Edith Hofer

Master's Thesis

Component-based Web Development

Integration of JPF into JBoss

Institute for Genomics and Bioinformatics,
Graz University of Technology
Petersgasse 14, 8010 Graz, Austria
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Zlatko Trajanoski

Supervisor:
Dipl.-Ing. Dr. Gerhard Thallinger

Evaluator:
Univ.-Prof. Dipl.-Ing. Dr. Zlatko Trajanoski

Graz, May 2007

Abstract

German

Für Bio-Wissenschaften werden in zunehmendem Maße Web Applikationen entwickelt, welche die Wissenschafter bei der Verwaltung und Auswertung von Daten unterstützen.

Diese Applikationen beinhalten oft Teile, die auch in anderen Web Applikationen in ähnlicher Form wieder gefunden werden können.

Um Zeit und Entwicklungsaufwand zu sparen, sollten für die Erstellung einer neuen Applikation bereits existierende Komponenten von anderen Applikationen wiederverwendet werden.

Ein Modul, das für die Zusammenstellung einer neuen Applikation benutzt wird, bsteht aus der Business Logik, den Definitionen der Tabellen in der Datenbank und dem Web Interface.

Das Ziel dieser Arbeit ist es herauszufinden, ob dieser Komponenten-basierte Ansatz mit den Technologien, welche auf dem Institut für Genomik und Bioinformatik Verwendung finden, vereinbart werden kann. Die zur Entwicklung von Web Applikationen auf dem Institut verwendeten Technologien sind u.a. AndroMDA, Struts, J2EE, Spring, Hibernate und Tapestry.

Die mit dem Java Plugin Framework (JPF) erstellte Emersion Plattform ermöglicht eine einfache Integration von Komponenten in ein Web-Applikations-System. Deshalb wurde diese Technologie für die Erstellung von Komponenten-basierten Web Applikationen gewählt. Damit diese Plattform in den JBoss Application Server integriert werden konnte, musste diese modifiziert werden. Anschließend wurde getestet, ob kleine Applikationen, welche die Technologien des Instituts verwenden, mit Hilfe der modifizierten Emersion Plattform, zu einer größeren Web Applikation zusammengesetzt werden können.

Stichwörter: Komponenten-basierte Software Entwicklung, JPF, Emersion

Abstract

English

An increasing number of web applications is developed for the research in life sciences. These web applications which support the researcher in management and analysis of data often contain parts which can be found in other web applications in a similar form. To save time and development effort, existing components of other web applications could be reused when a new application is developed. A module which is used for the composition of a new application should consist of the business logic, the database table definitions and the web interface.

The goal of the thesis is to assess the feasibility of this component-based approach in relation to the technologies which are used at the Institute for Genomics and Bioinformatics e.g. AndroMDA, Struts, J2EE, Spring, Hibernate and Tapestry.

The Emersion platform based on the Java Plugin Framework (JPF) provides a simple mechanism for the integration of components into a web application system. Therefore it was chosen as the technology for building component-based web applications. The platform had to be modified so that it could be integrated into the JBoss Application Server. Then it was tested whether small applications using the institute's technologies could be composed to a larger web application with the help of the modified Emersion platform.

Keywords: Component-based Software Development, JPF, Emersion

Contents

List of Figures

Glossary

AOP Aspect Oriented Programming is a programming technique which allows the description of software issues, like logging, which cut across the software system so that the source code can be kept clear of these issues.

API Application Programming Interface describes the methods and variables through which a object can be accessed.

CBSD Component-based Software Development makes it possible to develop a software system by assembling components.

CBWE Component-based Web Engineering enables to build a web application by the composition of components.

COM Component Object Model is a programming model which defines the way objects can interact within an application or between applications.

CORBA Common Object Request Broker Architecture makes it possible that applications which are written in different languages and which are located on different computers can work together.

COTS Commercial-off-the-Shelf Components are software components that can be bought from the vendor who produces them in serial-production.

DCOM Distributed Component Object Model is a protocol that enables program components to communicate via a network.

DNS Domain Name Service resolves Internet domain names into IP numbers.

DS Declarative Services are binded and unbinded to an application via injection.

EAR Enterprise Application Archive is a Java J2EE file format for packing modules like WAR archives into a single archive in order to enable the simultaneous deployment of the modules in an application server.

EJB Enterprise Java Bean is server-side component that encapsulates business logic for modular development of enterprise applications.

ESB Enterprise Service Bus is a communication infrastructure for applications which follow the SOA principle.

GUI Graphical User Interface enables the interaction of users with the application through graphical devices.

HAR Hibernate Archive is a file format used for packing Hibernate classes and Hibernate mapping files.

IDL Interface Definition Language is a language that is used to describe the interface of a software component.

J2EE Java 2 Enterprise Edition is a programming platform for developing and running distributed multi-tier Java applications.

JAAS Java Authorization and Authentication Service is a Java based security framework.

JAR Java Archive is ZIP file type which contains compiled Java classes and the metadata associated with these classes.

JMS Java Messaging Service is an API with methods that allow Java application components to create, send, receive, and read messages.

JMX Java Management Extensions provide tools for the management and monitoring of Java applications, system objects, devices and service oriented networks.

JNDI Java Naming and Directory Interface is an API which enables clients to lookup and discover objects and data via their name.

JPF Java Plugin Framework is a framework for the development of component-based Java applications.

JSP Java Server Pages is a technology which makes it possible that Java code fragments can be embedded in simple HTML code.

JVM Java Virtual Machine is the runtime environment in which Java code is executed.

LDAP Lightweight Directory Access Protocol is a protocol to access directory listings.

MDA Model Driven Architecture disconnects the business and application logic of an application from the underlying platform technology.

MDR Metadata Repository is a database containing metadata.

MOM Message Oriented Middleware is a software which connects applications based on asynchronous messsage exchange.

ORM Object-relational Mapping is the conversion of objects into entries of relational databases and vice versa.

OSGi Open Services Gateway initiative technology offers a component-based, service-oriented environment for the development of applications.

POJO Plain Old Java Object is a simple Java object.

RMI Remote Method Invocation enables Java objects to invoke methods on objects which are located in another JVM.

SAR Service Archive is a JBoss specific JAR file type used to pack services to ensure that these services are started in an early phase of the server start-up process.

SOA Service-Oriented Architecture is a set of services which are able to communicate with each other.

TCP/IP Transfer Control Protocol/Internet Protocol is a collection of protocols which are responsible for the data transfer in the Internet.

UCL Unified Class Loader is a JBoss specific classloader which inherits the URLClassLoader.

UML Unified Modeling Language is a standardized language for the modeling of objects.

WAR Web Application Archive is a JAR file type which contains web modules.

XML Extensible Markup Language is an open standard of the World Wide Web Consortium (W3C) designed as a data format for structured document interchange on the web.

Chapter 1

Introduction

An increasing number of web applications is developed for the research in life sciences. These web applications support the researcher in management and analysis of data. They often contain parts which can be found in other web applications in a similar form.

For instance these applications usually have a "User Management" part which handles the user data, grants access permissions and controls the authentication mechanism. Often web applications have a "Publications" part to manage documents like papers or theses with contents related to the application's domain. Additionally there are often modules like "Sample preparation" or "Sample description" which allow the user to process data obtained by biological experiments via a web application.

Usually such modules are developed from the scratch for each new application. This is bad practice because the developer cannot focus her attention on the actual function of the application but has to spend time and effort on implementing parts of applications that have been implemented before.

To save time and development effort, existing parts of other web applications could be reused when a new application is developed. Existing components or modules which cover one of the common areas and newly developed modules could be combined to a new application.

A module which is used for the composition of a new application should consist of the business logic, the database table definitions and the web interface. It should be possible to integrate all those three parts of the module unmodified into the new application just like a library is integrated into conventional applications.

A library integrated into a conventional application contains only classes that can be used by the application whereas in the component-based approach, a module also consists of database tables and the web interface. When a module is integrated into an application, the integration of the web interface or the definition of relationships between database tables of different plugins may cause problems.

1.1 Objectives

The goal of the thesis is to assess the feasibility of this component-based approach in relation to the technologies which are used at the Institute for Genomics and Bioinformatics. These technologies include AndroMDA, Struts, J2EE, Spring, Hibernate and Tapestry.

The specific goals of this thesis are:

1. Literature and the Internet should be searched a suitable technology to build a web application by composing existing modules with newly developed ones.
2. The technology which was found during the research phase should be tested whether it can interoperate with the institute's technologies mentioned above.

Chapter 2

Background

This chapter provides an overview of component-based software development. Definitions of components are given and the advantages and disadvantages of component-based software development are listed. Moreover component technologies are introduced briefly. The last part of the chapter deals with advanced technologies to build component-based web applications.

2.1 Component-based Software Development (CBSD)

Gaedke [1] gives us a simple, yet significant definition of component-based software development:

> ”CBSD aims at assembling large software systems from previously developed components (which in turn can be constructed from other components).”

component-based software development has been an issue in software engineering since this discipline was first introduced at the first NATO Software Engineering Conference in Garmisch Partenkirchen in Germany in 1968. CBSD was an approach to overcome the software crisis. Krueger [2] explains the software crisis as

> ”the problem of building large, reliable software systems in a reliable, cost-effective way”.

In his white paper, which was also presented at the NATO Software Engineering Conference, McIlroy [3] states that the software industry, which at

that time was less industrialized than the hardware branch, would benefit from a software components sub-industry.
According to his paper such a software components sub-industry would ensure that a developer gets standard software components which fit her needs concerning performance, precision etc.
Those components should have a high quality and robustness. Further they should be built in such a way that they go together with other components. Thus relieving developers of larger systems of the need to care about how the components work and enable them to concentrate on their real tasks.
As areas for reusable components McIlroy suggested among others, text processing, two and three dimensional geometry and storage management.

Years later the Internet entered the computing world and though the world wide web was originally a medium for the distribution of information in a document-centric form, with the development of web technologies software development for the Internet became a huge branch of the software engineering discipline. Of course component-based software development also became an issue for developers of web applications. Gaedke defines Component-Based Web Engineering (CBWE) [1] as

> "the cost-effective production of (high quality) web applications using a defined process that includes systematic reuse of components and domain knowledge."

2.1.1 Components

The term "software component" is not clearly defined. Szyperski gives us two definitions [4]:

> "Software components are binary units of independent production, acquisition and deployment that interact to form a functioning system."

and

> "A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed separately and is subject to composition by third parties."

Usually a component has three attributes [5]:

- a component is an element of a system that is nearly independent and replaceable with a non-trivial, encapsulated functionality
- a component accomplishes its function in the context of an accurately defined architecture
- documented and well-defined interfaces are used to access a component.

Generally speaking, there are two types of components. Components which were explicitly developed to be reused and components which were not explicitly developed for reuse, for instance complete, stand-alone applications, but which are reused nevertheless.
The first type can be divided into two subtypes: Commercial-off-the-Shelf (COTS) components which are produced commercially by a third-party vendor. They can be bought ”off the shelf” from this vendor and then integrated into a system without any modification.
The second subtype are components whose interfaces can be adjusted to existing systems or other components.
Components can be fine-grained as well as coarse-grained. Fine-grained components may be visual elements such as a button or a label whereas coarse-grained components can represent entire applications [6].
When a component is too small or too trivial, its benefits may not be worth the management costs of applying component-based design. On contrary when the component is too large or too complex it is difficult to maintain high quality [7].

2.1.2 Component-based Software Development

Graef's [8] definition of component-based software development:

> ”A fundamental requirement for the automatic adaption of applications in a flexible and evolution oriented way is the availability of all necessary functionality and features as independent building blocks or components.”

With component-based development it is possible to divide large and complex software systems into smaller, less complex modules. Since these modules have an encapsulated function they can be decoupled from each other and thus be implemented in parallel by different developers, regardless of location, independently of each other's work. Thus development time is reduced.

Moreover component-based technology promises higher quality because the specific components can be tested individually with unit tests and the component assembly has to be tested as well with integration tests [9].
However in general the testing effort is reduced because of the unit tests of each component.
Another advantage is that software systems which consist of several components are more flexible than monolithic software systems. If a software system has to fulfill new requirements which were not anticipated, it is easier to recompose the components than to modify a monolithic system [10].
Component compositions are often easier to maintain because each component can be maintained independently.
Developers of component-based software can focus more on the specific context of the application. They do not need to worry how and whether the components work because usually the components themselves have been developed by experts. Thus the developers can concentrate on more complex requirements related to the specific field of the application [11].
Two of the main reasons why developers write component-based applications are the extensibility and the reusability.

Reuse

Frakes' definition of software reuse [12]:

> "Software reuse is the use of existing software or software knowledge to construct new software."

For Gaedke [1] the main target of component-based web development is the cost-effective, systematic reuse of software components. Software reuse is desirable because many systems are not unique, share common elements and differ in confined areas only.
Although software systems tend to become bigger and more complex, customers expect low development costs, less development time, and high reliability.
Software reuse helps to achieve these goals. It allows efficient software development with improved productivity and quality.
The reliability is increased because components which have been tested thoroughly and worked well for one system may be reused in another system. This leads to a higher quality of the composed application [11].
Instead of developing a component from the scratch, an already existing element with the same functionality can be used and thus time is saved. This also accelerates development of new software. Due to reuse less code has to

be written. The saved time and effort can be used to improve other aspects of the software such as robustness and correctness [13].

But there are some drawbacks developers have to face when reusing software. Very often it is difficult to find and to select the right component with the desired functionality. Higher maintenance costs may occur and the module may become incompatible with new technologies.
The software may be bloated unnecessarily because single components can have functions which are not used by the composed system.
Software reuse is often made difficult due to insufficient documentation or bad programming style. Furthermore abstractions for large and complex software components are usually very complex. Developers who want to reuse this components either have to know the abstractions a priori or they must invest time to study the abstraction [14].
Sophie Ramel [15] classifies the types of reuse into two categories: conceptual reuse, which is a reuse of ideas, and program reuse.

- Conceptual reuse
  - Reuse of models
  - Reuse of architectures
- Program reuse
  - Reuse of frameworks
  - Reuse of code
  - Reuse of components

Conceptual Reuse This type of reuse is also called design reuse. There are three main reasons that speak for design reuse. Since one of the earliest phases of software development is the design of the system, many errors that affect the whole development process can be eliminated beforehand in this early phase. Moreover, the system may be easier to understand and thus easier to develop and to maintain if a familiar design is reused.
Due to reuse of design code reuse is often supported because code that is used for an application with a certain design may also be useful for another application with the same design [16].

An example for reuse of models are so called ”design patterns”. In programming, especially in object-oriented programming, there are design tasks which occur very often and thus have to be implemented very often. A design

pattern describes a very basic solution to such a recurring problem. A core solution can be adapted to varying forms of the problem. It may be used over and over again as long as the core problem corresponds to the description of the design pattern [17].
The main advantage of design patterns is they provide solutions which have been developed and permanently improved by many experienced developers over a larger period of time. This means that the solution has been optimized and is definitely not the solution one would come up with when looking at the problem for the very first time.

Reuse of architectures is the reuse of software at the architectural level. Architecture reuse can be realized with Model Driven Architecture (MDA). The developer can define an architecture of an application which may be reused even on other platforms. The implementation is generated automatically from the application model [14]. An example for an MDA framework is AndroMDA which will be presented in Section 3.4.

