INF - 3981

Master's Thesis in Computer Science

Argos Container, Core and Extension Framework

Dan Peder Eriksen

June, 2007

Faculty of Science
Department of Computer Science
University of Tromsø

INF - 3981

Master's Thesis in Computer Science

Argos Container, Core and Extension Framework

Dan Peder Eriksen

June, 2007

Abstract

With the emergence of the internet and e-commerce in the 90's new common problems arose when developing applications that span the internet. These common problems include among others scalability, robustness, networking, database usage and heterogeneity. Software developers creating internet applications saw themselves reinventing the wheel repeatedly. This lead to the creation of middleware systems that aimed to solve these common problems.

This thesis will present Argos which uses a different way of building middleware systems. Argos is able to provide tailored, flexible and extensible middleware support using reflection, dependency injection, Java Management Extensions (JMX) notifications and hot deployment. The result is a platform capable of tackling domain specific challenges. It provides rapid development of feature rich applications for managing and processing information.

Argos has gone through thorough testing proving production stability.

Acknowledgements

I would like to thank my supervisors Arne Munch-Ellingsen and Anders Andersen for invaluable input and helpful discussions.

My dear friend Anne Staurland Aarbakke for proof reading and the incredible working environment she created at the lab during the last year and a half.

Finally my parents, Lisbeth Grande Eriksen and Svein Birger Eriksen, for always supporting and trusting me.

List of Acronyms

ACID Atomicity Consistency Isolation Durability

AOP Aspect Oriented Programming

API Application Programming Interface

APMS A Personal Middleware System

AWT Abstract Window Toolkit

BSD Berkeley Software Distribution

BOA Basic Object Adapter

CCM CORBA Component Model

COMS COntext Management System

CORBA Common Object Request Broker Architecture

DOM Document Object Model

EJB Enterprise Java Beans

HTML Hyper Text Markup language

HQL Hibernate Query Language

IDL Interface Description Language

J2SE Java 2 Standard Edition

JAXB Java Architecture for XML Binding

JCP Java Community Process

JDBC Java Database Connectivity

JEE Java Enterprise Edition

JME Java Micro Edition

JMX Java Management Extensions

JMS Java Message Service

JSE Java Standard Edition

JSR Java Specification Requests

JSP Java Server Pages

JVM Java Virtual Machine

GUI Graphical User Interface

MVC Model View Control

NST Norwegian Centre for Telemedicine

OMG Object Management Group

ORM Object Relational Mapping

ORB Object Request Broker

POA Portable Object Adapter

POJO Plain Old Java Object

QoS Quality of Service

RDMS Relational Database Management System

RMI Remote Method Invocation

RMI-IIOP Remote Method Invocation - Internet Inter-ORB Protocol

RPC Remote Procedure Call

SQL Structured Query Language

TCP Transfer Control Protocol

XHTML eXtensible HyperText Markup Language

XML eXtensible Markup Language

XML-RPC Extensible Markup Language - Remote Procedure Call

XSLT eXtensible Stylesheet Language Transformations

SOAP Simple Object Access Protocol

UML Unified Modeling Language

UP Unified Process

URL Uniform Resource Locator

W3C The World Wide Web Consortium

WS Web Services

WSDL Web Service Description Language

XP Extreme Programming

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Middleware is a software layer that resides between the operating system and applications and is usually found on each side in a distributed system. Distributed middleware systems have gained a lot of momentum since they first emerged in the early 90's and are today widely used for web applications doing e-commerce and integration. The most popular systems at the time of writing are Java Enterprise Edition (JEE) and .NET.

In this thesis a new approach for building middleware platforms will be explored. In doing so a *"tracer bullet implementation"* [1] named *Argos* will be built. A tracer bullet resembles a prototype except that the tracer bullet is not thrown away like a prototype is. Tracer bullet implementations are covered in [1]. The goal is to make Argos easier extendable than current systems. Argos will achieve this by creating a core which can be extended at run-time with the use of the JMX notification model, reflection and deployability.

1.1 Background

With the emergence of the internet and e-commerce in the 90's new common problems arose when developing applications that span the internet. These common problems include among others scalability, robustness, networking, database usage and heterogeneity. Software developers creating internet applications saw themselves reinventing the wheel repeatedly. This lead to the creation of middleware systems that aimed to solve these common problems.

One of the first widely used middleware systems was Common Object Request Broker Architecture (CORBA). CORBA enables software components written in several languages running on different machines to interoperate. The CORBA 1.0 specification was released in October 1991. Since then it has gone through multiple revisions, the latest one being CORBA 3.0.3 in 2004.

Today middleware is at the core of most web applications. The two most widely used systems include Sun's JEE (previously known as J2EE) and Microsoft's .NET. Both middleware platforms offer similar services to the application programmer. These services are among others Remote Procedure Call (RPC), load balancing, transparent fail-over, backend integration, transactions, clustering, thread handling, security, system management, resource pooling and caching. It is important to note that JEE itself is not an actual middleware system but a composition of several open standards. JEE is covered in detail in section 2.2. There are multiple competing implementations of JEE, both open source and

commercial: Web sphere[1], JBoss[2] and Weblogic[3].

The University of Tromsø has a history of doing research in distributed systems and middleware, e.g. The Arctic Beans project [2]. One of the latest projects is A Personal Middleware System (APMS) [3]. The APMS is a middleware platform designed to enable application programmers to rapidly implement pervasive services based on sensor data. The platform is a personal middleware system and can omit requirements enterprise systems have to meet. One of the major requirements an enterprise system has to meet is scalability, which can be disregarded in the APMS. The result is more freedom for developers when creating components, e.g. creating threads, reading files and loading native libraries is allowed in APMS. Such limitations occur in the enterprise systems such as Enterprise Java Beans (EJB) 3.0 [4] because it makes it extremely difficult to handle scalability in a sensible manner. Open connections to sockets or files and multiple threads makes it difficult to put components in a waiting pool.

1.2 Problem Definition

The goal of this thesis is to build a middleware platform that is easy to extend with new functionality in addition to its intrinsic functionality. It should as a start include deployable functionality for web applications, Web Services (WS), database persistence, service management and service distribution. This goal can be split in two intermediate aims:

- Build a *Core* that enables the addition and extension of system functionality at runtime.
- Build system services (deployable functionality) to extend the functionality of the Argos core proving that it can be extended to include new functionality.

  When implementing system services that extend the functionality of Argos existing open source products should be used rather than implementing everything from scratch, e.g. a web container. The system services will be used to bootstrap these open source products and make them available to other services.

1.3 Method and approach

To support the software development process we have chosen to use Scrum, Extreme Programming (XP) and the Unified Modeling Language (UML). How these methodologies shaped the development process is covered in section 7.1.

1.3.1 Scrum

Scrum [5] is a light weight agile method for software development. The name, Scrum, comes from the game rugby, where scrum is a way to restart the game after an accidental infringement.

When a project using Scrum starts, a *Product Backlog* is created. The product backlog is

[1] http://www-306.ibm.com/software/websphere
[2] http://www.jboss.org
[3] http://www.bea.com

a list of all requirements for the final product. The backlog items are prioritized by their importance, the most important backlogs are implemented first. The product backlog never is finalized; it evolves along with the software changing as the requirements change. All participants in the project can add items to the backlog, e.g. developers, users, sales and customers. However, only the *Product Owner* can prioritize the items effectively deciding what will be done by the developers.

Scrum divides the available developers into one or more *Scrum Teams* depending on how many developers are available. It is recommended by [6] that teams consist of 5-8 members. Scrum teams work within a limited time period called a *Sprint*. Within this period a scrum team will take on as many product backlog items as they think can be turned into a working product by the end of the sprint. Every sprint must end with a working product.

Scrum teams maintain a list of task to complete by the end of the sprint. This list is called *Sprint Backlog*. The tasks are moved from the product backlog into the sprint backlog. The status of the tasks in the sprint backlog is maintained as progress is made. This makes it possible to track the progress in a *Burndown graph*. Figure 1.1 shows a burndown graph. The x-axis shows how many hours of work are needed to complete the sprint and the y-axis is the time. Ideally the graph should be linear from top left to bottom right.

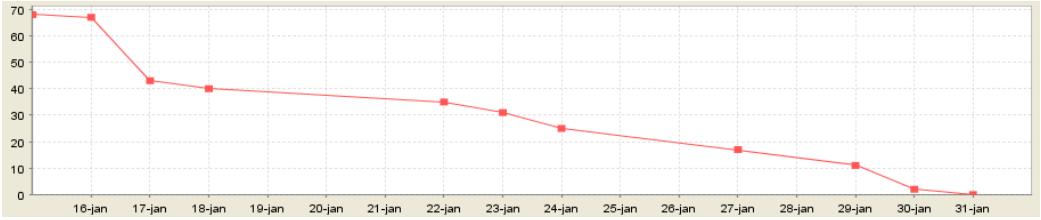

Figure 1.1: Burndown Graph

To coordinate and track progress each team has a daily meeting called *Daily Scrum*. The daily scrum is held by a *Scrum Master*. During the daily scrum each member of the scrum team has two answer three questions:

1. What have you done since the last daily scrum?

2. What do you plan to do until the next daily scrum?

3. Are there any impediments?

Daily scrum meetings are short (5 min) and only the three questions above should be answered. Often these meetings can turn into design discussions. The scrum master is responsible for keeping the meeting quick and to the point.

When a sprint has ended a *Sprint Review Meeting* is held with the scrum team and the management. During this meeting it is decided if what the team has build during the sprint should be built on, scavenged or thrown away. Then the whole process is started again and this keeps going until the project is finished.

1.3.2 Extreme Programming

Extreme Programming (XP) [7] is a light weight agile software development methodology. XP, unlike traditional methodologies, places a higher value on adaptability than predictability.

The planning process in XP is called the *planning game*. The planning game itself consist of two processes: *release planning* and *iteration planning*. The release planing defines the requirements and time frame for a release. Programmers and the customer take part in the release planning. During the iteration planning the requirements are transformed into tasks. Once every task related to a requirement are completed the requirement is fulfilled. Customers do not take part in the iteration planning.

XP uses *Pair Programming*. Pair programming means that two programmers sit at one workstation and work together. One writes code while the other observes. It is recommended that within the development team pairs frequently change [7]. The benefits of pair programming are:

- Keep each other on task.
- Clarify ideas.
- Take initiative when a partner is stuck.
- Holds each other accountable to the team's practices.
- Two persons has knowledge about one piece of code.
- Easier bug discovery.

When problems are solved in XP they are solved in the simplest way possible. Source code is then refactored into the best possible solution for the same problem. Due to the constant change of code *Test Driven Development* has become an important part of XP. Test driven development uses automated test to drive the development process [8]. There is comprehensive use of *Unit Tests*. Unit tests are functions used to evaluate individual units of source code. These unit tests are incorporated into the build process revealing changes to source code that has broken implemented functionality.

1.3.3 Unified Modeling Language

Unified Modeling Language (UML) [9] is a modeling language used to create an abstract graphical model of a system. UML is mostly used for modeling software but is not restricted this one area. UML is standardized by Object Management Group (OMG). When building software UML is used in defining requirements and the design a system. There are 13 types of diagrams used in UML. These diagrams can be split into three types: structure, behavior and interaction.

Only activity diagrams will be used in this thesis. Activity diagrams describe activities and the flow between activities. The reasons for only using one type of diagram are multiple. This project is a small project in terms of software development, spanning 5 months and with one developer. Heavy modeling usage is thus not needed as processes as Unified Process (UP) specifies. Though UP is most often used in large projects where it is possible

to argue that modeling is required. The method used in this project, XP, does not value design much. Designing should be limited to the minimum, and this we have followed. Activity diagrams were used in chapter 5. Activity diagrams gave the wanted level of detail and overview to describe Argos properly.

1.4 Outline

The outline in this master's thesis includes common deliveries in a software project:

- **Chapter 2: Related Work**
  This chapter gives an overview of past and present work being done in middleware systems.
- **Chapter 3: Requirements**
  This chapter defines the requirements for the Argos middleware container.
- **Chapter 4: Technology**
  This chapter presents technologies used in Argos.
- **Chapter 5: Design and Implementation**
  This chapter explains design and implementation of Argos.
- **Chapter 6: Testing**
  This chapter presents the testing done on the Argos core.
- **Chapter 7: Discussion**
  This chapter discusses all results which has been obtained and concludes.
- **Chapter 8: Conclusion**
  This chapter concludes the discussion in chapter 7.

Chapter 2

Related Work

In this chapter related work in the topic of middleware will be presented. Middleware is the software layer that resides between the operating system and applications on each side of a distributed system. Middleware also often supplies services to the application programmer, e.g. transactions and data persistence.

2.1 Common Object Request Broker Architecture

CORBA is a standard [10] maintained by Object Management Group (OMG). OMG consists of over 700 companies and organisations including all the major vendors distributed object technology. The superior goal of CORBA is to allow heterogeneous software components to interoperate in a distributed setting. In this section the object model and architecture of CORBA will be introduced.

2.1.1 Architecture

CORBA uses a distributed object model [11] and the CORBA specification [10] defines the architecture. Objects provide services defined in an interface. Interfaces are defined in Interface Description Language (IDL) [12]. Figure 2.1 shows how requests involving a client, Object Request Broker (ORB) and server object are done.

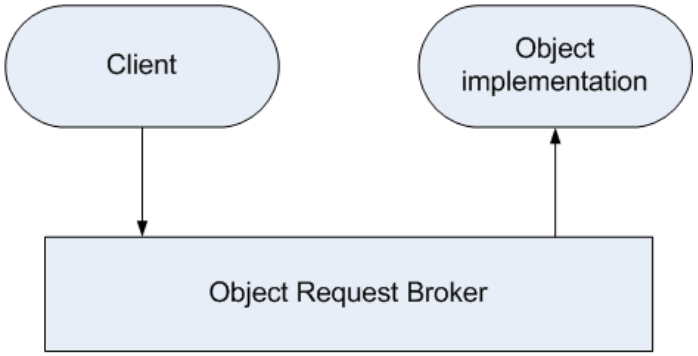

Figure 2.1: Object Request Broker

Object Request Broker

When a client wants to access a server object it does so through the Object Request Broker (ORB). The ORB is responsible for locating the server object and performing the function call. The ORB implements this independent of programming languages. A client written in e.g. java can be serviced by a server object written in C++. Client to server communication is *Location Transparent*. To the client it seems like using a local object when it in fact it is using a remote object.

When a client is doing function calls to a server object, the local object it uses is a proxy. The proxy is called *client stub*. The client stub forwards the call to the ORB which then locates the server object and forwards the call to a *server skeleton*. The server skeleton is another proxy, but resides on the server side with the server object. The skeleton does the actual call to the server object and propagates the result back to the client through the ORB. When requests are done and results are propagated back marshalling and unmarshalling occurs since the results are transmitted across a network. In CORBA marshalling and unmarshalling is performed by the client stubs and server skeletons.

The CORBA specification [10] defines two object adapters, Basic Object Adapter (BOA) and Portable Object Adapter (POA). Object adapters are responsible for object activation and deactivation. BOA was the first adapter to be specified in the CORBA specification. However BOA was inadequately specified leading to incompatible CORBA implementations. As a response OMG later provided the specification for POA. The POA defines activation and deactivation more precise and also supports persistent objects.

2.1.2 Object Model

The CORBA object model is defined in [11]. The specification includes Objects, types, attributes and operations.

Objects

Every object in CORBA has a unique identifier. When a client does a request on a CORBA object it needs to pass the server objects unique identifier to reference the correct object. Client to server communication has location transparency in CORBA, meaning the actual server object location is hidden from the client. CORBA implement location transparency by making all communication go through the ORB, which hides heterogeneity and distribution.

Types

The object model in CORBA is statically typed, meaning all attributes, parameters and return values have a static type. These types are defined by CORBA IDL. The motivation is that static typing enables type safety to be checked when strongly typed programming languages are used. Examples of strongly typed languages using CORBA are Java and Ada.

Object types in CORBA are defined using the keyword *interface* in an IDL file. Object types must have a unique name within its *scope*. The scope is defined by using the keyword *module*. The interface contains attributes, operations, type and exception definitions. To build more complex types like objects a small set of atomic types are needed. These atomic

predefined types are: boolean, short, long, float, char and string. A hello world example interface is given below to illustrate the use of IDL:

Example: IDL Hello World Operation

```
module HelloApplication {
    interface Hello {
        string sayHello();
    };
};
```

Attributes

Attributes in CORBA are used to make the state of a server object accessible to clients. Attributes are bound to two operations, one for reading the value and the other for modifying the value of the attribute. Attributes are defined in IDL. Not all attributes has to have two operations bound to them, CORBA supports read only attribute also. An example of using attributes in our hello world IDL file is illustrated below:

Example: IDL Attribute

```
module HelloApplication {
    interface Hello {
        attribute string greeting;
        string sayHello();
    };
};
```

Operations

The CORBA specification [10] supports operations. Operations have name, return type, parameter list and a list of exceptions the operation may raise. There are three different kinds of parameter modes: in, out and inout. The parameter modes define which way data flows:

- **in**: client to server.
- **out**: server to client.
- **inout**: both client to server and server to client.

An example of an operation which may raise an exception is given below:

Example: IDL Exception

```
module HelloApplication {
     exception MissingName{};
    interface Hello {
         string sayHello(in string name) raises MissingName;
    };
};
```

2.1.3 CORBA Component Model

The CORBA Object Model enables software applications to invoke operations on distributed objects without concern for object location, programming language or OS platform. The CORBA Object Model has the following limitations [13]:

- No standard way to deploy object implementations.
- Limited standard support for common CORBA server programming patterns.
- Limited extension of object functionality.
- Availability of CORBA Object Services is not defined in advance.
- No standard object life cycle management.

To address the limitations in the object model CORBA Component Model (CCM) was added to the 3.0 specification of CORBA. CCM extends the object model by defining features and services to: implement, manage, configure and deploy components commonly used. These services include support for: transactions, security, persistence and event services. CCM allows greater reuse and flexibility for CORBA applications.

2.2 Java Enterprise Edition

Java Enterprise Edition (JEE) is the Java platform targeting enterprises. JEE is a set of open standards for technologies targeting the needs of enterprises. Sun Microsystems define JEE as following: *"Java Platform, Enterprise Edition (Java EE) is the industry standard for developing portable, robust, scalable and secure server-side Java applications."*. Open standards in JEE, and java technology in general, are defined in Java Specification Requests (JSR)'s. These JSR's are developed through the Java Community Process (JCP). JCP is a formalised process which makes it possible for interested parties to take part in future versions and features on the java platform. A JSR goes through several reviews before it becomes final. When a JSR becomes final the JCP executive committee votes on the JSR. JCP is itself described in JSR 215 [14].

In JEE there are over twenty JSR's; including JSR's for web services, web applications, enterprise application technologies, management and security. In the following sub sections Java Servlets (JSR 154) [15], Java Server Pages (JSP) (JSR 245) [16], EJB 3 (JSR 220) [4] and Java Persistence Application Programming Interface (API) (JSR 220) [17] will be presented.

2.2.1 Enterprise Java Beans

EJB is an middleware architecture for the development and deployment of component based business applications. In the JSR 220 specification [4] it says *"Applications written using the Enterprise Javabeans architecture are scalable, transactional, and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification."* EJB focuses on development of business applications and includes support for transactions, security, clustering, integration and much more. EJB covers many advanced topics, however these will not be covered. Rather the fundamental

parts of the technology will be covered. For a more in dept coverage of the topics not covered [18] is recommended reading.

The component model in EJB consists of the *Triads of Beans*, Session Beans, Entity Beans and Message-driven Beans. Since version 3.0 of EJB the triad of beans became Plain Old Java Object (POJO)'s. The main focus of EJB 3.0 was on the programmer and ease of development. These were the lessons learned from EJB 2.1 and the rise in popularity of light weight JEE containers such as Spring [19].

The component model in EJB poses certain constraints when creating components. Components need to conform to these requirements so the EJB container is able to provide all the services the specification demands. Examples of what the components are not allowed to do are:

- open/read/write local files.
- open sockets.
- create/start/stop/suspend threads.
- load native libraries.
- use Abstract Window Toolkit (AWT) functionality.

A full list of restrictions is found in chapter *21.1.2 Programming Restrictions* of [4].

Session Beans

Session beans handle interaction with clients. There are two types of session beans: stateless and statefull. Stateless session beans contain no *conventional state*. Conventional state meaning there can be state but the state can not be specific to any one client. Stateless session beans allow concurrent access to the bean by multiple clients. Statefull session beans however contain conventional state. A statefull session bean is instantiated per client by the EJB container. When the client is done interacting with the session bean it is destroyed.

Sessions beans interact with clients using Remote Method Invocation - Internet Inter-ORB Protocol (RMI-IIOP) and WS. WS enables communication in heterogeneous distributed systems. For more information about WS in EJB JSR 109 [20] can be consulted. Below two examples of session beans are given. Both examples are small hello world samples, one of a stateless bean and the other of a statefull bean.

Example: Stateless session bean

```
package hello;

import javax.ejb.Stateless;

@Stateless
public class HelloBean implements hello.HelloLocal {
    public String hello() {
        return "Hello, world!";
    }
}
```

The stateless session example has one method which is a simple hello world method.

Example: Statefull session bean

```
package hello;

import javax.ejb.Stateful;

@Stateful
public class Hello2 implements hello.Hello2Local {
    private String greet;

    public Hello2() {
        greet = "world";
    }

    public String hello() {
        return "Hello, " + greet + "!";
    }

    public void setGreet(String greet) {
        this.greet = greet;
    }
}
```

The statefull hello world session bean contains state. The state in the example is who to greet.

According to Sun Microsystems EJB tutorial [21] session beans should be used in the following circumstances:

- At any given time, only one client has access to the bean instance.
- The state of the bean is not persistent, existing only for a short period (perhaps a few hours).
- The bean implements a web service.

And statefull session beans should be used if:

- The bean's state represents the interaction between the bean and a specific client.
- The bean needs to hold information about the client across method invocations.
- The bean mediates between the client and the other components of the application, presenting a simplified view to the client.
- Behind the scenes, the bean manages the work flow of several enterprise beans.

Entity Beans

Entity beans are components which model some entity. The entity is persisted to a database. Entities can e.g. be a customer, order, or product. Basically anything that needs to be persisted. Entity beans are stored in Relational Database Management System (RDMS) and each entity bean has their on table in the database. Entity beans make it easy to develop a layer on top of a database that stores and retrieve data. Developing java applications working against a relational database using Structured Query Language (SQL) and Java Database Connectivity (JDBC) is too time consuming and requires a lot of boiler plate code. With entity beans writing SQL is not needed as the EJB container does Object Relational Mapping (ORM).

Entity beans differ from session beans is several ways:

- Entity beans have a primary key that is distinct from their object reference.
- Entities have persistent, client-visible state.
- Entities are not remotely accessible.
- An entity beans lifetime is independent of the applications lifetime.

The following small example of an entity bean models a bank account where it is possible to deposit and withdraw money.

Example: Entity Bean

```
package bank;

import javax.persistence.*;

@Entity
public class Account {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long accountNumber;
    private String ownerName;
    private int balance;

    public void deposit(int amount) {
        balance += amount;
    }

    public int withdraw(int amount) {
        if(amount > balance) {
            return 0;
        }
        else {
            balance -= amount;
            return amount;
        }
    }
}
```

In this example the EJB container will map this entity bean to the following table in the RDMS.

