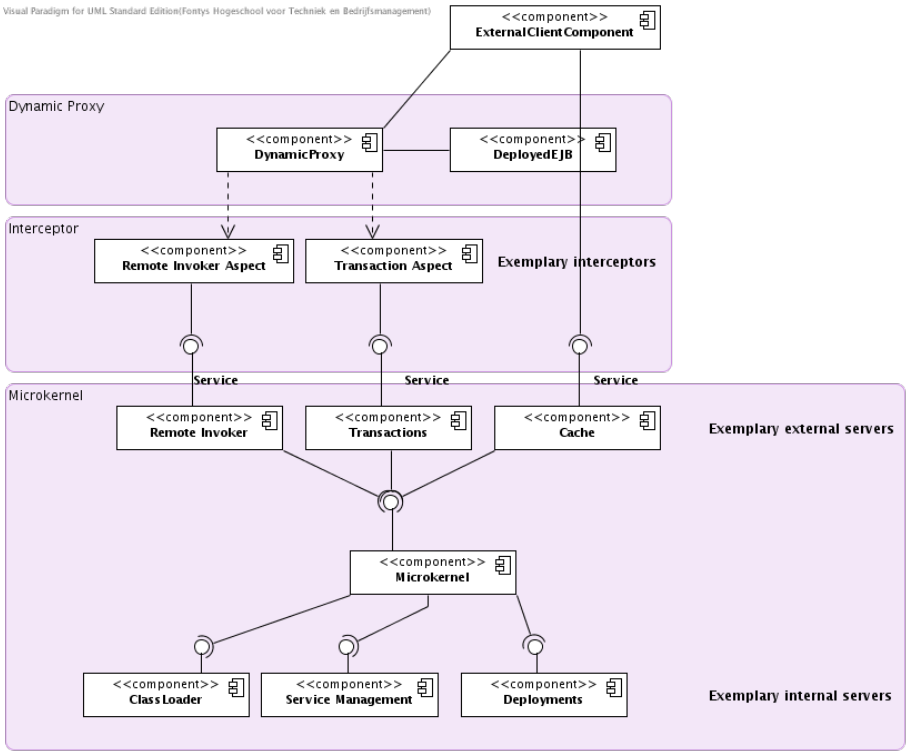

Figure 2: Core architecture

Although figure 2 optically reminds one of a Relaxed Layered system, the Microkernel pattern describes the architecture better. The JBoss server may be used as part of business applications that do use the layers pattern, though.

Two of the major concerns of the JBoss stakeholders are related to changeability and adaptability. The Microkernel pattern is used to satisfy these concerns. The functional core of the architecture is the so called Microcontainer which can be seen as a microkernel, representing the central component of the system. It implements central services like classloading, deployment and service management. On top of the microcontainer, services are deployed as external servers. These services include remote invoking, transaction- and cache management. Besides the provided services, any other service can be deployed on top of the microkernel by providing a predefined service interface. The microkernel implementation used here is JMX (Java Management Extension), services are implemented using so called Managed Beans (MBeans).

Some of the services that provide functionality to satisfy cross cutting concerns of component base enterprise applications, are wrapped in so called inter-

ceptors. These interceptors provide the means for Aspect Oriented Programming (AOP). The pattern, which can be seen here, is the Message Interceptor pattern. Interceptors can be combined in a chain to provide business component behaviour around plain components. The variant chosen in the JBoss server is the Chain-of-Responsibility variant. Each interceptor calls the next following interceptor. An intercepting-context is passed among the invocations. Please refer to the Interceptor pattern in the pattern section for further details and diagrams. Interceptors can be configured to be used by dynamic proxies, invokers and containers, that provide a runtime environment for business components. The combination of the Dynamic Proxy pattern and the interceptors is prominent in the JBoss Remoting package. Clients always use business components through so called dynamic proxies. It is *dynamic*, because the behaviour of the proxy can be changed at runtime. Method calls on the proxy object are delegated to a proxy handler. The proxy handler itself is configured to invoke a chain of interceptors. Finally, the remote-invocation-interceptor is used as a Broker to forward requests to the Application Server itself, which also invokes a chain of interceptors, before actually invoking the target method on a business component. Further explanations and detailled diagrams can be found in the dynamic proxy pattern explanation in the pattern section.

Subsuming the combination of patterns that is explained in figure 2 is *Microkernel*, *Message Interceptor*, *Dynamic Proxy* and implicitly *Broker*. Other identified patterns, which are less prominent in the architecture, are explained in the upcoming sections.

5 Pattern Documentation

This section describes how the possible patterns (and their variants) used to design the architecture of the JBoss system. Each pattern is represented by a table whose structure is inspired by [6]. The tables provide details on the reasons for choosing the corresponding patterns with respect to the system requirements.

Since the amount of JBoss documentation is enormous, we started with a global overview of the system and then identified patterns in a few specific subsystems. Figure.3[2] shows how JBoss implements a simple J2SE application in a J2EE environment. The core pattern in the architecture is the *Microkernel* which provides essential functionality, including the possibility to deploy various kinds of services in the form of *Interceptors*. One of the promenient interceptor services is the JBoss Remoting framework which implements the *Client-Server*, *Broker*, *Factory* and *Active Repository* pattern. The *Dynamic Proxy* extends the functionality of the remoting services to make JBoss a enterprise oriented system.