Program Reuse According to Ramel [15], a framework provides structures which a developer may reuse in her programs. These structures provide certain functionalities like building a web page or the management of database access. An advantage is that different functionalities are combined in a single framework.
A framework provides standard interfaces and additional configuration files. Examples for frameworks used in web based applications are Spring and Apache Struts.

Code reuse is what every developer does from time to time: She copies a fragment of her previously written code, be it a few lines or a whole class, to insert it into her own code. The inserted code, usually just a small amount, is often modified to suit the needs of the new program. With this technique the developer often reuses her own code and less frequently the code of another developer. This type of reuse is generally not recommended because it is complicated to transfer changes in the original code to the inserted code of the new program and vice versa.

Reuse of Components New software systems may be composed by reusing already existing software components.
Krueger [2] observes that a software reuse technique is only effective when reusing the components is easier than writing new components.
That means that a suitable component has to be found quickly. Therefore

components may be stored in repositories, so called component catalogues, with detailed descriptions of what the component does and sophisticated search functions. Thus enabling easy and fast finding of components which fit the developer's needs. Additionally it has to be just as easy and fast to integrate the component into an existing system.
According to Monroe [16] the reuse of implementation-level code can enhance the economics of software development. However, there are limitations to code reuse. For instance it is not always clear under which circumstances (e.g. expected interaction protocol, scheduling constraints, etc.) the component is intended to work.
In order to invite reuse, a component must be well-tested, efficient and well-documented. Moreover it should be reusable in different contexts. A component's quality regarding software reuse can be determined from the following characteristics [11]:

- **Portability**: the ease of adaption to a new system and the ease of installation in a new system.
- **Applicability**: the controllability and understandability of a component as well as the ease of learning of a component.
- **Efficiency** regarding space, time and resources.
- **Functionality**: the adequacy, interoperability and accuracy of the component's functions.
- **Reliability**: the error tolerance, error frequency and traceability of errors in a component.
- **Maintainability**: the changeability, stability, testability of a component and up to which extend it is possible to analyze the component.

Another indicator of the quality of the component may be the reuse frequency. The number of applications in which the component is used is a reliable sign of its usefulness.
There are four steps to be accomplished until a software component can be successfully reused [2]:

- **Abstraction**
  Usually software components cannot be reused exactly the same way as they were used in another system. Thus an abstraction that removes details that were only necessary for a certain system and that emphasizes on important information is necessary to reuse a component in a variety of contexts.

- **Selection**
  Software developers have to locate, compare and select reusable software components.

- **Specialization**
  The abstract component is specialized via parameters, transformations, constraints or any other kind of refinement to meet the requirements of the new application.

- **Integration**
  The components are put together forming a new, complete software system. The integration is done via component technologies, module interconnection languages or by calling functions or methods of the component. Therefore the API (Application Programming Interface) of the component can be used directly or a standard API which the component implements may be used.

  There are often obstacles which obstruct the composition of components. Integration is difficult if the vocabulary that is used in the component is different than the vocabulary used in the application. To deal with that problem ontologies may be used.

  It can be difficult to integrate the component into the business process of the whole application. Sometimes even the business process has to be changed so that the component fits. Moreover incompatible technologies can cause severe problems.

  When the architectures of the component and the application are too different it is extremely difficult to integrate the component into the application [15].

Black Box Reuse vs. White Box Reuse Black box reuse [15] means that the developer who assembles the components has no knowledge about how the component is working internally. To be able to add such a component to a software system and thus to reuse its implementation, the component needs to have well-documented interfaces.
Commercial-off-the shelf components are examples for black box components. Unfortunately COTS components often do not follow standard interfaces which makes it very difficult to integrate them into an application.

The white box approach denotes that the developer knows how the component works internally. The programmer has access to the source code and its methods and functions. If they do not suit her needs the developer may modify them. The disadvantage of being able to modify the component is the modification makes it difficult to update or maintain the component.

Extensibility

An application that is component-based is usually easier to extend than a monolithic piece of software. Extensibility is very important because an application is never really finished. There are always improvements to make and new features to implement. Software can be modified more easily if it is designed to be extensible.
Building a new software system is less effort if the base system is extensible [13]. Extensibility enables software developers to develop variants of a system by taking a base system, which shares a common structure and functionality with other systems, and equipping this base system with possibly different components.

Extensibility Requirements There are certain prerequisites which make construction, deployment and evolution of components of an extensible system easier.

- To prepare the system and components for programming in the large, the mechanism for the composition and the evolution of the components have to scale well.
- As little as possible adaption code should be needed when a component is reused in a different context.
- A component should be extensible even if not all potential future extensions are known.
- The coexistence of different versions of a component is required and can be achieved through a versioning mechanism.

Zenger [13] differentiates between three extensibility mechanisms: white box extensibility, gray box extensibility and black box extensibility.

White Box Extensibility The extension of the software system takes place through modification or addition to the source code. It is the most flexible and least restrictive kind of extensibility. White box extensibility can be classified into two types: open box extensibility and glass box extensibility.

To apply open box extensibility the developer must have access to the source code because the modifications are directly inserted into the original source code.
Changing the source code, particularly if the code was written by someone else and if the code and the functions of the system is not well-understood, is prone to errors and exhausting.
Open box extensibility is usually applied when a development team is fixing bugs, refactoring internal code or producing the next version of its own software system. Open box extensibility is also often used when programmers try to derive a variation of an existing piece of open source software.
They then copy the source code and apply the changes to the copied code. This way of extending software simplifies the development of a new member of a software family but it makes maintenance more difficult. For instance if code in the original software is modified, e.g. a bug is fixed, then of course it is desirable that this modification, which leads to an improved system, is also applied to the extended system.
Glass box extensibility is applied when is the source code is available and the developer can study the code.
Unlike in open box extensibility in glass box extensibility the developer does not modify the source code. She separates her modifications from the original system so that the original system is not affected. Since the extensions are well-separated from the original system both the extensions and the original system are easier to understand and maintain. It is also possible to combine modified versions of the original system with extensions that were originally developed for the unmodified original system.
Object-oriented frameworks are an example of glass box extensibility.

Gray Box Extensibility The compromise between the white box approach and the black box approach is called gray box approach. Contrary to the white box extensibility, in this approach the source code is not available but the binary of the code is available.
Nevertheless a system can be extended in a similar way to the glass box approach. An abstract documentation of the system's interface needs to be provided. Of course this is not a simple interface but a specialization interface which

> "lists all the abstractions that are available for refinement and specifies how extensions interact with the original system [13]."

As a mixture of the white box and black box extensibility the source code is not fully revealed nonetheless details about the implementation of the original system and the extensions are provided.

Black Box Extensibility In the black box approach the software is completely encapsulated and the implementation details are hidden. The developer has no access to internal details of the system's source code or the architecture of the system. Deployment and extension of the system is possible only through the specification of the system's interface.
The mechanism which allows extension is directly implemented in the system, respectively the mechanism is part of the design. The design of such an extensibility mechanism requires that all imaginable extension scenarios are taken into consideration. Since it is often impossible to consider all possible future scenarios this extensibility type is more limited than the white box extensibility.
An advantage of black box extensibility is the ease of extending such a system because it is not necessary to know internal details of the application. System configuration applications or application specific scripting languages are used to extend such a black box system. When it comes to object-oriented frameworks, interfaces of components are used to support the black box approach. Via object composition these components can then be added to the framework. Black box extensibility is usually applied when a development team develops proprietary components or frameworks whose source code must not be made public and when additionally other developers nevertheless should have the possibility to customize the system or to extend the functionality of the system.

2.1.3 Component Technologies

One of the first approaches to realize component-based programming was object-oriented programming. A complex application is divided into smaller parts, the objects. Object-oriented programming supports reuse. For instance classes may be stored in class libraries (reuse libraries) [10].

But object-oriented programming could not satisfy the programmer's needs completely because a single class often is not as useful and the complex relationships between classes make it impossible to develop and test the objects independently of each other [18].
An advanced approach of component-based software development are component technologies which are based on object-oriented programming. Component technologies are technical utilities (frameworks, middleware, architectures, etc.) that were developed to improve reuse [14]. Examples for component technologies are:

CORBA

CORBA is short for Common Object Request Broker Architecture and was initially released in 1991. It is an open standard for application interoperability.
CORBA manages component interoperability. Due to CORBA, applications can communicate with each other across different programming languages, implementations, locations and platforms.
CORBA is a so called middleware or integration software that can be integrated with other technologies like J2EE (Java 2 Enterprise Edition) or COM (Component Object Model).
In the CORBA specification of 2004 [19] it is stated that

> "the object request broker is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request and to communicate the data making up the request."

The developer defines the interface of those classes that want to make their methods available for remote access with an Interface Definition Language (IDL).
An IDL compiler creates classes which do not contain any implementation. From these classes the developer can then derive her own classes and implement them. When another application wants to access an object, its developer gets the IDL interface, the IDL compiler creates the classes in the adequate programming language and the programmer may make a remote method call on one of the objects of the other application. That is the client can send a request containing an operation, a target object, and parameters which can be used to pass data to the target object. The name of the method to be invoked and the parameters of the request are stored in a binary buffer and the content of the buffer is sent to the server process which owns the target object [20].

COM

Microsoft's Component Object Model (COM) describes how the components interact with each other. COM depends on the windows platform but is language independent. According to the COM specifications [21]

> "COM is used by developers to create re-usable software components, link components together to build applications, and take advantage of Windows services."

COM is a binary standard which facilitates reuse of written code. A COM component has to have a well-defined interface which can be used to apply the functions of the component.
The interface is basically a pointer to a table with all the functions a COM component provides. The COM technologies comprise COM, COM+, DCOM, and ActiveX controls.
Today a lot of parts of the COM platform have been replaced by the Microsoft .NET initiative [4].

Java Beans and Enterprise Java Beans

In 1996 Sun Microsystems presented its answer to the other component technologies: the Java Beans. They support the composition of an application out of smaller components. It is a platform neutral component architecture for the Java 2 Platform. Sun's web site [22] gives the following definition:

> ”Components (Java Beans) are reusable software programs that you can develop and assemble easily to create sophisticated applications.”

Instances of such beans may be assembled at design time by a tool, for example an application builder and the composition then works like a single application at runtime [4].
A Java Bean may contain many objects but it can be treated like a single object. Java Beans are intended for reusable client-side component development. A Java Bean can be a simple GUI element, e.g. a button, a more complex visual component or even an element without a visual appearance of its own at all [23].
Enterprise Java Beans (EJBs) are intended for reusable server-side component development. They encapsulate an application's database access and business logic.

2.1.4 Service-Oriented Architecture (SOA)

Service-Oriented Architecture is an evolution of component-based development and its technologies like COM, CORBA and J2EE [24].
The Service-Oriented Architecture (SOA) is described by Ortiz [25]:

> ”An SOA enables flexible connectivity and communication among applications by representing each as a service with an interface that lets them communicate readily with each other.”

A service in this context has an explicit, implementation independent interface and encapsulates a reusable business function. With SOA it is possible to integrate existing systems, applications and users in a flexible architecture that can be easily adjusted to changing requirements. Three roles determine the service-oriented architecture:

- Service Provider provides a description of the service and publishes the service in the Service Broker so that it is available to other services
- Service Requestor searches for a service in the service descriptions of the Service Broker and connects to the service when an appropriate service has been found
- Service Broker contains a registry with service descriptions and with the help of the registry it links the service requestor to the service provider.

The benefits of the use of a service-oriented architecture include interoperability, efficiency and standardization. SOA supports interoperability because it enables applications to share data and functionality even if they run on different systems. Using a Service Oriented Architecture is efficient because with SOA existing applications may be reused [26].

Web Services are a concrete implementation of the SOA principles. Web Services are software components which run on different servers and which are loosely coupled to form a single application [27].
They make their functions available to other applications or services via application interfaces and an industry standard network. There are other ways to implement SOA, e.g. by using CORBA or COM Technologies.

2.2 Spring OSGi

The Spring OSGi framework provides a way of building component-based applications following the Service Oriented Architecture.
It is based on the Spring Application Framework, an open source, layered Java / J2EE application framework [28].
The main objective of Spring is to provide an effective way of managing the business objects of an application. It intends to facilitate the use of J2EE and provides a consistent configuration handling throughout applications.
The Spring Framework makes unit testing of applications easy. Moreover it allows the user to build applications using Plain Old Java Objects (POJOs)

instead of using exclusively EJBs [29, 30].

Spring OSGi makes it possible to develop Spring applications which run in an OSGi framework. OSGi[31] stands for Open Services Gateway initiative and is a technology which was developed by the OSGi Alliance, which is a consortium of technology innovators. It is a dynamic module system for Java and provides

> "a service-oriented, component-based environment for developers and offers standardized ways to manage the software life cycle" [32].

The OSGi technology makes it possible to compose large applications from small, reusable and collaborative components [31].
This technology in cooperation with Spring facilitates component-based development and allows a dynamic service model. It provides a simple programming model for developers of enterprise applications that want to benefit from the features of the OSGi platform. Spring OSGi provides the following advantages:

- Spring facilitates the splitting of application logic into modules.
- Several versions of a module can be deployed at the same time.
- Modules are able to automatically find and use services which other components in the system provide.
- Modules can be dynamically deployed, updated and removed from the running system.
- The Spring Framework is responsible for the instantiation, configuration, composition and decoration of components within and across modules.

In Spring OSGi the Spring-based application logic is packed as bundles. A bundle contains one Spring application context and a Spring-based application may consist of multiple bundles.
A bundle is by default completely protected. Classes within the bundle can neither be accessed with reflection nor with classloading experiments. A bundle is able to export packages containing Spring beans. Only these exported beans are visible to other bundles. This visibility constraints help to avoid unintended coupling between components and make independent development of modules possible [33, 34]. A bundle also has to define on which

other bundles it depends [35]. A bundle can also export services e.g. logging service, that are then available to other bundles. Additionally OSGi services may be imported into bundles.

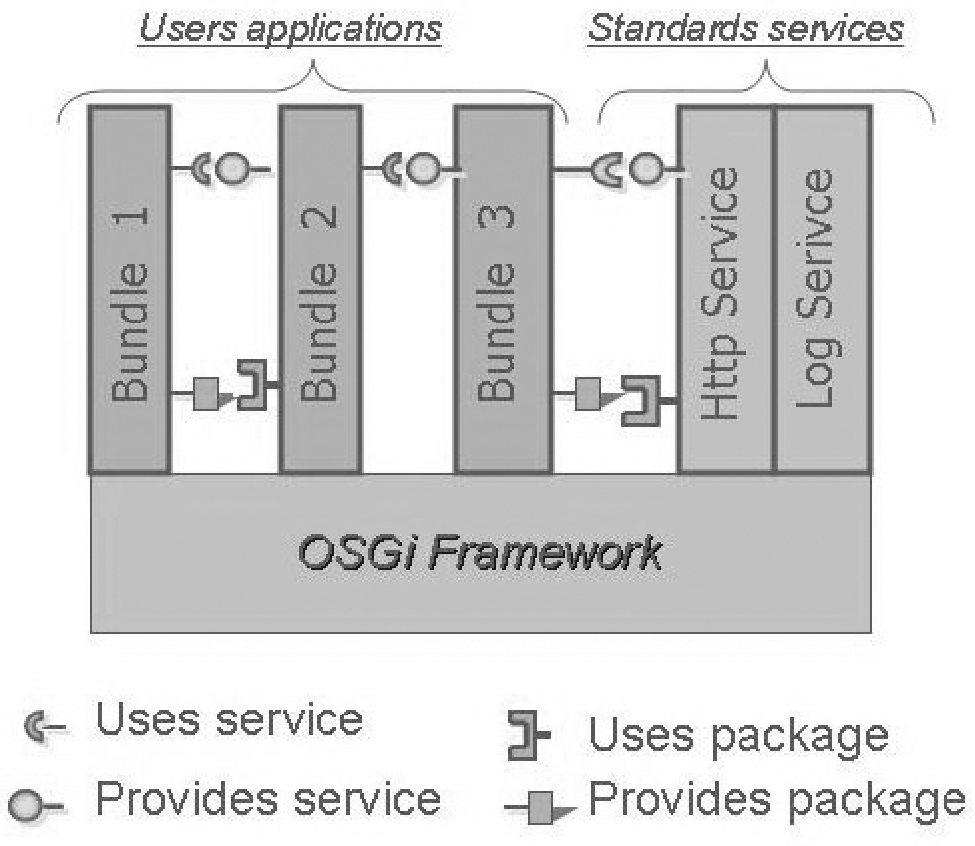


Figure 2.1: The OSGi Architecture [36]: An OSGi application consists of bundles which can export and import packages and services.