Account Entity Object Relational Mapping

ACCOUNT NUMBER (PK)	OWNER NAME	BALANCE

Table 2.1: Account Entity Object Relational Mapping

Since entity beans can not be used by clients session beans interact with the clients and session bean use the entity beans. An example of a session using the Account entity bean is given below.

Example: Stateless Session Bean using Entity Bean

```
package bank;

import javax.ejb.Stateless;
```

```
import javax.persistence.*;

@Stateless
public class AccountManagerBean implements bank.AccountManagerLocal {
    @PersistenceContext
    private EntityManager manager;

    public Account openAccount(String ownerName) {
        Account account = new Account();
        account.setOwnerName(ownerName);
        manager.persist(account);
        return account;
    }

    public void close(long accountNumber) {
        Account account = manager.find(Account.class, accountNumber);
        manager.remove(account);
    }

    public int getBalance(long accountNumber) {
        Account account = manager.find(Account.class, accountNumber);
        return account.getBalance();
    }

    public void deposit(long accountNumber, int amount) {
        Account account = manager.find(Account.class, accountNumber);
        account.deposit(amount);
    }

    public int withdraw(long accountNumber, int amount) {
        Account account = manager.find(Account.class, accountNumber);
        return account.withdraw(amount);
    }
}
```

This session bean allows a client to create an account, deposit and withdraw money. After the account has been updated (balance adjusted) the source code does not explicitly update the entity component to the persistent manager. This is done by the container automatically.

Message-driven Beans

A message-driven bean is an EJB component which receives Java Message Service (JMS) messages as well as other types of messages. When asynchronous data handling is needed message-driven beans is used in EJB. RMI-IIOP is default way of invoking EJB's. RMI-IIOP is however not used in messaging due to a few limitations. Two examples are:

- Asynchrony: A RMI-IIOP client must wait while the server performs the processing, meaning it is a synchronous operation.
- Support for multiple senders and receivers: RMI-IIOP is for client to server communication and not multiple senders and receivers.

Instead of using client to sever communication in messaging there is a middle man, the message middleware. Applications are able to send messages to the middleware in an asynchronous manner. The middleware then handles relaying messages to one or more receivers.

Figure 2.2: Middleware Messaging

2.2.2 Java Servlets

Java servlets is as anything else in JEE defined in its own specification. Servlets are defined in JSR 154 [15], the latest version is 2.5. Servlets are programs running on a web server and they build web pages. Building web pages at run-time is useful for a number of reasons.

- The data changes frequently.
- The web page is based on user submitted data.
- The web page uses information from a database or other resources.

Servlets (usually) generate Hyper Text Markup language (HTML) or eXtensible HyperText Markup Language (XHTML) (a stricter standard than HTML) output but can also generate eXtensible Markup Language (XML) output parsed with eXtensible Stylesheet Language Transformations (XSLT) by the browser. A small hello world example with HTML is given below.

Example: Java Servlet

```
package hello;

import java.io.*;
import java.net.*;

import javax.servlet.*;
import javax.servlet.http.*;

public class Hello extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet Hello</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Hello, world!</h1>");
        out.println("</body>");
        out.println("</html>");
        out.close();
    }
}
```

The result in a web browser is seen in figure 2.3.

The servlet consist of normal java code, the code runs within a servlet container. The servlet container interacts with the web browser and invokes the right servlet based on the browsers request. The output from the servlet is then sent back to the browser. Since this is normal java code it is possible to e.g. connect to a database or lookup EJB components and use them. Both are often done in production systems to build the web page. Servlet

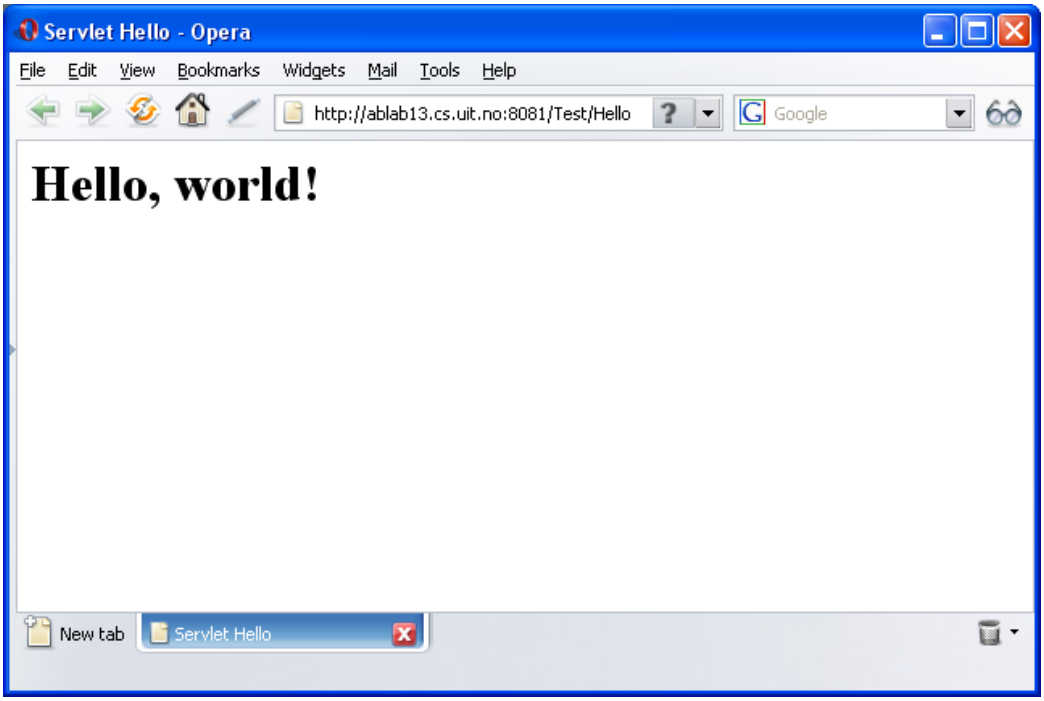

Figure 2.3: Hello World Servlet

are used for creating visual pages, session beans handling the logic and entity beans for handling persistence. This is referred to as Model View Control (MVC). MVC is a three layer architecture. It creates separation between persistence (model), presentation (view) and logic (control). MVC makes it possible to change any of the three parts independently of each other.

Looking closely at the example above it can prove quite time consuming doing an applications presentation layer with servlets. There is a lot of boiler plate code with the out.println() statements. Also, servlets are not very web designer friendly as most web designers do not know how to program. And after all servlets are meant to handle the view of the web application hence having a web designer actually understand what is going on would be a desirable property. Seeing this JSP was created.

JSP is basically web pages with embedded java code in them. When the servlet container gets a request for a JSP page it preprocesses the JSP page and turns it into a servlet. The servlet then is compiled into byte code and then the request is handled by the servlet. A JSP of the same hello world servlet example looks like this:

Example: JSP

```
<html>
    <head>
        <title>JSP Hello</title>
    </head>
    <body>
        <%= "<h1>Hello, world!</h1>" %>
    </body>
</html>
```

Writing a JSP is a lot easier than writing a servlet. There is more to JSP than just embedded java code in web pages, but this will not be covered.

2.3 A Personal Middleware System

A Personal Middleware System (APMS) [3] is as the name suggests a personal middleware system. It has evolved from the COntext Management System (COMS) container developed at the University of Tromsø. The APMS does not aim to solve all the enterprise issues such as scalability, security, clustering and transactions. The APMS middleware container is developed in the java programming language. When developing on the APMS platform the application programmer creates *services* and *components*. Services consist of zero or more components, a deployment descriptor, external applications, desktop widgest and dashboards. The deployment descriptor specifies what components should be loaded, how they should be configured and connected to each other. Components are POJO's. The application programmer can use annotations on methods and attributes to change how the component will behave. It is up to the APMS container to extract annotations from components and handle the components as is specified by the annotations.

In this section the architecture of APMS and the component model will be explained along with a few examples.

2.3.1 Architecture

The APMS container is built upon JMX. This has some really nice implications. It enables the container and the services running on it to be instrumented and controlled at run-time. The container is not a black box where magic happens, it becomes possible take a look inside the box and see the magic as it unfolds.

The APMS architecture is built from eleven components. The components are shown in figure 2.4. The components can be split in two main groups, deployment and services. The components in the deployment group take part in the deployment process of services. Everything from reading configuration files, verifying relationships to class loading. The service group is used by the application programmer created services (from now on called user services). The service group provides the application programmer with common tools which eases the creation of user services on the platform.

The APMS includes a web container (Jetty 5)[1] and hibernate[2]. Most services today, even personal services, have a need to conway their information and services. The web is a good tool for information distribution because of its accessibility and interoperability. Hibernate was included to simplify persisting data for services running on the APMS . Hibernate is an ORM tool written in java. It makes it easy to store and retrieve data from databases. Hibernate has support for EJB entity beans, but this support is not included in the APMS.

[1] http://www.mortbay.org
[2] http://www.hibernate.org

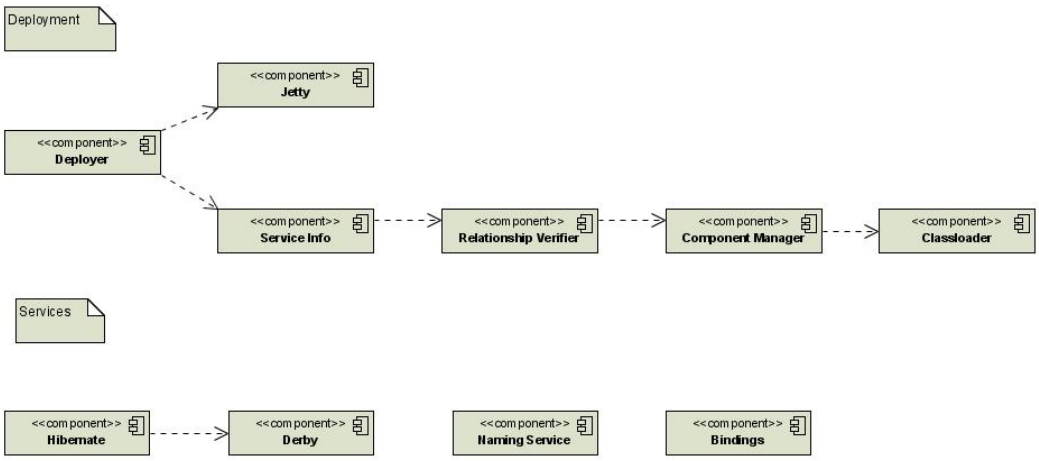

Figure 2.4: APMS Architecture

2.3.2 Component Model

The APMS, as stated, does not try to solve issues enterprises face. This has implications for how the component model is realised. Restrictions put in EJB when it comes to creating threads, creating sockets and reading from files and such is not included in the component model of APMS. Any and everything goes. This is a real benefit when collecting sensor data. To collect sensor data the application programmer has to be able to created threads, sockets and read to/from local files.

Services consist of among others components (class files) and a deployment descriptor (XML file). These files are packed into a jar file called the service file. To deploy a service file it has to be copied to the APMS deploy folder. The APMS will then discover the new service file and start the deployment process, see figure 2.4.

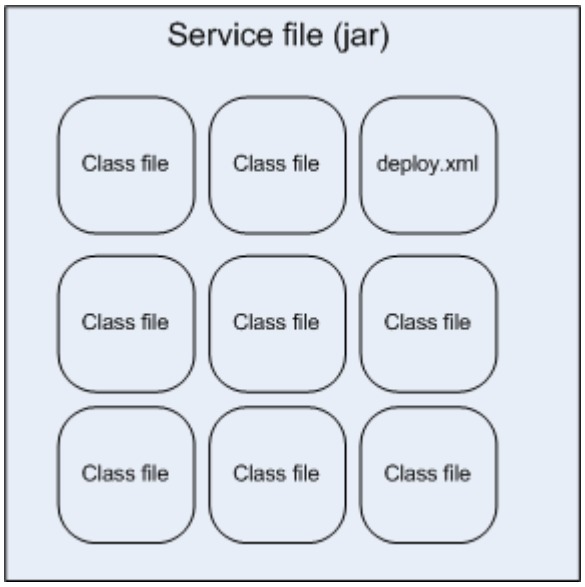

Figure 2.5: Service File

When the APMS starts the deployment of the service it reads the deployment descriptor. The deployment descriptor tells the container how to setup the service. What components should be instantiated and how they should be configured.

When creating services with multiple components these components often need a way to communicate. The APMS is built using JMX.JMX includes a notification model. The APMS takes advantage of this by enabling components to receive and broadcast notifications internally in the container itself and even to components running on different containers. This is a big improvement from the old COMS container where components communicated through a database.

The following example is a small sample application of a service that could not be implemented using EJB.

Example: APMS Component

```
//Imports omitted

public class Weather {
    private String timestamp;

    private double temperature;
    private double pressure;
    private double humidity;

    private double windDirection;
    private double windSpeed;
    private double windGust;

    private Icon webCamera;

    @Execute(30)
    public void collectData() throws IOException {
        //Get weather data
        StringBuffer sb = new StringBuffer();
        Socket socket = new Socket("weather.cs.uit.no", 44101);
        InputStream in = socket.getInputStream();
        int temp = 0;
        while((temp = in.read()) != -1) {
            sb.append((char) temp);
        }
        in.close();

        //Format weather data
        String result = sb.toString();
        while(result.indexOf("  ") != -1) {
            result = result.replaceAll("  ", " ");
        }

        //Extract weather data
        StringTokenizer st = new StringTokenizer(result, " ");
        String time = st.nextToken();
        if(timestamp == null || !timestamp.equals(time)) {
            timestamp = time;
            windSpeed = Double.parseDouble(st.nextToken());
            windGust = Double.parseDouble(st.nextToken());
            windDirection = Double.parseDouble(st.nextToken());

            temperature = Double.parseDouble(st.nextToken());
            humidity = Double.parseDouble(st.nextToken());
            pressure = Double.parseDouble(st.nextToken());

            URL url = new URL("http://weather.cs.uit.no/" +
                        "wcam0_snapshots/wcam0_latest.jpg");
```

```
            webcamera = new ImageIcon(url);
        }
    }

    //Getters for object variables omitted
}
```

The source code defines a class Weather with a method collectData(). The method is marked with the annotation @Execute(30). This tells the APMS that every time a component of type Weather is created the method collectData() should be called every 30 seconds. Inside the collectData() method the component connects to an external sensor using a socket, reads data and puts the data into instance variables. The component itself is instrumented using JMX, this makes the instance variables accessible to other components in the container and to remote components.

To actually load this service with a single component a deploy descriptor is needed. The deploy descriptor for looks like this:

Example: APMS Service Descriptor

```
<?xml version = "1.0" encoding="UTF-8" standalone = "yes"?>

<service name="Weather" version="1.0">
    <deploy>
        <component name="Weather" class="weather.Weather" />
    </deploy>
</service>
```

The deployment descriptor simply states that the service called weather consist of one component. It is also possible to do advanced things like configuring services, setting dependencies and creating listeners through the deployment descriptor.

To actually view the data of the weather service a JMX browser has to be used. In figure 2.6 Argus[3] is being used to display the weather service running on the APMS.

2.4 Summary

With the emergence of the internet and e-commerce in the 90s new common problems arose. These common problems include among others scalability, robustness, networking, database usage and heterogeneity. As a result new kinds of systems were created to tackle these problems. In this chapter related work in middleware systems was presented. CORBA was one of the first middleware systems to be widely adopted. CORBA's superior goal was to allow heterogeneous software components to interoperate in a distributed setting. When CORBA was released it was the first system solving these issues.

Java Enterprise Edition (JEE) is at the time of writing the most used middleware system. JEE is not an actual system but a set of open standards defined by JCP. There exist multiple commercial and open source implementations of JEE. The JEE defines standards for web services, web applications, enterprise application technologies, management, security and persistence. The success of JEE is related to java being easy to use and secure (e.g. no

[3]http://sourceforge.net/projects/argusjmx

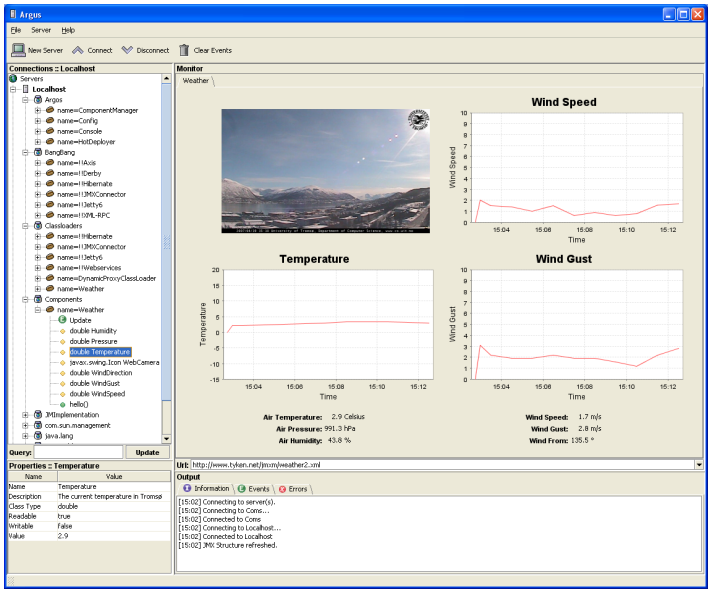

Figure 2.6: Weather Service

buffer overflows). There also is a huge open source community surrounding java. The open source community has created a lot of libraries which makes it easy to do powerful things fast in java.

The A Personal Middleware System (APMS) is a middleware system developed at the University of Tromsø. The APMS does not aim to solve all the enterprise issues such as scalability, security, clustering and transactions. Rather the focus is on personal, pervasive and sensor related services. Since the APMS does not try to solve the enterprise issues it gives the application programmer more freedom in its component model. This enables programmers to read local files and opening sockets, which is needed when working with sensors.

Chapter 3

Requirements

In this chapter the functional requirements of the Argos core and system services will be presented. Non-functional requirements are omitted. By the book requirements should not contain technology specific requirements. This rule is broken in this requirement specification. Argos has as a requirement to be built upon JMX. JMX is set as a requirement because it makes it possible to monitor and control Argos. Argos also has to be built in modular components which is a real benefit.

3.1 Argos requirements

In this section functional requirements for the Argos core will be specified. The requirements can be divided into six main parts: notification model, dependency injection, life cycle support, deployment, JMX related and miscellaneous requirements. Due to the amount of requirements posed to Argos each of the mentioned main parts has been given their own sub section for readability.

3.1.1 Notification model

Argos has to support a notification model so system services can extend the functionality of Argos it self. This is realised by letting system services react to events in the core. Also, this makes asynchronous component to component communication possible internally in Argos, and between Argos containers. Notifications are used because new notifications listeners can be added at run-time.

Requirement MK-1: Emit Core Notification

Requirement Id:	**MK-1**.
Requirement Name:	Broadcast Core Notification.
Priority:	High.
Goal:	Core to system service communication (one way).
Testing:	See requirement MK-2.
Description:	To be able to extend the functionality of the core, system services needs to be informed of events within the core. E.g. service being loaded and components successfully loaded.

continued on next page

continued from previous page

Table 3.1: Requirement MK-1: Emit Core Notification

Requirement MK-2: Register to Core

Requirement Id:	**MK-2**.
Requirement Name:	Register to Core.
Priority:	High.
Goal:	System services to receive core events.
Testing:	1. Create a system service listening for core events. 2. Load another service The system service receiving an event means this requirement is fulfilled.
Description:	For system services to receive events from the core a way to register is needed.

Table 3.2: Requirement MK-2: Register to Core

Requirement MK-3: Deregister to Core

Requirement Id:	**MK-3**.
Requirement Name:	Deregister to Core.
Priority:	High.
Goal:	Make the core stop sending events to a system service.
Dependencies:	MK-2.
Description:	When a system service is unloaded the core should stop sending events to it.

Table 3.3: Requirement MK-3: Deregister to Core

Requirement MK-4: Component Broadcast

Requirement Id:	**MK-4**.
Requirement Name:	Component Broadcast.
Priority:	High.
Goal:	Component to component communication.
Dependencies:	MK-2.
Testing:	See requirement MK-5.
Description:	Components in many services may need to communicate together. Since all components are loaded by the container getting references to each to is not easy. Letting components send notifications to each other makes them able to interact.

continued on next page

continued from previous page

Table 3.4: Requirement MK-4: Component Broadcast

Requirement MK-5: Register to Component

Requirement Id:	**MK-5**.
Requirement Name:	Register to Component.
Priority:	High.
Goal:	Component to component communication.
Testing:	1. Create a component which sends notifications. 2. Create a component receiving notifications If the notification is received this requirement is fulfilled.
Description:	Components can send notifications (MK-4), but they also need a way to receive notifications sent by other components.

Table 3.5: Requirement MK-5: Register to Component

Requirement MK-6: Deregister to Component

Requirement Id:	**MK-6**.
Requirement Name:	Deregister to Component.
Priority:	High.
Goal:	Stop notifications being received by an unloaded component.
Description:	When a component is unloaded it should no longer be receiving notifications from other components. This requirement makes sure components can deregister their listeners so they no longer receive notifications.

Table 3.6: Requirement MK-6: Deregister to Component

3.1.2 Dependency Injection

Dependency injection is in no way vital to the system as all the functionality it brings to the table can be gotten with code lines. However with the ease of development gains dependency injection has it is worth including in the Argos core.

Requirement MK-7: Component Injection

Requirement Id:	**MK-7**.
Requirement Name:	Component Injection.
Priority:	Normal.
Goal:	Let's one component get the reference to another component.
Testing:	1. Create a component with one method. 2. Create another component which gets the first component injected into itself. If the second component is able to call a method in the first component this requirement is fulfilled.

continued on next page

continued from previous page

Description:	A component may need access to data when it suits it self and not the source of the data. Which is the case when communication is done through broadcasting notifications. The component then needs a reference to the other component so it can call methods to it directly.

Table 3.7: Requirement MK-7: Component Injection

Requirement MK-8: Component Meta Data Injection

Requirement Id:	**MK-8**.
Requirement Name:	Component Meta Data Injection.
Priority:	Normal.
Goal:	Easy access to own component meta data.
Testing:	Create a component and use component meta data injection. If the component meta data is not null this requirement is fulfilled.
Description:	Some components may need access to their own meta data (mostly system services). Dependency injection of the component meta data solves that need.

Table 3.8: Requirement MK-8: Component Meta Data Injection

Requirement MK-9: Inject Notification Proxy

Requirement Id:	**MK-9**.
Requirement Name:	Inject Notification Proxy.
Priority:	Normal.
Goal:	Easy access to a components own notification proxy.
Testing:	Create a component with the notification proxy injected into it. If the proxy is not null this requirement is fulfilled.
Description:	When sending notifications a component needs to use a notification proxy. To ease getting a hold of the correct proxy dependency injection can be used.

Table 3.9: Requirement MK-9: Inject Notification Proxy

3.1.3 Life Cycle Support

In this sub section we will look at the requirements related to life cycle support in Argos. Life cycle support is needed by components that use resources during their execution. To make sure these resources are setup properly at start and closed when unloading, life cycle support is used.

Requirement MK-10: Component Load Invocation

Requirement Id:	**MK-10**.
Requirement Name:	Component Load Invocation.
Priority:	High.

continued on next page

continued from previous page

Goal:	Let components initialize themselves.
Testing:	Create a component with a method that should be invoked when the component is loaded. If the method is invoked this requirement is fulfilled.
Description:	Components may need to setup resources before they are ready to do their tasks. This requirement lets them do just that.