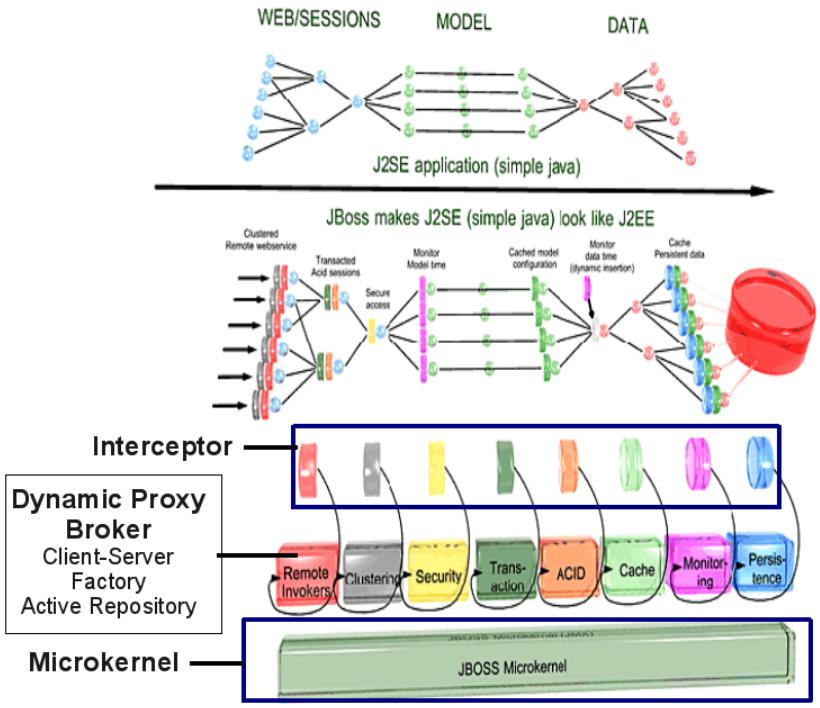

Figure 3: JBOSS Deployment Architecture

[2]retrieved from http://www.jboss.com/products/jbossas/architecture

5.1 JBoss Core

This section deals with patterns which describe the essential components of the JBoss architecture. Both the *Microkernel* and *Interceptor* allow for very high flexibility implying that various kinds of applications can be developed using these components.

5.1.1 Microkernel

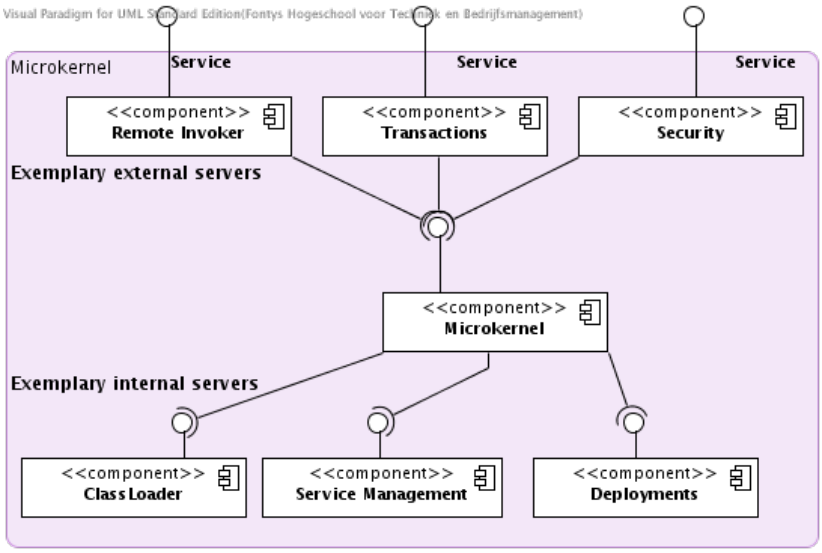

Figure 4: The JBoss microkernel

Pattern Section	Comments
Name	Microkernel
Problem	Two basic forces were taken into consideration when designing the Jboss server. 1. The Application Server platform must cope with continuous software evolution. The services provided by the Jboss are among others specified in the Java Community Process and new versions emerge rapidly (2 years). 2. The application platform should be portable, extensible and adaptable to allow easy integration of emerging technologies. The Jboss server should be configurable for extremely lightweight application type setups like junit testcase or Mobile Applications, as well as for full feature application servers.
Category	Adaptable Systems
Context	Development of several applications that user similar programming interfaces that build on the same core functionality.

Variants	The variant chosen to satisfy the forces is documented in [4]. There it is the base variant of the Microkernel pattern. In this variant Client and Server (Service Provider) communicate directly. Messages are not passed through the microkernel on every request.
Solution	The Jboss developers decided to encapsulate the fundamental services needed by a full featured J2EE Application Server, as described in the J2EE specification, in a Microcontainer, as shown in figure 4. The internal servers (services) provided by the microcontainer are the following: 1. Class loading 2. Deployments 3. State management 4. Lifecycle and Dependency management 5. Configuration 6. Service management Other services provided by the application server are implemented as external servers. For every external server, a Service Proxy is created. The instance of the service proxy is bound to the client. Some well known external servers in the Jboss download versions are: 1. AOP 2. Security and Identity Management 3. Remoting 4. EJB3 5. Transactions 6. Web-Services This is just a small excerpt from existing external servers.
Rationale	The JBoss Application Server uses the microcontainer to integrate enterprise services in order to provide a standard J2EE environment. If additional services are needed, then they can simply be deployed on top of the container to provide the needed functionality. Likewise any services that are not needed can be removed simply by changing the configuration. Since JBoss Microcontainer is very lightweight and deals with POJOs (Plain Old Java Object) it can also be used to deploy services into a Java ME runtime environment. This opens up new possibilities for mobile applications that can now take advantage of enterprise services without requiring a full J2EE application server.