An OSGi service is an object with a public interface and it may have a set of queryable properties. Such a service is registered in the OSGi Service Registry. The OSGi Service Registry contains all the services which should be visible across the system. Bundles can dynamically register and unregister their services in the Service Registry and may ask the Service Registry for a service and use this service. The standard Spring proxy mechanism deals with the dynamically added and removed services.
Applications which are developed with Spring OSGi need to find a way to cope with dynamic services. The Declarative Services specification (DS) is a mechanism to deal with the dynamic services. DS is a model which facilitates

writing OSGi services because it takes care of the registration of services and the dependencies between services of an application [34].

The main goal of Spring OSGi is the development of enterprise applications and among those, the development of web applications. The Spring OSGi developer team promises that in the future it will be possible to easily implement and deploy Spring web applications in an underlying OSGi infrastructure. Currently Spring OSGi is tested with web applications using the Equinox Incubator OSGi provider because which is the only OSGi container that supports web applications [32]. There are plans to extend this to other web containers like Tomcat, WebSphere, WebLogic or any other application server since an OSGi container can be embedded into another container [33].

2.3 Enterprise Service Bus Architecture

The Enterprise Service Bus Architecture provides an infrastructure for applications that are developed using the Service Oriented Architecture approach described in Section 2.1.4.
Keen et al. [37] define the Enterprise Service Bus

> "as providing a set of infrastructure capabilities, implemented by middleware technology, that enable the integration of services in a SOA."

The Enterprise Service Bus is a concept for such an infrastructure. It allows the developer to compose a heterogeneous application [37].
With ESB software, applications which run on different platforms, are written in different programming languages, are using different programming models, different data formats or programming interface can communicate through a connectivity infrastructure.

To support the principles of SOA, the following integration paradigms must be realized by the Enterprise Service in one infrastructure [37]:

- Service-oriented architectures: The communication between the modules is carried out through well-defined, explicit interfaces and uses underlying message and event communication models.
- Message-driven architectures: The communication between the applications takes place via message exchange.

- Event-driven architectures: In order to communicate, the sender triggers the message independently and the receiver processes the message independently.

The ESB works in a similar way as a hardware bus. Data is sent along a common pipe to which all the applications are connected.

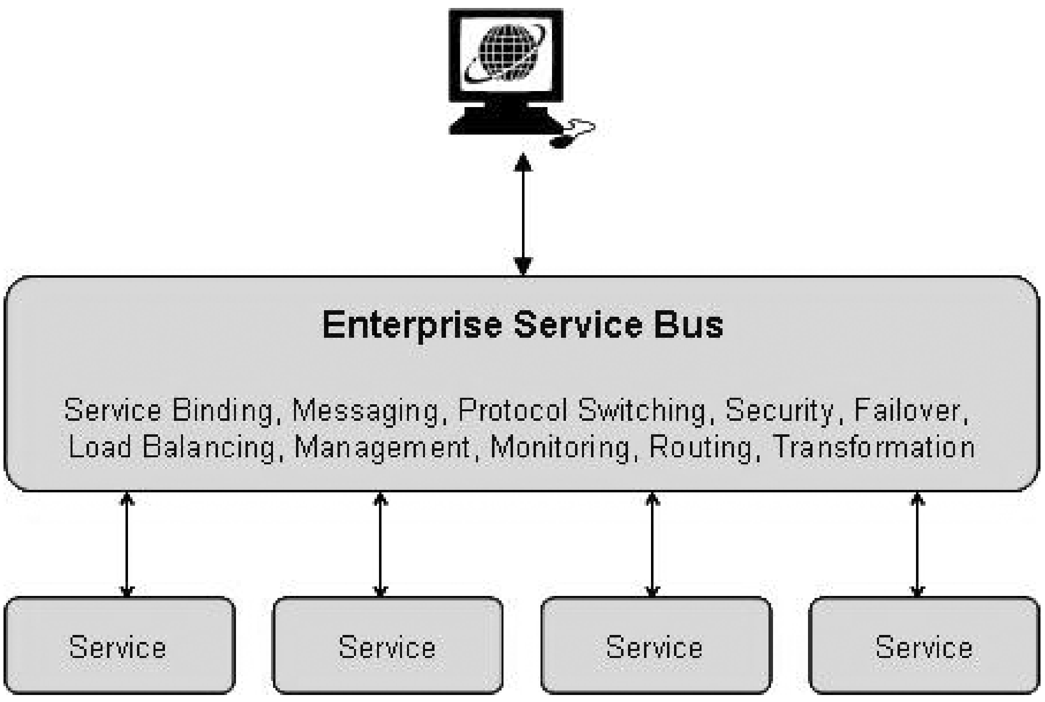

Enterprise Service Bus
Service Binding, Messaging, Protocol Switching, Security, Failover, Load Balancing, Management, Monitoring, Routing, Transformation
Service
Service
Service
Service

Figure 2.2: The ESB Architecture [38]: Applications are connected to a common pipe through which the data is sent and which takes care of routing, transformation etc.

First, the sending application tries to access the receiving application via a locater and then it sends the message. The Enterprise Service Bus is responsible for transforming the data formats as well as for routing, acceptance and processing of messages [25]. Transformations and routing are carried out decentralized before the bus is given the messages to deliver them. The applications can communicate with each other through the exchange of messages in a loosely coupled network. For this asynchronous messaging a Message Oriented Middleware (MOM) like Java Messaging Service (JMS) is necessary.
An ESB has to support Web Services, whose advantage is their interfaces specified in a special XML dialect and are thus independent of platform and programming language.

Moreover, an Enterprise Service Bus should be able to integrate systems and applications that run on different geographical locations. Properties of more advanced ESB implementations are security with encryption and authentication, robustness, i.e. reliability, fault tolerance, business process management, work flow, business rules and content based routing [39].
There are several advantages of ESB. It configures the services when applications are integrated in such a way that they can be reused whenever necessary. Since the Enterprise Service Bus architecture enables a standardized and automated way to handle the communication between applications, it is easy to extend and compose existing services. The architecture of ESB is decentralized, which means that extensions to a system may be added whenever and wherever needed.
A disadvantage of the ESB that it usually takes some time to learn how to use an ESB optimally. This means that a return on investment must not be taken for granted for the first few projects.
Breitling [40] draws the conclusion that ESB is interesting for a whole enterprise but for smaller applications it is not reasonable to use such an infrastructure.

Commercial vendors of ESB implementations are for instance IBM's WebSphere ESB [41], Oracle ESB [42] and BEA AquaLogic® Service Bus [43]. There are also open source ESBs available, for instance Sun's Open ESB or JBoss ESB (see Section 6.1).

2.4 Java Plugin Framework (JPF)

The Java Plugin Framework enables a developer to implement component-based applications. Unlike Spring OSGi and ESB, JPF is not based on Service Oriented Architecture.
The Java Plugin Framework (JPF) [44] is an open source project which provides an environment for building component-based Java applications, which uses a similar plugin framework approach as Eclipse 2.x [45].
With JPF any kind of Java application can be developed, be it a simple GUI application, a command line tool or a J2EE application.

2.4.1 The Main Features of JPF

- JPF makes it possible to develop an application as a composition of modules.
- An application developed with JPF is easily extendable.

- JPF facilitates the reuse of code. Code and resources may be shared among different applications.
- It is possible to define dependencies between modules.
- JPF allows "Hot Deployment": plugins my be added to the JPF system or removed from the JPF system during runtime.
- JPF provides an integrity check which tests the registered plugins for consistency when JPF is started.
- There is no predefined structure for applications built with JPF. Developers can organize their plugins as they want.

An application developed with JPF usually consists of several plugins. A plugin has a name and a version identifier and contains code and resources. It provides a well-defined import interface which declares all the plugins on which this plugin depends on. It also provides a well-defined export interface which declares which code and resources may be used by other plugins. Such a plugin also has so called "extension points", which indicate where other plugins may be plugged in.

2.4.2 Emersion

Emersion [46] is a platform based on the Java Plugin Framework. Its purpose is to develop and deploy server-side applications and web sites. These applications can be component-based. In order to be able to execute Java Servlets and Java Server Pages, the Tomcat web container is embedded into Emersion.

Chapter 3

Methods

JPF and the Emersion platform were chosen to assess the component-based development approach of web applications. In order for these technologies to be used at the institute they have to work with JBoss. To validate the approach, it was tested with AndroMDA and Hibernate. All these technologies are explained in this chapter in more detail.

3.1 JPF

The following sections show how a module can be added to a JPF-based application and which tools are provided by JPF. Moreover the most important components of JPF are introduced.

3.1.1 Plugin Manifest

A plugin may consist of Java code and resources. Resources can be all types of files, e.g. images, XML files, text files, etc. A plugin has an ID, a version identifier, an import interface, an export interface and a interface where it can be extended.
An application that is built with JPF is usually composed of several of those plugins. To add a plugin to such an application it must be connected to an extension point of an existing plugin. An extension point is a JPF feature that makes an application extensible. A plugin can have zero, one or more extension points, according to the architecture of the application. JPF is responsible for automatically discovering and loading these plugins.
To make a plugin visible to the JPF system, a XML file called *plugin manifest* has to be inserted into the plugin folder. The plugin manifest contains all necessary information to enable JPF to make all the plugins work together

resulting in a consistent application for the user. The plugin manifest file is described in more detail in Section 6.2.

3.1.2 The JPF System

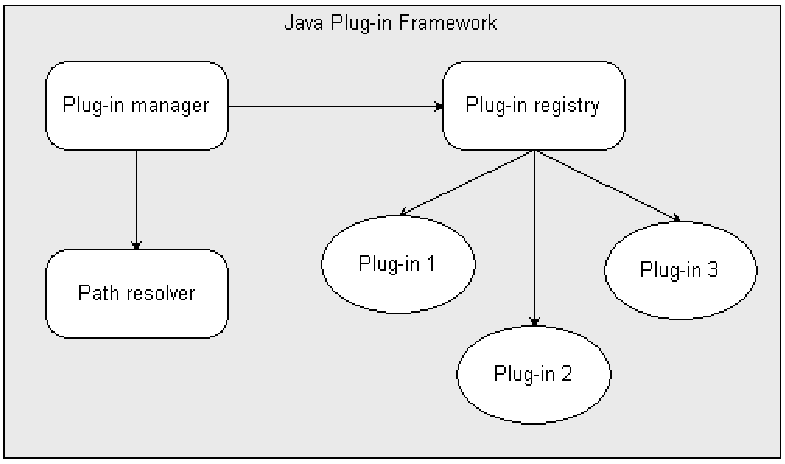


Figure 3.1: Java Plugin Framework [44]: All the plugins are registered in the plugin registry. The plugin manager locates the code of the plugins with the help of the path resolver and then loads the code when the client requests it.

Three objects are the backbone of JPF: the plugin registry, the plugin manager and the plugin class loader.

Plugin Registry

When the application is started, all the plugins that are discovered by JPF are added to the plugin registry. This means the information provided by the plugin manifest is stored in the plugin registry. Plugins may also be registered or unregistered at runtime which is known as *hot deployment*.

Plugin Manager

"The plugin manager is the runtime system of the framework [44]".
Usually only one instance of such a plugin manager exists in an application. The most important task of the plugin manager is loading the plugin code when the client requests it. If an entry class to the plugin is specified, the manager also calls this class. Furthermore the plugin manager is responsible for managing the dependencies between plugins.

Plugin Class Loader

The class loader used in JPF is a subclass of `java.net.URLClassLoader`. The classloading mechanism of JPF is also known as *scoped classloading*. This means for every plugin a custom classloader is created and all classes and resources are loaded with this classloader. The main responsibility of the plugin classloader is to add the code and resources belonging to the plugin, which were specified in the manifest XML file, to the classpath. That is, all files and folders declared in a `<library>` tag within the plugin manifest are appended. One of the main characteristics of JBoss is the ability to specify other plugins in the plugin manifest on which the current plugin depends. The plugin can see the code and resources of those plugins. When the plugin classloader is about to load a class that is in one of those imported plugins, it delegates this work to the classloader of the package the class belongs to. The classpath of the current plugin is extended to include the classpath of the imported plugin. In fact, the current plugin does not need to import any plugin which one of its other imported plugins or the imported plugins of those plugins has already imported. The classloading will be delegated to the classloader of the plugin which provides the requested code or resources. To make this more clear, the developer of JPF gives the following explanation [44]:

> ”Lets say a plugin PluginA introduces a classA (this class is included in a plugin directory hierarchy described by a JPF plugin manifest). Now you are developing another plugin, PluginB, and add another class, classB, to this plugin. You want to reference classA in your classB code, so you need to declare a plugin dependency. You can do this by making an entry in the JPF manifest of the plugin PluginB that says ”PluginB depends on PluginA”. This is done in the prereqisites/imports section of the JPF manifest. JPF handles finding and loading classA when it is first called. The magic lies in the classloaders created by JPF. They extend the classpath of PluginB so that it includes the classpath of PluginA. So the developer doesn’t have to worry about finding classes and can use the basic code that follows in classB:
>
> No further work is necessary to make ClassA visible for ClassB code.”

However at compile time it is necessary that ClassA is in the classpath of ClassB.

```java
import ClassA;

public void ClassB(){

 public ClassB() {
 }

 public static void main(String [] args) {
   ClassA clsA = new ClassA();
 }
}
```

Listing 3.1: Even if ClassA is in another plugin than ClassB it can be simply imported and used by ClassB

3.1.3 Tools

The JPF provides several tools that make the developer's life easier.

- **Integrity Check Tool:**
  This Ant [47] task checks whether the plugin and its manifest are consistent. For instance, if all the files that are mentioned in the manifest exist in the plugin.

- **Documentation Tool:**
  In JavaDoc [48] manner, this tool automatically creates the documentation of the plugins.

- **Plugin Archive Tool**
  A *plugin archive* is a file in ZIP format which contains all packed plugins and special descriptors. These descriptors enable the fast extraction of meta data of plugins without unpacking the entire archive.

- **Single File Plugin Tool** This tool makes it possible to package every plugin and every plugin fragment of a plugins collection into a single ZIP archive file.

- **Manifest Info Tool**
  The job of this tool is to read data from the plugin manifest and to add this data to the properties of the application.