Table 3.10: Requirement MK-10: Component Load Invocation

Requirement MK-11: Component Unload Invocation

Requirement Id:	**MK-11**.
Requirement Name:	Component Unload Invocation.
Priority:	High.
Goal:	Let components close open resources.
Testing:	Create a component with a method that should be invoked when the component is unloaded. If the method is invoked this requirement is fulfilled.
Description:	Enables a component to clean up its resources when being unloaded. This makes sure resources are cleaned up properly (if used correctly by components).

Table 3.11: Requirement MK-11: Component Unload Invocation

Requirement MK-12: Create Repeated Invocation

Requirement Id:	**MK-12**.
Requirement Name:	Create Repeated Invocation.
Priority:	High.
Goal:	Make it possible for a component to perform repeated tasks at given intervals.
Testing:	Create a component with a method that should be invoked repeatedly. If the method is invoked this requirement is fulfilled.
Description:	Components may need to do work at certain intervals. This enables the component to e.g. check data streams for data at regular intervals.

Table 3.12: Requirement MK-12: Create Repeated Invocation

Requirement MK-13: Destroy Repeated Invocation

Requirement Id:	**MK-13**.
Requirement Name:	Destroy Repeated Invocation.
Priority:	High.
Goal:	Stop a repeated method invocation.
Description:	When a component is unloaded the repeated method invocation has to stop.

continued on next page

continued from previous page

Table 3.13: Requirement MK-13: Destroy Repeated Invocation

3.1.4 Deployment

All components and services needs to be loaded at some point, this is where the deployment process starts. Not only do they need to be loaded, they also need to be unloaded and reloaded since Argos should support *hot deployment*. The deployment process in Argos is a complicated process. This comes to view by the amount of requirements it has to fulfill. Hot deployment is supported because it makes it easier for developers to test services as they develop. They do not have to stop and start the middleware to update services in the middleware. Also in a production setting updating one service will not disturb other services running on the middleware.

Requirement MK-14: Load System Component

Requirement Id:	**MK-14**.
Requirement Name:	Load System Component.
Priority:	High.
Goal:	Load a system component into the system.
Dependencies:	MK-2, MK-10 and MK-12.
Testing:	Create a system component and deploy it. If it loads this requirement is fulfilled.
Description:	Loading a system component is a bit different than loading a normal component. System components can get notifications from the core thus this has to be setup if needed.

Table 3.14: Requirement MK-14: Load System Component

Requirement MK-15: Unload System Component

Requirement Id:	**MK-15**.
Requirement Name:	Unload System Component.
Priority:	High.
Goal:	Unload a system component from the system.
Dependencies:	MK-6, MK-11 and MK-13.
Description:	This requirement with requirement MK-14 enables hot deployment of system components.

Table 3.15: Requirement MK-15: Unload System Component

Requirement MK-16: Load Component

Requirement Id:	**MK-16**.
Requirement Name:	Load Component.
Priority:	High.
Goal:	Load a component into the system.
Dependencies:	MK-2, MK-10 and MK-12.

continued on next page

continued from previous page

Testing:	Create a component and deploy it. If the component is loaded this requirement is fulfilled.
Description:	This requirement makes sure that user components can be loaded by the system.

Table 3.16: Requirement MK-16: Load Component

Requirement MK-17: Unload Component

Requirement Id:	**MK-17**.
Requirement Name:	Unload Component.
Priority:	High.
Goal:	Unload a component from the system.
Dependencies:	MK-6, MK-11 and MK-13.
Description:	Makes sure components can be unloaded from Argos.

Table 3.17: Requirement MK-17: Unload Component

Requirement MK-18: Read Component Meta Data

Requirement Id:	**MK-18**.
Requirement Name:	Read Component Meta Data.
Priority:	High.
Goal:	Extract meta data about a component.
Description:	This requirement makes sure that the meta data for a component is extracted from the service file.

Table 3.18: Requirement MK-18: Read Component Meta Data

Requirement MK-19: Read Service Meta Data

Requirement Id:	**MK-19**.
Requirement Name:	Read Service Meta Data.
Priority:	High.
Goal:	Extract service meta data.
Dependencies:	MK-18.
Description:	This requirement makes sure that the meta data for a service is extracted from the service file.

Table 3.19: Requirement MK-19: Read Service Meta Data

Requirement MK-20: Validate Service Meta Data

Requirement Id:	**MK-20**.
Requirement Name:	Validate Service Meta Data.
Priority:	High.
Goal:	Verify service meta data.
Dependencies:	MK-19.

continued on next page

continued from previous page

Description:	After the meta data for a service is read there is a need to verify that all required information is present.

Table 3.20: Requirement MK-20: Validate Service Meta Data

Requirement MK-21: Verify Component Dependencies

Requirement Id:	**MK-21**.
Requirement Name:	Verify Component Dependencies.
Priority:	High.
Goal:	Verify component dependencies.
Dependencies:	MK-18.
Description:	Ensures that all component dependencies are present so the component can be loaded.

Table 3.21: Requirement MK-21: Verify Component Dependencies

Requirement MK-22: Verify Service Dependencies

Requirement Id:	**MK-22**.
Requirement Name:	Verify Service Dependencies.
Priority:	High.
Goal:	Verify service dependencies.
Dependencies:	MK-19.
Description:	Ensures that all service dependencies are present so the service can be loaded.

Table 3.22: Requirement MK-22: Verify Service Dependencies

Requirement MK-23: Component Circle Dependencies

Requirement Id:	**MK-23**.
Requirement Name:	Component Circle Dependencies.
Priority:	High.
Goal:	Identify component circle dependencies.
Dependencies:	MK-18.
Testing:	Create a service with two components depending on each other and deploy the service. If the service is not loaded this requirement is fulfilled.
Description:	When circle dependencies are present there is no way to know the order of loading. Thus circles has to be identified and the deployment has to be stopped.

Table 3.23: Requirement MK-23: Component Circle Dependencies

Requirement MK-24: Service Circle Dependencies

Requirement Id:	**MK-24**.

continued on next page

continued from previous page

Requirement Name:	Service Circle Dependencies.
Priority:	High.
Goal:	Identify service circle dependencies.
Dependencies:	MK-19.
Testing:	Create two services depending on each other and deploy the services. If the services are not loaded this requirement is fulfilled.
Description:	When circle dependencies are present there is no way to know the order of loading. Thus circles has to be identified and the deployment has to be stopped.

Table 3.24: Requirement MK-24: Service Circle Dependencies

Requirement MK-25: Service Load Order

Requirement Id:	**MK-25**.
Requirement Name:	Service Load Order.
Priority:	High.
Goal:	Determines the correct load order of services.
Dependencies:	MK-19.
Testing:	Create two services where one depends on the other. If the service with no dependencies is loaded first this requirement is fulfilled.
Description:	When services have dependencies and the dependencies are not already loaded, the correct load order has to be determined. All dependencies for a service has to be place before it can be loaded.

Table 3.25: Requirement MK-25: Service Load Order

Requirement MK-26: Component Load Order

Requirement Id:	**MK-26**.
Requirement Name:	Component Load Order.
Priority:	High.
Goal:	Determines the correct load order of components.
Dependencies:	MK-18.
Testing:	Create two components where one depends on the other. If the component with no dependencies is loaded first this requirement is fulfilled.
Description:	When components have dependencies and the dependencies are not already loaded, the correct load order has to be determined. All dependencies for a component have to be place before it can be loaded.

Table 3.26: Requirement MK-26: Component Load Order

Requirement MK-27: Component Unload Order

Requirement Id:	**MK-27**.

continued on next page

continued from previous page

Requirement Name:	Component Unload Order.
Priority:	High.
Goal:	Determine component unload order.
Dependencies:	MK-18.
Description:	Component A depends on component B. When unloading, component A needs to be unloaded before B. If component B is unloaded first component A will be in an invalid state with missing dependencies.

Table 3.27: Requirement MK-27: Component Unload Order

Requirement MK-28: Service Unload Order

Requirement Id:	**MK-28**.
Requirement Name:	Service Unload Order.
Priority:	High.
Goal:	Determine service unload order.
Dependencies:	MK-19.
Description:	Service A depends on service B. When unloading, service A needs to be unloaded before B. If service B is unloaded first service A will be in an invalid state with missing dependencies.

Table 3.28: Requirement MK-28: Service Unload Order

Requirement MK-29: Create Class Loader

Requirement Id:	**MK-29**.
Requirement Name:	Create Class Loader.
Priority:	High.
Goal:	Create a class loader for a service.
Dependencies:	MK-19.
Description:	To be able to instantiate components a class loader for the service file is needed.

Table 3.29: Requirement MK-29: Create Class Loader

Requirement MK-30: Remove Class Loader

Requirement Id:	**MK-30**.
Requirement Name:	Remove Class Loader.
Priority:	High.
Goal:	Unload service class loader.
Description:	Destroys a service class loader that no longer is being used.

Table 3.30: Requirement MK-30: Remove Class Loader

Requirement MK-31: Load Service

Requirement Id:	**MK-31**.
Requirement Name:	Load Service.
Priority:	High.
Goal:	Load a service.
Dependencies:	MK-14, MK-16, MK-21, MK-23, MK-26 and MK-29.
Testing:	Create a service and deploy it. If the service loads this requirement is fulfilled.
Description:	This requirement makes sure that services can be loaded.

Table 3.31: Requirement MK-31: Load Service

Requirement MK-32: Unload Service

Requirement Id:	**MK-32**.
Requirement Name:	Unload Service.
Priority:	High.
Goal:	Unload a service.
Dependencies:	MK-15, MK-17, MK-28 and MK-30.
Description:	This requirement makes sure that services can be unloaded. If MK-31 is also fulfilled hot deployment of services is possible.

Table 3.32: Requirement MK-32: Unload Service

Requirement MK-33: Ignore Service File

Requirement Id:	**MK-33**.
Requirement Name:	Ignore Service File.
Priority:	High.
Goal:	Ignore a service file.
Description:	Makes sure that a file that has failed loading before does not get loaded again.

Table 3.33: Requirement MK-33: Ignore Service File

Requirement MK-34: Deploy Service File

Requirement Id:	**MK-34**.
Requirement Name:	Deploy service file.
Priority:	High.
Goal:	Deploys services files.
Dependencies:	MK-25 and MK-31.
Testing:	Create a service and deploy it. If the service is deployed this requirement is fulfilled.
Description:	The container needs to be able to take service files and load them. This requirement makes sure that it is possible.

Table 3.34: Requirement MK-34: Deploy Service File

Requirement MK-35: Undeploy jars

Requirement Id:	**MK-35**.
Requirement Name:	Undeploy jars.
Priority:	High.
Goal:	Undeploys service files.
Dependencies:	MK-28 and MK-32.
Description:	This requirement makes sure that it is possible to unload services when seeing that services files that were loaded are missing from the deployment directory.

Table 3.35: Requirement MK-35: Undeploy jars

Requirement MK-36: Redeploy Jars

Requirement Id:	**MK-36**.
Requirement Name:	Redeploy Jars.
Priority:	High.
Goal:	Update services that have changed.
Dependencies:	MK-34 and MK-35.
Description:	When services files in the deployment directory are updated the services should be redeployed.

Table 3.36: Requirement MK-36: Redeploy Jars

3.1.5 Java Management extensions related

To make sure Argos and the services loaded does not become a black box some kind of instrumentation is needed. In this sub section the requirements posed to instrumentation will be covered. Instrumentation will enable will enable application programmers and system administrators to look into the services and components as they are running. This will make debugging easier and also make it possible to change how services and components behave at run-time.

Requirement MK-37: Instrument Component

Requirement Id:	**MK-37**.
Requirement Name:	Instrument Component.
Priority:	High.
Goal:	Enable instrumentation of components in JMX.
Testing:	Create a component and deploy it. If the meta data about the component is available in the MBean server this requirement is fulfilled.
Description:	To enable instrumentation and control of components they have to be made available via JMX. This makes it possible to access components through JMX.

Table 3.37: Requirement MK-37: Instrument Component

Requirement MK-38: Remove Component Instrumentation

Requirement Id:	**MK-38**.
Requirement Name:	Remove Component Instrumentation.
Priority:	Low.
Goal:	Remove instrumentation of components in JMX.
Description:	When components get unloaded they should no longer be instrumented through JMX.

Table 3.38: Requirement MK-38: Remove Component Instrumentation

Requirement MK-39: JMX Method Invocation

Requirement Id:	**MK-39**.
Requirement Name:	JMX Method Invocation.
Priority:	High.
Goal:	Method invocation via JMX.
Dependencies:	MK-37.
Testing:	Create and deploy a component. If methods in the component are callable through JMX this requirement is fulfilled.
Description:	This requirement enables invocation of methods in components through JMX.

Table 3.39: Requirement MK-39: JMX Method Invocation

Requirement MK-40: Impact

Requirement Id:	**MK-40**.
Requirement Name:	Impact.
Priority:	Low.
Goal:	Add meta data about a component
Dependencies:	MK-37.
Testing:	Create a component with impact set on a method. If meta data about the method in the component is accessible through JMX this requirement is fulfilled.
Description:	Enable components to inform, using JMX, what the impact of a method invocation will have on the component.

Table 3.40: Requirement MK-40: Impact

Requirement MK-41: Description

Requirement Id:	**MK-41**.
Requirement Name:	Description.
Priority:	Low.
Goal:	Add meta data about a component.
Dependencies:	MK-37.
Testing:	Create a component using a description. if the meta data is accessible through JMX this requirement is fulfilled.

continued on next page

continued from previous page

Description:	Enable components to attach descriptions to their methods and attributes so they become visible through JMX.

Table 3.41: Requirement MK-41: Description

Requirement MK-42: Instrument Core

Requirement Id:	**MK-42**.
Requirement Name:	Instrument Core.
Priority:	High.
Goal:	Instrument the Argos core using JMX.
Dependencies:	MK-37.
Testing:	This requirement is fulfilled if the core is accessible through JMX.
Description:	This requirement makes sure that the core can be introspected and controlled via JMX.

Table 3.42: Requirement MK-42: Instrument Core

3.1.6 Miscellaneous

All requirements that do not fit under the other five categories of requirements have been placed in this sub section. Most of these requirements are not related to each other, e.g. logging, configuration and naming services. They are however all important to the system as a whole.

Requirement MK-43: Config Lookup

Requirement Id:	**MK-43**.
Requirement Name:	Config Lookup.
Priority:	High.
Goal:	Make Argos and services configurable.
Description:	For components to be configurable through a config file the config file has to be read. The values read have to be accessible to components through a look up.

Table 3.43: Requirement MK-43: Config Lookup

Requirement MK-44: Override Component Attributes

Requirement Id:	**MK-44**.
Requirement Name:	Override Component Attributes.
Priority:	Normal.
Goal:	Override attributes in components.
Dependencies:	MK-43.
Description:	When configuring components through the config file the values set in the config file should override the values set in the service. If not the user has to unpack the service file and chance is content and then pack the file together.

continued on next page

continued from previous page

Table 3.44: Requirement MK-44: Override Component Attributes

Requirement MK-45: Add to Naming Service

Requirement Id:	**MK-45**.
Requirement Name:	Add to Naming Service.
Priority:	High.
Goal:	Enable component and service lookup.
Testing:	Create and deploy a component. If the component is accessible from the naming service after being loaded this requirement is fulfilled.
Description:	Adds a component or service to the naming service enabling component/service lookup.

Table 3.45: Requirement MK-45: Add to Naming Service

Requirement MK-46: Remove from Naming

Requirement Id:	**MK-46**.
Requirement Name:	Remove from Naming.
Priority:	High.
Goal:	Remove a component or service from the naming service.
Description:	This requirement makes it possible to removed components and services added to the naming service.

Table 3.46: Requirement MK-46: Remove from Naming

Requirement MK-47: Naming Service Lookup

Requirement Id:	**MK-47**.
Requirement Name:	Naming Service Lookup.
Priority:	High.
Goal:	Get components and services.
Dependencies:	MK-45.
Testing:	Create a component and deploy it. If the component is accessible from the naming service this requirement has been fulfilled.
Description:	Once components and service have been added to the naming service looking up the components and service should be possible.

Table 3.47: Requirement MK-47: Naming Service Lookup

Requirement MK-48: Setup Security Policy

Requirement Id:	**MK-48**.
Requirement Name:	Setup Security Policy.
Priority:	High.
Goal:	Create a security policy.

continued on next page

continued from previous page

Description:	Sets up the security policy so Remote Method Invocation (RMI) calls are possible.

Table 3.48: Requirement MK-48: Setup Security Policy

Requirement MK-49: Setup Logging Framework

Requirement Id:	**MK-49**.
Requirement Name:	Setup Logging Framework.
Priority:	High.
Goal:	Setup the logging framework.
Description:	Sets up the logging framework, enabling logging for components.

Table 3.49: Requirement MK-49: Setup Logging Framework

Requirement MK-50: Graceful Shutdown

Requirement Id:	**MK-50**.
Requirement Name:	Graceful Shutdown.
Priority:	High.
Goal:	Unload all components before shutting down.
Dependencies:	MK-32.
Description:	Detects shutdown and stops all services gracefully.

Table 3.50: Requirement MK-50: Graceful Shutdown

3.2 Web Container

In this sub section all requirements posed to the integrated web container will be listed. The system service will use a third party implementation of a web container, there for the requirements listed for the actual web application is not listed. Later in this section when *web application* is mentioned what is meant is JSP, servlets and any resources associated with them.

Requirement MC-1: Start Web Container

Requirement Id:	**MC-1**.
Requirement Name:	Start Web Container.
Priority:	High.
Goal:	Bootstrap the web container.
Description:	Binds the web container to specific port enabling web browsers to connect.

Table 3.51: Requirement MC-1: Start Web Container

Requirement MC-2: Stop Web Container

Requirement Id:	**MC-2**.

continued on next page

continued from previous page

Requirement Name:	Stop Web Container.
Priority:	High.
Goal:	Shuts down the web container.
Description:	When the service is unloaded the web container has to be stopped.

Table 3.52: Requirement MC-2: Stop Web Container

+

Requirement MC-3: Add Web Application

Requirement Id:	**MC-3**.
Requirement Name:	Add Web Application.
Priority:	High.
Goal:	Add new web application.
Description:	Makes it possible for other services to add web applications to the web container.

Table 3.53: Requirement MC-3: Add Web Application

Requirement MC-4: Remove Web Application

Requirement Id:	**MC-4**.
Requirement Name:	Remove Web Application.
Priority:	High.
Goal:	Remove a web application.
Description:	Removes a running web application from the web container.

Table 3.54: Requirement MC-4: Remove Web Application

3.3 Web Services

In distributed systems RPC is used for invoking methods across machines. Web services have become the most used way of doing RPC in recent years due to being platform independent. In distributed systems that are heterogeneous such a property is much needed. We want Argos to be able to communicate with other machines and devices e.g. mobile phones. The requirements to make this possible are listed later in this section. There will not written own libraries for doing RPC but rather existing frameworks will be integrated.

Requirement WS-1: Create End Point

Requirement Id:	**WS-1**.
Requirement Name:	Create End Point.
Priority:	High.
Goal:	Create end point.
Description:	Creates a end point for Simple Object Access Protocol (SOAP) and Extensible Markup Language - Remote Procedure Call (XML-RPC) making it possible to call web services.

continued on next page

continued from previous page

Table 3.55: Requirement WS-1: Create End Point

Requirement WS-2: Close End Point

Requirement Id:	**WS-2**.
Requirement Name:	Close End Point.
Priority:	High.
Goal:	Close web service end point.
Description:	When a service using WS is removed the SOAP and XML-RPC end points should be closed.

Table 3.56: Requirement WS-2: Close End Point

Requirement WS-3: Add SOAP Method

Requirement Id:	**WS-3**.
Requirement Name:	Add SOAP Method.
Priority:	High.
Goal:	Expose method.
Dependencies:	WS-1.
Description:	Exposes a method in a component as a SOAP method.

Table 3.57: Requirement WS-3: Add SOAP Method

Requirement WS-4: Remove SOAP Method

Requirement Id:	**WS-4**.
Requirement Name:	Remove SOAP Method.
Priority:	High.
Goal:	Remove exposed method.
Description:	Makes a method in a component no longer invocable with SOAP.

Table 3.58: Requirement WS-4: Remove SOAP Method

Requirement WS-5: Add XML-RPC Method

Requirement Id:	**WS-5**.
Requirement Name:	Add XML-RPC Method.
Priority:	High.
Goal:	Expose method.
Dependencies:	WS-1.
Description:	Exposes a method in a component as a XML-RPC method.

Table 3.59: Requirement MK-5: Add XML-RPC Method

Requirement WS-6: Remove XML-RPC Method

Requirement Id:	**WS-6**.

continued on next page

continued from previous page

Requirement Name:	Remove XML-RPC Method.
Priority:	High.
Goal:	Remove exposed method.
Description:	Makes a method in a component no longer invocable with XML-RPC.

Table 3.60: Requirement WS-6: Remove XML-RPC Method

3.4 Object Relational Mapping

In most middleware systems persistence is an important service offered to the application programmer. Argos is no different here. To make the application programmer able to create services with large amount of data which are accessible in an easy way data persistence is needed. To offer this service Hibernate will be made available with the Apache Foundation created database Derby. The requirements to integrate hibernate and derby is listed below.

Requirement OR-1: Initialize Database

Requirement Id:	**OR-1**.
Requirement Name:	Initialize Database.
Priority:	High.
Goal:	Setup database.
Description:	Makes it possible to initialize the database.

Table 3.61: Requirement OR-1: Initialize Database

Requirement OR-2: Unload Database

Requirement Id:	**OR-2**.
Requirement Name:	Unload Database.
Priority:	High.
Goal:	Stops the database.
Description:	Makes it possible to stop the database.

Table 3.62: Requirement OR-2: Unload Database

Requirement OR-3: User Authentication

Requirement Id:	**OR-3**.
Requirement Name:	User Authentication.
Priority:	High.
Goal:	Secure access to the database.
Description:	Secures the database with a user name and password.

Table 3.63: Requirement OR-3: User Authentication

Requirement OR-4: Database Port

Requirement Id:	**OR-4**.
Requirement Name:	Database Port.
Priority:	High.
Goal:	Remote access to the database.
Description:	Makes the database available through a Transfer Control Protocol (TCP) port.

Table 3.64: Requirement OR-4: Database Port

Requirement OR-5: Find and Add O/R Files

Requirement Id:	**OR-5**.
Requirement Name:	Find and Add O/R Files.
Priority:	High.
Goal:	Add mapping files in hibernate.
Description:	Finds all O/R files in a jar file and adds them to hibernate.

Table 3.65: Requirement OR-5: Find and Add O/R Files

Requirement OR-6: Create Session

Requirement Id:	**OR-6**.
Requirement Name:	Create Session.
Priority:	High.
Goal:	Give components access to hibernate session.
Description:	Makes it possible for components to get sessions from hibernate so they can work against the database.

Table 3.66: Requirement OR-6: Create Session

Requirement OR-7: Stop Hibernate

Requirement Id:	**OR-7**.
Requirement Name:	Stop Hibernate.
Priority:	High.
Goal:	Stop hibernate.
Description:	Makes it possible for Argos to stop hibernate if the service should be unloaded.

Table 3.67: Requirement OR-7: Stop Hibernate

Requirement OR-8: Update Policy

Requirement Id:	**OR-8**.
Requirement Name:	Update Policy.
Priority:	High.
Goal:	Set hibernates update policy.
Description:	Makes it possible to change the update policy of hibernate.