Consequences	1. External servers do not need to be ported to a new software environment. Only the microcontainer has to be ported, which improves adaptability. 2. The microcontainer based architecture is very flexible and extensible. Extending the system with additional capabilities only requires the addition or extension of servers. 3. The microcontainer based architecture is complex in design and implementation. It is a non-trivial task to analyse and predict the basic mechanisms that must be provided by the microcontainer. As a result it is likely, that services were forgotten and need to be implemented bypassing the microcontainer. The effort for refactoring might be higher than in a layer based system for example.
Related patterns	Layers, Interceptor

5.1.2 Interceptor

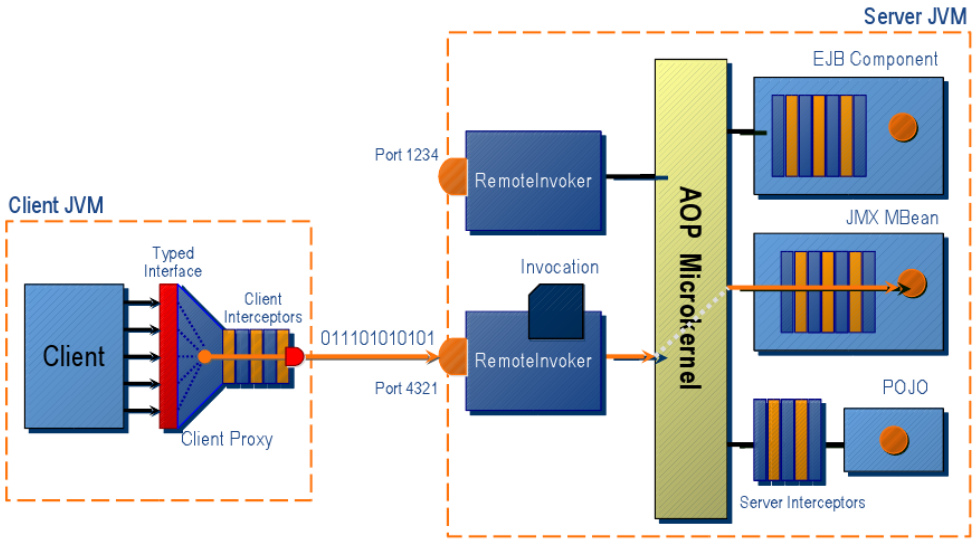

Figure 5: Interceptors and Invokers

Pattern Section	Comments
Name	Interceptor
Problem	The JBoss started as a server, providing a runtime environment for Enterprise Java Beans (EJB). The main challenge here was to wrap the specified EJB services like transaction, security, logging, profiling, caching and others, around a client's call of an EJB method. The developers needed to create a flexible and, for the user, easy-to-change implementation, as the EJB services are optional services the users must be able to choose among.
Category	Aspect Oriented Programming
Context	Separation of cross-cutting concerns in enterprise Java applications.
Variants	The variant of the interceptor pattern used in the JBoss server, see figure 5, is the *Message Interceptor* pattern as documented in [7]. A chain of interceptors is applied in an indirection layer. Some interceptors can be configured to be invoked before the invocation of the target component, others may be configured to be invoked afterwards. The pattern is used to implement the concept of *container based deployment* as specified in the EJB core specification. A container can be seen as runtime environment for EJB components. In the JBoss server, containers are specified as a chain of interceptors that are applied before a method on a bean component is invoked.

Solution	The interceptors are invoked *before* and *after* a method invocation. It is done using the dynamic proxy pattern. The handler of the dynamic proxy gets to know the chain of interceptors that need to be processed before and after a method on the target object, hidden by the proxy is invoked. The interceptors themselves are typically used to invoke services, registered as external servers in the microkernel architecture.
Rationale	The JBoss Application Server uses interceptors to wrap enterprise services around method invocations on deployed components. If additional services have to be invoked, then they can simply be plugged in as interceptors in the interceptor chain. Likewise any services that are not needed can be removed simply by changing the configuration.
Consequences	1. Services needed for deployed components can be configured very flexible and extensible. 2. Interceptors may introduce single points of failures. If one interceptor crashes the whole chain of interceptors is interrupted. 3. An architecture that makes excessive use of interceptors is complex in behaviour. The behaviour of the system is hard to predict, as interceptors are dynamic components invoked at runtime. 4. The concept of AOP, that is followed using interceptors is also currently not accurately supported by IDEs.
Related patterns	Microkernel, Dynamic Proxy

5.2 JBoss Enterprise

The section describes how JBoss can be used as an enterprise oriented application server. The *Dynamic Proxy* makes it possible to build enterprise level applications.