- **Version Update Tool**
  With this tool, the version number and optionally the version name will be increased if the plugin has been modified since the last time this task was called.

3.1.4 Boot Library

To start a JPF-based application a boot procedure is necessary. The application developer may write its own boot procedure where he has to implement a few basics to enable JPF to run. A simpler and faster way is to use the boot library written by the JPF developer and packed in `jpf-boot.jar`.
First the boot procedure loads the system properties and the application configuration which is located in the `boot.properties` file. This properties file allows to set several parameters of the JPF system. For instance, the user may specify which implementations of the application initializer or the error handler she wants to use.
If no special configuration is needed, the only parameter the developer has to set is `org.java.plugin.boot.applicationPlugin`. This is the id of the plugin that is used to start the modularized application. It is the entry point to the application.
The next steps in the boot procedure is the creation of an instance of the error handler and the configuration of the logging system.
Afterwards all plugins are gathered, the plugin manager is instantiated and the integrity of the plugins is checked. Next, the application plugin which is the entry point of the application is located and initialized. Then the application is started.

3.2 Emersion

Emersion [46] is a Web Applications Integration Platform based on the Java Plugin Framework. It is possible to deploy Java Server Pages applications or Java Servlet application into Emersion as plugins. The Emersion platform consists of several plugins. A plugin adds functionality or resources or both to the system. The relations between plugins are declared through dependencies which are described in XML files.

3.2.1 Architecture of Emersion

The core of the platform is the `org.emersion.platform` plugin. It is responsible for the initialization and the start of the entire platform.
It has one extension point called `PlatformApplication` where the `org.emersion.platform` plugin can be extended with the web server plugin called `org.emersion.platform.webserver`.
The web server plugin is constructed as a Java Servlet and JSP container which is implemented in a container-independent way.

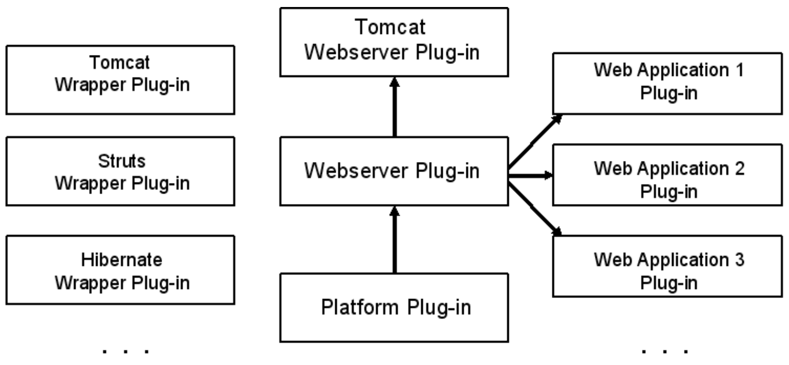


Figure 3.2: The Emersion Architecture: The Platform Plugin is the core of the Emersion platform. It is responsible for starting and initializing the entire application. The Webserver Plugin is connected to the Platform plugin. Into the Webserver plugin the Tomcat web server is plugged in as well as the web applications. The wrapper plugins contain classes of e.g. Tomcat or Struts. They are not plugged into another plugin but may be imported as dependencies by plugins.

To achieve this, the `WebServer` extension point, the `WebContext` extension point and the `WebContextFragment` extension point are inserted into the `org.emersion.platform.webserver` plugin.
The implementations of Java Servlet and JSP containers can be plugged into the `WebServer` extension point. In the version which is available for download, Apache Tomcat is plugged into the `WebServer` extension point.
The web applications which are written by the developer can be plugged into the `WebContext` extension point. Usually the web applications may be added to the system without modifications.
The `WebContextFragment` extension point can be used to add plugin fragments (see Section 6.2).
Due to this extension point resources of web applications may be split between different plugins. Embedded Tomcat is used as the Java Servlets and JSP container.
Additionally there are so called "wrapper plugins" which only contain the libraries of e.g. Tomcat or Struts. In case a plugin needs to access the classes of a wrapper plugin, the wrapper plugin has to be declared in the `<import>` part of the plugin manifest.

3.3 JBoss Application Server

The JBoss Application Server (JBoss AS) is an open source J2EE platform for the development and deployment of web applications, Java enterprise applications and portals [49]. It is part of the JBoss Enterprise Middleware System (JEMS). JEMS is an extendable and scalable product suite with completely integrated and tested middleware products including JBoss AS, Apache Tomcat, JBoss Cache, Hibernate and JBoss Eclipse IDE [50].

Whereas a web server like Tomcat just sends the web pages of an application to the client, the application server adds the business logic to the application. The application server provides a container in which server-side applications are deployed and run. It generates HTML code which is sent to the client. It is also possible to link a database to the application server. Application servers supporting J2EE enable communication with EJBs and WebServices. In order to be able to display Java Servlets and JSPs the Tomcat web server is embedded in JBoss. Other J2EE servers are IBM's WebSphere and BEA's Weblogic [51].

3.3.1 JBoss Architecture

The main parts of the JBoss architecture are the JMX MBean server and the MBeans which are pluggable component services. Services which are not required may be removed and custom MBeans can be added to the system [52].
A managed bean (MBean) is a Java object which implements an MBean interface. The MBean of a resource provides information needed to control the resource by the management application. The information can be attribute values which are accessed through their name, operations and functions to be invoked, notifications, events as well as the constructors for the MBean's Java class.
The MBeans are registered with the MBean server. The MBean server is responsible for making the MBean's management interface available for management applications. To register a MBean on the MBean server it must have an unique name by which it is identified by the management applications.
In JBoss, the JMX MBean server is used as a micro-kernel. The JBoss components are integrated into JBoss via registration in the MBean server. That means the micro-kernel is just a framework and the components add the functionality [53].
The JMX console or Java Management Console provides a live view of the server and its MBeans. It offers information about the server and it is pos-

sible to change the server's configuration in the JMX console [52]. This includes memory consumption, the amount of free memory in the JVM and the number of active threads of a JBoss instance [53].

3.3.2 JBoss Features

JBoss provides many services e.g. logging, security, object persistence, etc. Some of them are discussed below.

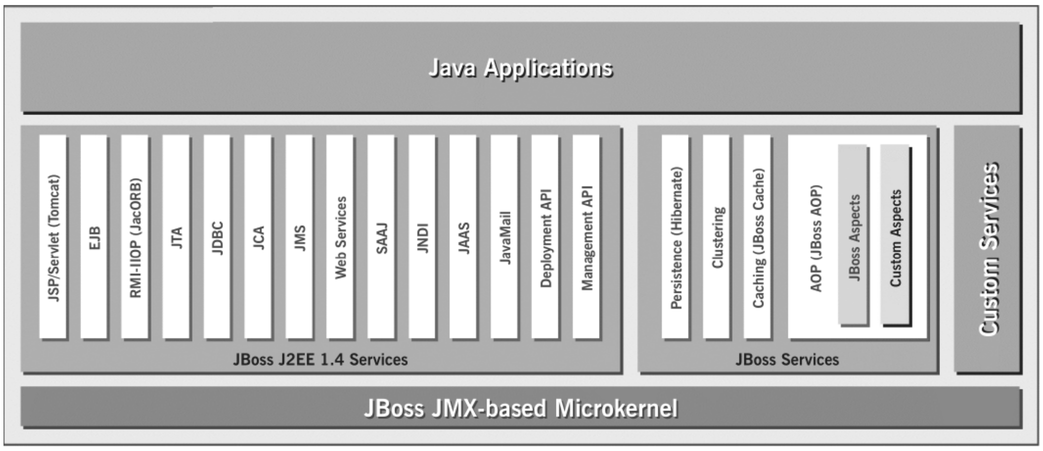


Figure 3.3: The JBoss Architecture [51]: The services JBoss provides, like the web server, security, persistence, etc. are registered with the microkernel and can be used by the web applications deployed in JBoss.

Security

In JBoss it is possible to control user's access to application and what operation those users may perform in the application. The Java Authorization and Authentication Service (JAAS) provides an API for user authorization and authentication [53].

EJB (Enterprise Java Beans) Container

The EJB container is responsible for managing a particular class of EJBs. In JBoss, for each EJB type a container instance exists. The EJB Deployer is in charge of the creation and initialization of the EJB containers.
When an EJB jar is deployed the EJB Deployer verifies the EJBs, creates the container for each type of EJBs and initializes the container. To verify the EJB it is checked that the required home and remote, local home and

local interfaces are available and that the objects in these interfaces have the right types. The application, which starts all the containers, is started and the EJBs are made available to the clients when the EJBs are deployed successfully. In case the deployment of the EJB fails an exception is thrown and the application is not started [53].

Hibernate Integration

Hibernate is an object persistence framework which is able to map Java objects into tables of relational databases and vice versa. JBoss provides a hibernate service which makes the Hibernate framework libraries available to all applications which are running on the JBoss AS.
JBoss is able to manage Hibernate sessions by registering them as a MBeans. This facilitates the configuration of Hibernate. The Hibernate classes and the Hibernate mapping files can be packed separately in a HAR archive or together with the web application in another archive [53].

JNDI Naming Service

JBoss uses the JNDI naming service. JNDI stands for Java Naming and Directory Interface. The naming service is responsible for locating objects and services in the application server. Moreover external clients use the naming service to locate services in the server.
The Java Naming and Directory Interface is a standard API providing a single interface for many naming services, e.g. DNS, LDAP, RMI registry, and file systems. The API is split into two parts, the client API and the service provider interface. With the client API, the naming services can be accessed. On the other hand, the user can create a custom JNDI implementation for naming services using the service provider interface [53].

Transactions

In the JBoss Application Server Guide [53] a transaction is described as

> "a unit of work containing one or more operations involving one
> or more shared resources".

A JBoss transaction has four characteristics [53]:

- **Atomicity** means that it is not possible to perform a part of an transaction. When the whole transaction cannot be executed, then none of the parts must be executed.

- **Consistency** means that the system must be in a stable condition when the transaction has been accomplished.
- **Isolation** means that transactions must be independent of each other. It must be impossible for a transaction to see the partial results of another transaction until this transaction is completed.
- **Durability** means that as soon as the transaction is committed, its results are made persistent. Thus even if the server should crash the changes are not lost.

Enterprise Entity Beans transactions have those four attributes and due to JBoss' aspect-oriented approach POJO transactions also have those attributes [51].

Clustering

Clustering means that an application can run on multiple servers at the same time. With clustering the load is distributed across several cluster nodes. If a server fails the application can still be accessed through other servers. To improve an application's performance simply additional cluster nodes have to be added. In JBoss a cluster node is a JBoss server instance [53].

JBoss Cache

The main target of the JBoss Cache is to improve the performance of applications. Java objects that are frequently accessed are cached and thus unnecessary database ac The JBoss Cache is distributed, transactional and enables AOP(Aspect Oriented Programming). Business objects, so called Entity Beans, are accessed with this cache.
The cache is implemented as an object tree. When an object is added to a tree it is possible to add it to other trees in the cluster to enable the replication of data across the borders of the process.
The cache is also responsible for the replication of HTTP sessions with Tomcat and the replication of JNDI. To improve the management of the AS and its subsystems the JBoss cache is able to combine all JBoss caches to a single cache [51].

3.3.3 JBoss Configuration

There are three predefined configurations of JBoss: ”minimal”, ”default” and ”all”.

As its name gives away, the "minimal" configuration only provides the minimal services which are needed to start JBoss. The logging services, a JNDI server and an URL deployment scanner that searches for new deployments are available. Tomcat is not included in this configuration.
The "default" configuration contains all the services which are necessary for the deployment of web applications. However there are no clustering services available.
The "all" configuration provides all the JBoss services. Moreover the developer may also add her own configuration
[52].

3.3.4 Deployers

JBoss scans regularly for new application deployments in the deploy folder or any folder the developer specifies. When JBoss finds an application to be deployed, the deployment request is sent to the `MainDeployer`. The `MainDeployer` checks whether there is a sub deployer that can handle the deployment. When a sub deployer is found, the `MainDeployer` assigns the deployment task to the sub deployer.
Examples for SubDeployers are

- the `AbstractWebDeployer`, which handles the deployment of web application archives (WARs),
- the `EARDeployer`, which handles the deployment of enterprise application archives (EARs),
- the `EJBDeployer`, which handles the deployment of Enterprise Bean JARs,
- the `JARDeployer`, which handles the deployment of library JAR archives,
- the `HARDeployer`, which handles the deployment of hibernate archives,
- the `SARDeployer`,which handles the deployment of JBoss MBean service archives, and (SARs) [53].

3.3.5 JBoss ClassLoading

The classloading architecture of JBoss enables easy sharing of classes over the boundaries of deployment units with the help of shared classloading repositories. It also enables the deployment of applications and services while the JBoss AS is running.

The most important classloader in JBoss is an extension of the `URLClass-Loader` called `UnifiedClassLoader3` (UCL). The parent for any UCL is a `NoAnnotationClassLoader` which extends `URLClassLoader`.
When the server is started, a single instance of a `NoAnnotationClassLoader`, is created, which is responsible for loading the classes in the libraries contained in the <JBossHome>/lib/ folder.

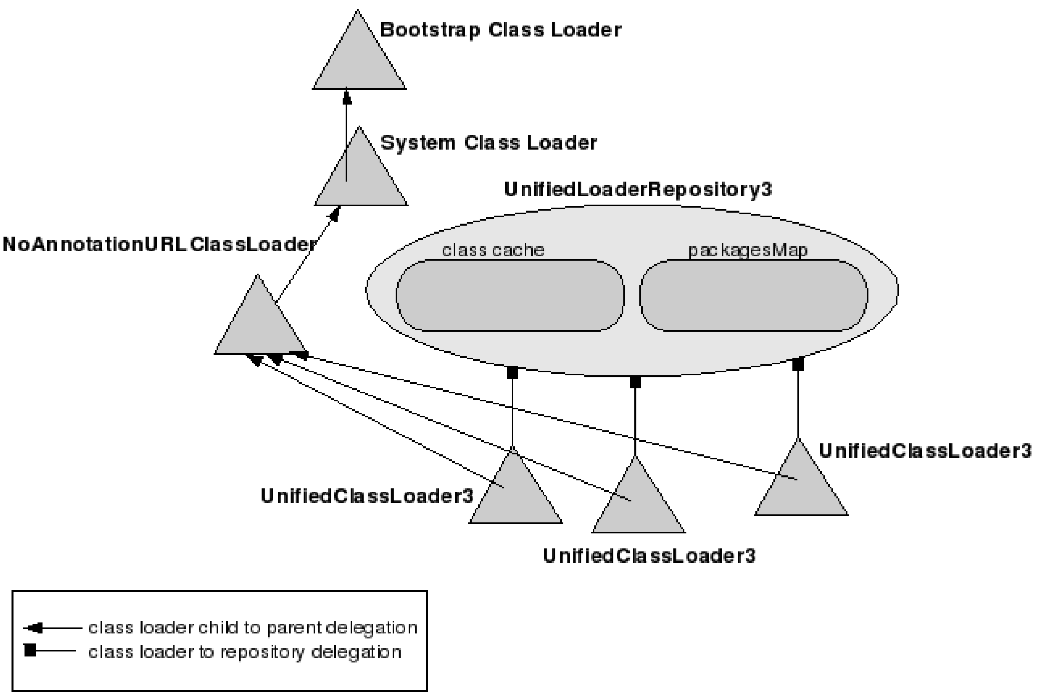


Figure 3.4: JBoss ClassLoading [54]: When a ***`UnifiedClassLoader`*** *wants to load a class, first the cache of the* ***`UnifiedLoaderRepository`*** *associated with the* ***`UnifiedClassLoader`*** *is searched for the class. If the class can not be found there, the* ***`UnifiedClassLoader`*** *itself tries to load the class or delegates the classloading to its parents. If the class cannot be loaded, the other* ***`UnifiedClassLoaders`*** *associated with the* ***`UnifiedLoaderRepository`*** *are asked to load the class. If the class cannot be loaded by any of the* ***`Unified-ClassLoaders`*** *in the Repository, an Exception is thrown.*

The system classloader is the parent of the `NoAnnotationClassLoader` and the parent of the system classloader is the bootstrap classloader [54].
Additionally there are classloading repositories, for instance the `Unified-LoaderRepository3`, the `BasicLoaderRepository` and the `Heirarchical-LoaderRepository`. Please note: `HeirarchicalLoaderRepository` is spelled correctly. A loader repository usually contains several `UnifiedClassLoaders`

whereas a `UnifiedClassLoader` belongs to only one loader repository. The `UnifiedLoaderRepository` is an MBean which has methods to display the information of classes and packages. The JMX console can be used to get a view of this MBean [53].
Several URLs which are used to load classes and resources can be associated with a single UCL. The default configuration ensures that there is only one `UnifiedLoaderRepository3` that contains all the `UnifiedClassLoaders`. When an `UnifiedClassLoader` tries to load a class the following tasks are performed:

1. The cache of the loader repository which is associated with the UCL is searched for the class.

2. If the class cannot be found in the cache the UCL tries to load the class by calling the `URLClassLoader`'s loadclass method. The URLs of the `UnifiedClassLoader` and its parent class loader are searched for the class. In case the class is found it is added to the cache of the loader repository which belongs to the `UnifiedClassLoader` and returned.