Table 3.68: Requirement OR-8: Update Policy

3.5 Remote Method Invocation Connector

By default it is not possible to connect to the MBean server (JMX) from external applications. This is a feature is wanted since it makes it easier to monitor what is going on inside Argos. To make this possible a RMI connector will be opened. This Connector makes it possible to connect to Argos from external applications. The requirements to such a system service are listed below.

Requirement C-1: Open RMI Port

Requirement Id:	**C-1**.
Requirement Name:	Open RMI Port.
Priority:	High.
Goal:	Open RMI port.
Description:	Opens an RMI port enabling external applications to connect to the Argos MBean server.

Table 3.69: Requirement C-1: Open RMI Port

Requirement C-2: Close RMI Port

Requirement Id:	**C-2**.
Requirement Name:	Close RMI Port.
Priority:	High.
Goal:	Close RMI port.
Description:	Closes the RMI port.

Table 3.70: Requirement C-2: Close RMI Port

3.6 Service Distribution

When a large number of users use the same service the service manufacturer, and users, has a need to keep the software up to date. To make this possible and to show the capabilities of Argos in a distributed setting a system service for service distribution will be developed. The implementation of this service should be as a prototype. The requirements are listed below.

Requirement SD-1: Add Service to Repository

Requirement Id:	**SD-1**.
Requirement Name:	Add Service to Repository.
Priority:	High.
Goal:	Add service to repository.
Description:	Adds a new service to the repository, making it accessible for all current users to get the updated version of the service.

Table 3.71: Requirement SD-1: Add Service to Repository

Requirement SD-2: Remove Service from Repository

Requirement Id:	**SD-2**.
Requirement Name:	Remove Service from Repository.
Priority:	Low.
Goal:	Remove service from repository.
Description:	Removes a service from the repository, no longer making it available.

Table 3.72: Requirement SD-2: Remove Service from Repository

Requirement SD-3: Search for Service

Requirement Id:	**SD-3**.
Requirement Name:	Search for Service.
Priority:	High.
Goal:	Search in the repository.
Description:	Enables users to search the repository for new services.

Table 3.73: Requirement SD-3: Search for Service

Requirement SD-4: Download Service

Requirement Id:	**SD-4**.
Requirement Name:	Download Service.
Priority:	High.
Goal:	Makes repository services available for download.
Description:	Enables Argos containers to download a new service or new version of the service.

Table 3.74: Requirement SD-4: Download Service

Requirement SD-5: Register For Updates

Requirement Id:	**SD-5**.
Requirement Name:	Register For Updates.
Priority:	High.
Goal:	Register for repository updates.
Description:	Makes it possible to get events on services that has been updated so they can be downloaded when updates becomes available.

Table 3.75: Requirement SD-5: Register For Updates

Requirement SD-6: Deregister For Updates

Requirement Id:	**SD-6**.
Requirement Name:	Deregister For Updates.
Priority:	Low.
Goal:	Deregister for repository updates.
Description:	Makes it possible to deregister so the container no longer will get events about a service being updated.

continued on next page

continued from previous page

Table 3.76: Requirement SD-6: Deregister For Updates

Requirement SD-7: Install Service

Requirement Id:	**SD-7**.
Requirement Name:	Install Service.
Priority:	High.
Goal:	Install service.
Description:	Installs a newly downloaded service in Argos.

Table 3.77: Requirement SD-7: Install Service

3.7 Administration GUI

To ease the use and administration of Argos (and to prove system services with a Graphical User Interface (GUI) can be created) a GUI tool will be developed as a system service. The tool should enable the user to control all services and components running on the platform. The implementation should be a prototype.

Requirement GUI-1: Display Service Info

Requirement Id:	**GUI-1**.
Requirement Name:	Display Service Info.
Priority:	High.
Goal:	Expose service information to the user.
Description:	Displays all relevant information about a service in the GUI.

Table 3.78: Requirement GUI-1: Display Service Info

Requirement GUI-2: Display Component Info

Requirement Id:	**GUI-2**.
Requirement Name:	Display Component Info.
Priority:	High.
Goal:	.Display component information
Description:	Displays all relevant information about all components in the GUI.

Table 3.79: Requirement GUI-2: Display Component Info

Requirement GUI-3: Component Configuration

Requirement Id:	**GUI-3**.
Requirement Name:	Component Configuration.
Priority:	High.
Goal:	Enable run-time configuration of components from the GUI.
Description:	Enables the user to configure components from the GUI.

continued on next page

continued from previous page

Table 3.80: Requirement GUI-3: Component Configuration

Requirement GUI-4: Display Panels

Requirement Id:	**GUI-4**.
Requirement Name:	Display Panels.
Priority:	High.
Goal:	Displays Argus panels in the GUI.
Description:	This requirement makes sure that Argus panels can be loaded in the GUI.

Table 3.81: Requirement GUI-4: Display Panels

Requirement GUI-5: Stop Service

Requirement Id:	**GUI-5**.
Requirement Name:	Stop Service.
Priority:	High.
Goal:	Stop a service.
Description:	Enables the user to stop a service from the GUI.

Table 3.82: Requirement GUI-5: Stop Service

Requirement GUI-6: Start Service

Requirement Id:	**GUI-6**.
Requirement Name:	Start Service.
Priority:	High.
Goal:	Start a service.
Description:	Enables the user to start a service from the GUI.

Table 3.83: Requirement GUI-6: Start Service

3.8 Summary

In this chapter all functional requirements for the Argos core and system services was defined. The Argos core has to fulfill fifty requirements. These requirements can be split into five parts: notification model, dependency injection, life cycle support, deployment and JMX related. Most of the requirements in the Argos core also include a test plan. The test plans explain how it is possible to verify that the functionality of the requirement has been fulfilled.

Functional requirements for the six system services: web container, web services, object relational mapping, remote method invocation connector, service distribution and administration were also defined. The requirements were mostly related to boot strapping existing libraries. Test plans for system services were not included.

Chapter 4

Technology

In this chapter technologies which are used and are related to the implementation of Argos are presented. The motivation for using these technologies will not be substantiated in this chapter. Technologies such as XML[22], SOAP [23], XML-RPC [24], RMI [25] and relational databases [26] are considered fundamental and it is assumed that the reader has knowledge about these technologies. Reading section 4.1 is important for understanding chapter 5. As Argos is open source all technologies used in the implementation are also open source.

4.1 JMX

Java Management Extensions (JMX) provides tools for building distributed and modular applications offering monitoring and management. JMX was included in Java Standard Edition (JSE) from version 5. JMX is defined as an open standard in JSR 003 [27]. JMX is used by a lot of JEE containers, such as JBoss[1], WebSphere[2] and WebLogic[3].

The goal of JMX is to offer a standardized interface for monitoring and managing java applications. JMX helps system administrators and developers administer and control applications at run-time by e.g. enabling:

- monitoring critical events, performance problems, fault conditions and statistics.
- changing properties which controls how the application behaves.
- changing debugging level and logging and running debug methods without restarting the application.

In figure 4.1, found in [27], the relationship between the components in JMX is shown. The figure consists of three levels: Distributed, agent and instrumentation.

4.1.1 Instrumentation Level

The instrument level declares a specification which describes how the application programmer can create a JMX managed resource. These resources are called Managed Beans or *MBeans*. MBeans have *attributes* which are readable and writable. Such an attribute may

[1] http://www.jboss.org
[2] http://www.ibm.com/software/websphere
[3] http://www.bea.com

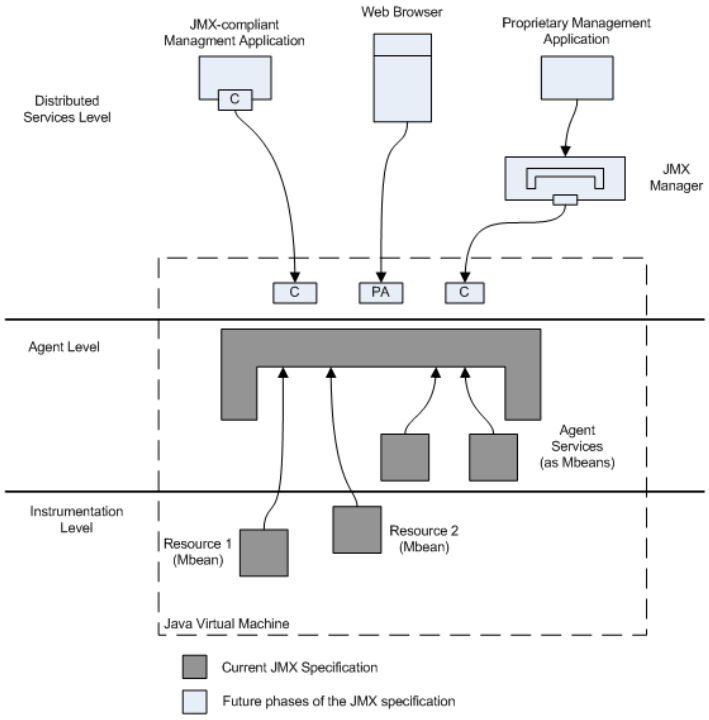

Figure 4.1: Relationship Between the Components of the JMX Architecture

be the debug level, changing its value can turn on debugging without restarting the application. MBeans also have *operations*. Operations are what are called methods in a java object. These operations can be invoked through JMX. The last thing MBeans have is *notifications*. JMX keeps track of which notifications the MBean can broadcast and handles the relaying of the notification.

There are four kinds of MBeans: Standard, Dynamic, Model and Open. To create an MBean a java class needs to implement one of four interfaces, depending on which type of MBean it is. Implementing the methods defined in these interfaces can be quite time consuming.

4.1.2 Agent Level

The agent level contains the *MBean server*. The MBean server is where all MBeans are registered and it provides an entry point for managing the MBeans. In the MBean server it is possible to:

- register/deregister MBeans.
- search for MBeans based on various criteria.
- get MBean meta data.
- invoke operations.
- read/write attributes.
- register/deregister listeners to MBeans.

With all these capabilities it is possible to monitor and manage MBeans.

4.1.3 Distribution Level

The MBean server is located in the agent level. The MBean server however is only available on the Java Virtual Machine (JVM) it runs on. To make it possible to monitor and manage applications remotely the distributed layer is introduced. The distributed layer defines an interface for creating *connectors*. A connector is a proxy for the MBean server that other applications can connect to remotely. How applications interact with the connector is not defined. Only the interaction between the connector and the MBean server is defined. RMI is currently the only implementation offered in JSE but work is being done to offer a connector which uses Web Services (WS) (JSR 262 [28]).

When a connector is defined and added to the MBean server remote monitoring and management becomes possible. In figure 4.2 an application connecting to an MBean server using the RMI connector can be seen. The figure shows an MBean called HotDeployer which has a set of notifications and attributes defined.

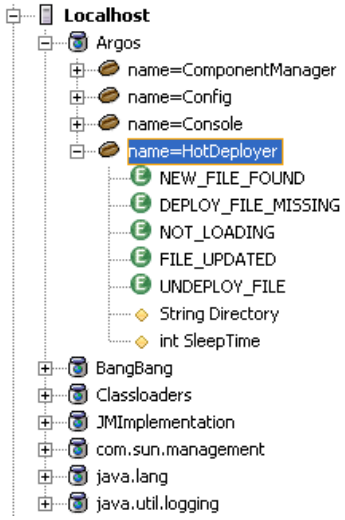

Figure 4.2: Monitoring an MBean remotely

4.2 Jetty

Jetty[4] is a full featured web server written in java. The web server also includes a servlet container based on Tomcats Jasper engine[5]. Jetty is developed and maintained by Mortbay Consulting. There are four defining features of jetty: simplicity, efficiency, embeddability and pluggability.

- **Simplicity:** Jetty is easy to setup and configure.
- **Efficiency:** Jetty is designed to have small footprint and scale well under stress.

[4]http://www.mortbay.org/
[5]http://tomcat.apache.org/tomcat-5.5-doc/jasper-howto.html

- **Embeddability:** Jetty is designed to be component that can easily be embedded into other java applications.
- **Pluggability:** Jetty is architected for pluggability which allows different implementations of core parts in Jetty.

Jetty is in Argos used to serve static documents and dynamic web applications using JSP and servlets.

4.3 Apache Axis

Apache Axis[6] is web application for creating SOAP services in java. Axis version 1.4, which is used in Argos, has support for the following specifications:

- The World Wide Web Consortium (W3C) SOAP version 1.1 [29] and 1.2 candidate recommendation [23].
- W3C Web Service Description Language (WSDL) v1.1 [30].

Axis is used in Argos to make it possible to create SOAP services.

4.4 Apache XML-RPC

Apache XML-RPC[7] is a web application for creating XML-RPC services according to the XML-RPC specification [24]. Version 3 of Apache XML-RPC is being used in Argos. It supports:

- all primitives in java.
- calendar objects.
- all Serializable objects.
- Java Architecture for XML Binding (JAXB) objects.

The web application can also run in streaming mode which greatly improves how resources are handled. Apache XML-RPC is in Argos used to create XML-RPC services.

4.5 Hibernate

Hibernate[8] is an object/relational persistence and query service written in java. Hibernate let's programmers create classes that can be persisted to a relational database. In hibernate the application programmer has to specify the mapping in *hibernate mapping files* (.hbm.xml). These files are XML files and specifies how hibernate should map a class to a

[6] http://ws.apache.org/axis
[7] http://ws.apache.org/xmlrpc
[8] http://www.hibernate.org

table in the relational database. In hibernate there is support for all major databases such as: Oracle 10g[9], MSSQL[10], PostgreSQL[11] and many more.

To access objects stored in the relational database hibernate let's the application query objects using the query language Hibernate Query Language (HQL). However, if there is a need to write SQL hibernate also let's applications do just that.

In Argos hibernate is used with Apache Derby to offer data persistence.

4.6 Apache Derby

Apache Derby[12] is an open source relational database implemented in java. Derby is being maintained by the Apache Foundation and was originally donated to open source by IBM, at the time called Cloudscape. Derby has a small footprint and can easily be embedded in java applications. Derby can also run in a standard client/server mode where clients access Derby through a TCP port using JDBC. Derby supports:

- Atomicity Consistency Isolation Durability (ACID).
- Referential integrity.
- Transactions.
- Unicode.
- Temporary tables.
- Triggers.
- Functions.
- Procedures.
- External routines.

Apache Derby with Hibernate is in Argos used to offer data persistence.

4.7 JUnit

JUnit[13] is an open source unit testing framework for applications written in java. A unit test is a function used to evaluate individual units of source code. JUnit let's developers create unit test and then run them. After the tests are have been executed JUnit presents a report of which tests passed and which tests failed. If a test fails JUnit supplies the developer with the reason for the test failure. Such a reason can e.g. be an exception or that the assertion failed. The unit tests are normally include in the build process of an application.

[9] http://www.oracle.com/database
[10] http://www.microsoft.com/sql
[11] http://www.postgresql.org/
[12] http://db.apache.org/derby
[13] http://www.junit.org

4.8 Class Loading in Java

Class loading in java is a large and complex topic. In this sub section the information required to understand section 5.5.6 (Class loading in Argos) will be presented.

Class loading gives java two fundamental compelling features: dynamic linking and dynamic extension. Dynamic linking allows types to be added into the JVM incrementally. Dynamic extension allows the decision as to which types are loaded into the JVM to be determined at run-time. This means that an application can use types that were unknown or did not even exist when the application was compiled.

A class is defined by its byte code and identified by its name and the class loader that defined the class. That means two classes with the same name can be loaded in two different class loaders and coexist. In java there is not one class loader but a tree of class loaders. By default there is only one node in the tree, the system class loader which defines the classes found in the java API and any jar files added to the class path when the application was started. All class loaders have a parent except the system class loader. When defining a class the JVM starts at the leaf nodes (class loader) and works its way to the system class loader until it finds a class loader who can define the class. If no class loader can define the class a ClassNotFoundException is raised. Class loaders are added to the tree by the application programmer.

4.9 Summary

In this chapter technologies used in Argos are presented. Java Management Extensions (JMX) is used in Argos as a foundation, there for understanding JMX is important for understanding chapter 5. JMX provides tools for building distributed and modular applications offering monitoring and management. JMX consist of three layers: instrumentation, agent and distribution. The instrumentation layer use MBeans monitor and control resources, the agent level serves as an entry point for all manageable resources, and the distributed layer makes the agent level accessible remotely. Open source technologies related to web applications, RPC and data persistence were also presented in this chapter.

Chapter 5

Design and Implementation

In this chapter the design and implementation of Argos will be presented. The methodology when building Argos has followed Scrum and XP. Scrum and XP does not put much weight on design compared to writing code. This is why design and implementation has been put in the same chapter. Argos with its system services is a large and complex system. The most important parts in the Argos core are the deployment process and class loading. These along with an overview of Argos will be presented in this chapter.

5.1 Platform Choice

JMX was a requirement (see chapter 3), the choice of platform could then only fall on Java Standard Edition (JSE). Experience from the APMS has shown that JMX is beneficial when creating a middleware platform. Configuration management of any middleware system is a burden. JMX is a reusable for exposing applications to local and remote management tools. In JMX it is possible to query configuration settings and change them at run-time. JMX also includes common services such as class loading, timers and notifications. JMX gives a clear cut component model to build the middleware platform on and also supplies the middleware platform with remote instrumentation and control.

5.2 Limitations

Argos with its system services is a large and complex system. It is out of this chapter's scope to explain every choice made and details of the whole system. Instead, details of the most important parts of the core will be explained and modelled.

5.3 System Overview

In figure 5.1 an overview of Argos core and the default loaded system services is outlined. The core and all system services are described in their own sub sections.

The circles are core components and system components. The squares are used to show a group of components, either the Argos core or services. The blue arrows shows which components use each other, e.g. the SOAP component using the Jetty component to expose web services. The red arrows show how system components hook into the core and react to events, e.g. Jetty reacts when the core sends out notifications regarding new services being loaded by extracting any web applications from the service file and deploying them to Jetty.

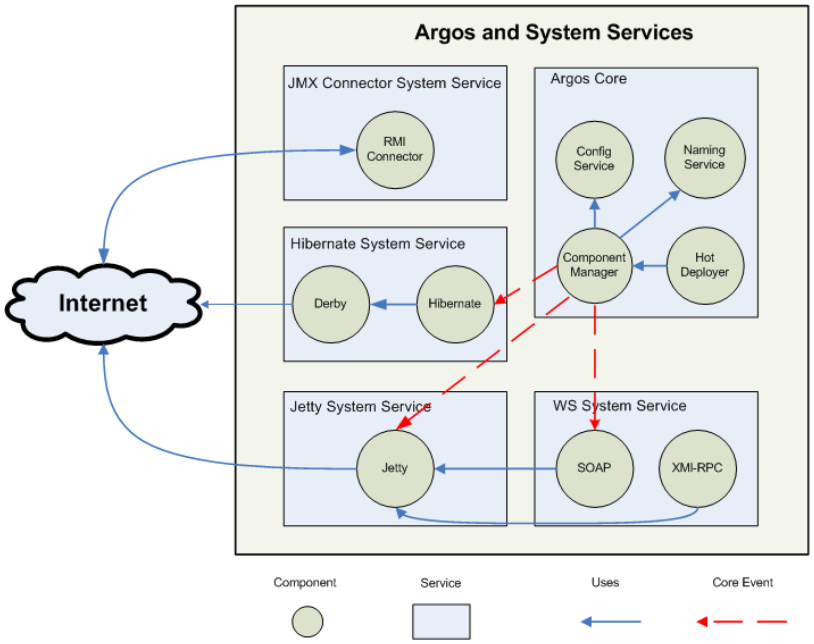

Figure 5.1: Argos Overview

5.3.1 Argos Core

The Argos core is as seen in figure 5.1 quite simplified. The core is defined to the absolute minimum of what is needed to bootstrap the system and create and deploy services. The components shown in the figure will be given a thorough examination in 5.5 and 5.5.6. The core is built on the microkernel pattern [31]. The microkernel pattern includes functionality that enables components running separately from each other to communicate. It also provides interfaces that enables other components to access its functionality.

The core broadcasts notifications when certain events happen, e.g. when services are loaded and unloaded. This is how it communicates with system services. System services are then able to receive these notifications and react to them extending the functionality of the Argos core. The motivation for using notifications are three-fold. JMX defines notifications in its specification. Argos building on JMX can the use notifications out of the box. When broadcasting notifications the broadcaster does not need to know who receives the notifications. At compile time the Argos does not know what system services it will be communicating with, they might not even be created. Thirdly it is possible to add and remove NotificationListener's at run-time. The Argos core needs this feature to supporting hot deployment. Notifications thus makes it possible to extend the functionality of the core while little implementation work has to be done.

5.3.2 JMX Connector

The JMX Connector system service is a simple service which does one important task. When loaded, it opens up RMI on a given port to the MBean server and when unloaded it closes the port. When opening the RMI port the service makes it possible for remote users to connect to the Argos container with JMX, enabling instrumentation and control. This system service was created because it enables remote Argos containers and other management tools to interoperate with the Argos container running this system service.

5.3.3 Hibernate

The Hibernate system service has wrapped the whole hibernate library and when loaded, it bootstraps hibernate and the java database derby. Derby is setup on a TCP port making it possible to connect to it and query the database from tools like DBvisualizer[1]. Derby was setup like this to enable application developers to connect to the database and see the actual data inside it. The hibernate system service was implemented to provide persistence to developers creating services on Argos.

Hibernate is setup to use derby as the default database, however other databases can be used. What is important to note about the hibernate component is that it listen to events from the core. When new services are loaded and the core broadcasts an event the hibernate component reacts by extracting any hibernate mapping files from the newly loaded service and adds them to hibernates mapping.

5.3.4 Jetty

When the jetty system service is loaded it starts the Jetty 6 web container which is included as a library inside the service. The jetty service reacts to events from the core by extracting any web applications found inside a loading service and deploys it in the web container. When services are unloaded the web application associated with the unloading service is also unloaded. This system service was created so developers could easily create web applications.

5.3.5 Web Services

The web service system service, from now called WS service, is the only system service which depends on another system service, jetty. The WS service uses the two apache developed libraries (Axis (SOAP) and XML-RPC) for creating web services. These two libraries are built as servlets and thus require web container, hence jetty is a dependency. The WS service, as hibernate and jetty, springs into action when new services are deployed. It searches for components that should be exposed using web services and if found these are deployed as web services.

The internet consist of heterogeneous systems and devices. To enable all these systems and devices to interoperate with Argos communication that tackles heterogeneity had to be added. Web services was added as a system service to Argos solving this problem.

5.4 Package and class structure

The classes that Argos is composed of are put into ten packages:

- argos
- argos.annotation
- argos.config
- argos.deploy
- argos.logging

[1] http://www.minq.se/products/dbvis

- argos.metadata
- argos.naming
- argos.proxy
- argos.util
- argos.util.graph

Each one of the packages and a subset of the classes will be described in their own subsection.

5.4.1 argos

The main job of classes in this package is related to starting and stopping the core. The argos package consists of two classes:

- **Argos:** Bootstraps the core.
- **ShutdownHook:** Detects shutdown by the user and gracefully stops all services currently running.

5.4.2 argos.annotation

The argos.annotation package is a large package containing many classes (annotations) but has no functional behavior. Components, methods and attributes can be marked with annotations to tell the core how to handle the component. Below a subset of the classes found in this package and what effects they have are explained:

- **Component:** Used to inject one component into another.
- **Description:** Used to add documentation components, operations and attributes via JMX.
- **Execute:** Used to mark a method for repeated invocation.
- **Init:** Tells the core to invoke this method when initializing the component.
- **NotificationHandler:** Tells the core that any notifications sent to this component should be relayed to this method.
- **NotificationSender:** Used to inject a notification proxy (for broadcasting notifications) into a component.
- **Unload:** Methods marked with this annotation will be called when a components is being unloaded.