5.2.1 Dynamic Proxy

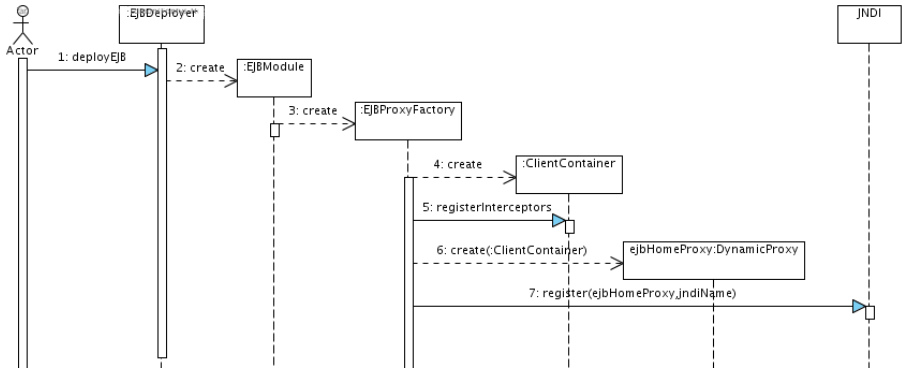

Figure 6: Creation and population of dynamic proxy on server side

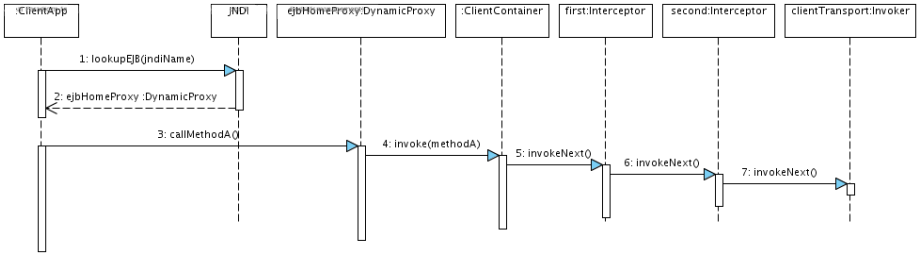

Figure 7: Invocation through dynamic proxy on client side

Pattern Section	**Comments**
Name	DYNAMIC PROXY
Problem	For any developer who builds an application on JBoss, the mechanism for invocation of a remote service should be similar to a local function call. This capability would allow developers to construct distributed systems with ease. The ability of the JBoss environment to provide remote service transparency is an essential need.
Category	Distributed Communication
Context	Client applications should be able to make remote invocations in a transparent manner.

Variants	The stated problem can be solved using a variant of the DYNAMIC PROXY pattern. The presence of JAVA invocation handlers provides the functionality for the given variant (instead of the standard PROXY pattern).
Solution	The proxy element of the DYNAMIC PROXY pattern serves as a substitute for a target POJO on the remote server. The proxy is an object that can implement a list of specified interfaces at run time when it is created using Java reflection. The first step is to construct a POJO which implements one or more interfaces that are to be exposed for remote method invocation. A transporter server is then wrapped around the POJO to expose it remotely. On the client side, in order to be able to call on the target POJO remotely, a client transporter is used. The client transporter takes in the locator url to find the target POJO (same as one used when creating the transporter server) and the interface for the target POJO, on which the remote method invocations are to be made. The return from this create call is a dynamic proxy which can be cast to the same interface type supplied. This is a result of the target POJO being serialised and sent to the remote client across network. At this point, any method can be invoked on the returned object, which will then make the remote invocations using JBoss Remoting. This mechanism is also used by JBoss to provide EJB functionality. In the EJB specification, EJBHome implements the bean home interface and EJBObject implements bean remote interface. EJBObject interface presents a client's view of EJB and it is the responsibility of container provider to generate the EJBHome and EJBObject. In the design of JBoss EJB container, there is NO EJBHome and EJBObject object implementation at all. It is the proxy element that takes on the role of EJBHome and EJBObject. The creation of the dynamic proxy is illustrated in figure 6. An example of an invocation is shown in figure 7.
Rationale	By simply providing the locator url of the remote service and the given interface a client application can obtain a proxy object, which can be called on directly. This provides location transparency implying ease in development.

Consequences	1. It is possible to create multiple target POJOs using the transporter server in clustered mode, implying scalability. 2. Clustering also allows for automatic, seamless failover of remote method invocations improving availability. 3. Since the application developer can call on proxy objects directly, usability improves. 4. Changes in the target POJO can done without any impact implying changeability. 5. Since JAVA reflection is used extensively in generating the proxy, reliability can decrease.
Example uses	An example of the proxy pattern is a reference counting pointer object. In situations where multiple copies of a complex object must exist, the proxy pattern can be adapted to incorporate the Flyweight Pattern in order to reduce the application's memory footprint. Typically one instance of the complex object is created, and multiple proxy objects are created, all of which contain a reference to the single original complex object. Any operations performed on the proxies are forwarded to the original object. Once all instances of the proxy are out of scope, the complex object's memory may be deallocated.

5.3 JBoss Remoting

This section deals with the patterns related to client/server communication. The communication functionality is implemented by the JBoss Remoting[2] API which provides the ability for making synchronous and asynchronous remote calls (*Client-Server*, *Broker*, *Factory*) and automatic discovery of remoting servers (*Active Repository*).