3. If the `UnifiedClassLoader` or his parent cannot find the class then the repository which is associated with the `UnifiedClassLoader` is searched for other UCLs which might be able to load the class. It can be quickly determined which UCL is capable of loading a class because a mapping from package names with their classes and the UCLs exists. This mapping is updated when a UCL is added to a repository.

   The discovered `UnifiedClassLoaders` are capable of loading classes in the order they have been added to the repository. If the class cannot be found a `ClassNotFoundException` is thrown, otherwise the class is simply returned.

Scoped Repositories

JBoss provides a flat classloading model. The classes which make up an application can be used by other applications because the classloader that is responsible for the classes is in the same repository as the classloader of the other applications. When any of the other applications try to load a class it may be found in the repository's cache when it has been loaded before, or it can be loaded by a suitable classloader in the repository.

The flat model can cause problems especially when a version conflict exists. Then the classes have to be isolated from other deployments by so called scoping.

In JBoss it is possible to create a loader repository for each deployed application. Such a repository is a `HeirarchicalLoaderRepository` implementation. It is a child repository of the main `UnifiedLoaderRepository` that cannot share classes. When a class is loaded, first the `UnifiedClassLoader` instances of the `HeirarchicalLoaderRepository` are searched for the class before the classloading task is delegated to the default `UnifiedLoaderRepository3`. This means that an application with a scoped repository can still access the classes of the UCLs in the root repository [54][53].

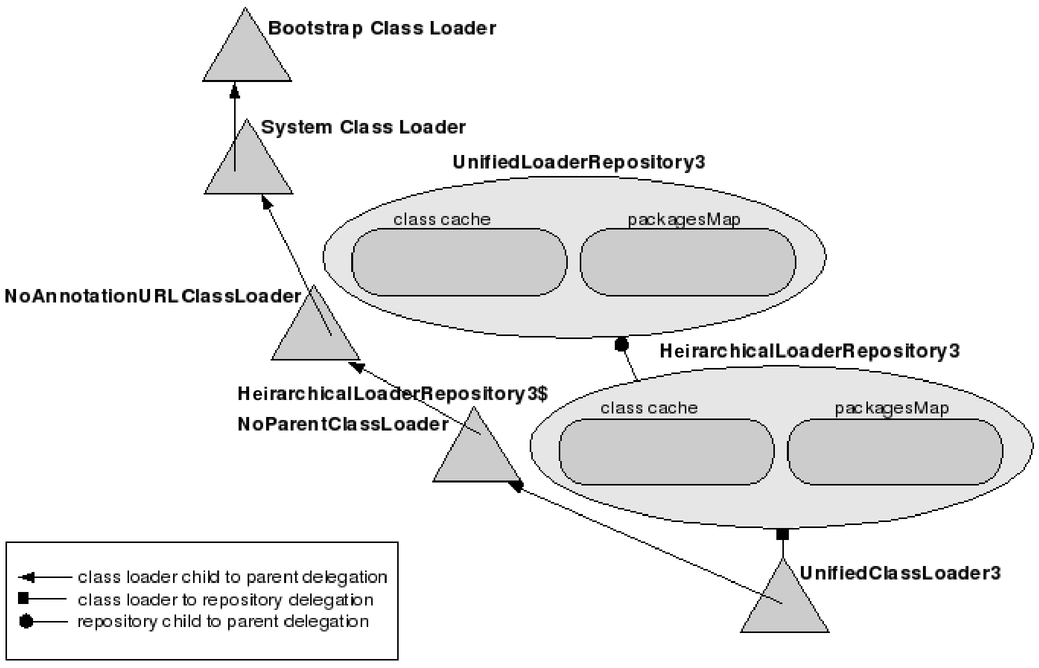

Bootstrap Class Loader
System Class Loader
UnifiedLoaderRepository3
class cache
packagesMap
NoAnnotationURLClassLoader
HeirarchicalLoaderRepository3
HeirarchicalLoaderRepository3$
NoParentClassLoader
class cache
packagesMap
UnifiedClassLoader3
class loader child to parent delegation
class loader to repository delegation
repository child to parent delegation

Figure 3.5: JBoss ClassLoading with a Scoped Repository [54]: When a ***`UnifiedClassLoader`*** *which is associated with a scoped repository (****`HeirarchicalLoaderRepository`****) wants to load a class, first the cache of the scoped repository is searched for the class. If the class can not be found there, the cache of the parent of the scoped repository, the* ***`UnifiedLoaderRepository`****, is searched. If the class cannot be found there either, the* ***`UnifiedClassLoader`*** *itself tries to load the class. If the class cannot be loaded, the other* ***`UnifiedClassLoaders`*** *associated with the scoped repository are asked to load the class. If they are not able to load the class, the* ***`UnifiedClassLoaders`*** *of the* ***`UnifiedLoaderRepository`*** *try to load the class. If the class cannot be loaded by any of the* ***`UnifiedClassLoaders`****, an Exception is thrown.*

Advantages and Disadvantages

Advantages

- It is possible to access classes across the boundaries of deployments.
- This model offers many possibilities for the future, e.g. dependency and conflict detection and the partitioning of repositories into domains.

Disadvantages

- The developer has to be careful with packaging of applications because if there a duplicate classes in a loader repository, ClassCastExceptions or linkage errors may occur.
- When different applications use different versions of the same class, this class needs to be isolated in a separate EAR or WAR for which a HierarchicalLoaderRepository is created.

Classloading and Deployers

The UCL for a deployment is created by the main deployer. Only the topmost deployment unit, e.g. an EAR, gets an UCL. The deployment units contained in that EAR, for instance a WAR or a JAR, just add their classpaths to the UnifiedClassLoader of the parent deployment unit, the EAR.
A `WARDeployer` only adds the WAR archive to the classpath of the UnifiedClassLoader. The class loaders of the Servlet container loads the classes from the `WEB-INF/classes/` and `WEB-INF/lib/` folders. Since the `UnifiedClassLoader` of the WAR is the parent classloader of the Servlet container's class loaders, the classloaders of the servlet container delegates the loading of the classes in `WEB-INF/classes/` and `WEB-INF/lib/` to the WAR UCL. The classloaders of the web container are not included in the shared classloader repository and thus cannot be seen by other components. In case this classes need to be available to other applications as well, they have to be included into a SAR or EJB deployment or into a JAR which is referenced through a manifest classpath entry [53].

3.3.6 Embedded JBoss

The JBoss micro-container can also be used as a stand-alone micro-container outside of JBoss and is called *embedded JBoss* [53].
Embedded JBoss was developed to be able to execute the JBoss kernel and the JEMS services in a non-application server environment with a classloader

not controlled by JBoss. The goals of the embedded JBoss project is the ability to run JBoss in a stand-alone Java application within JUnit tests in a stand-alone Tomcat instance or in other application servers.
Embedded JBoss uses the new JBoss 5 kernel and supports JNDI, EJB 3.0, JBoss Security, JBoss Messaging, JMX MBeans and JBoss TS.
In contrary to the JBoss AS it is not possible to deploy an application while embedded JBoss is running. WAR and EAR packaging types are not supported, files can only be packed in JARs. The default JBoss Classloading model with `UnifiedClassLoader`s and Repositories was removed. The embedded version of JBoss does not generate classloaders. Only system classpath and the context classloader are used for deployment and execution of applications [55][56].

3.4 AndroMDA

Model Driven Architecture disconnects the business and application logic of an application from the underlying platform technology. By using UML (Unified Modelling Language) platform-independent models of an application's business functionality and behavior are created. These models can then be realized on almost any platform via MDA [57].

AndroMDA [58] is an open source Model Driven Architecture (MDA) framework which creates source code based on UML models. AndroMDA takes an UML model and with the help of the Metadata Repository (MDR) it builds a metamodel out of it. Velocity then takes the instantiated objects of the metamodel and generates the text output according to the generation rules which are provided by the cartridges. The output may be generated for any platform e.g. .NET, Spring, J2EE etc.
AndroMDA provides cartridges for popular technologies such as Java, Struts, Spring and Hibernate which are ready to use. Since AndroMDA is able to generate output for any computer language and any architecture, the developer may build her own cartridges or modify existing cartridges to suit her needs.
AndroMDA is not only able to generate source code. Any sort of text output may be produced. Examples for outputs are web pages, database scripts, configuration files, e.g. for object-relational mapping.
To develop an application with AndroMDA the following steps have to be performed [59]:

1. The architecture of the application has to be designed.

2. The business domain has to be modelled in a platform independent way.

3. Before the code is generated the developer adds additional information to the model.

4. The code is generated.

5. After the generation of the code the developer has to implement the business logic.

3.5 Hibernate

Hibernate [60] is a framework for object-relational persistence and querying. It provides object-relational mapping (ORM) for applications written in Java. This means Java classes can be mapped to database tables of relational databases. To support the object-relational mapping Java data types can be mapped to SQL data types. Hibernate relieves the developer of a lot of coding work and allows her to concentrate on the business logic.
XML mapping documents are used to define the object-relational mapping and generate the database tables. With Hibernate it is possible to create any entity association be it one-to-many, one-to-one or many-to-many. Bidirectional any unidirectional associations are supported as well.
Hibernate is compatible with any database which is JDBC compliant including MySQL, PostgreSQL, HypersonicSQL, Oracle, Sybase, SAP DB, and Ingres. Hibernate provides an object-oriented query language with a SQL-like syntax and queries in the database's SQL dialect.
The Hibernate service within JBoss is compatible to J2EE and can be configured and managed with JMX. Hibernate provides a so called transparent persistence which means that the developer only needs to take care of the appropriate configuration while Hibernate deals with the storage and retrieval of persistent objects.

Chapter 4

Results

To find suitable technologies which could be used to develop component-based web applications literature and the Internet were searched.
Three technologies were found, Spring OSGI, ESB and Emersion based on JPF. It was decided that Emersion and JPF should be tested whether they were able to work together with the other technologies used at the institute.
To asses the feasibility of JPF and Emersion with the technologies used at the institute, first the Emersion platform was tested with a few technologies and it was extended with the embedded JBoss.
In order to be able to run component-based web applications in the JBoss AS a modified version of Emersion was deployed in JBoss and then validated with the technologies at the institute.
The composition of web applications using Hibernate into a single application was successfully tested.
It was verified that J2EE applications built with AndroMDA are able to run correctly in the modified Emersion/JBoss environment but it was not enough time to build a composition of two or more J2EE applications.

4.1 Research

The search for technologies suitable to develop component-based web applications resulted in three candidate technologies: Spring OSGi, ESB and Emersion based on JPF.
Emerstion and JPF were then chosen to be tested with the instiute's technologies because of the following reasons:

- A plugin can consists of the business logic, database tables and the web interface.

- A plugin can be integrated into the system via the plugin.xml file without a lot of effort.
- The dependencies of plugins can be easily declared in the plugin.xml file.
- The plugins usually can be integrated into the composed web application without modifications.

An application connected to an ESB may also consist of business logic, database tables and web interface. Additionally there exists an ESB implementation of JBoss, the JBoss ESB (see Section 6.1), which is able to work together with the JBoss application server. However, it takes some time to learn how to use an ESB optimally and according to Breitling [40] the use of an ESB is reasonable only if it is applied in the entire enterprise and not just in small application.

With Spring OSGi only the business logic can be composed of modules. Moreover the development of web applications has not yet been tested thoroughly. Spring OSGi based web applications have so far only been tested with the Equinox Incubator OSGi provider and not with JBoss or any other application or web server.

4.2 Embedding JBoss into Emersion

Struts is included in the Emersion platform but it was also possible to build applications with plugins that used Hibernate, Spring and Tapestry. Since there was only Tomcat embedded in Emersion, J2EE applications do not work in Emersion.
To be able to test whether Emersion functions with J2EE application it was necessary to add the embedded JBoss (see Section 3.3.6) to Emersion.
To embed JBoss into the Emersion platform it was necessary to create three new plugins: the `JBoss Wrapper` plugin, the `ApplicationServer` plugin, and the `JBoss ApplicationServer` plugin.
The `JBoss Wrapper` plugin contains the libraries of the embedded JBoss.
The ApplicationServer plugin includes an `ApplicationServer` interface which is actually implemented in the `ApplicationServer` class contained in the JBoss `ApplicationServer` plugin.
Both the interface and the implementation are similar to the web server plugins. The `ApplicationServer` interface in the `ApplicationServer` plugin

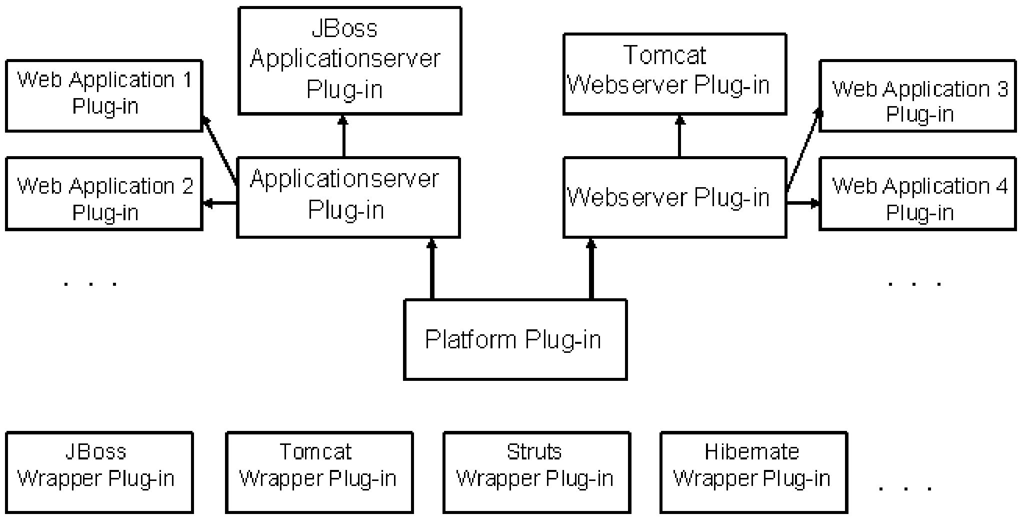

JBoss Applicationserver Plug-in
Tomcat Webserver Plug-in
Web Application 1 Plug-in
Web Application 2 Plug-in
Applicationserver Plug-in
Webserver Plug-in
Web Application 3 Plug-in
Web Application 4 Plug-in
. . .
. . .
Platform Plug-in
JBoss Wrapper Plug-in
Tomcat Wrapper Plug-in
Struts Wrapper Plug-in
Hibernate Wrapper Plug-in
. . .