5.4.3 argos.config

The argos.config package contains only one class, Config. The Config class reads a config file when Argos starts and enables the core, system components and components to look up config values. The Config class make it possible for the Argos core and system service to be configured through a configuration file.

5.4.4 argos.deploy

This package is the work horse of the core. All classes found in this package are related to the deployment process. Deployment is needed to setup services properly with all required resources. The important classes found in this package are:

- **HotDeployer:** Discovers new services that should be loaded.
- **ComponentManager:** Starts the deploy and undeploy processes.
- **JarClassloader:** Handles class loading for services.

The deployment process will be covered in detail in section 5.5.

5.4.5 argos.logging

The argos.logging package contains classes that are related to console and log formatting. When debugging service it is important to use the logging framework. Logs of errors, warnings and information is then kept persistent.

- **Console:** Overrides System.out and broadcasts notifications about everything that is printed to the console.
- **ConsoleFormatter:** Formats all logging that goes to the console.

5.4.6 argos.metadata

In the argos.metadata package classes that contain and handle the meta data of services and components are located. These classes parse XML files found in the service jar files and extract the information found making it available for the core. The core needs data from this package to deploy services properly.

- **ServiceMetainfo:** Parses the service deployment XML file.
- **ComponentMetainfo:** Contains meta data about a component.
- **InformationMissingException:** Thrown when required information is missing, such as component name.

5.4.7 argos.naming

The argos.naming package consists of one class, NamingService. It keeps track of all components, services, proxies, meta data that are currently loaded and allows the core to search for them in multiple ways. Lookup mechanisms are needed by the Argos core during deployment for checking dependencies and by components to interact with other components.

5.4.8 argos.proxy

In argos.proxy the class DynamicProxy and various interfaces that it implements resides. DynamicProxy is the class that hooks components into JMX. It does so by implementing the DynamicMBean interface. When JMX "asks" for which operations, notifications and attributes the component has the DynamicProxy uses reflection on the component to find out what should be exposed. Any call to operations and attributes made to DynamicProxy is relayed to the component and the result is returned. Acting as a proxy between POJO component and JMX. DynamicProxy is covered in detail in section 5.5.5.

5.4.9 argos.util

argos.util contains classes give various utilities to other classes. Such as file copying, unzipping, cleaning temp folders and much more.

5.4.10 argos.util.graph

The classes in argos.util.graph handle graph analysis in Argos. It is used for finding and identifying circle dependencies with in services and components. Finding circles is important to make sure that the core does not go into an infinite loop when loading services and components recursively.

5.5 Deployment

In this section a detailed look at the deployment of services and components in Argos will be given. UML diagrams will be used to explain the deployment process. The Argos core supports hot deployment. Hot deployment means that services and components can be loaded, unloaded and reloaded without shutting down the core. This gives the Argos core two good properties. It eases the development process of services running on Argos because the core does not have to shutdown to reloaded services. Restarting the Argos core with all system services does not take long, however if it is done frequently enough it proves cumbersome and time consuming. Secondly, since the core does not have to be restarted it enables services not affected by the hot deployment to remain running unaffected by the other services that are being hot deployed. This increases up-time.

5.5.1 Deployment Overview

Before a service and its components can be started Argos needs to verify that it can load the service. The process of verifying that it can load it will be explained in this sub section. Figure 5.2 shows an activity diagram of the deployment process. The diagram has been slightly simplified to make it more readable. The start service action which is quite complex has been given its own diagram (figure 5.3) and is explained in section 5.5.2.

Figure 5.2 consist of eleven actions, each action will be explained in their own sub section.

List files in deploy folder

Argos has a deploy folder. In this folder all services files a put. Argos needs to know when new services files are added or existing files are updated. This action does just that.

Filter ignored files

When Argos has tried to load a service and the service for some reason fails this service should not be tried loaded again. The list of files Argos has from the previous action thus has to be filtered, removing all service files that has been tried loaded but failed. If files that fail are tried loaded again they will just fail again. Argos would then try to load the service file repeatedly.

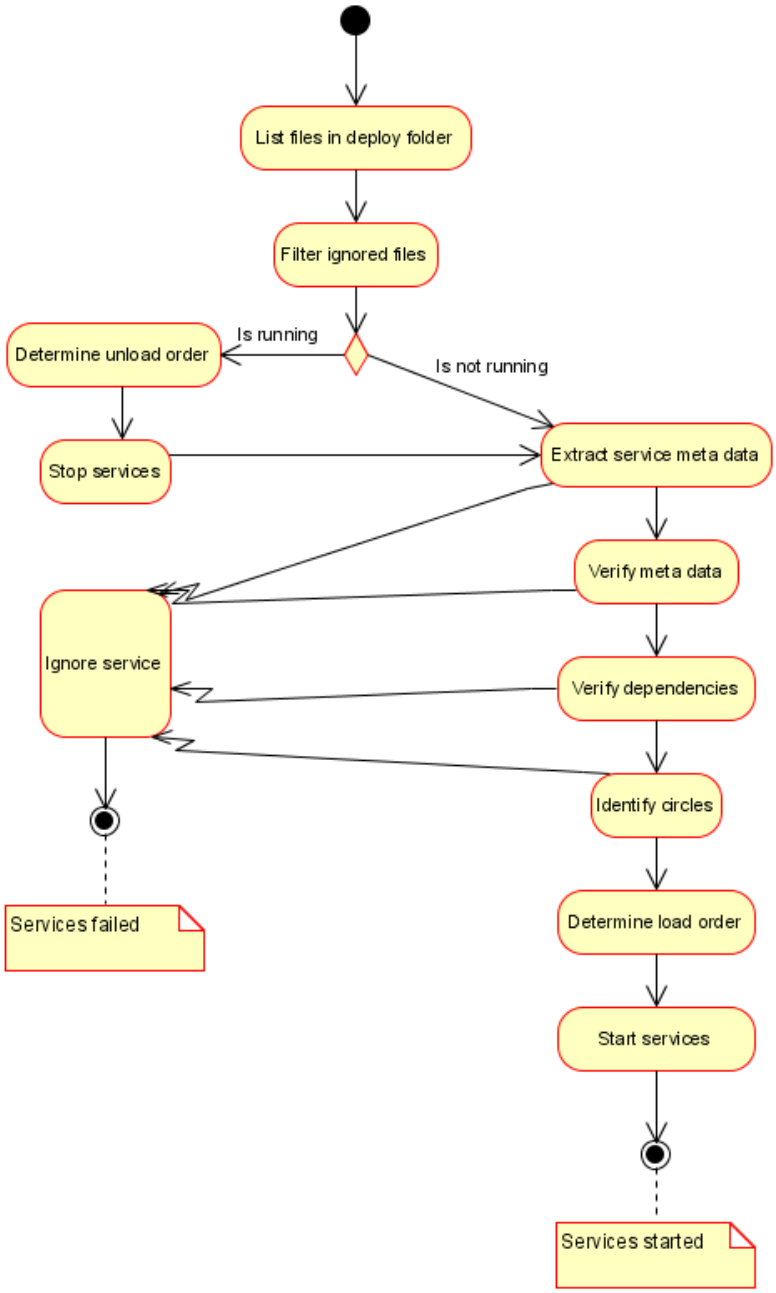

Figure 5.2: Deployment Activity Diagram

Branch

The list of files from the previous action can be divided into two groups. Service files that has already been loaded and service files that has not been loaded. The service files that has already been loaded has to be stopped before they can be deployed again. This branch splits service files in these two groups, one for deployment and one for redeployment.

Determine unload order

Services may have dependencies among themselves. Before services can be stopped a unloaded order has to be created to preserve the integrity of services while they are running. This step uses recursion with Java's sorting API to determine the unload order.

Stop service

This action stops all services passed to it. Stopping a service includes removing: listeners, registered MBeans, class loaders and stopping all components in the service. When stopping components life cycle method invocations have to be done. After the service has been stopped it can be reloaded.

Extract service meta data

Every service file in Argos has to contain a deployment descriptor (deploy.xml). The deployment descriptor is extracted from the service file and parsed using the Document Object Model (DOM) reader that comes with JSE. The deployment descriptor contains information about: service name, component instantiation, component names, component listeners, configurable attributes, dependencies and more. This information is parsed from the deployment descriptor and put into java objects called ServiceMetaInfo and CompoentMetaInfo.

Verify meta data

The "Extract service meta data" handles obtaining the meta information found in the deployment descriptor. When the information has been extracted Argos needs to verify that the sufficient amount of data to loaded the service is present. Meta information such as service name, component name and component class are required to complete the deployment process. This action checks that all required information is in place. If a service has not defined the required meta information the service will be stopped from deployment by raising an exception.

Verify dependencies

Services have the option to specify dependencies on one or multiple services. Before the deployment process is started Argos will check if the dependencies of a loading service is already loaded or if they are in the same load queue as the service it is checking. If dependencies are found to be missing Argos can not supply the functionality (given from depending services) that the loading service requires. The deployment of the loading service is the stopped by raising an exception.

Identify circles

Services can have dependencies on each other. This can lead to service A depending on service B while service B depends on service A. The question is, which service is loaded

first? There is no way of knowing. When dependency circles are found among services the services are removed from the deployment process as Argos is unable to load them. Also, Argos uses recursion when figuring out load order. Not removing dependency circles would lead to Argos entering eternal loop, finally crashing when the stack is full.

Determine load order

Services may have dependencies among themselves. Before services can be started a loaded order has to be created to make sure that all required dependencies of services are loaded when the service itself is loaded. This step uses recursion with Java's sorting API to determine the load order.

Start service

This action starts a service. Starting a service involves starting all components, setting up listeners, configuring attributes and much more. This action is described in its own activity diagram in section 5.5.2.

Ignore service

If a service fails the validation for loading it is added to Argos's ignore list. Services on the ignore list will not be loaded. This is done so services that have failed loading will be loaded again, they would just fail again and again and again.

5.5.2 Start service

An activity diagram for starting a service is shown in figure 5.3. Starting a service is done by going through six actions. There are also two actions related to error handling. Each action is explained below. The diagram shows how they are related.

Copy service file

Before class loading can proceed the service file has to be copied to a temp directory. The class loader used in Argos extends URLClassLoader. This class loader keeps jar files open so it is able to access the content of the file fast. On windows open files can not be deleted or rewritten, hence if the class loader loads the jar file in the deploy directory, the user will not be able to overwrite the file. Overwriting the file is done when doing hot deploying.

Create class loader

Based on the dependencies described in the deployment file of the service a class loader is created. This action can result in three different class loaders based on how the dependencies are defined. If there are no dependencies a class loader with only the service file is created. If the service has one dependency a class loader with the service file is created. This class loader also has its parent class loader set to the class loader of the service that the loading service depends on. The third option occurs when the service depends on multiple services. Then a ChainClassLoader is created. The ChainClassLoader contains the service file and the class loader of all depending services.

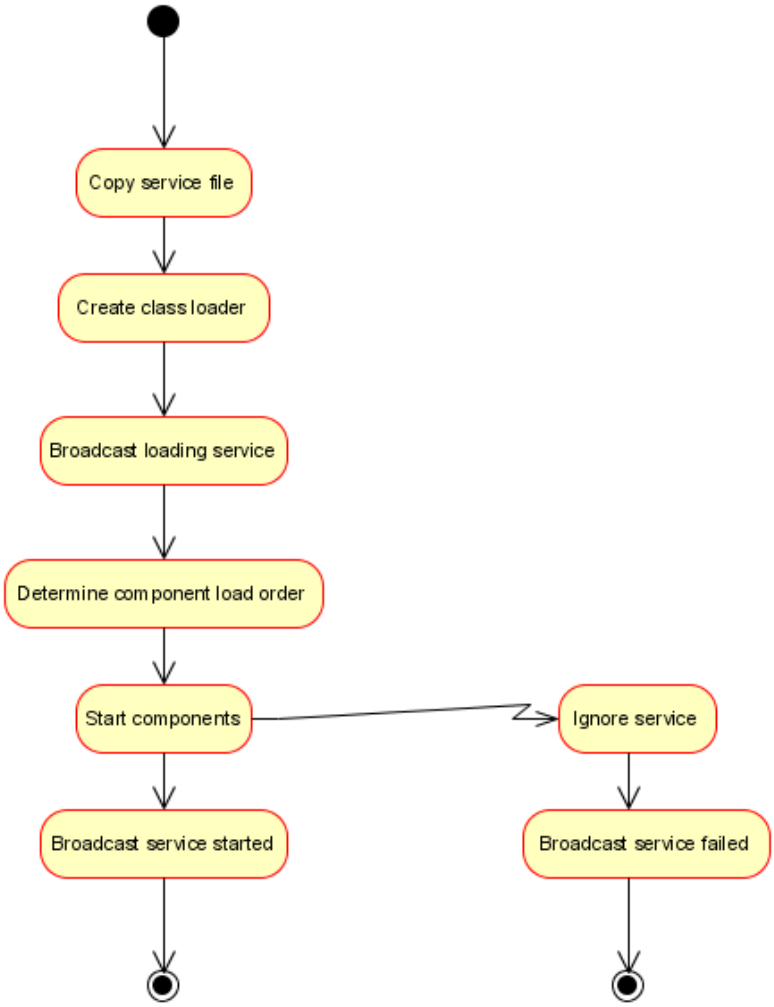

Figure 5.3: Start Service Activity Diagram

Broadcast loading service

Right before the core starts creating the components it broadcasts an event. Interested system services will then receive this event enabling them to any actions they require before components are started. At the current time of writing no system services use this event. However it is important to include as many event types as possible as it is impossible to know what events system services that has not been written yet need.

Determine component load order

Components can have dependencies (like services), thus they have to be loaded in the right order to preserve the integrity of components that are started. This action uses the same code as when the service load order is determined.

Start components

The actual starting of components includes instantiation the component, setting configurable attributes, setting up listeners, invoking life cycle methods and much more. The starting of components is covered in detail in its own section, section 5.5.3.

Broadcast service started

When the service has been started the core broadcasts this to all system services so they are able to act on this event. This is the event that all system services at the time of writing uses. They use the event among others to look for annotations in components and find web applications in the service file.

Ignore Service

If a service fails the validation for loading it is added to Argos's ignore list. Services on the ignore list will not be loaded. This is done so services that have failed loading will be loaded again, they would just fail again and again and again.

Broadcast Service Failed

When a service fails loading an event is broadcasted from the core so system services can clean up any resources they have allocated to the service that failed. Not currently used in any system service as all services react to the event that is broadcasted when a service has been successfully loaded.

5.5.3 Start Component

The activity of starting a component is shown in figure 5.4. Every component in a service will have to go through these steps before they are considered loaded successfully.

Register MBean Class Loader

RMI calls can be made through JMX. For RMI to be successful JMX needs to be able to get the reference to class loaders of the MBean that is being invoked. Thus all components have their class loader registered as an MBean in the MBean server. JMX is then able to get the right class loader when MBeans are invoked with RMI.

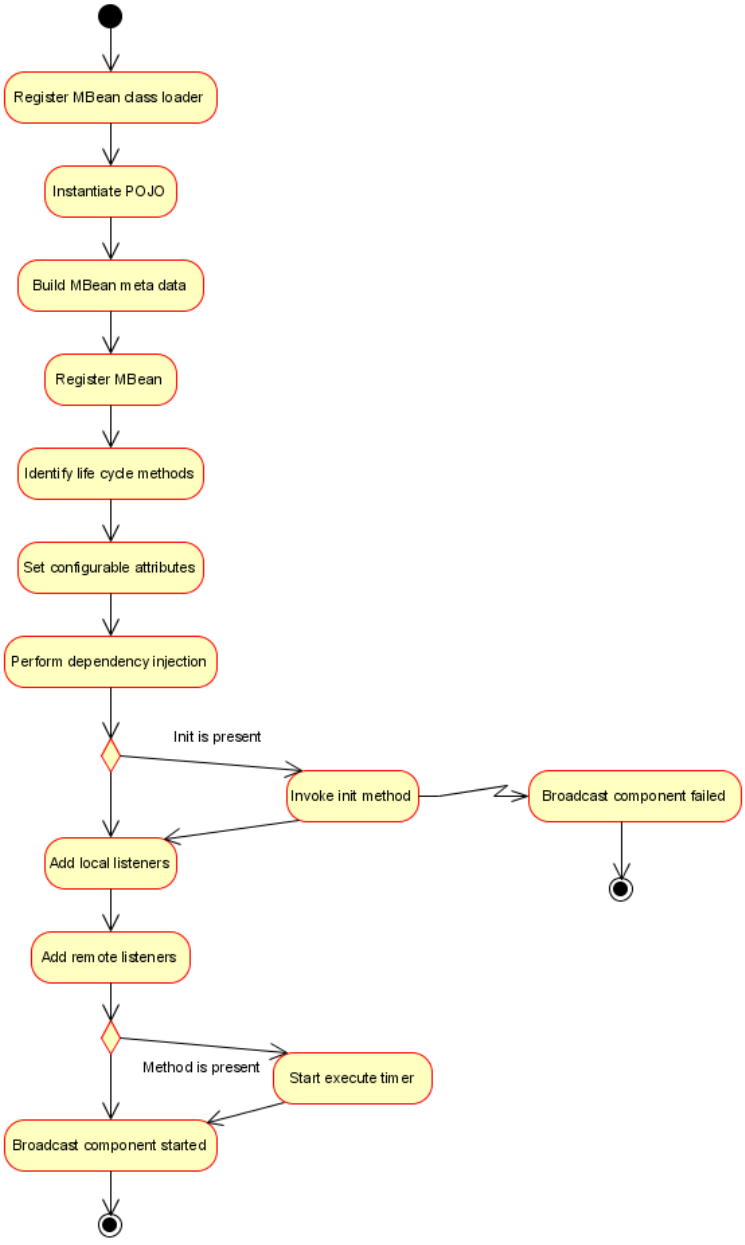

Figure 5.4: Start Component Activity Diagram

Instantiate POJO

The next step in starting a component is to create a instance of the component. The class name is extracted from the meta information reading in a previous step. The reflection API is then used to instantiate the component. The component needs to have defined a public constructor with no parameters or the instantiation will fail.

Build MBean Meta Data

Before a MBean can be created from the component the meta information required by JMX needs to be extracted. This information is found in the byte code of the component. Reflection is used to extract the required meta information. This action is covered in more detail in section 5.5.5. Reflection is used to give JMX the necessary information about the component. This information includes operations, attributes and notifications.

Register MBean

When all meta information required from JMX has been collected from the component it can be registered as an MBean. Once registered as an MBean it is possible to monitor and manage the MBean both locally and remotely.

Identify Life Cycle Methods

The Argos core supports life cycle by invoking methods in components when certain events happen. Currently the core supports three life cycle methods: init, execute and unload. Init is called right before a component is loaded. Execute methods are methods which are called repeatedly at given intervals. Unload is called right before a component is unloaded. Methods marked with annotations identifying life cycle methods are found in this action, by using reflection.

Set Configurable Attributes

Argos support setting the values of configurable attributes in components from the deployment descriptor. Information gathered during reading the deployment descriptor is used in this action and values are set accordingly. It is also possible to override values set in the deployment descriptor in the configuration file of Argos. The config component of the Argos core is used for this.

Perform Dependency Injection

The Argos core supports dependency injection of: components, class loaders, component and service meta information. During this step reflection is used to find annotations related to dependency injection. If any are found the NamingService in the Argos core is used to find the appropriate resource and the resource is injected into the component.

Init Branch

During a previous step (Identify Life Cycle Methods) life cycles methods where identified. If a method is tagged with init a branch is done.

Invoke Init method

If the branch occurs the init method of the component is invoked by the core. In this method a component can setup any resources used during the life span of the component. Such resources can be database connections, sockets or file streams. If the invocation of the method fails the service the component is in, will be stopped completely.

Add Local Listeners

Component to component communication can be done in two ways in Argos. Either through direct method calls or through notifications. For notifications to work, listeners have to be setup. Listeners are defined in the deployment descriptor of the service. Any local listeners defined for the component is setup properly in JMX. Local listeners meaning both the sending and receiving components are contained in the MBean server of Argos.

Add Remote Listeners

Any remote listeners defined for the component is setup properly in JMX. Remote listeners meaning both the sending and receiving components are contained in different Argos containers.

Execute Branch

During a previous step (Identify Life Cycle Methods) life cycles methods where identified. If a method is tagged with execute, a branch is done.

Start Execute Timer

Argos supports repeated invocation of methods in components. This enables component to do repeated work such as reading data from a socket or writing to files. A JMX timer is created. This timer broadcasts notifications at given intervals. These notifications are passed to the ComponentRunner of the component. When the ComponentRunner gets the notification it invokes the correct method in the component.

Broadcast Component Started

The core broadcasts that the component has been started so system services can react to the event.

5.5.4 Component Overview

In this sub section a more architectural overview of the objects involved per component and how information flows to and from components will be given. Figure 5.5 shows an overview of the objects created per component and how they interact with the POJO component. When referring to POJO in this section it means the application developed component. Components were chooses to be POJO's because they are simple to implement. The application programmer needs little knowledge of Argos to create components. Also it makes it possible to port existing applications quite easily to Argos.

JMX acts as the bus for everything that happens. In the MBean server three components are registered, the class loader, JMX timer and dynamic proxy. The POJO itself is not registered as an MBean.

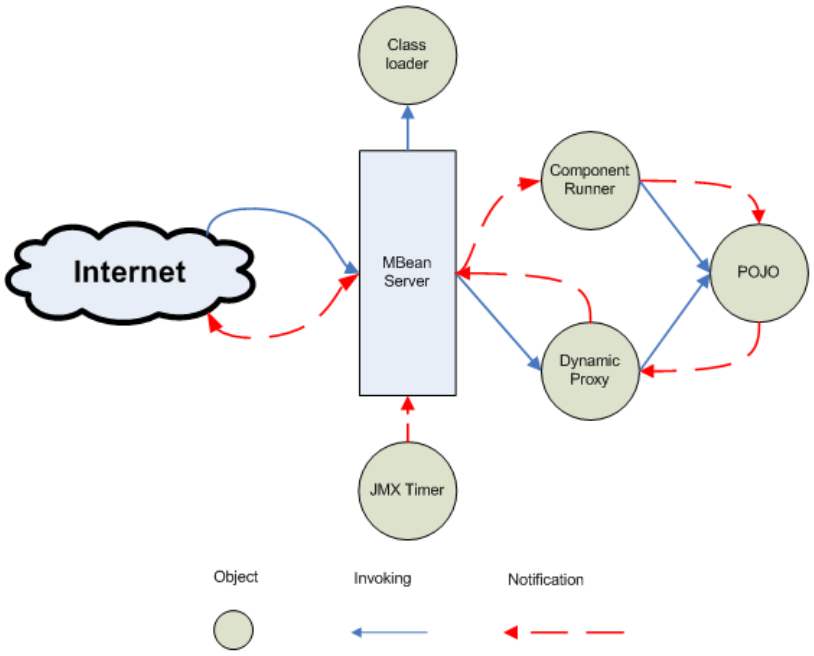

Figure 5.5: Component Overview

The dynamic proxy acts as a proxy between JMX and the POJO. POJO's can not be added as MBeans since they do not have the require interfaces implemented. This is why the proxy is created. The proxy behaves like the POJO and has the required interfaces implemented so it can be added as an MBean. This way the POJO can have simple programming model (no interfaces have to be implemented) while still getting the benefits of JMX (instrumentation and control).