5.3.1 Client - Server

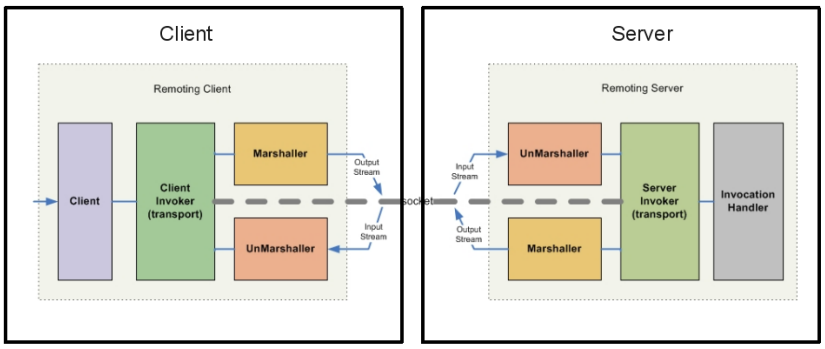

Figure 8: Client - Server Decoupling

Pattern Section	**Comments**
Name	CLIENT-SERVER
Problem	Since JBoss is in itself an application server, a system based on clients connecting to the server is an inherent feature of JBoss and is an essential need for all concerned stakeholders.
Category	Remote Invocation
Context	Clients can connect to application servers and perform remote invocations on services provided by the server.
Variants	The variant used here is the basic variant of Client-Server pattern as described in [3]

Solution	The J2EE platform acts as a server capable of handling web, EJB and basic remoting requests. The Client Tier can be one or more applications or browsers. There can be additional sub-tiers on the server side including the Enterprise Information System (EIS) tier which links to existing applications, files, and databases. As can be seen in figure 8[2], the client application calls the remoting client API to make an invocation request to a service on a remote server. The request is directed to the appropriate invocation handler on the remote server, which handles the request and generates a response, which is sent back to the client.
Rationale	An application server framework needs a distinction between client applications and server processes, to benefit from the advantages of a distributed system. This allows distinction between application written on the client side and server invocations, implying better management of functionalities.
Consequences	1. Decoupling of the functionalities into client and server elements allows for better changeability of either component without affecting the other. 2. The single point of interaction also improves integratability of the client - server components. 3. Since the remote server can provide services for multiple clients, reusability improves. 4. Due to heavy dependence on the remote server, reliability and availability can be adversely affected.
Example uses	JBoss Remoting can be used to implement content management systems where multiple clients perform remote invocation calls to obtain specific content from a data management application hosted on a central server.

5.3.2 Broker

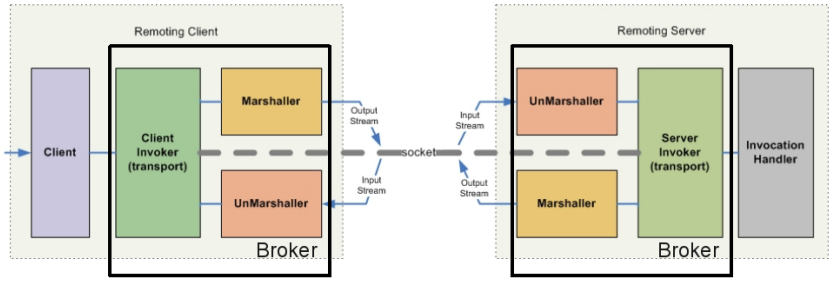

Figure 9: Invoking/Marshalling using the Broker Pattern

Pattern Section	Comments
Name	BROKER
Problem	One important feature of any application server is its ability to handle different methods of client-server communication across networks. For the benefit of all users of JBoss it is necessary to hide the communication details, so that they can focus on developing applications that efficiently tackle their domain specific problems. This aspect has been given considerable importance in the development of JBoss.
Category	Remote Invocation
Context	Server identification should be performed in a manner which allows for remoting servers to be easily identified and called upon. Communication details regarding transport protocols, data (un)marshalling and serialisation should be hidden from the user.
Variants	The stated problem can be solved using a variant of the BROKER [3] pattern. In this variant the requester (Client) hides the communication details by calling on the the appropriate invoker which in sequence calls on the marshaller / unmarshaller to handle the request / response.

Solution	Server identification can be done (via an Invoker-Locator object) using a simple string with a URL based format (e.g., socket://myhost:5400). This is all that is required to either create a remoting server or to make a call on a remoting server. As seen in figure 9[2], the broker element, on both client and server, consists of an invoker which handles transport details and a component that performs marshalling/unmarshalling of data. When a user calls on the Client to make an invocation, it will pass this invocation request to the appropriate client invoker, based on the transport specified by the locator url. The client invoker will then use the marshaller to convert the invocation request object to the proper data format to send over the network. On the server side, an unmarshaller will receive this data from the network and convert it back into a standard invocation request object and send it on to the server invoker. The server invoker will then pass this invocation request on to the users implementation of the invocation handler. The response from the invocation handler will pass back through the server invoker and on to the marshaller, which will then convert the invocation response object to the proper data format and send back to the client. The unmarshaller on the client will convert the invocation response from wire data format into standard invocation response object, which will be passed back up through the client invoker and Client to the original caller
Rationale	The client can easily identify the server as well as specify the transport protocol by providing a simple locator url. The same locator url is also used to create and initialise a remote server. The broker element takes the information embedded within the locator url and constructs the underlying remoting components needed to build the full stack required for either making or receiving remote invocations. By restricting the transport layer specifications to the locator url, the broker element ensures that user applications on the client and invocation handling applications on the server are not impacted by the underlying details relating to communication.