Figure 4.1: JBoss embeddded in Emersion: The Platform plugin is the core of the Emersion platform. The Webserver plugin and the Applicationserver plugin are connected to the Platform plugin. The embedded Tomcat web server and Java Servlet and JSP applications are plugged into the Webserver plugin whereas the embedded JBoss and web applications that need the embedded JBoss because they for instance consist of EJBs, are plugged into the Applicationserver plugin. Wrapper plugins contain the libraries of e.g. embedded JBoss, Tomcat or Struts.

defines methods such as starting the application server, stopping the application server, deploying the web application, undeploying the web application, etc. Also it was based on the interface in the web server plugin.
To implement the methods in the JBoss `ApplicationServer` class the embedded JBoss API [61] was used.
It was then successfully tested with a small application. The application consists of two plugins, each containing an Entity bean.

4.3 Embedding a JPF Application into JBoss

At the institute all web applications run in the JBoss application server (see Section 3.3). Thus to enable compositions of web applications within the institutes software environment an application based on JPF and similar to Emersion would have to be ”embedded” in JBoss.
Therefore the Emersion platform and the Java Plugin Framework had to be modified to be able to deploy and run web applications consisting of modules in JBoss.

First it was necessary to modify the classloader implemented in JPF. Due to the special classloading architecture of JBoss it was not possible to use the original JPF classloader.
Second it had to be ensured that the modified version of the Emersion starts when JBoss is started up. Afterwards the architecture of the new platform application had to be defined.

4.3.1 JPF PluginClassLoader

The classloading in JPF is designed in a way that each plugin has its own classloader. All the classes and resources that belong to the plugin are loaded by this classloader. In Emersion a web application plugin is also loaded by the same plugin specific classloader. This is possible because the Tomcat embedded into Emersion does not provide its own classloading mechanism. Therefore the developer can easily set the classloader for each web application.
When a JPF application with web applications as plugins is running in JBoss, it is not possible to simply assign a classloader to a plugin so that all the plugin's classes and resources are loaded by this classloader because of complex classloading architecture in JBoss.
Thus the `PluginClassLoader` which is implemented in JPF had to be modified to allow it to be used within JBoss.

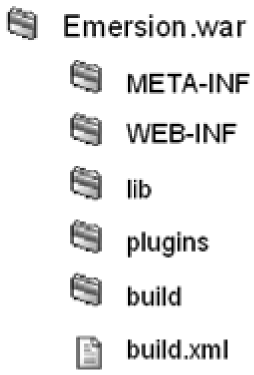
The most elementary modification concerned the superclass of the `Plugin-ClassLoader`. In JPF the `PluginClassLoader` extends the `URLClassLoader`. The `URLClassLoader` loads classes and resources from a path containing URLs which refer to directories as well as JAR files.
In order to integrate the `PluginClassLoader` into the JBoss classloading architecture (see Section 3.3.5), the `PluginClassLoader` must not inherit the `URLClassLoader` but JBoss' `UnifiedClassLoader3`.
The three most important methods of the `PluginClassLoader` are `getUrls()`, `collectImports()`, and `loadClass()`.
The `static getUrls(final PluginManager manager, final PluginDe-scriptor descr)` method returns a URL array. This array contains all the URLs that make up the plugin application for instance EARs, WARs, and JAR libraries. The `getUrls` method is necessary to add these URLs to the plugin specific classloader. This method was not changed in the new implementation of the `PluginClassLoader`.
The `getImports` method collects the imported plugins. These are the plugins on which the application depends. This method which just adds the plugins to hashmaps was slightly modified into a public method which returns a string array with all the plugin IDs. This allows the plugins to be called from outside the `PluginClassLoader` class (e.g. in the Emersion platform).
PluginClassLoader's `loadClass(final String name, final boolean re-solve)` method performs the usual classloading procedure as follows: First the `PluginClassLoader` tries to find the class. If the class can not be found loading is forwarded to the parent classloader.
Before the classloading procedure is started, it is first checked whether the class to load is on the blacklist. A blacklist is part of the new implementation. The developer may define classes to be on the black list, if these classes for any reason should not be loaded with the plugin's classloader.
In case the class is on the blacklist, the loading of this class is not handled by the classloader of the plugin but is immediately delegated to the parent classloader. This is necessary to avoid classloading problems that can be caused by the fact that all the classes which are contained in the default loader repository are also included in each scoped repository. This mainly concerns classes from logging libraries or EJB libraries which can not be found by the `UnifiedClassLoader`s of the scoped repository when the parent delegation is turned off resulting in a `NoClassDefFoundError`.

4.3.2 Integration of JPF into JBoss AS

The next challenge was to "embed" the JPF application into the JBoss Application Server. To achieve this the deployment folder of the Emersion platform was modified into a folder with a .WAR ending. The required `META-INF` and `WEB-INF` folders were inserted into the `WAR` folder. Then the `Emersion.war` folder was added to the JBoss deploy folder. This ensured it gets deployed and runs just like any other web application in JBoss.

The Folder and File Structure of the JPF Application

The folder and file structure of the application is similar to the structure of the Emersion files and folders.



Emersion.war
META-INF
WEB-INF
lib
plugins
build
build.xml

*Figure 4.2: Structure of Emersion.war: The **WEB-INF** folder contains the listener and the **web.xml** file. Various libraries are contained in the **lib** folder. The **plugins** folder contains the plugins and the **build** folder contains the compiled plugins. The **build.xml** file is used to compile the plugins and to copy the plugins from the **plugins** folder to the **build** folder.*

- The `WEB-INF`-folder contains the listener implementation and the `web.xml` file in which the listener is declared.
- The `lib`-folder contains various libraries.
- The `plugins`-folder is the so-called plugin-repository containing these.
- The `build`-folder contains the compiled plugins (in case they are not readily compiled in the plugin-repository) which are then used to compose the application.
- The `build.xml` file is an ant file used to compile the plugins and copy them from the `plugins`-folder to the `build`-folder.

The ServletContextListener

For the modified Emersion platform to run properly it is necessary that it is started at the same time as JBoss. Therefore a `ServletContextListener` was implemented. The listener was inserted into the `WEB-INF`-folder of Emersion.war and a listener entry was added to the `web.xml` file. The task of the listener is to invoke the main method of JPF's boot class.
The boot class is the class which starts the JPF application. Its `initApplication` method configures and initializes the application.
Most of the initialization parameters are fetched from the `boot.properties` file. The user can define for instance which implementations should be used for the plugin registry or the path resolver, or which repositories the application should scan for plugins. The only parameter which is absolutely necessary is the name of the plugin which represents the entry point to the entire JPF-based application.
The `initApplication` method of the boot class calls a method of the ApplicationInitializer class. The `ApplicationInitializer` class is responsible for the initialization of the application.
First all the folders which may contain plugins, the so-called plugin repositories, are scanned and the plugins with their files and folders are collected. The located plugins are then made available to the system by the plugin manager. Then an instance of the plugin which forms the entry point to the application is created and returned to the `Boot class` which in turn starts the entry point plugin. The entry point plugin consists of just a simple Java file which has to implement a pre-defined JPF interface.

4.3.3 The Architecture of the JPF Application

The heart of the modified Emersion platform is the core plugin. The core plugin is also the entry point to the application and is specified in the `boot.properties` file. All plugins which contain parts of the web application are plugged into the extension point of that core plugin.
In order to be able to access classes of any other plugin a plugin has to specify on which plugins it depends on in the `plugin.xml` file.

4.3.4 The Core Plugin

The core plugin is started during the startup process of JPF.
Since the modified version of Emersion is running in JBoss AS it is not possible to simply assign a classloader to each plugin and load the classes and resources with this `PluginClassLoader`.

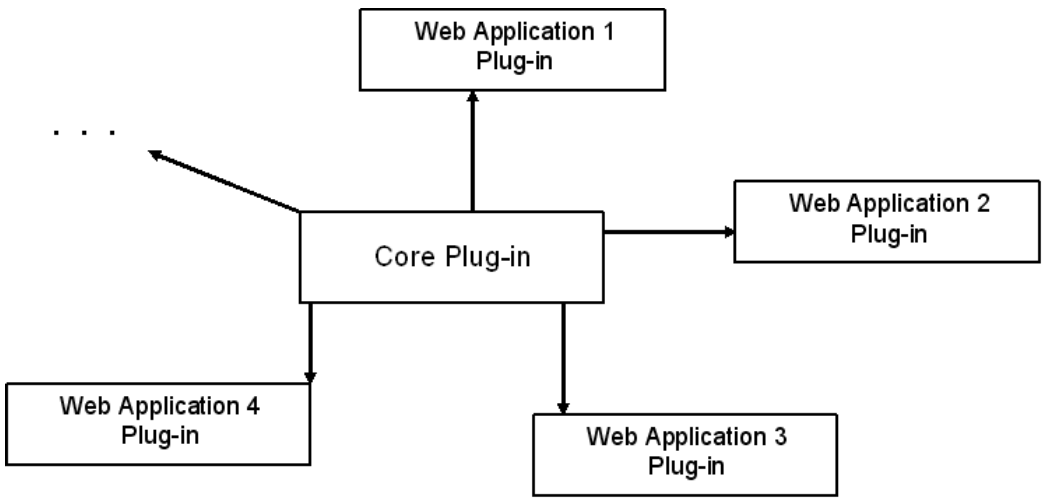


Figure 4.3: Architecture of the modified Emersion Platform: The modified Emersion platform consists of a core plugin which is started during the JBoss start-up process and into which all the web applications are plugged in.

Instead the classloading of the JPF-based application has to be modified to work within to the JBoss classloading architecture. This is achieved by implementing certain classloading tasks in the core plugin. In a conventional JPF application these tasks are performed by the `PluginClassLoader`.

Plugin.xml

The two most important tags of the `plugin.xml` which define the core plugin are the `<plugin>`-tag and the `<extension-point>`-tag.
The `<plugin>`-tag specifies the name of the plugin, its version and the class-name. The class parameter declares the class which is instantiated and its methods called when the plugin is started.

The `<extension-point>`-tag provides the functionality to add multiple web applications to the modified Emersion platform.

```
<plugin id="org.emersion.platform"
        version="0.0.1"
        class="org.emersion.platform.CorePlugin">
   <runtime>
      <library ... />
         ...
   </runtime>

   <extension-point id= ...>
      ...
   </extension-point>
</plugin>
```

Listing 4.1: The ***<plugin>*** *tag defines the name of the plugin and the version number and the class which is beeing instantiated at the start-up process of the plugin. The* ***<runtime>*** *tag and the* ***<extension point>*** *tag are emdedded into the* ***<plugin>*** *tag.*

```
<extension-point id="WebContext" extension-multiplicity="any">
 <parameter-def id="id"
                multiplicity="one"
                type="string" />
 <parameter-def id="dataLocation"
                multiplicity="one"
                type="string" />
</extension-point>
```

Listing 4.2: The ***<extension-point>*** *tag provides the functionality to add multiple web applications to the modified Emersion platform.*

CorePlugin.java

The `CorePlugin.java` class' `startApplication()` method is called when the core plugin is started.
The `startApplication()` method first collects all the plugins which are plugged into the `WebContext` extension point. Then the following steps are performed for each plugin:

1. The `PluginClassLoader` which JPF creates for each plugin is determined.

2. Each plugin gets its own scoped repository. A scoped repository is a MBean of the type `HeirarchicalLoaderRepository3` with a unique name. This MBean is added to the MBeanServer of JBoss.A scoped repository does not allow any other repository to access its classes. In the modified Emersion platform it is required that plugins only share code or resources when it is explicitly declared by adding this URLs to the `<library-tag>` of the `<runtime>` part in the `plugin.xml`. To be able to control which URLs can be accessed by other plugins it is necessary that a scoped repository is created.

3. The classloader of the plugin is registered with the scoped repository to ensure that this classloader is used when any of the plugin's classes is requested.

4. The classes and libraries which belong to the plugin are determined by the `getURLs()` method of the `PluginClassLoader`. They are then added to the classloader and subsequently added to the repository.

5. It is necessary that the plugin cannot only load its own classes but it must also be able to load the classes belonging to plugins it depends on. Since the repositories of these plugins do not share their classes too, these classes must be added to the scoped repository of the requesting plugin.

   Therefore all the plugins on which the plugin depends are determined with the `getImports()` method of the PluginClassLoader.

   Then the classloaders of those imported plugins are determined and added to the scoped repository of the plugin.

6. Every deployment unit of the plugin for example an EAR, a WAR, a JAR, etc. is deployed in JBoss by calling the MainDeployer's deploy method.

4.3.5 Validation of the Approach

It is necessary to verify whether applications built with different technologies are able to run as plugins correctly in JBoss and the modified Emersion platform.
"Running correctly" means that the applications work as they should and use the correct classloaders.
The next step was to test if two plugins work together correctly. It had to be determined if the classes of imported plugins were loaded correctly by the plugin specific classloader.

The classloading works correctly for simple JSP and Java Servlet applications that are built with Struts or Tapestry.

However it was found out that the loading of user interface resources such as JSP files and Servlet files can not be handled by the `PluginClassLoader`. The loading of these files is handled by the Tomcat web server service of JBoss and it is never delegated to the JBoss classloading system. This means that those files cannot be loaded by the `PluginClassLoader` which is a part of the JBoss classloading system.

Hibernate

Plugins that use Hibernate (see Section 3.5) as their persistence mechanism work well in the JBoss and modified Emersion environment. All the classes are loaded with the correct classloader.
To test whether two Hibernate plugins can work together correctly with the correct classloaders a simple example was used.
The example web application consists of a two plugins where each plugin has only one database table, business classes and a user interface.
The task was to find out if the plugins could access objects of another plugin and load them correctly. It turned out that it is possible to retrieve objects of another plugin no matter whether a one-to-one, one-to-many or many-to-many relationship exists between the fields of the database tables of the different plugins.