DynamicProxy relays any invocation done via JMX to the POJO (blue arrow) and returns the result to JMX. DynamicProxy is also used to send notifications. The proxy is registered as an MBean thus the notification has to come from the proxy and not the POJO. The POJO gets a reference to the proxy using dependency injection and sends the notification to the proxy which relays it to JMX (red lines).

Receiving notifications in the POJO is not done through DynamicProxy but an object called ComponentRunner. The DynamicProxy could not be used to receive notifications. Classes that receives notifications needs to implement the interface NotificationListener. This interface extends Serializable so JMX are able to move the notification listeners to other MBean server (remotely) if necessary. DynamicProxy uses reflection heavily, thus using the class Method. The Method class does not implement Serializable. This makes it impossible to move DynamicProxy to remote MBean servers. ComponentRunner was instead used as the NotficationListener since it contained no object variables that were not serializable.

ComponentRunner implements the NotificationListener interface and is registered in the MBean server. When it receives notifications they are relayed to the POJO using reflection. There are two types of arrow going from ComponentRunner to the POJO, one red and one blue. The blue one is present as the ComponentRunner gets notifications from the

JMX Timer thread. The listener then invokes the execute method of the POJO instead of replaying the notification. All other notifications are just relayed to the POJO.

The class loader of the component is shared among all components in the service. This class loader is registered as an MBean in JMX MBean server. The MBean server is then able to resolve classes used in e.g. RMI calls made to the component through JMX.

5.5.5 Dynamic Proxy

DynamicProxy is one of the most important classes in Argos. It registers itself in the MBean server and reflects the behaviour of the user component to the MBean server. This makes it possible to register a user created component as an MBean without having to implement the interfaces required by JMX. Getting the power of JMX, monitoring and management, while keeping a simple component model.

In JMX it is possible to create four types of MBeans: standard, model, open and dynamic. A standard MBean has to implement an interface defining methods exposed in JMX. Standard MBeans were not chosen because of the extra work posed on the application programmer by having to define an interface with instrumented methods. Model MBeans could have been used to give the same functionality as DynamicProxy. However in a model MBean it is not possible to control what methods and attributes that get instrumented, everything gets instrumented. An open MBean could also have been used to some extent. The problem with open MBeans is that only methods with scalar types (String, boolean, int and so on) can be instrumented. The choice of MBean type fell on a dynamic MBean. When using a dynamic MBean with the proxy pattern and reflection any and every java object can be instrumented and controlled. Also it makes it possible to control what methods should be instrumented if needed.

In DynamicProxy the proxy pattern [32] is used. A proxy is a class functioning as an interface to something else. It is used when there is a need to represent a complex object by a simpler one. A proxy usually has the same methods as the object it represents. Method calls are then simply relayed to the object the proxy represents.

When DynamicProxy is instantiated it takes on argument in its constructor. The argument is the user component it will act as a proxy for. DynamicProxy then uses reflection on the user component to extract all the information it needs to mimic the user component to JMX. This information includes constructors, methods, attributes and any notifications the component may broadcast. When the information has been collected DynamicProxy passes it on to JMX completing the instrumentation of the MBean. From this point any method invocations made to the MBean (DynamicProxy) is relayed to the POJO.

Using the proxy pattern with reflection proves very powerful as it makes it possible to turn any java object into an MBean.

5.5.6 Class loading in Argos

Class loading gives java two fundamental compelling features: dynamic linking and dynamic extension. An application can use types that were unknown or did not even exist when the application was compiled. This is the case for Argos. In Argos there are two types of class loaders used depending on the dependencies defined in the service deployment file:

- OpenClassloader
- ChainClassloader

The open class loader is used in two cases: when a service has no dependencies and when a service has one dependency. In the case of the service having no dependencies the open class loader simply contains the service file which makes it able to resolve all classes found in the service file. If the service has one dependency the open class loader will have its parent set to the class loader of the service it depends on.

The chain class loader is used in the case of a service having dependencies on multiple services. A class loader can have only one parent and this causes a problem. The solution is to create a class loader which contains multiple other class loaders (the service class loaders the service depends on). When the chain class loader is "asked" by the JVM to define a class it relays the call to the other class loaders one by one until it finds a match.

5.6 Meta Model

Argos keeps a meta model of services and components. This model can be split in two. Argos keeps a set of meta information while the MBean server also has one set. Together these sets constitute the Argos meta model. Argos needs the meta model to figure out how services should be deployed and how JMX should instrument components in the service.

The MBean servers set of meta information consists of components (MBeans), component descriptions, operations, operation descriptions, operation impact, attributes and notifications. The meta model information is centred around components and what they can do. The set of meta information Argos holds has a different characterisation. It holds meta information centred around the deployment process and answers the following questions:

- which components/services should be loaded.
- what order should they be loaded (dependencies)?
- which components should be connected to other components?

Together the meta information in the MBean server and Argos define components/services, how they should be loaded and how they interact.

5.7 Summary

In this chapter the design and implementation of Argos was presented, with the main focus on the Argos core. Since scrum and XP was used as development methodologies it was natural to merge design and implementation into one chapter. The Argos core is responsible for the deployment of services. A service with the default setup of Argos consist of components, web applications, web services, Java Micro Edition (JME) applications and Argus panels. There are two types of services: system and user services. System services extend the functionality of Argos using annotations and reflection.

Chapter 6

Testing

There have been four types of test carried out on Argos: informal testing, beta testing, unit testing and performance testing. Testing during the development of Argos has been done by creating services and deploying these to check that implemented functionality works as it should (informal testing). A unit test is a function used to evaluate individual units of source code. JUnit was used for unit testing. Unit testing was not done during the development but added after the implementation was done. The reason to not follow unit testing as XP states was because of time constraints. In retrospect this was a bad choice. This will however be discussed in chapter 7. Performance testing of the deployment process and component to component interaction was performed, results are presented in section 6.3.

6.1 Unit testing

Unit testing was done using JUnit (see section 4.7) and was limited to the Argos core. In chapter 3 all functional requirements which were unit testable contains a testing plan. The test plan describes how the requirement can be tested to ensure that the requirement has been fulfilled. In total there were 19 unit tests created testing 25 functional requirements in the core. Some of the functional requirements that were not explicitly tested were implicitly tested through other requirements.

Figure 6.1 shows Eclipse using JUnit to run the unit tests. The unit tests can be seen to the left in the figure. All 19 unit tests have been executed and passed, verifying that all testable requirements have been implemented according to the requirements. JUnit does not support test ordering. Before and after each test is done a service is loaded and unloaded. Each individual unit test is explained below (MK-X refers to requirements defined in chapter 3):

Name:	**Testing**	**Description**
Service Loaded	MK-14, MK-16, MK-21, MK-34, MK-45 and MK-47	Verifies that a service is loaded correctly.

continued on next page

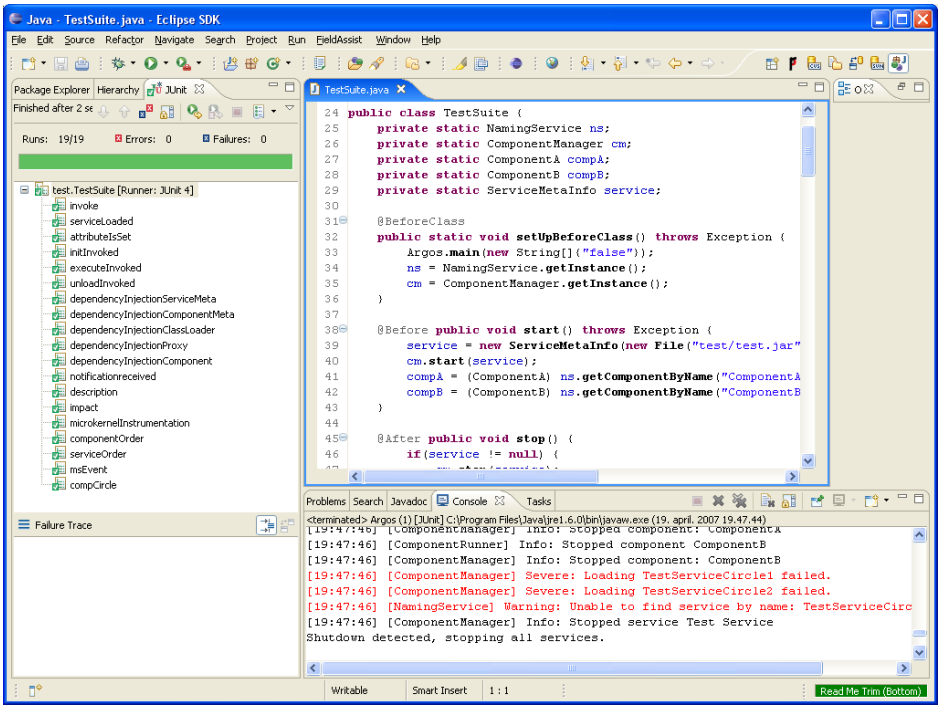

Figure 6.1: Argos unit tests

continued from previous page

Attribute Set	MK-44	Verifies that attributes are set correctly.
Init Invoked	MK-10	Verifies that attributes are set correctly.
Execute Invoked	MK-12	Verifies that methods marked with @Execute are invoked.
Unload Invoked	MK-11	Verifies that methods marked with @Execute are invoked.
Dependency Injection Component Meta	MK-8	Verifies that component meta data is correctly injected into a component.
Dependency Injection Proxy	MK-9	Verifies that notification proxy is correctly injected into a component.
Dependency Injection Component	MK-7	Verifies that a component is correctly injected into a component.
Notification Received	MK-1, MK-2, MK-4 and MK-5	Verifies that notifications can be received.
Descriptions	MK-41	Verifies that descriptions in JMX is set properly on components.
Impact	MK-40	Verifies that impact in JMX are set properly on operations.
Invoke	MK-37 and MK-39	Verifies that operations on components can be invoked through JMX.
Core Instrumentation	MK-42	Verifies that all components in the core are instrumented.

continued on next page

continued from previous page

Component Order	MK-26	Verifies that components with dependencies are loaded in the correct order.
Service Order	MK-25	Verifies that services with dependencies are loaded in the correct order.
Core Notification	MK-2 and MK-1	Verifies that the core can broadcast notifications.
Component Circle	MK-23	Verifies that dependency circles are discovered.

Table 6.1: Unit Tests

These 19 unit tests do not test every aspect of the Argos core. There are certain requirements that can not be checked with unit tests. An example is the graceful shutdown requirement (MK-50). It is impossible to run a software test to check if the container is shutdown gracefully. This is because as the JVM has to be shutdown, it effectively also shuts down the unit testing framework. The unit tests created verifies that the functionality for all requirements which have a testing plan associated with them are fulfilled, see chapter 3 for details.

6.2 Beta testing

After implementing the Argos core and some basic system services Argos was taken into use by end users. The end users were two students (Espen Ekvang and Mats Mortensen), and Argos has also been in use at Norwegian Centre for Telemedicine (NST) and Telenor Research and Innovation. They used Argos as a building block when writing their master thesis. According to [33] beta testing is best done with many beta testers as testers are less likely to be biased. Getting a sufficient amount of beta testers was not a top priority. Even though there only were two beta testers their help in identifying bugs (see figure 6.2) has resulted in Argos becoming far more stable. Many bugs that I could not have found on my own were discovered. Not all bugs that were discovered were reported in the tracker on the sourceforge site (figure 6.2). Most were fixed on the fly after verbal communication of the bugs.

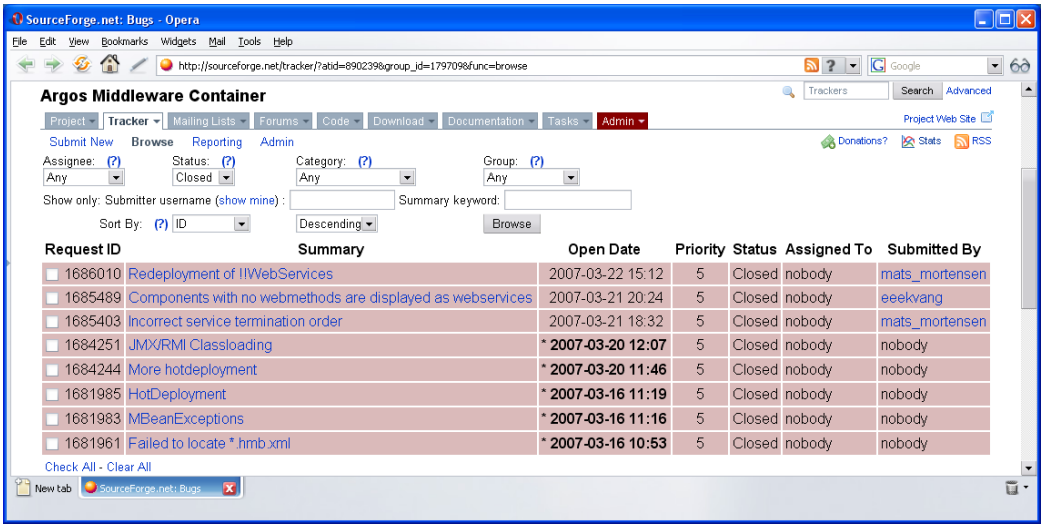

Figure 6.2: Bugs discovered in beta testing

6.3 Performance Testing

Performance testing of the deployment process, component to component interaction and component scheduling has been done. The results are presented in their own sub section. All testing was done on a PC with: 2 GB RAM, Intel Core Duo 2,40 GHz running Windows XP.

6.3.1 Deployment Performance

Deployment testing was done to find out two things:

- How many components could be loaded?
- How does the component process scale as the number of components increases?

For the testing, a service with 100 to 50 000 components was loaded and the time to load was measured. Testing was done with the default setup of Argos. The results were not as expected. The time to deploy components grew exponentially as the number of components increased, seen in the blue graph in figure 6.3. The source code for the deployment process then was reviewed. During this review the circle finding algorithm, used to find circle dependencies, was fine tuned. The algorithm which uses recursion did a lot more method calls than necessary, the redundant method calls were removed. This resulted in the performance going from exponentially to linear, as seen in the red graph in figure 6.3.

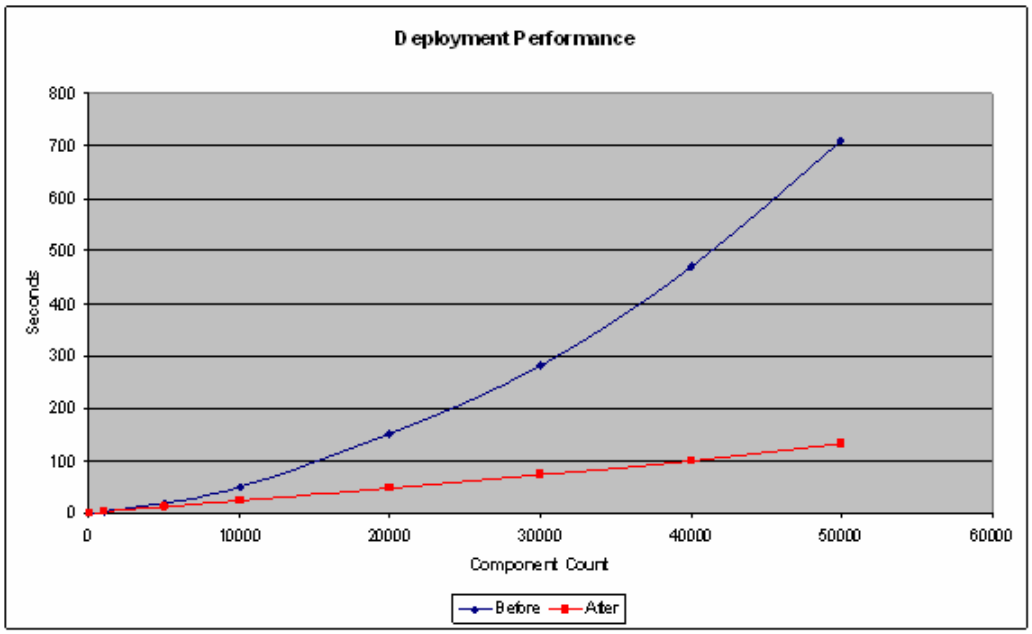

Figure 6.3: Deployment Performance

This testing also showed that the amount of components that could be loaded was related to the heap size used by the JVM. Components are java objects, these objects occupy space in the heap. When the 20 thousand mark had been passed the heap size was adjusted to use 256 MB, instead of 64 MB which is the default. This was done since the JVM could not place more components in the heap since it became full.

6.3.2 Interaction Performance

Components can interact in three different ways: notifications, JMX invocations and method invocations. Each way of interacting has its advantages and disadvantages. When using notifications the component broadcasting the notification does not need to know who receives the notification. New components receiving notifications can be added at run-time. Using JMX to do method invocations makes it possible to call methods on local and remote components. While normal method invocation is limited to local components. Normal method invocations are however a lot faster than JMX invocations.

To test the performance properties of the three ways components can interact, a service was created. This service contained two components which interacted in all three ways and measured the performance. The results are shown in figure 6.4

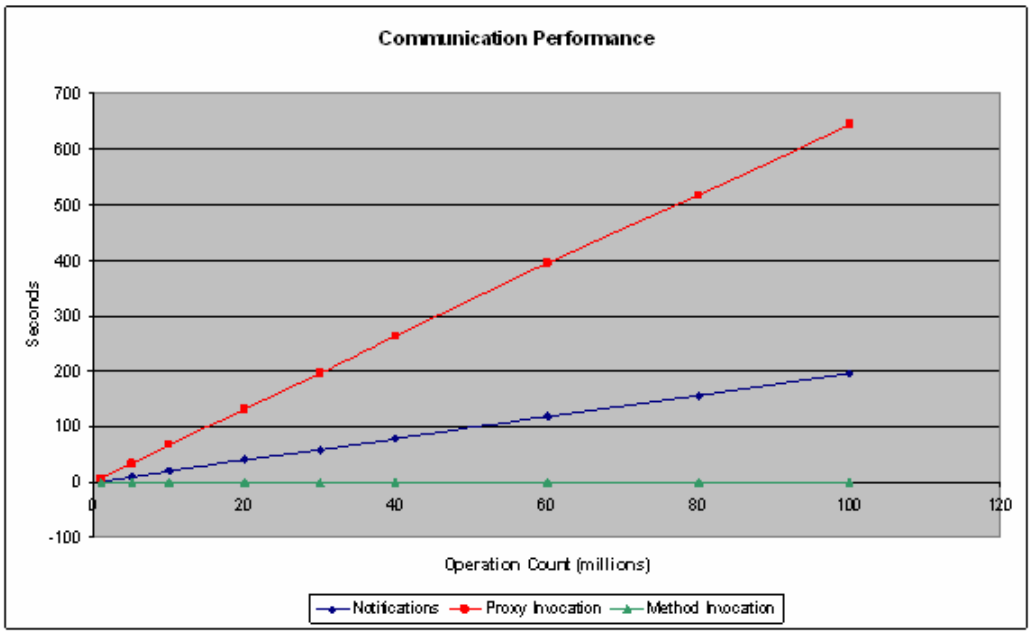

Figure 6.4: Interaction Performance

The slowest form of interaction is invocation through JMX by far, taking almost three times as long as notifications sent through JMX. Because of the big difference and that both ways of interacting used JMX it was suspected that the implementation of DynamicProxy where the method call is carried out using reflection was not as effective as it could be. This suspicion proved wrong however. A HashMap was used to do the lookup of methods that should be called. This was replaced with a binary search algorithm instead and tests were performed again. The results were almost identical. This implies that what takes time is the actual method call done with reflection. This is however not confirmed by any tests. The overhead of one invocation is on average 0.00643625 mili seconds.

Notifications had much better performance than JMX invocations. The reason for this is that reflection is not used and there is no need to identify what methods have to be called. It is just a series of method calls and the overhead of passing the notification through the MBean server. The overhead of broadcasting one notification is on average 0.00196812 mili seconds.

Normal method invocations are by far the fastest. However these invocations only work when both components are found in the same container. JMX invocations work when components are placed on different Argos containers, possibly on different machines, and also makes it possible to add interceptors (discussed in chapter 7). Notifications work when components are found on the same and different containers. Notifications also make it possible to change interaction patterns at run-time since the broadcasting component does not need to know which components receive the broadcasted notifications.

6.3.3 Component Scheduling

Components can define a method which will get invoked at set intervals. A test was created to verify how far off the invocations were compared to the defined intervals. The results are shown in figure 6.5.

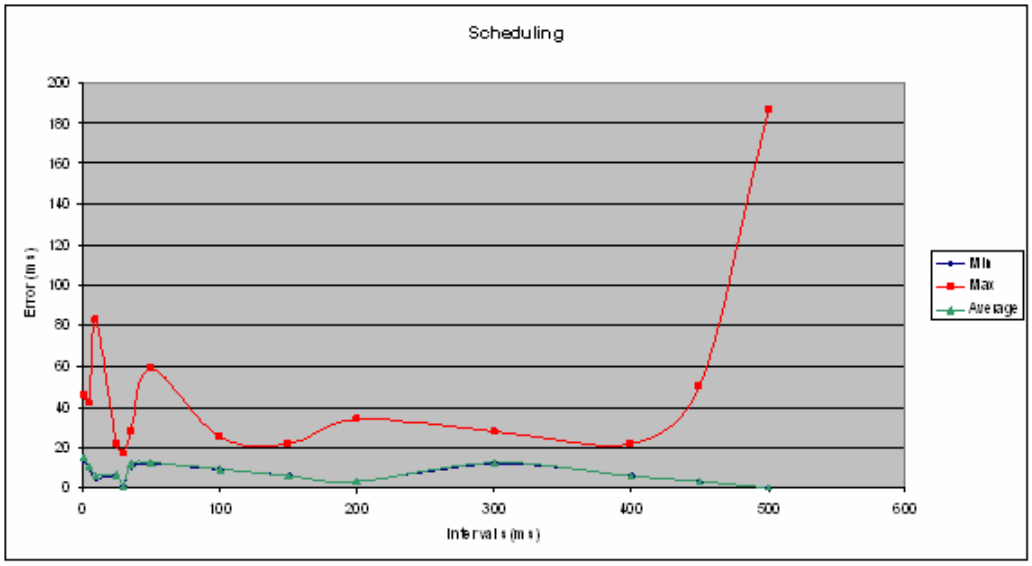

Figure 6.5: Component Scheduling

In the test 1400 components where used. All components had the same interval set. The interval was set between 500 milliseconds to 1 milliseconds (X-axis). The three graphs shows the min, average and max values in terms of how much Argos missed the interval by. For our use the results were acceptable as on average the interval was only off by a few milliseconds. There were however some rare cases where the largest miss was 181 milliseconds. This was most likely due to garbage collection done by the JVM.

6.4 Summary

The were four types of tests performed on Argos. Unit testing and performance testing was limited to the core while informal testing and beta testing was done on the core and all system services created. The beta testing was done by two students writing their master thesis. It proved useful as several bugs were discovered and corrected as a result of the beta testing. The performance testing done on the deployment process uncovered a bug in

the circle finding algorithm (dependencies) resulting in exponential scaling. The bug was corrected and near linear scaling was achieved.

Chapter 7

Evaluation

In this chapter Argos and the methodologies used when building argos will be evaluated. The evaluation of the Argos implementation will be limited to the Argos core.

7.1 Method

Scrum was one of the methodologies used during the development of Argos. We did some modifications so scrum would fit us better. The size of the scrum team was just one person. The need for communication was not as important as with a large team. Because of this daily scrum was done twice a week. The sprint length was also modified to 2 weeks instead of 30 days. The reason for this is that it was difficult to plan 30 days a head.