Consequences	1. Solution to hide communication details improves integratability between the clients and servers as the broker element provides a single point of contact between the two. 2. Changeability of communication functionality improves as the broker element can be adapted without affecting the client / server application layers. 3. Since the broker element provides a simple interface to the client application usability improves. 4. Adaptability, reliability and availability of the system can be affected, since the broker element presents itself as a single point of failure.
Example uses	If a user decides to change the transport protocol from sockets to http, the only change required will be the locator url (e.g. from 'socket://myhost:5400' to 'http://myhost:80').

5.3.3 Factory

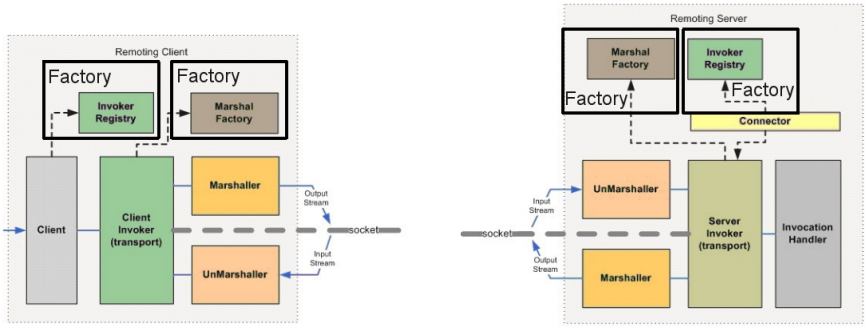

Figure 10: Invoking/Marshalling using the Factory Pattern

Pattern Section	**Comments**
Name	FACTORY
Problem	The feature that makes an application server interesting to use, is its ability to handle different transport protocols as well as different data formats for wire transfer and serialisation implementations. Consequently, it is critical that JBoss consider them as essential.
Category	Remote Invocation
Context	Different protocols that transport the same remoting API should be pluggable. Provided protocols should include Socket, RMI, HTTP(S), Multiplex, Servlet and BiSocket. Different implementations of (un)marshalling and serialisation for data streams should be easily integrated.
Variants	The stated problem is solved using a variant of the FACTORY pattern. The transportation level is split into client and server factories, each of which provide separate functionalities, whereas the (un)marshalling level consists of a single factory.

Solution	The transport implementations within JBoss remoting, called invokers, are responsible for handling the wire protocol to be used by remoting clients and servers. As seen in figure 10[2], the JBoss remoting loads client and server invoker implementations (within the InvokerRegistry) using factories (TransportClientFactory / TransportServerFactory). The invokers are generated, based on the locator url provided by the client API, which in turn calls the MarshallFactory, where the marshaller/unmarshaller is generated based on the data type information in the locator url.
Rationale	The factory design allows the implementation of different invoking and marshalling methods.
Consequences	1. The FACTORY pattern improves changeability with respect to the invokers and (un)marshallers, as the transport details can be changed / adapted without affecting the other components. 2. As the factory provides a standard interface, integratability becomes easier as all implementation details are hidden behind this interface.
Example uses	In the case of remote invocation on the server 'myhost' on port 5400 using sockets and marshalling data type as serializable, the locator url would be 'socket://myhost:5400/?datatype=serializable'. To perform the remote call, the client begins by passing the locator url to the *InvokerRegistry* which returns a *ServerInvoker* instance, to handle the transport of data. The *ServerInvoker*, in turn, obtains a *SerializableMarshaller* instance from the *MarshalFactory* to convert the data into the required wire format for the remote invocation call.

5.3.4 Active Repository

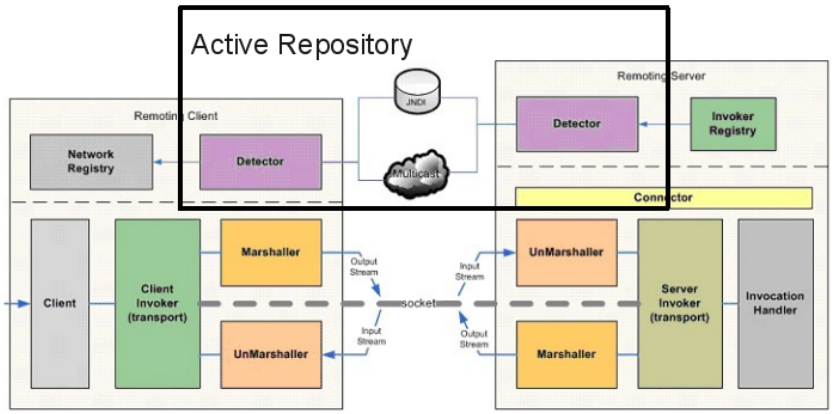

Figure 11: Discovery using the Active Repository Pattern

Pattern Section	Comments
Name	ACTIVE REPOSITORY (EXPLICIT / IMPLICIT INVOCATION)
Problem	To ensure availability of services deployed on remote servers, the JBoss architecture should ideally include a mechanism which allows clients to discover when particular services are not available and seamlessly switch over to the ones that are.
Category	Remote Invocation
Context	Client applications should be able to automatically detect remoting servers as they come on and off line.
Variants	The stated problem can be solved using a variant of the ACTIVE REPOSITORY [3] pattern. JBoss remoting provides notification functionalities via the EXPLICIT INVOCATION [3] (multicast broadcast) pattern or IMPLICIT INVOCATION [3] (JNDI server binding), in which case the PUBLISH-SUBSCRIBE [3] pattern is used.