However it is not possible to keep the plugins completely independent of each other.
In the Hibernate configuration file `.hbm.xml` the relationship between two database table fields are defined. At deployment time a web application needs to know with which other application' database table fields it has a

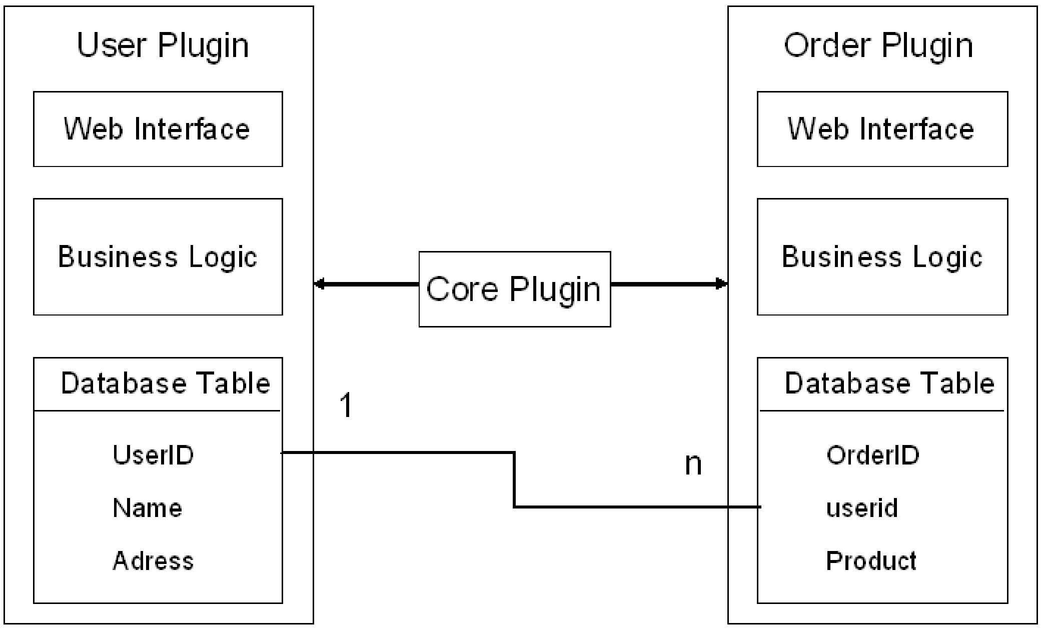


Figure 4.4: This is an example for an application that was used to test the feasibility of Hibernate in combination with JPF. Each plugin consists of one database table, the business logic and a web interface. A 1:n relationship exists between the `UserID` column of the `user` table and the `userid` column of the `order` table.

relation. That means it has to be able to read the `.hbm.xml` configuration file of the other application. To achieve this, the `.hbm.xml` file of on application has to be inserted into the other application. This means that the second plugin cannot be reused unmodified in another application without the first plugin.

AndroMDA and J2EE

Finally it was tested how the JBoss and modified Emersion environment copes with J2EE applications that were designed and built with AndroMDA (see Section 3.4).
The sample application was developed using the AndroMDA Project Template [59]. This is a framework for the rapid development of J2EE applications with AndroMDA. The main part of the application is a Java Entity Bean. An Entity Bean is a persistent business object. In the example the bean is called "MyCustomer" and has an Id field and a name field. The content of these fields are stored in the database. The two main tasks of the business logic are to create a new `MyCustomer` object and add it to the

database and to retrieve all `MyCustomer` objects that are stored.
Such an application runs in JBoss with the modified Emersion correctly and uses the correct `PluginClassLoaders`. To avoid classloading problems with classes od external libraries, the blacklist Section 4.3.1 was added to the Implementation of the `PluginClassLoader`.
However if there are two such plugins in a modified Emersion environment, then it is not possible to have duplicate EJB names. Each EJB name in the composed application must be unique.

It has not yet been tested whether two or more AndroMDA and J2EE plugins can be composed to a single application that works correctly.

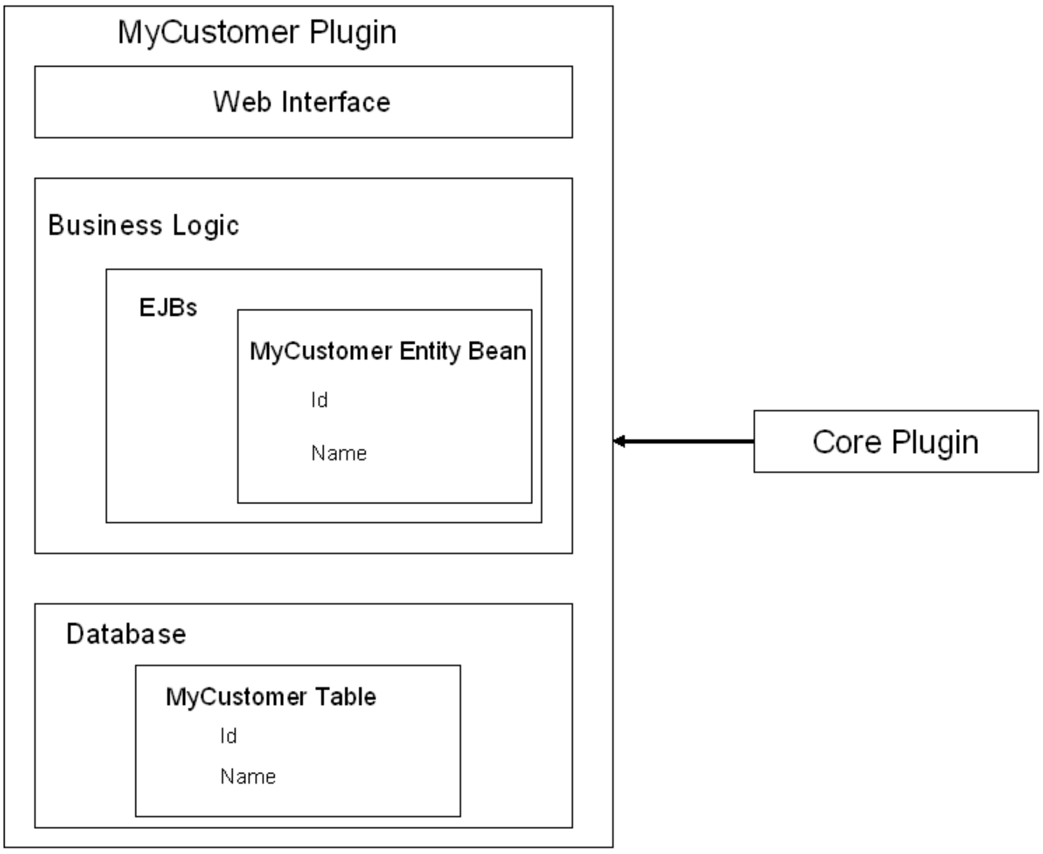


Figure 4.5: The J2EE application that was tested consists of a single application that was plugged into the core plugin. This application contains the MyCustomer Entity Bean with an ID field and a name field.

Chapter 5

Discussion

The first task of the thesis was to search for a suitable technology for developing component-based web applications.
Three technologies were taken into consideration: Spring OSGi, Enterprise Service Bus Architecture and JPF with the Emersion platform.
JPF in combination with the Emersion platform was then chosen to be tested with technologies like Struts, Hibernate, Tapestry, J2EE and AndroMDA.
First the Emersion platform with Tomcat and JBoss embedded was tested with those technologies. The Emersion platform then had to be modified so that it could be used inside of JBoss.

5.1 Technologies for the Development of Component-based Web Applications

Spring OSGi modules are able to find and use services provided by other modules automatically. Additionally they can be added and removed dynamically from the running system.
The Spring OSGi concept only allows to split the business logic into modules. That means a module does not consist of all three layers, the business logic, the web interface and the database tables.
The development of web applications has not yet been tested thoroughly. Spring OSGi based web applications have so far only been tested with the Equinox Incubator OSGi provider.

Enterprise Service Bus is a technology which allows applications written in different programming languages, using different data formats or programming interfaces or are located on different servers to communicate with each other via a connectivity infrastructure provided by the ESB.

JBoss provides an ESB implementation, the JBoss ESB (see Section 6.1). The JBoss ESB is able to work together with the JBoss application server and with the technologies that are compatible with JBoss like Struts, Spring, Hibernate, J2EE, AndroMDA and Tapestry.
However it takes some time to learn how to use an ESB optimally and according to Breitling [40] the use of an ESB is reasonable only if it is applied in the entire enterprise and not just in small application.

The Emersion platform which is based on JPF was then examined in more detail because it is easy to integrate a plugin into the system and declare its dependencies with the plugin manifest file. Moreover plugins can be integrated into the system without modifications. A plugin in JPF may consist of the web interface, the business logic and the database table definitions.

5.2 Assessment of the collaboration of JPF and Emerision with Web Technologies

It was assessed whether the Emersion platform was compatible with the technologies used at the institute e.g. Struts, AndroMDA, J2EE, Spring, Hibernate and Tapestry.
With the Emersion platform it was possible to compose web applications using Struts, Hibernate, Spring and Tapestry. After JBoss was emdedded into the Emersion platform, it was also possible to build component-based applications with plugins containing EJBs.

Since it is desireable that component-based web applications can also run in the JBoss Application Server which is used at the institute, the next step was to integrate a modified version of the Emersion platform into JBoss.
It was possible to compose web applications with plugins using Struts and Tapestry. Plugins that use Hibernate could also be composed but some of the plugins could not be integrated completely unmodified. J2EE applications built with AndroMDA are able to run in the modified Emersion and JBoss environment with the correct classloader but it has not yet been tested whether it is possible to successfully compose two or more plugins that use J2EE.

It has been verified that plugins can be easiliy integrated into an application with the modified Emersion platform. Except for AndroMDA and J2EE which are still to be tested, all required technologies are compatible with the modified Emersion platform in JBoss. A disadvantage is the loading of JSPs and Java Servlets cannot be handled by the PluginClassloader because this is the task of the Tomcat service in JBoss. Furthermore the core plugin has a structure that does not allow to deploy plugins while the application is running. As the Hibernate example shows, it is not possible to always keep the plugins independent of each other.

5.3 Outlook

There is still work to do until it is definitely verified that JPF can be used to build component-based webapplications with modules that were developed using the institute's technologies mentioned above.
First it has to be tested whether it is possible to compose applications out of two or more plugins which use J2EE and are built with AndroMDA.
Next the modified Emersion platform should be tested in JBoss 5 which was released a few month ago. At the moment the modified Emersion platform is integrated into JBoss 4.

Chapter 6

Appendix

6.1 ESB Implementations

6.1.1 Open ESB

Khire, Liu and Naderzad [62] describe Open ESB as a standard, decentralized integration infrastructure. It provides normalized message routing and proxying and is based on asynchronous XML message exchanges. Management and monitoring are centralized.
With Open ESB it is possible to compose Web Services and enterprise applications to a loosely coupled system. A true Service-Oriented Architecture can be implemented with Open ESB because the application composition can be composed and recomposed seamlessly.

The current version, Open ESB 2.0 Beta, consists of the following components [63]:

- JBI Framework: This framework implements a JBI instance. JBI is short for Java Business Integration. It consists of a platform and a service assembly. The platform is extensible and pluggable and makes the cooperation between integration technology and Web services possible. The service assembly is a document which describe services, artifacts and routing of a SOA application. [64]

- BPEL Service Engine: The Business Process Execution Language (BPEL) is utilized to organize the processes in a composed application.

- Java EE Service Engine enables the EJB services to connect with the hosting application server.

- XSLT Service Engine is used for transforming XML documents with the help of XSL style sheets.
- Intelligent Event Processor Service Engine is used for event notification and event triggers and manages real-time business event collection and processing.
- SQL Service Engine enables SQL services to other JBI components.
- File Binding Component offers a transport service to a file system and makes it possible that the JBoss environment can interact with the file system.
- FTP Binding Component enables messaging using the FTP protocol.
- HTTP Binding Component responsible for connecting a JBI instance to external web services and for connecting those external services to the JBI instance.
- JDBC Binding Component makes it possible to configure and connect to databases that support the JDBC 3.0 API specification.
- JMS Binding Component supports Java Messaging Service (JMS) for the transport of messages.
- SMTP Binding Component enables the configuration and connection to SMTP servers and clients within the JBI environment.
- WebSphere MQ Binding Component enables the configuration and connection to WebSphere MQ servers within a JBI environment.

So far Open ESB only works with GlassFish and Sun Application Server. Only an experimental version of Open ESB runs on JBoss. [64]

6.1.2 JBoss ESB

This ESB was developed with the JBoss Enterprise Middleware Suite (JEMS) technology and is available as open source since July 2006. The core of JBoss ESB is Rosetta.
Rosetta was build to make it easier to interoperate between different components, applications and services. It also offers an infrastructure and tools that can be configured to function with different transport mechanisms, for instance email or JMS. Furthermore a general purpose object repository, a way to log the interactions and pluggable data transformation mechanisms

are provided. JBoss ESB is Java specific. [65] It provides a base transport mechanism, a pluggable architecture, a transformation engine for transforming data formats, a service registry and it supports several messaging services. In JBoss ESB everything is a logical service. At the architectural level these services interact with messages. [26]
The messaging infrastructure (MI) makes up the core of JBoss ESB. The messaging infrastructure is abstract, multiple different implementations may be provided. For instance not only JMS can be used but also a pure Web Service deployment may be supported. In JBoss ESB the mapping of service description and service contract to the technology is dynamic and configurable. This means that many SOA implementation technologies are supported. Moreover JBoss ESB supports several implementations of registry, e.g. UDDI. Such a registry is needed to publish, discover and consume a service. The ESB also provides a versioning service. It is possible to integrate existing services into the ESB environment without changing these services. The core of JBoss ESB consist of the following parts:

- Message Listener and Message Filtering Message Listeners listen for messages and forwards them to a pipeline which filters the messages and routes to the adequate message endpoint.
- Data transformation is carried out by the SmooksTransformer action processor. Smooks is a transformation implementation and management framework which permits the developer to write the transformation logic in XSLT, Java, etc.
- Content Based Routing Service.
- Message Repository to save the messages and events that are exchanged in the ESB.

Not all service that can interact via the ESB need to be implemented using JBoss ESB. There is an Interoperability Bus within ESB that makes it possible to plug such ESB-unaware services into the ESB.
The communication between services that are aware of JBoss ESB, that is, services which were developed with JBoss ESB, is realized through messages. Such a message has a standardized format for information exchange. It consists of a header, a context, a body and attachments.
The header includes routing and addressing information.
The content of the context are session related information, e.g. transaction or security contexts.

The body can contain a byte array for arbitrary data. The way the data of this array is interpreted by the service must be specified by the service itself. Additionally it may contain a list of objects which have arbitrary types. The specific object type defines how the objects are serialized to or from the message body when the message is transmitted.

```java
public interface Body
{
public void add (String name, Object value);
public Object get (String name);
public Object remove (String name);
public void setContents (byte[] content);
public byte[] getContents ();
public void replace (Body b);
public void merge (Body b);
}
```

Listing 6.1: The Body of a message can contain a byte array for arbitrary data and objects.

An attachment may for instance contain binary document formats like zip files, images, audio files, etc.

In order for applications which are not ESB-aware to be able to communicate with services within the ESB and vice versa, the concept of a gateway is implemented in the ESB. This is a server which is able to accept messages from ESB-unaware services and relays them to their destination. The gateway accepts arbitrary objects which are contained in files, for instance JMS messages or SQL tables whereas the ESB listeners are only able to process normalized JBoss ESB messages. The gateway takes the object from the ESB-unaware service and constructs an ESB message object from it. This message is sent to an ESB-aware target service. The target service is determined at configuration time and at runtime the registry returns the correct address for the target service.