The sprint backlog forced us to plan ahead. Every sprint had clear goals of what had to be achieved. Having clear goals proved helpful as there always were objectives to work towards. The effects of having goals would be more visible in a team. Multiple members working towards the same goals would create synergy effects. The combined effort of multiple members working together would be larger than the individual efforts combined. The daily scrums provided visibility to the process and made sure that progress was made. Also, the burndown graph made it quite clear if progress was as expected, slower or even faster. Progress became apparent and visible. With visibility came control. If progress halted it was identified at an early stage making it possible to adapt.

As mentioned, with scrum visibility is achieved. Visibility is not only positive. Everything that is done becomes apparent. Progress has to be made and is checked on a daily basis through daily scrum. Some might find the constant demand of results stressful, resulting bad performance and even physical effects such as headaches. Stress was experienced during the development process. However it was a result of poor estimates. When sprint backlogs were created each task was given an estimate of how long it would take to complete. It became obvious early during implementation that estimates were horribly wrong. Prior to undertaking the implementation tasks it was difficult to anticipate the complexity. As more experience was gained estimates became more accurate. Towards the end of the development process the estimates became quite accurate. Having a more accurate relation between hours of work and the burndown graph resulted in increased motivation.

The use of scrum surpassed our expectations. It guided the development process in the right direction. When tasks took longer than expected due to complications it was quickly identified. This made it possible to carry out counter measures getting the development

back on track.

Extreme Programming (XP) was the second methodology used during the development of Argos. In XP the focus is on writing code rather than designing and then implementing. The problem is first solved in the easiest way possible. The source code is then refactored from the easiest solution to the best solution. Central in XP is test-driven development and pair programming. We did not follow XP by the book. Pair programming was not used as there only was one developer. Unit testing was not taken into use until the end of the development process.

The focus on implementing instead of going through design modeling and documentation was motivating. Most developers consider this to be more satisfying. This does not however exclude modeling from the process. When encountering complex implementation task modeling was used to gain an overview of the problem. Once an overview of the problem was established the implementation was written.

Unit testing was not undertaken until the end of the development process. This proved to be a big mistake. During the development there were multiple occasions where existing functionality was broken. This was a result of changing a component that was implicitly coupled to another component, the second component then broke. Often the broken functionality was not discovered until much later as no formal testing was applied to the build. The lesson learned is that unit testing is a big part of XP for a good reason.

The methodologies that were followed successfully guided the development process, ensuring that the project was finished well before the deadline.

7.2 Testing

There have been four types of tests performed during the development of Argos: informal, beta, unit and performance. The testing of Argos will be evaluated before Argos itself. The performance tests performed will be used in the evaluation of Argos.

Informal testing means that we manually tested functionality as the development process progressed. This was done instead of unit testing. Unit testing was applied after Argos 1.0 was released on Sourceforge[1]. Choosing to do informal testing instead of unit testing was a big mistake. A lot of time could have been save by applying unit testing as XP requires from the get go. Bugs could have been discovered at an earlier time. The tedious tasks of manually checking functionality could have been avoided. Due to the above reasons tests plans were written and unit testing was applied. Maintenance and extension of Argos in the future can then be done with unit testing.

Beta testing was the testing that proved most effective. The beta testers found bugs and more importantly came with helpful suggestions for improving Argos. Having this user input changed requirements during the development. Adapting to this was not a problem as the development methodologies used were agile. In figure 6.2 a set of the bugs found during beta testing can be seen. Both Norwegian Centre for Telemedicine (NST) and Telenor (Norwegian telecom company) has flagged that they will take Argos into use on several

[1] http://www.sourceforge.net/projects/jargos

research projects. Hopefully this will result in more feedback that can help improve Argos further.

Performance testing was done on two fundamental parts of Argos: deployment and communication. Performance testing was done to find the limits of Argos. The requirements of Argos did not demand that it scales like enterprise solutions have to. Still, finding the limits of Argos could prove, and did prove, valuable.

The performance of deployment showed that the amount of components Argos is able to handle is related to the amount of memory available. At the most 50 000 components were running simultaneously with no hick ups. After doing some optimisations to algorithms used in the deployment process near linear scaling of the time to deploy components was achieved. On average a component took 2.47 mili seconds to load which is quite good. The deployment process could be optimised, however the current results are satisfying as long as scaling is linear.

Components in Argos are able to interoperate by: notifications, JMX invocations and method calls. The performance of all three was measured. Normal method calls was fasted by far, notifications came in second and JMX invocations was the slowest. Normal method calls are really fast as there is almost no overhead in the operation. It is just a matter of changing the program counter to the location in memory where the code for the method is located. Notifications are slower than method calls since notifications have to pass through the MBean server, ComponentRunner and is then passed to the component using reflection. There are three jumps from component to component and an invocation using reflection. Each jump take time, but most of all reflection takes a lot of time. The communication pattern when doing a JMX call is the same as when notifications are used, three jumps in total. However, DynamicProxy gets the name of the method that should be invoked as a String. DynamicProxy then has to find the correct method using reflection and invoke it. Finding the correct method takes a lot of time resulting in JMX invocations being the slowest of the three. Each of these three communication forms have other properties associated with them. They will be discussed in the next section.

Argos has gone through rigorous testing proving the quality of Argos. The quality is simply that of a production system. Performance testing has also shown that Argos can be used in near real-time applications.

7.3 Argos core

In this section the Argos core and the functionality it gives developers will be evaluated.

JMX is used as the foundation of Argos. All components loaded in Argos are also loaded as MBeans. MBeans can not be created from any java object in JMX, certain interfaces have to be implemented. We did however not want application programmers of Argos to implement these interfaces as it forces unnecessary work on the developer. A Dynamic MBean class was created, it uses reflection to implement the proxy pattern between the MBean server and POJO components. It then becomes possible to introspect and manage any component. The negative with this compared to application programmers creating the MBeans themselves is that method invocations become slower. However the performance testing shows that it is just a small overhead.

7.3.1 JMX Notifications

The Argos core broadcasts notifications to system services about events happening in the core. The system services are then able to act on these notifications expanding the functionality of the core. Notifications have the property that the broadcaster does not need to know who receives the notifications. Since the Argos core does not know what system services it will be used with, this is a good match. This makes it possible to develop system services later without making changes to the core. As an effect of using notifications JMX has to be used. As shown in the previous section it has some performance impacts.

Using the proxy pattern on components has other implications. It makes it possible to intercept method calls made to components. It could then be possible to add interceptors before and after methods are invoked. These interceptors could even change the arguments methods are called with and their return values. Examples of functionality interceptors could add include:

- **Security:** Before carrying out the method call checking authorisation is done. If the authorisation has not been issued an exception may be raised.
- **Logging:** Logging which methods are called with what arguments.
- **Caching:** Statistics of what is requested can be built and from this statistics caches can be created.
- **Redundancy:** Redundancy in e.g. a persistence layer can be introduced by creating an interceptor that stores the given data in a separate location.
- **Quality of Service (QoS):** Components can be overridden completely and exchanged with other implementations based on QoS requirements.

This is closely related to Aspect Oriented Programming (AOP) [34], interceptors are often used as a mechanism in AOP. AOP is supported by some JEE contains e.g. JBoss[2]. AOP not currently support in Argos.

7.3.2 Hot Deployment

The Argos core support hot deployment, adding/removing/updating system and user services at run-time. Hot deployment has two major advantages. It becomes faster to develop services using the container since every time a service gets added functionality the whole container does not have to be restarted. Only the service that has been updated needs to be reloaded. If there are many services running in the container the start up time can become time consuming and annoying for the developer. The other advantage of hot deployment is that services that are not affected by the service that should be updated can keep running uninterrupted. Increasing the availability of the container and the loaded services. However hot deployment has some disadvantages. Hot deployment is hard to implement. During the development of Argos most of the time was used to fix class loading issues which related to hot deployment. It is extremely hard to get right. Also when using libraries that does not support hot deployment problems are encountered. The hibernate system service is one. Hibernate does not support hot deployment. This effectively makes any service that uses hibernate none hot deployable. There are ways to get around this using class loading. It

[2] http://www.jboss.org

requires changes in the service model of Argos. To get around hibernate not supporting hot deployment it is possible to create one instance of hibernate for every service that uses hibernate. Thus, when a service is reloaded it gets a new fresh instance of hibernate. Persistence object can then have been redefined as hibernate has never seen them before and will not complain about the class having changed.

When comparing Argos to CORBA and JEE only Argos and JEE containers support hot deployment. Hot deployment in JEE is however limited to user services. It is not possible to hot deploy system functionality which extends the system. The possibility to add system services makes it possible to create Argos configurations that fit specific needs. These needs can change over time and Argos can then with hot deployment continue to fulfill these needs without being restarted and reconfigured. Argos is compared to JEE containers much smaller and easier to customise.

7.4 Usage Experience

One problem with middleware containers is debugging. When a container is needed to actually run the code it makes debugging a lot harder as the code runs in the container. Both Argos and JEE containers suffer from this. One thing that makes it easier to debug and test is the use of POJO's. To some degree it is then possible to test and debug components without starting the container. Enabling remote instrumentation is a lot easier in Argos than in JEE containers, e.g. JBoss. The remote instrumentation helps with debugging as it gives insight into components as they are running on Argos. This is something the beta testers really liked.

From a developers view Argos will seem easier to understand than JEE containers as it is a smaller system. JEE containers are target for enterprises while Argos is targeting services which do not need the advanced support that JEE containers provide. Argos focuses on rapid development, system services allows for creating and reusing services with commonly needed functionality. An open and none restrictive component model allows for easy access to local and remote resources. While services developed on JEE can not do this.

7.4.1 Argos Compared to Related Work

When comparing services written for Argos to creating the application in standard java Argos has some advantages and some disadvantages. The disadvantage is that in a normal java application the developer has full control. This however means that the developer needs to write everything him/her self also. Which could be a waste of time. When developing on Argos the developer has the option of using system services. System services give the developer functionality at no cost. This can drastically increase the development time. Also common things as logging, configuration and management are already enabled for the developer, at no time cost.

The design of the component model in Argos reflects that services to be provided to a small number of simultaneous users. Argos allows services to access the file system, open incoming and outgoing sockets connections and to create threads. Enterprise does not allow this since it is extremely hard to handle scaling in a sensible manner (components can not be put in a waiting pool). Scalability is addressed in JEE containers, but not in Argos. Argos allows the developers more freedom when creating services.

Argos provides a smaller system and a non-restrictive component model compared to JEE. Argos becomes easier to learn and more rapid to develop on.

7.5 Summary

Scrum and XP were used during the development of Argos. The use of these methodologies proved highly motivating and gave great visibility as progress was made or stagnated. Unit testing was not used from the start even though it is a big part of XP. Not using unit testing was a big mistake as a lot of time was spent fixing bugs that could have been easier identified with unit testing. Beta testing has been done on Argos and the system services developed. This beta testing proved highly useful. A lot of bugs were discovered that we could not have discovered on our own. Also helpful suggestions on how to improve Argos was made by the beta testers. The result was a more stable and user friendly system. Argos with its system services and hot deployment makes it possible to tailor Argos configurations to satisfy specific needs. When these needs change it is possible to adapt Argos at run-time to satisfy the new requirements. Argos does not focus on scalability as enterprise containers. Argos gives developers more freedom in their development than enterprise solutions do.

Chapter 8

Conclusion

This chapter summarises this thesis by first concluding if the problem definition has been fulfilled. Then, future work is presented.

8.1 Achievements

The problem definition from section 1.2 states:

> *The goal of this thesis is to build a middleware platform that is easy to extend with new functionality in addition to its intrinsic functionality. It should as a start include deployable functionality for web applications, WS, database persistence, service management and service distribution. This goal can be split in two intermediate aims:*
>
> - *Build a Core that enables the addition and extension of system functionality at run-time.*
> - *Build system services (deployable functionality) to extend the functionality of the Argos core proving that it can be extended to include new functionality.*

We have developed middleware platform which can easily be extended. The Argos core is the minimum configuration of Argos, it is a small base including: service and component model, deployment, component to component communication and logging. The core is extend with system services, system services developed include web applications, database persistence, service management, service distribution and web services. These system services can be added, removed or updated at run-time. User created services take advantage of the functionality system services add to Argos.

All requirements in chapter 3 have been fulfilled. At the current time of writing there exist one limitation in the implementation. The limitation is concerning hot deployment in hibernate. It can be solved, but requires an addition to the service model (will be mentioned in future work). The quality of the software written is excellent and stable enough to be used in production systems. Also the amount of work that has gone into Argos is well over what is expected in a master thesis. The use of sloc [35] has shown that all source code in Argos, including testing, constitute 16.36 months of work.

Argos has been used by two students in their master thesis, Norwegian Centre for Telemedicine (NST) and Telenor Research and Innovation.

The Argos core and system services provide tailored, flexible and extensible middleware support. The result is a platform capable of tackling domain specific challenges. It provides rapid development of feature rich applications for managing and processing information from various sources. Information sources can be sensors, local file system, web services, sockets, databases and so on. Information can be collected from these sources, processed and made available using web services, web applications or JMX.

Argos and all system services are released under the Berkeley Software Distribution (BSD) licence and can be acquired from sourceforge[1].

8.2 Future Work

In this section issues that should be explored to improve Argos is presented:

- **Interceptors:** Between DynamicProxy and component it is possible to create an interceptor layer. In this layer interceptors can be added and invoked before and/or after methods on components are invoked. This allows for separation of concerns where issues such as security, logging, caching, redundancy and much more can be explored.

- **Service standards:** Currently services are implementation only. It is possible to create services that confirm to a standard, e.g. a web application standard. This standard would define the service. There could be multiple implementations of a standard, e.g. Tomcat or Jetty for the web application standard. This would allow user services to depend on standards and not actually implementations of services. The service and standards would compare to objects and interfaces in java.

- **Security:** In the current implementation security is not considered. How Argos can become more secure should be explored.

- **Administration GUI Prototype:** The administration GUI system service was developed as a prototype and is thus not useful for anything but demo purposes. Developing a new system service from the lessons learned when developing the prototype should be undertaken.

- **EJB3 Persistence:** The current data persistence support uses hibernate. The developer then has to maintain a mapping file and the class that should be mapped. Hibernate has an additional library called hibernate annotations. When using this library only the class file has to be maintained. This would make it easier for developers to create services using databases, Hibernate annotations support the EJB 3 persistence specification [17].

- **Service Model:** Currently there are some limitations in hot deployment when using libraries that do not support hot deployment. The service model can be changed to load a system service once every time a user service depends on it. This would solve

[1] http://sourceforge.net/projects/jargos

the hot deployment issue since every time the user service is hot deployed it gets a new version of the system service it depends on.

- **System Services:** To make it easier for developers on Argos to create services new system services could be created. Examples of functionality such system services could provide is: AOP, transactions, HTTP management and resource pooling.
- **Cooperating Argos Containers:** The topic of cooperating Argos containers has not been explored much. Argos containers are able to communicate quite easy, how can this be exploited to create new applications?
- **Service Distribution Prototype:** The service distribution system service was developed as a prototype and is thus not useful for anything but demo purposes. Developing a new system service from the lessons learned when developing the prototype should be undertaken.

8.3 Contribution

Argos is able to provide tailored, flexible and extensible middleware support using reflection, dependency injection, JMX notifications and hot deployment. The result is a platform capable of tackling domain specific challenges. It provides rapid development of feature rich applications for managing and processing information. As a result of this thesis one article [3] has been published. There is also a second article [36] written and submitted, however at the current time it is not known if the article has been accepted.

Bibliography

[1] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.

[2] Anders Andersen, Gordon S. Blair, Vera Goebel, Randi Karlsen, Tage Stabell-Kulø, and Weihai Yu. Arctic Beans: Flexible and open enterprise component architectures. In *NIK 2001*, Tromsø, Norway, November 2001.

[3] Arne Munch-Ellingsen, Bjørn Thorstensen, Dan Peder Eriksen, and Anders Andersen. Building pervasive services using flock sensor network and flock container middleware. In *The IEEE 21st International Conference on Advanced Information Networking and Applications (AINA-07), PCAC07 Workshop*, Niagara Falls, Canada, 2007. IEEE CS Press.

[4] EJB 3.0 Expert Group, Linda DeMichiel, and Michael Keith. Jsr 220: Enterprise javabeans^{TM}, version 3.0 ejb core contracts and requirements, 2006. http://jcp.org/aboutJava/communityprocess/final/jsr220.

[5] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2002.

[6] Henrik Kniberg. Scrum and xp from the trenches, 2006. http://www.crisp.se/henrik.kniberg/ScrumAndXpFromTheTrenches.pdf.

[7] Kent Beck and Cynthia Andres. *Extreme Programming Explained Embrance Change*. Addison-Wesley, second edition edition, 2005.

[8] Kent Beck. *Test-Driven Development By Example*. Addison-Wesley, 2003.

[9] Ken Lunn. *Software Development with UML*. Palgrave Macmillan, 2003.

[10] Object Managment Group. Corba component model specification, 2006. http://www.omg.org/cgi-bin/doc?formal/06-04-01.

[11] Object Managment Group. Object management architecture guide, 1995. http://www.omg.org/docs/ab/97-05-05.pdf.

[12] Object Managment Group. Omg idl syntax and semantics, 1997. http://www.omg.org/docs/formal/02-06-39.pdf.

[13] Nanbor Wang, Douglas C. Schmidt, and Carlos ORyan. Overview of the corba component model, 2000.

[14] Sun Microsystems Inc. Jcp 215: Process document, 2004. http://jcp.org/aboutJava/communityprocess/final/jsr215.

[15] Danny Coward and Yutaka Yoshida. Java^{TM}servlet specification version 2.4, 2003. http://jcp.org/aboutJava/communityprocess/final/jsr154.

[16] Pierre Delisle, Jan Luehe, and Mark Roth. Javaserver pages^{TM}specification version 2.1, 2004. http://jcp.org/aboutJava/communityprocess/final/jsr245.

[17] EJB 3.0 Expert Group, L. DeMichiel, and M. Keith. Jsr 220: Enterprise javabeans^{TM}, version 3.0 java persistence api, 2006. http://jcp.org/aboutJava/communityprocess/final/jsr220.

[18] Rima Patel Sriganesh, Gerald Brose, and Micah Silverman. *Mastering Enterpise JavaBeans^{TM}3.0*. Wiley Publishing Inc., 2006.

[19] Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, and Rick Evans. Spring java/j2ee application framework, 2006. http://static.springframework.org/spring/docs/2.0.x/spring-reference.pdf.

[20] Jim Knutson and Heather Kreger. Jsr 109: Web services for j2ee, version 1.0, 2002. http://jcp.org/aboutJava/communityprocess/final/jsr109.

[21] Sun Microsystems Inc. The j2ee^{TM}1.4 tutorial. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/EJBConcepts3.html.

[22] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau. Extensible markup language (xml) 1.0, 2006. http://www.w3.org/TR/2006/REC-xml-20060816/.

[23] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. Soap version 1.2 part 1: Messaging framework, 2003. http://www.w3.org/TR/soap/.

[24] Dave Winer. Xml-rpc specification, 2003. http://www.xmlrpc.com/spec.

[25] Sun Microsystems Inc. The remote method invocation tutorial. http://java.sun.com/docs/books/tutorial/rmi.

[26] Michael Kifer, Arthur Bernstein, and Philip M. Lewis. *Database Systems: An Application Oriented Approach*. Addison Wesley, 2nd edition edition, 2005.

[27] Sun Microsystems Inc. Java^{TM}management extensions (jmx^{TM}) specification, version 1.4. http://jcp.org/aboutJava/communityprocess/mrel/jsr003/index3.html.

[28] Sun Microsystems Inc. Web services connector for java^{TM}management extensions (jmx^{TM}) agents. http://www.jcp.org/en/jsr/detail?id=262.

[29] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1, 2002. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[30] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1, 2001. http://www.w3.org/TR/wsdl.

[31] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[32] Steven John Metsker and William C. Wake. *Design Patterns in Java*. Addison-Wesley Professional, 2006.

[33] Robert Krull. Is beta better? *Proceedings of IEEE professional communication society international professional communication conference and Proceedings of the 18th annual ACM international conference on Computer documentation: technology and teamwork*, 2000.

[34] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[35] http://www.dwheeler.com/sloc/.

[36] Arne Munch-Ellingsen, Anders Andersen, and Dan Peder Eriksen. Argos, an extensible personal application server. 2007.

[37] Sun Microsystems Inc. Java^{TM}logging overview, 2001. http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html.

[38] Hibernate reference documentation, 2006. http://www.hibernate.org/5.html.

[39] Sun Microsystems Inc. The j2ee 1.4 tutorial, 2005. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/.

Appendix A

Programmer's Manual

In this programmer's manual the reader will learn how to download, install and build Argos and develop applications on it.

A.1 Intended Audience

The intended audience for this programmer's manual is advanced java developers. It is recommended that the developer has a basic knowledge of java, core java packages (e.g. IO and networking), JMX, hibernate, JSP, servlets, SOAP and XML-RPC.

A.2 Prerequisite

The following prerequisites are needed:

- JSE 6 or newer.
- Ant 1.6 or newer for building the source.
- Eclipse 3.2 or higher is not needed by Argos but it is used in this programmer's manual.

A.3 Getting Argos up and running

In this section you will learn how you can get, build and run Argos.

A.3.1 Obtaining Argos

Argos is an open source project released under the BSD license[1]. There are several hosting sites on the internet offering free hosting for open source projects. Argos uses Sourceforge.net since it offers: a free web site, version control, forums, file release control, bug tracking and feature requests. On the web site[2] you also can download Argos. If you find bugs, want new features or just want to get involved in the project go to the site and contribute.

There are two versions available from the sourceforge site: source and binary distributions.

[1] http://www.opensource.org/licenses/bsd-license.php
[2] http://sourceforge.net/projects/jargos

Names should be something like Argos_1_0_src.zip and Argos_1_0_bin.zip where the first is the source version and the later is the binary version. The difference between the two is that the source version has to be built before you can use it while the binary version can be started out of the box.

A.3.2 Building

If you have downloaded the source version of Argos you need to build it before you can actually run it. To build Argos ant 1.6 or newer is required. When you unpack the zip file a new directory name ArgosSource is created. This directory contains the following directories:

- Argos
- BangBang Hibernate
- BangBang Jetty
- BangBang JMXConnector
- BangBang Webservices

The Argos directory is the core while the other directories are various system services. A system service is a service that extends the functionality in Argos, e.g. the web service system service which adds support for web services (SOAP and XML-RPC).

Assuming you have ant installed and correctly configured you do the following to build Argos:

1. Open a console and go into the ArgosSource/Argos directory.
2. Type "ant".

The output should look like in figure A.1. If you have successfully built Argos you can move on to the next section which explains how to start Argos.

A.3.3 Starting

If you have downloaded the binary version (and unpacked it) or successfully been able to build the source version of Argos you are now ready to start Argos. But firsts lets have a look at the directories and files in the Argos directory that matters to you:

- **deploy**: The deploy directory contains all services that should be loaded. When you create a service jar file this is where you place your file to make Argos load it.
- **derby**: By default Argos comes with Apache Derby 10.1. All data put inside derby is stored in the derby directory.
- **logs**: Argos uses the logging framework which was introduced in Java 2 Standard Edition (J2SE) 1.4. All logging information is stored in this directory.
- **temp**: Argos needs to store various temp files while running. These files are stored in the temp directory.