Solution	To add automatic detection, a remoting Detector will need to be added on both the client and the server side as well as a NetworkRegistry to the client side. With respect to figure 11[2], when a Detector on the server side is created and started, it will periodically pull from the InvokerRegistry all the server invokers that it has created. The detector will then use the information to publish a detection message containing the locator and subsystems supported by each server invoker. The publishing of this detection message will be either via a multicast broadcast or a binding into a JNDI server. On the client side, the Detector will either receive the multicast broadcast message or poll the JNDI server for detection messages. If the Detector determines a detection message is for a remoting server that just came online it will register it in the NetworkRegistry. The NetworkRegistry houses the detection information for all the discovered remoting servers. The NetworkRegistry will also emit a notification upon any change to this registry of remoting servers. The change to the NetworkRegistry can also be for when a Detector has discovered that a remoting server is no longer available and removes it from the registry.
Rationale	In this variant the Detector object on the server side acts as the central repository to all the client Detector objects that subscribe to it. Both the multicast broadcast and/or the JNDI server binding done by the server detector provides, to all connected clients, timely information regarding available services. The client application is then made aware of the changes and an appropriate reaction can be implemented.
Consequences	1. Since the information concerning services is consumed by all subscribed clients, reusability is improved. 2. Loose coupling of server and client detectors implies better changeability. 3. Clients running on different platforms can subscribe to the server detector, implying enhanced integratability. 4. Since it is possible for a single client to connect to many services and vice versa, adaptability, reliability and availability of the system is improved.

6 Quality Attribute Evaluation

The earlier section provided details on the patterns extracted from the JBoss architecture and their related consequences with respect to the quality attributes. This section gives an overview of how well the quality attributes are addressed by the patterns identified, along with corresponding recommendations. This corresponds to step five of the PBAR method

	Microkernel	Interceptor	Client-Server	Broker	Factory	Active Repository	Dynamic Proxy
Reliability	+	1)	-	-		-	-
Adaptability	+			+	+	+	
Availability		1)	-	-		+	+
Changeability	+	+	+	+	+	+	+
Integratability			+	+	+	+	
Reusability	+		+			+	
Usability	-	-		+			+

Figure 12: JBoss Patterns Vs Quality Attributes Matrix. Note 1) is explained in the appropriate section

The matrix in figure 12 gives a brief summary of the impact (positive/negative) that the patterns have on the quality attributes. The quality attributes given in the matrix correspond to the stakeholder concerns and are presented in the matrix in order of ascending importance. The attributes are discussed in the upcoming sections.

6.1 Reliability

Most of the patterns extracted from the architecture affecting reliability, introduce singular entities on the communication path. The entities include (but are not restricted to) the remote server in the CLIENT-SERVER pattern, the broker element in the BROKER PATTERN, the repository component in the ACTIVE REPOSITORY pattern and the proxy object in the DYNAMIC PROXY pattern. These entities become bottlenecks in the case of overload and can therefore affect the performance of the system. Data throughput and timing issues may occur in that case. Another major reason for decreased reliability is the number of potential indirections that are present in the current system.
The reliability of the system seems to be negatively impacted by the chosen combination of patterns. One of the main reasons that reliability fails is due to the fact that client invocations target a single service on the remote server. If that target service is unavailable, the remote invocation fails. However, clustering of remote services is already provided by the JBoss architecture. Therefore, clustering combined with discovery of services (provided by the ACTIVE REPOSITORY pattern) and network transparency (provided by the BROKER pattern), makes the system more fault tolerant and hence more reliable.
Note 1) Whilst the INTERCEPTOR pattern as such can hurt reliability, specific interceptors(e.g. transaction interceptor), can greatly improve reliability.

6.2 Adaptability

The current JBoss architecture is highly adaptable in a number of system components. These include communication protocols and service discovery mechanisms. This is mainly achieved by the BROKER, FACTORY and ACTIVE REPOSITORY patterns.
It appears that the combination of chosen patterns gives a highly adaptable system. Since the system is highly configurable, it can have a major impact on other quality attributes. For example, smart use and configuration of interceptors, communication and discovery can lead to a more reliable system. E.g it becomes more easy to do failure recovery.

6.3 Availability

Interceptors, remote servers and broker elements are essentially single points of failure. This implies that the patterns that bring these elements into the architecture have an adverse effect on the availability of the system. On the other hand, discovery of services by the ACTIVE REPOSITORY and clustering of services made possible by the DYNAMIC PROXY helps in reducing downtime. Note 1) Availability of the system is partially addressed by the chosen patterns. One of the areas of improvement could be the implementation of specific interceptors.