JBoss ESB Configuration

The two main elements the developer has to configure are providers and services. The <providers> tag includes all bus providers of an instance of the ESB. A <provider> may have several <bus> definitions and various properties.
Within the <services> tag all services of the ESB instance are defined. A service has a name and a category under which it is stored in the Service

Registry, and a human readable description. A service has listeners and actions.
The listener's attributes are its name, a reference to the id of the bus through which the listener receives messages, the limitation of active message processing threads and whether it is an ESB-aware listener or a gateway.
The task of an action is to process the content of a received message. The action is defined through its name, the name of the class which implements the action and the name of the method which processes the message.

6.2 JPF Plugin Manifest

The main parts of the plugin manifest file are described here.

<plugin>

This tag contains all other elements of the manifest file. Furthermore the plugin id, the plugin version, the plugin vendor, the plugin class and the plugin docs path can be specified here, see Listing 6.2.

```
<plugin id="org.test" version="0.0.1" class="org.test.MyTest">
```

Listing 6.2: The **`<plugin>`** *tag defines the name od the plugin as well as the version number and the main class of the plugin.*

<requires>

All other plugins the current plugin depends on must be listed here. This means listing those plugins containing code or resources which are needed by the plugin. The ID of the imported plugin has to be declared and there are several attributes that can be defined, for example ”exported”, ”optional” and ”reverse-lookup”.

If the attribute ”exported” is enabled plugins depending on the current plugin will be able to ”see” the imported plugin. Setting the attribute ”optional” means the imported plugin is not essential for this plugin, and thus no runtime exception is thrown if the plugin to be imported does not exist. When ”reverse-lookup” is set to true the imported plugin is also able to see the code of the current plugin.
The default value of all three attributes is ”false”.

```
<requires>
  <import plugin-id="org.myplugin" exported="true"
    reverse-lookup = true/>
</requires>
```

Listing 6.3: The `<requires>` tag declares the name of the plugin to be imported and whether plugins depending on the current plugin will be able to "see" the imported plugin as well as whether the imported plugin is also able to see the code of the current plugin.

<runtime>

Code and resources which the plugin contributes to the application are specified here. The user has to define a name for the code or exported resource. Moreover the path to the resource or code has to be specified and it also has to be specified whether it is code or resource. A version may also be declared.

```
<runtime>
  <library id="jsp-api" path="lib/jsp-api.jar" type="code">
      <export prefix="*"/>
  </library>
</runtime>
```

Listing 6.4: The `<runtime>` tag contains `<library>` tags which define the name and the path of code or resources of the plugin. It is also defined whether it should be allowed to export code or resources.

The export tag means that this code (or resource) can be imported in another plugin.

<extension point>

This manifest element specifies one or more points where other plugins can be plugged in. The extension point requires a name and the parent plugin id, the parent plugin point and the extension multiplicity may be declared. The extension multiplicity describes how many plugins may extend this extension point. There are four possibilities:

- any: an arbitrary number of extensions can be added at this point.
- one: just one extension can be added at this extension point.
- one-per-plugin: only one extension of a plugin may be added.

- none: it is not possible to add an extension at this extension point because this is an abstract extension point that can only be inherited by another extension point.

An extension point can also define parameters whose values must be provided by an extension. It is up to the user which parameters he wants or needs. A parameter may be a plugin id, a extension point id, a value from a predefined list, a number, a string, a date, a time, etc.

```
<extension-point id="Context">
  <parameter-def id="id"/>
  <parameter-def id="isSystem" multiplicity="none-or-one"
   type="boolean"/>
</extension-point>
```

Listing 6.5: The `<extension-point>` *tag can define parameters whose values must be provided by an extension. It is up to the user which parameters he wants or needs.*

<extension>

The extension is the functionality that is added at the extension point. It must be specified in which plugin the extension is specified, the id of the extension point must be specified and the extension itself must be named. Also the values of the parameters which are expected by the extension point must be defined.

```
<extension plugin-id="org.test" point-id="Context" id="root">
  <parameter id="id" value="standard-root"/>
  <parameter id="isSystem" value="false/"/>
</extension>
```

Listing 6.6: The `<extension>` *tag declares in which plugin the extension is specified. The id of the extension point must be specified and the extension itself must be named. Also the values of the parameters which are expected by the extension point must be defined.*

<plugin fragment>

The plugin fragment tag contains the manifest of a plugin fragment. A plugin fragment belongs to some plugin and contributes code and / or resources to this plugin. A plugin fragment is simple a part of the plugin that has its own manifest file. Nevertheless the plugin and the plugin fragment are loaded with the same class loader.

The plugin fragment must specify its id and among others attributes like version, docs path, vendor, the id and version of the plugin it belongs to.

```
<plugin-fragment id="org.myfragment" version="1.2.4"
plugin-id="org.myplugin" plugin-version="1.2.4">
  <runtime>
    <library id= .../ >
  </runtime>
  <extension plugin-id=..>
    <parameter id= .. />
  </extension>
  <extension plugin-id=..>
     ...
  </extension>
</plugin-fragment>
```

Listing 6.7: The ***<plugin-fragment>*** *tag must specify its id and among others attributes like version*

<doc>

With the <doc>-tag documentation for any part of the plugin (runtime, plugin fragment, etc.) may be created. Additional information may be provided to create something like a javadoc for plugins by automatically processing these tags. The documentation can be simple text or a link to another document.

```
<extension-point id="MyExtensionPoint">
  <parameter-def id="class">
    <doc caption="application class">
      <doc-text>Should implement interface A
      and have empty public constructor</doc-text>
    </doc>
  </parameter-def>
</extension-point>
```

Listing 6.8: The ***<doc caption>*** *tag enables the creation of documentation for any part of the plugin.*

Bibliography

[1] Gaedke M and Rehse J. Supporting compositional reuse in component-based web engineering. In *SAC '00: Proceedings of the 2000 ACM symposium on Applied computing*, 927–933, New York, NY, USA, 2000. ACM Press.

[2] Krueger CW. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[3] McIlroy D. Mass-produced software components. In Naur P and Randell B, editors, *Software Engineering*, 138–155. Scientific Affairs Division, NATO, 1968. Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968.

[4] Szyperski C. *Component Software - Beyond Object-Oriented Programming.* Addison-Weseley, Boston, MA, USA, 1999.

[5] Brown AW and Wallnau KC. The Current State of CBSE. *IEEE Software*, 15(5):37–46, 1998.

[6] Nierstrasz O and Lumpe M. Komponenten, komponentenframeworks und gluing. *HMD - Theorie und Praxis der Wirtschaftsinformatik*, 197:8–23, 1997.
http://www.iam.unibe.ch/ scg/Archive/Paper-
s/Nier97aKomponentenUndGluing.pdf.

[7] Cai X, Lyu M, Wong K, and Ko R. Component-Based Software Engineering: Technologies, Development Frameworks and Quality Assurance Schemes. In *APSEC '00: Proceedings of the Seventh Asia-Pacific Software Engineering Conference*, 372, Washington, DC, USA, 2000. IEEE Computer Society.

[8] Graef G and Gaedke M. Construction of adaptive web-applications from reusable components. In *EC-WEB '00: Proceedings of the First*

International Conference on Electronic Commerce and Web Technologies, 1–12, London, UK, 2000. Springer.

[9] Martin L. *Visuelles Komponieren und Testen von Komponenten am Beispiel von Agenten im elektronischen Handel.* PhD thesis, University of Technology, Darmstadt, 2003.

[10] Sametinger J. Classification of Compostition and Interoperation. Poster Session, OOPSLA ´96, San Jose, CA, 1996. http://www.se.jku.at/publications/pdf/TR-SE-96.17.pdf.

[11] Schmietendorf A, Dumke R, Dimitrov E, and Nakonz S. Bewertungsaspekte der komponentenorientierten Softwareentwicklung am Beispiel von Java-Komponenten. Preprint, 2002. Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg.

[12] Frakes WB and Kang K. Software Reuse Research: Status and Future. *IEEE Transactions on Software Engineering*, 31(7):529–536, 2005.

[13] Zenger M. *Programming Language Abstractions for Extensible Software Components.* PhD thesis, Swiss Federal Institute of Technology, Lausanne, 2004.

[14] Atkinson C, Bär H, Bayer J, Bunse C, Girard JF, Gross HG, Kettemann S, Kolb R, Kühne T, Romberg T, Seng O, Sody P, and Tolzmann E. *Handbuch zur komponentenbasierten Softwareentwicklung.* Fraunhofer Institut Experimentelles Software Engineering, Kaiserslautern, 2003. http://app2web.fzi.de/themen/ap4/cbse_handbuch.pdf.

[15] Ramel S. Software Reuse in Free Software: State-of-the-Art. http://libre.tudor.lu/results/FOSSSoftwareReuse-StateOfTheArt-v1.0.pdf,
2005.

[16] Monroe RT and Garlan T. Style-based reuse for software architectures. In *ICSR '96: Proceedings of the 4th International Conference on Software Reuse*, 84, Washington, DC, USA, 1996. IEEE Computer Society.

[17] Gamma E, Helm R, Johnson R, and Vlissides J. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Weseley, Boston, MA, USA, 1998.

[18] Osterrieder C. *Komponentenmodelle für Web-Anwendungen*. Master's thesis, University of Salzburg, 2004.

[19] Object Management Group. Common Object Request Broker Architecture: Core Specification, 2004.
http://www.omg.org/technology/documents/corba_spec_catalog.htm .

[20] McHale C. *CORBA Explained Simply*. Xhaus.com, Reading, UK, 2004.

[21] COM: Component Object Model Technologies.
http://www.microsoft.com/com/default.mspx
May 15th, 2007

[22] JavaBeans.
http://java.sun.com/products/javabeans/
May 15th, 2007

[23] Sun Microsystems. Java Beans Specification, 1997.
http://java.sun.com/products/javabeans/docs/spec.html.

[24] Nickull D. Service Oriented Architecture, 2005.
http://www.adobe.com/enterprise/
pdfs/Services_Oriented_Architecture_from_Adobe.pdf.

[25] Ortiz S. Getting on board the Enterprise Service Bus. *IEEE Computer*, 40(4):15–17, 2007.

[26] JBoss ESB.
http://labs.jboss.com/jbossesb/
May 15th, 2007

[27] Severiens T. Tutorial: WebServices, 2003.
http://www.aki-
dpg.de/Dokumente/Bad_Honnef_2003/webservicestutorial.pdf.

[28] Spring Application Framework.
http://www.springframework.org
May 15th, 2007

[29] Johnson R. Introduction to the Spring Framework, 2005.
http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework.

[30] Tate B and Gehtland J. *Spring: A Developer's Notebook*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2005.

[31] OSGI-The dynamic module system for Java.
http://www.osgi.org
May 15th, 2007

[32] Spring-OSGI.
http://www.springframework.org/osgi
May 15th, 2007

[33] Raible M. Spring-OSGI with Adrien Colyer, 2006.
http://raibledesigns.com/rd/entry/tse_spring_osgi_with_adrian.

[34] Open SOA Collaboration. Power Combination: SCA, OSGi and Spring, 2007.
http://www.osoa.org/download/attachments/250/Power _Combination_SCA_Spring_OSGi.pdf?version=3.

[35] Kolb B, Lippert M, and Wütherich G. Spring and OSGi: Plattform der Zukunft, 2006.
http://www.it-agile.com/fileadmin/docs/WJAX2006-SpringOSGi.pdf.

[36] Overview of OSGi.
http://www-sop.inria.fr/oasis/Proactive/doc/release-doc/html/OSGi.html
May 15th, 2007

[37] Keen M, Acharya A, Bishop S, Hopkins A, Milinski S, Nott C, Robinson R, Adams J, and Verschueren P. *Patterns: Implementing an SOA Using an Enterprise Service Bus.* IBM.Com/Redbooks, Armonk, NY, USA, 2004.
http://www.redbooks.ibm.com/redbooks/pdfs/sg246346.pdf.

[38] Sample Architectures.
http://dev2dev.bea.com/pub/a/2006/01/ajax-portal-1.html?page=2
May 15th, 2007

[39] Wehner H. Was bringt ein Enterprise Service Bus, 2005.
http://www.computerwoche.de/produkte_technik/software/554063/.

[40] Breitling H. Open Source Enterprise Service Busses, 2006.
http://www.informatik.uni-hamburg.de/SWT/attachments/LVTermine/ESB%20Begriffe%20Konzepte%20Standards.pdf.

[41] WebSphere Software.
http://www-306.ibm.com/software/info1/websphere/index.jsp?tab=integration/esb
May 15th, 2007

[42] Enterprise Service Bus.
http://www.oracle.com/appserver/esb.html
May 15th, 2007

[43] BEA AquaLogic Service Bus.
http://www.bea.com
May 15th, 2007

[44] Olshansky D. Java Plugin Framework.
http://jpf.sourceforge.net
May 15th, 2007

[45] Eclipse.
http://www.eclipse.org
May 15th, 2007

[46] Olshansky D. Emersion Platform.
http://emersion.sourceforge.net
May 15th, 2007

[47] The Apache Ant Project.
http://ant.apache.org/
May 15th, 2007

[48] Javadoc Tool.
http://java.sun.com/j2se/javadoc/
May 15th, 2007

[49] JBoss.
http://labs.jboss.com/jbossas/
May 15th, 2007

[50] JBoss Application Server - Standards Based Infrastructure for the Enterprise, 2005.
http://www.jboss.com/pdf/JBossAS-EnterpriseInfrastructure.pdf.

[51] Azoff M. Application Server Technology AUDIT, 2005.
http://www.jboss.com/pdf/JBossAS-ButlerTechAudit.pdf.

[52] JBoss Group. Getting started with JBoss 4.0, 2006.
http://docs.jboss.org/jbossas/getting_started/v5/html/.

[53] JBoss Group. The JBoss 4 Application Server Guide, 2006.
http://docs.jboss.org/jbossas/jboss4guide/r5/html/.

[54] JBossClassLoadingUseCases.
http://wiki.jboss.org/wiki/Wiki.jsp?page=JBossClassLoadingUseCases
May 15th, 2007

[55] Embedded JBoss.
http://wiki.jboss.org/wiki/Wiki.jsp?page=EmbeddedJBoss
May 15th, 2007

[56] Burke B. Embedded JBoss: JBoss without the Application Server.
http://blogs.jboss.com/
May 15th, 2007

[57] OMG Model Driven Architecture.
http://www.omg.org/mda/
May 15th, 2007

[58] AndroMDA.
http://www.andromda.org
May 15th, 2007

[59] Truskaller T. *Data Integration into a Gene Expression Database.* Master's thesis, University of Technology, Graz, 2003.

[60] Relational Persistence for Java and .NET.
http://www.hibernate.org
May 15th, 2007

[61] Package org.jboss.ejb3.embedded.
http://docs.jboss.org/ejb3/embedded/api/org/jboss/ejb3/embedded/package-summary.html
May 15th, 2007

[62] Khire A, Liu L, and Naderzad A. Service Oriented Architecture using Open ESB.
http://www.snpnet.com/customer_pub/sun/SOA_OpenESB/
May 15th, 2007

[63] Service Oriented Business Integration.
http://java.sun.com/integration/openesb2_0/index.jsp
May 15th, 2007

[64] Open ESB.
https://open-esb.dev.java.net/
May 15th, 2007

[65] JBoss ESB 4.0 GA Programmers Guide, 2006.
http://labs.jboss.com/jbossesb/docs/4.0GA/manuals/html/ProgrammersGuide.html.