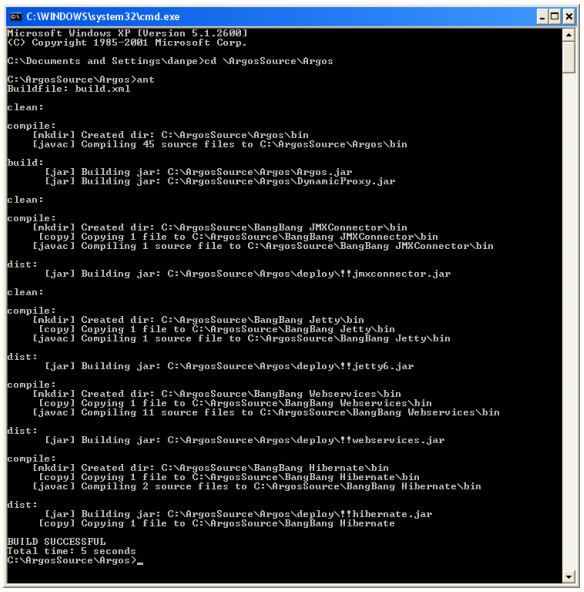

Figure A.1: Building Argos

- **Argos.jar**: The core jar file.
- **config.properties**: This is the config file of Argos and its system services.

Starting Argos is done by:

1. Open a console and go into the Argos directory.
2. Type ”java -jar Argos.jar”.

Doing this should result in what is seen in figure A.2.

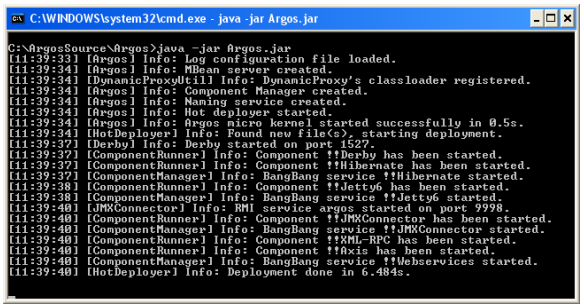

Figure A.2: Starting Argos

Congratulations, you have now successfully downloaded, installed and started Argos! Now let's look at how you can configure Argos.

A.3.4 Configuring

The file config.properties found in the root of the Argos directory. It is the configuration file of Argos. In the config file you are able to configure the core and the system services that are deployed. For the changes in the file to take effect Argos has to be restarted.

The start of the configuration file you should not touch, only the last section called "Component Configuration". The component configuration part looks by default like this:

Argos Configuration File:

```
############################################################
#   Component Configuration
############################################################
BangBang.Hibernate.updatePolicy = create-drop
BangBang.JMXConnector.port = 9998
BangBang.JMXConnector.serviceName = argos
BangBang.Jetty6.httpPort = 8080
BangBang.Derby.username = su
BangBang.Derby.password = pass
```

In this part you can change various ports, user name, password and hibernates update policy (covered in section A.6. The config parameters are really self explanatory and needs no more explanation.

A.4 Services

In Argos there are two types of services: *user services* and *system services*. Both types share a lot of similarities. A service consists of one or more components and a deployment descriptor. Components are Plain Old Java Object (POJO)'s and the deployment descriptor is a XML file (deploy.xml). The components (class files) and the deployment descriptor is packed into a jar file. This jar file is called the service file. The components follow its package structure, hence if a components package is org.myservice the class file of the component is found in the directory /org/myservice inside the service file. The deployment file, deploy.xml, is always placed in the root of the service file.

In section A.4.6 you will be given an ant build file which fixes all this for you automatically.

A.4.1 System Services

What separates a system service from a user service? There are two differences. System services provide other services with some functionality. Such as the jetty service which enables other service to create web applications using servlets and JSP. Secondly system services are able to react to notifications broadcasted by the core, user services do not have this option.

What actually separates a system service from a user service when implementing is that the service file starts with "!!", e.g. "!!Webservices.jar". This tells the core that this is a system service. Which makes the core load the service before user services and lets it receive notifications broadcasted by the core.

A.4.2 User Services

User services can take advantage of the extra functionality that system services add. Thus a user service can consist of more than mere components and deployment descriptors. With the default setup of Argos user services can in addition include web applications, mapping specifications (hibernate), JME applications, Argus panels and external applications and resources.

A.4.3 Components

Components in Argos are Plain Old Java Object (POJO), a simple java class. To add a bit of "life" to your components you need to use annotations. The Init annotation can be used on one method per component. Methods marked with this annotation will be invoked when the component is instantiated. Hence useful for setting up resources at the start of a components life cycle. The Execute annotation can also be used on one method per component. It tells the core that this method should be invoked repeatedly.

There are a lot more annotations and this will be covered in section A.5 with examples. The two most important ones being @Execute and @NotificationHandler. Both these annotations marks execution entry points in the component.

A.4.4 Deployment Descriptor

The deployment descriptor has to be name "deploy.xml" and be placed in the root of the service file. It gives the core information about:

- Service name and version, the service name has to be unique.
- Service dependencies, meaning which services has to be loaded for this service to work properly.
- Which components should be loaded, component name has to be unique.
- Component dependencies.
- What components should receive notifications from other components.
- Attribute configuration.

A common mistake done in the deployment descriptor is to not include dependencies on system services. If your service is suppose to have a web service it has to depend on !!Webservice. If the dependency is not listed the core will not setup the class loading properly and running the service will result in an error.

Lets look at an example deployment descriptor:

Example: Service Deployment Descriptor

```
<?xml version = "1.0" encoding="UTF-8" standalone = "yes"?>

<service name="!!Webservices" version="1.0">
    <depend on="!!Jetty6"/>
    <deploy>
        <component name="!!XML-RPC" class="argos.bangbang.xmlrpc.XmlRpc">
            <listen to="!!ComponentManager" />
        </component>
        <component name="!!Axis" class="argos.bangbang.axis.Axis">
            <listen to="!!ComponentManager" />
        </component>
    </deploy>
</service>
```

This is the deployment descriptor of the web service system service. The service and component names start with !!, meaning it is a system service with system components. When creating user services the "!!" is omitted.

The service tag includes a name and version attribute which both are self explanatory. On the next line you find the depend tag. This tells the core that this service has to be loaded after the service !!Jetty6 has been loaded and that !!Webservices parent class loader should be the class loader of !!Jetty6.

Inside the deploy tag you find the component tags. The component tags specifies components that should be loaded. They include name and the class name. In this deployment file you can see that both components listen to another component called !!ComponentManager. That is actually a part of the core. Any other component name loaded is possible to listen to as long as this service depends on the service where the other component is located.

You can also set the value of attributes for scalar types (String, boolean, int, double +++) in the deployment descriptor. However the above example does not do so. Lets look at the deployment descriptor of !!JMXConnector instead.

Example: Service Deployment Descriptor

```
<?xml version = "1.0" encoding="UTF-8" standalone = "yes"?>

<service name="!!JMXConnector" version="1.0">
    <deploy>
        <component name="!!JMXConnector"
            class="argos.bangbang.jmxconnector.JMXConnector">
            <attribute name="port" value="9998"/>
            <attribute name="serviceName" value="argos"/>
        </component>
    </deploy>
</service>
```

Here you can see that the attribute tag is used inside the component tag. The attribute tag includes a name and a value which lets the core know that the attribute called port inside the component should be set to 9998. For this to work the component needs to define the methods setPort(int) and setServiceName(String), if not an exception will occur.

A.4.5 Logging

This sub section is not intended to give you a deep understanding of the logging framework used in Argos. It is only intended to give you the minimum amount of information to successfully use it in your services. For a thorough look on the framework information can by found in [37].

When creating components it is encouraged to use the logging framework instead of System.out.println(). The framework is already setup for you so there is little work to be done to use the framework. To use the framework you include the following line in each of the java files where you want to output or log something:

Example: Creating a Logger

```
public class Test {
    private static final Logger logger = Logger.getLogger(Test.class.getName());
}
```

It is important that you change "Test.class.getName()" to reflect the class where the logger is created. If not it may look like the logging is coming from a different class than it actually is. Some examples on how logging is done are included in the example below:

Example: Using a Logger

```
logger.info("Hello, world!");
logger.warning("Some warning message here.");
logger.severe("An error just occurred!");

logger.log(Level.SEVERE, "An exception occurred!", exception);
```

Log messages of type info, severe and warning will be printed to the console and to the log file, while fine, finer and finest is only put in the log file.

A.4.6 Simple Example Service

In this section you will be presented a small and simple example of a service in a tutorial based manner. For this tutorial Eclipse 3.2 is required. The service will contain one component called "Time". The time component will every 10 seconds connect to a web service (time.soapware.org), get the current time and print it to the console. Simple enough, lets get cracking.

In eclipse create a new Java project called "My Simple Service". Once that is done right click your project, select new ➔ source folder. Name it "src". Your project will need to use classes in some other libraries, Argos and Axis (for doing the web service call). To setup the classpath correctly with these libraries right click your project and select build path. Here you need to add the jar files "Argos.jar", axis.jar and jaxrpc.jar. Once that is done we can start on the actual time component.

First let's create the component. Right click on your project and select new ➔ java class. Type in name "Time" and package "time" then press create. In our component we will use the @Execute(10) annotation to mark the fetchTime() method. This will tell Argos to invoke that method every 10 seconds. Inside the fetchTime() method we will use the axis SOAP client to do the web service call. The source code is given below:

Example: Time Component

```
package time;

//imports omitted

public class Time {
    private static final Logger logger = Logger.getLogger(Time.class.getName());

    public static final String END_POINT = "http://time.soapware.org/currentTime";
    public static final String METHOD = "getCurrentTime";

    @Execute(10)
```

```
    public void fetchTime() {
        Service service = new Service();
        try {
            Call call = (Call) service.createCall();
            call.setTargetEndpointAddress(new URL(END_POINT));
            call.setOperationName(METHOD);
            call.setReturnQName(XMLType.XSD_DATETIME);

            GregorianCalendar result = (GregorianCalendar)
                call.invoke(new Object[]{});
            logger.info(METHOD + "() returned: " +
                new Date(result.getTimeInMillis()));
        }
        catch(Exception e) {
            logger.log(Level.SEVERE, "Unable to call web service.", e);
        }
    }
}
```

In the fetchTime() method the web service call is made towards time.soapware.org. If an error occurs while doing the call the error is logged including the exception that is thrown. If the web service call is successful the result is printed.

The next thing we need to do is to create the deployment descriptor for our service. Right click on the source folder and select new ➔ file. Type in "deploy.xml". The deployment file should tell Argos that:

- Its name is "My Simple Service" and is versioned at 1.0.
- It depends on the web service system service.
- It is composed of one component.
- The component is called Time and is of class type "time.Time".

The list above results in the following deployment descriptor:

deploy.xml:

Example: Service Deployment Descriptor

```
<?xml version = "1.0" encoding="UTF-8" standalone = "yes"?>

<service name="My Simple Service" version="1.0">
    <depend on="!!Webservices"/>
    <deploy>
        <component name="Time" class="time.Time"/>
    </deploy>
</service>
```

That is it, our service is finally ready to be deployed to Argos! To build and deploy the service we will use an ant script. In the script you have to modify three variables before you are able to run it. Those three variables are "argos.dir", "lib.axis" and "lib.jaxrpc". "argos.dir" is the directory where Argos is located. "lib.axis" and "lib.jaxrpc" should point to where the axis.jar and jaxrpc.jar files can be found. Those two files are found in the BangBang Webservice lib directory. That means, you need to download the source version of Argos.

Let's create the build file. Right click on your project and select new ➔ file. Type in "build.xml" and press finish. Open the build.xml file and paste the following source into it. Remember to edit the three variables.

Example: Ant Build File

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="My Simple Service" default="build">
    <property name="argos.dir" value="../Argos"/>
    <property name="lib.axis" value="../BangBang Webservices/lib/axis.jar"/>
    <property name="lib.jaxrpc" value="../BangBang Webservices/lib/jaxrpc.jar"/>

    <property name="bin.dir" value="bin"/>
    <property name="src.dir" value="src"/>
    <property name="dist.file" value="\${argos.dir}/deploy/!!myservice.jar"/>

    <target name="build">
        <delete file="\${dist.file}"/>
        <delete dir="\${bin.dir}"/>
        <mkdir dir="\${bin.dir}"/>
        <copy file="\${src.dir}/deploy.xml" todir="\${bin.dir}"/>

        <javac srcdir="\${src.dir}" destdir="\${bin.dir}">
            <classpath>
                <pathelement location="\${argos.dir}/Argos.jar"/>
                <pathelement location="\${lib.axis}"/>
                <pathelement location="\${lib.jaxrpc}"/>
            </classpath>
        </javac>

        <jar destfile="\${dist.file}">
            <zipfileset dir="\${bin.dir}/"/>
        </jar>
    </target>
</project>
```

Now that the build script is created its time to run it. Right click on the file (build.xml) and select run as ➔ ant build. This will run the script which deploys your service to the deploy folder of Argos. You can now start Argos and see your service at work. The output should be as in figure A.3. Every ten seconds printing the current time.

A.5 Annotations

Annotations are a way of putting meta data into your java objects and was introduced in JSE 5. Annotations are used in components to tell Argos how the component should be handled. All annotations in Argos can be divided into five groups. Each group is explained in their own sub sections.

A.5.1 Dependency Injection

Dependency injection is a programming design pattern and architectural model. The pattern seeks to establish a level of abstraction. The architecture links the components rather than the components linking themselves. In Argos there exist four dependency injection annotations.

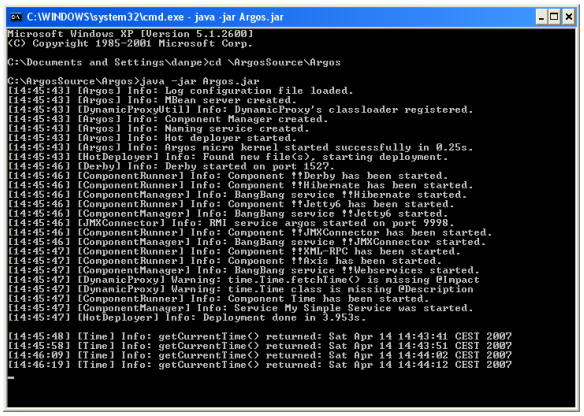

Figure A.3: Simple Service Output

In the examples below all object variables are public. They have to public so Argos can set them using reflection.

@Component is used to insert one component into another component. This makes it possible for the second component to directly call methods in the first component. The following code example injects the component "Hello" into component Test.

Example: Component Dependency Injection

```
public class Test {
    @Component("Hello") public Hello hello;

    @Execute(10)
    public void work() {
        hello.sayHello();
    }
}
```

@NotificationSender is used to insert the NotficationProxy for a component into it the component. This enables the component to broadcast notifications to other components.

Example: NotificationProxy Injection

```
public class Test {
    @NotificationSender public NotificationProxy proxy;

    @Execute(10)
    public void work() {
        proxy.send(new Notification());
    }
}
```

@ComponentMeta can be used to inject the meta data of a component into itself. This is mostly useful when creating system services as user services rarely needs to use the meta data that Argos keeps on all components.

Example: ComponentMetaInfo Injection

```
public class Test {
```

```
    @ComponentMeta public ComponentMetainfo metadata;
}
```

@ServiceMeta does the same thing as @ComponentMeta except that it injects the meta data of the whole service into the component.

Example: ServiceMetaInfo Injection

```
public class Test {
    @ServiceMeta public ServiceMetainfo metadata;
}
```

The last annotation for doing dependency injection that Argos makes available is @Service-Classloader. The annotation will inject the class loader of the service into a component. This annotation as the two previous annotations two is mostly useful when creating system services. The annotation can be used in this way:

Example: Service Class Loader Injection

```
public class Test {
    @ServiceClassloader public ClassLoader loader;
}
```

A.5.2 Life Cycle

Life cycle support is needed to give components scheduling time and to make components aware of when they are loaded and when they will be unloaded. This is in Argos done by using three annotations, @Init, @Execute and @Unload.

The @Init annotation must be placed on methods with no arguments. Any method marked with this annotation will be invoked by the container right after the component has been created. @Init is useful for setting up resources before a component springs to life. Example given below:

Example: Init

```
public class Test {
    private ServerSocket socket;

    @Init
    public void initialize() {
        socket = new ServerSocket(80);
    }
}
```

@Execute is an annotation used for repeatedly invoking a method. The marked method can not have any variables. @Execute is useful when some operation has to be run repeatedly, such as checking for new data on a stream. @Execute takes one argument, a double which tells Argos how often (in seconds) the method should be invoked.

Example: Execute

```
public class Test {
    private InputStream in;
```

```
    //Creating stream omitted

    @Execute(10)
    public void read() {
        byte[] buffer = new byte[1024];
        in.read(buffer);
        //Do something with buffer data
    }
}
```

A.5.3 Notification related

@NotificationInfo is used for exposing notifications this component may broadcast with JMX. In the JMX specification it says that components are not required to document any and every notification type they may broadcast. Thus this can be completely omitted. However it simplifies debugging when you can in a JMX browser connect to a component and see what notifications are being broadcasted. Without this meta information a JMX browser can not do this. The method marked with @NotificationInfo has to return an array of MBeanNotificationInfo objects. The example below is taken from the ComponentManager component found in the core of Argos.

Example: Notification Meta Data

```
@NotificationInfo
public MBeanNotificationInfo[] getNotificationInfo() {
    MBeanNotificationInfo[] not = new MBeanNotificationInfo[7];
    not[0] = new MBeanNotificationInfo(new String[]{SERVICE_STARTED},
             Notification.class.getClass().getName(),
             "Emitted when a new service is started.");
    not[1] = new MBeanNotificationInfo(new String[]{SERVICE_STOPPED},
             Notification.class.getClass().getName(),
             "Emitted when a service is stopped.");
    not[2] = new MBeanNotificationInfo(new String[]{SERVICE_LOADING},
        Notification.class.getClass().getName(),
             "Emitted when a service is loading.");
    not[3] = new MBeanNotificationInfo(new String[]{SERVICE_FAILED},
             Notification.class.getClass().getName(),
             "Emitted when a service failed loading.");
    not[4] = new MBeanNotificationInfo(new String[]{COMPONENT_STOPPED},
             Notification.class.getClass().getName(),
             "Emitted when a component has been stopped.");
    not[5] = new MBeanNotificationInfo(new String[]{STOPPING_COMPONENT},
             Notification.class.getClass().getName(),
             "Emitted when a component is being stopped.");
    not[6] = new MBeanNotificationInfo(new String[]{STOPPING_SERVICE},
             Notification.class.getClass().getName(),
             "Emitted when a service is being stopped.");
    return not;
}
```

@NotificationHandler is an important annotation. If components want to receive notifications from other components, how does Argos know how to give the notifications to the component? The answer is the @NotificationHandler annotation. A method marked with @NotificationHandler has to take one argument of type Notification. When a component gets notifications Argos then calls the method marked with @NotificationHandler with the notification as an argument. The example given below is taken from Jetty which receive notifications from the core.

Example: Notification Handler

```
@NotificationHandler
public void handler(Notification not) {
    //Adding web application omitted due to space.
}
```

A.5.4 Web Services

The @Webmethod annotation is the only annotation not found in the core. The annotation is located inside the !!Webservice.jar file (in the deployment folder of Argos). When creating web services this annotation is used. Creating web services is quite easy, just mark a method with @Webmethod and a web service will be created.

Example: Web Method

```
public class Test {
    @Webmethod
    public String hello() {
         return "Hello, world!";
    }
}
```

A.5.5 JMX related

In this sub section annotations related to tailoring JMX instrumentation are explained. By default any public method will be instrumented except methods defined by Object (all Objects extend java.lang.Object).

- **@Description**: Used to describe what methods and classes do.
- **@Impact**: Used to tell what the impact of a method invocation will be. Either ACTION, INFO or ACTION INFO.
- **@Instrument**: Used to explicitly tell Argos to instrument this method.
- **@InstrumentThisClassOnly**: Used to instrument only methods defined in this class and not methods defined in any super class.
- **@Removeinstrumentation**: Used to mark classes and methods telling Argos to explicitly not instrument the class or method.

A.6 Persistence

Hibernate is used in Argos as the object relational tool of choice. Hibernate is coupled with Derby 10.1. Using hibernate with all is capabilities is out of scope for this programmers manual. In this section you will learn how to access hibernate from your components. A tutorial can be found in [38]. It is important to note that hibernate does **not** support hot deployment. Hence, if you are using hibernate and you redeploy your service it will fail. You need to restart Argos to have your service properly redeployed.

Hibernate uses .hbm.xml files to do mapping from java objects to the database. To use hibernate in Argos you need to include your .hbm.xml files inside your service file. Where you place them does not matter as the hibernate system service searches the whole service file for .hbm.xml files and adds them to the mapping of hibernate.

When interacting with hibernate you interact with hibernates Session object. In Argos you can get your services session object like this:

Example: Hibernate Session Acquirement

```
public class Test {
    @ServiceMeta public ServiceMetaInfo meta;

    //@Execute(5)
    public void execute() {
        Session session = Hibernate.getSession(meta);
    }
}
```

It is important to note that the argument to getSession is the meta data of the service. This is to ensure that all components in a service use the same database to store their data.

Let's look at a small example inserting some data into the database using hibernate. First we need to create the class that will be stored in the database:

Example: Hibernate Class

```
package test;

public class Person {
    private Long id;
    private String firstName;
    private String lastName;
    private int age;

    //Setters and getters omitted
}
```

The person class is simple enough, basically a javabean class. The next thing we need to do is to create the hibernate mapping file.

Example: Hibernate Mapping File

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="test.Person" table="PERSON">
        <id name="id" column="PERSON_ID">
            <generator class="native"/>
        </id>
        <property name="age"/>
        <property name="firstName"/>
        <property name="lastName"/>
    </class>
</hibernate-mapping>
```

From the mapping file you can see that the id value will be generated automatically by hibernate and used as a primary key in the database. Beside from that all attributes in the

Person class is added so they will be mapped. The next thing we need to do is to create a component in Argos that uses the Person class and persist the data. Example given below:

Example: Using Hibernate

```
public class Test {
    @ServiceMeta public ServiceMetaInfo meta;

    //@Execute(5)
    public void execute() {
        Person p = new Person();
        p.setAge(25);
        p.setFirstName("Dan");
        p.setLastName("Eriksen");
        Session session = Hibernate.getSession(meta);
        Transaction trans = session.beginTransaction();
        trans.begin();
        session.save(p);
        trans.commit();
    }
}
```

First the person object is created and the values are set for the object. Then a session is gotten from hibernate, a transaction is started, the object is saved and last but not least the transaction is committed. For more detailed information about hibernate and its capabilities take a look at [38].

A.7 Web Applications

How to create web applications is out of scope for this programmer's manual. Jetty 6 which is the web container included by default in Argos has support for normal HTML pages as well as JSP and servlets. To include a web application in Argos you create a directory called "web" inside your service file and put all your web files there. The jetty system service will then deploy these files to jetty automatically. Your web application will then be made available on the following Uniform Resource Locator (URL): http://localhost:8080/ServiceName.

For a tutorial on web applications please take a look at Suns JEE tutorial [39].

A.8 Web Services

The web service system service in Argos offers not only SOAP but also XML-RPC. To create a web service mark a method in your component with the annotation @Webmethod. Remember to add dependency on the web service system service in your own service.

In SOAP the method then becomes available on the end-point:
http://localhost:8080/axis/services/ComponentName. In XML-RPC all methods are available on http://localhost:8080/xmlrpc

For a tutorial on web services please visit the JEE tutorial [39].