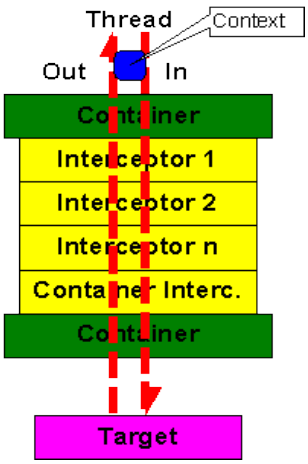

Figure 13: Interceptor Stack

As seen in figure 13 [3], in the JBoss architecture, interceptors are stateless components arranged in a stack, wherein every call proceeds through the stack from first to last. The stack is embedded in a specific container (e.g. MicroContainer or EJB Container). Each interceptor is responsible for invoking the next interceptor in the stack, as seen in the following code snippet,

[3] retrieved from
http://www.onjava.com/pub/a/onjava/2002/07/24/jboss_stack.html

```
import org.jboss.aop.advice.Interceptor;
import org.jboss.aop.joinpoint.Invocation;

public class HelloAOPInterceptor implements Interceptor {

   public Object invoke(Invocation invocation) throws Throwable {
         // PERFORM the INBOUND Tasks and when done go to the next
         // If the call has to be ended throw the appropriate exception

         System.out.print("Hello, ");

         // Call next Interceptor in the stack
         // PERFORM the OUTBOUND Tasks and when done return method
         // If the call shouldn't be complete throw and exception

         return invocation.invokeNext();
    }
}
```

This method of implementation allows for the entire stack to crash when the *invokeNext* method is not reached. This failure could occur in either of the inbound or the outbound tasks. This problem can be solved by implementing an iterator in the container itself. The iterator is responsible for invoking the interceptors in the right order and managing any failures if they occur. In this way, it is assured that the entire stack is executed or the failures (if any) are handled properly.

6.4 Changeability

One of the major advantages of the JBoss architecture is that it can be easily modified. A large number of components can be changed both at compile time as well as at run time. In fact all the patterns identified seem to have a positive impact on changeability.
It seems that the quality attribute of changeability is addressed very well by the chosen patterns. The decision to make the JBoss architecture highly changeable does negatively affect reliability (to some extent). However it remains a good design decision which should not be changed without good reason, because it fulfils an essential need of an application server, which is to be highly configurable.

6.5 Others

Most of the patterns support the integration of individual components and their reusability. However, due to the complexity of the JBoss architecture, usability from the developers point of view has a negative impact with respect to the core components. However the architecture also provides elements that hide unnecessary details implying ease in application development.

7 Recommendations

This section corresponds to the sixth step of the PBAR method. We provide recommendations of where additional, more detailed assessment should be made.

7.1 Negative Impact

Out of all the quality attributes addressed, reliability seems to be the worst affected. As already mentioned this problem can be partially solved by specific configurations of the system. However, more research needs to be done to mine applied patterns which have not been included our findings. The results of this research could eventually bring out patterns which have a positive impact on reliability and help in the analysis to resolve the problem of reliability.
Availability is also negatively impacted by some of the patterns, but we have managed to find patterns which partially solve the problem. Furthur research into the variants of the patterns could yield better results. Furthermore, the heartbeat pattern could possible be used on several places, in order to improve availability.

7.2 Conflicting Impacts

The CLIENT-SERVER and BROKER patterns both have a negative impact on reliability and availability, but they are retained in the architecture because of the benefits they bring to other quality attributes. Variants of these patterns (e.g RELIABLE BROKER) could be used to reduce the negative impacts.

7.3 QAs not addressed

Efficiency is an important requirement in any application server, but this requirement has not been addressed by our research. The reason for this being that more time and resources are required to test specific parts of the system with respect to time behaviour and resource utilisation in a real time setup. Compliance to efficiency standards is another area which has not been investigated by our research.

7.4 Subsystems with no patterns

As JBoss is a huge architecture framework, it is difficult to document every pattern implemented by it. This document presents the components and related patterns that represent the core architecture. Other subsystems not addressed in this document include JBoss RichFaces/Ajax4jsf, JBoss Cache, Hibernate etc. [1]

8 Conclusion

The JBoss Application Server proved to be a great source of architectural patterns. However, since the project is rather large, both in terms of source code and documentation, we had some difficulties to mine the most prominent ones. Overall, we are pleased with the patterns which we discovered. The JBoss team clearly invests a lot of time and resources in improving the architecture and overall quality of the project. Therefore, it was not trivial to come up with obvious improvements to the system. Whilst judging only the discovered QAs, there are some obvious flaws, e.g. like many patterns decreasing reliability, these problems cannot simply be solved by adding more patterns or replacing them, in our opinion. A more thorough understanding of JBoss is needed to make serious recommendations. Overall, the quality of JBoss seems very reasonably and we are curious about upcoming versions and their improvements.

References

[1] http://www.jboss.org/docs/.

[2] http://www.jboss.org/jbossremoting/docs/guide/2.2/html_single/index.html.

[3] P. Avgeriou and U. Zdun. Architectural patterns revisited - a pattern language. In *Proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPLoP 2005)*, July 2005.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, and Peter Sommerlad. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. 1996.

[5] Neil B. Harrison and Paris Avgeriou. Assessing quality requirements through pattern-based architecture reviews.

[6] Neil B. Harrison, Paris Avgeriou, and Uwe Zdun. Using patterns to capture architectural decisions. *IEEE Softw.*, 24(4):38–45, 2007.

[7] Uwe Zdun. Pattern language for the design of aspect languages and aspect composition frameworks. In *IEE Proceedings Software*, 2004.