Prototypical Realisation of a Swing Application within a Java EE Infrastructure as Eclipse-based Rich Client

BACHELOR THESIS

for examination as

Bachelor of Science

in the study course

Applied Computer Science - International Business Competence

at the Baden-Wuerttemberg Cooperative State University Mannheim

by

Alexander Wolz

7th of June, 2010

Matriculation Number:	222204
Academic Course:	TAI07ABC
Handling Period:	12 Weeks
Cooperative Company:	Fraport AG, Frankfurt am Main
Company's Supervisor:	Dipl.-Inf. Christian Seufert
University's Supervisor:	Prof. Dr. Eckhard Kruse

Wolz, Alexander:
Prototypical Realisation of a Swing Application within a Java EE Infrastructure as Eclipse-based Rich Client
Bachelor Thesis
Baden-Wuerttemberg Cooperative State University Mannheim
Period: 2010/03/15 - 2010/06/07

Sperrvermerk

Die vorliegende Arbeit des Studenten an der Dualen Hochschule Baden-Württemberg in Mannheim, Herrn Alexander Wolz (Matrikelnummer 222204; Kurs TAI07ABC) beinhaltet interne sowie streng vertrauliche Informationen der Fraport AG.

Die Weitergabe des Inhalts der Arbeit und eventuell beiliegender Zeichnungen und Daten im Gesamten oder in Teilen ist grundsätzlich untersagt. Es dürfen keinerlei Kopien oder Abschriften – auch in digitaler Form – angefertigt werden. Ausnahmen bedürfen der schriftlichen Genehmigung der Fraport AG.

Location, Date	Dieter Steinmann

Declaration of Authorship

I hereby confirm that I have authored this bachelor thesis independently and without the use of others than the indicated resources. It has not been submitted for a degree or examination at any other university.

Location, Date

Alexander Wolz

Acknowledgements

First of all I would like to thank my supervisor *Christian Seufert* who believed in me and continuously supported me in every problem that has occurred. I especially thank him for his encouragement and his accurate comments, which were of critical importance during this work.

In addition, I owe special thanks to *Jaqueline Dechamps*, *Christian Wrobel*, *Manuel Hegemann* and *Stefan Sabatzki*, who spent much time in giving advices and support to the actual topic.

Another thanks I owe to *Prof. Dr. Eckhard Kruse* for attending me from the university side and for constantly providing innovative encouragements.

I also would like to thank *Dieter Steinmann*, the department manager of IUK-AE4, for allowing me to write this bachelor thesis in his department.

Furthermore, I have to especially thank *Marina Wolf*, *Jan Wistuba* and *Max Wilhelm* for proofreading this bachelor thesis.

Abstract

At Fraport AG, the company-wide data warehouse *BIAF* is used, which is specialized in the analysis and reporting of operational data. In addition to its reporting system, BIAF supports an airport-map, in that the data of the airport operations are visualized in geopraphical maps.

The *Airport-Map* is a Java EE application, whose client is based on the Java Swing Technology. Unfortunately, its development turned out to be quite problematic. On the one hand, there are some incompatibilities between the deployed frameworks and the client's environment. On the other hand, and that's the more important topic, the client is difficult to maintain and to expand because there is no strict separation of model, logic and graphical user interface.

Therefore the task of this bachelor thesis is to evaluate the use of Eclipse RCP for the Airport-Map's client and to prototypically realise it.

Kurzzusammenfassung

Bei der Fraport AG wird ein unternehemsweites Data Warehouse *"BIAF"* eingesetzt, welches auf die Analyse und das Reporting von operativen Daten spezialisiert ist. In Ergänzung zu seinem Berichtswesen stellt BIAF eine Airport-Map zur Verfügung in der die Daten über den Flughafenbetrieb in geografischen Karten visualisiert werden.

Bei der Airport-Map handelt es sich um eine Java EE-Applikation mit einem auf Java Swing basierten Client. Die Entwicklung des Clients hat sich allerdings in der Vergangenheit als problematisch erwiesen. Zum Einen existieren einige Inkompatibilitäten zwischen den verwendeten Frameworks und der Clientumgebung. Zum Anderen gestaltet sich die Wartbarkeit und Erweiterbarkeit des Clients als schwierig, da keine strikte Trennung zwischen Modell und der graphischen Benutzeroberfläche (GUI) exisitiert.

Die Aufgabe für diese Bachelor-Arbeit besteht darin, den Einsatz von Eclipse RCP für die Clientapplikation der Airport-Map zu evaluieren und prototypisch umzusetzen.

List of Abbreviations

AODB	Airport Operational Database
AOP	Aspect Oriented Programming
API	Application Programming Interface
AWT	Abstract Window Toolkit
BI	Business Intelligence
BIAF	Business Intelligence Architecture Framework
CRUD	Create, Read, Update, Delete
EAR	Enterprise Application Archive
EJB	Enterprise JavaBeans
ETL	Extract, Transform, Load
FAST-MS	Frankfurt Airport Surveillance Traffic Management System
GIS	Graphic Information System
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines
IDE	Integrated Development Environment
ILOG	Intelligence Logicielle
IoC	Inversion of Control
IUK	Information and Telecommunication
JAAS	Java Authentication and Authorization Service
Java EE	Java Platform, Enterprise Edition
JMS	Java Message Service
JNDI	Java Naming and Directory Interface

JRE	Java Runtime Environment
MB	Megabyte
MVC	Model-View-Controller
OSGi	Open Services Gateway Initiative
POJO	Plain Old Java Object
RCP	Rich Client Platform
REST	Representational State Transfer
RMI	Remote Method Invocation
SAP	Systems, Applications, and Products in Data Processing
SAS	Statistical Analysis System
SOAP	Simple Object Access Protocol
Spring DM	Spring Dynamic Modules
SQL	Structured Query Language
SWT	Standard Window Toolkit
UI	User Interface
URL	Uniform Resource Locator
WAS	WebSphere Application Server
WSDL	Web Service Description Language
XML	Extensible Markup Language

List of Figures

Listings

Contents

1. Introduction

This chapter shortly gives an introduction to the workplace and environment, in which this bachelor thesis has been elaborated. In the first part, Fraport AG as well as the actual department are illustrated and afterwards, the motivation as well as outline for this thesis are covered.

1.1. Fraport AG

The *Fraport AG*, which was founded in 1924 as *"Südwestdeutsche Luftverkehrs AG"*, operates the largest airport in the Federal Republic of Germany - The Frankfurt International Airport. Fraport AG is an attendance provider, which offers a complete infrastructure and the corresponding services to its customers. The most important of the more than 500 customers are for example the different airlines, a variety of authorities, but also the German air-traffic control. Fraport's core competence is to provide and manage complex air-traffic hubs.

After the admission of air traffic at Frankfurt/Main in the early 1930's, the airport permanently experiences one after another and is subject to a constant growth. Today, the size of the airport ground is about 21 square kilometres that enfolds amongst others 3 runways, 2 passenger terminals and a baggage handling system with a total length of 73 kilometres.

In 2008 there were about 485.000 aircraft movements, which approximately corresponds to an average number of 1.300 starts and landings that have transported nearly 54 million passengers and 2 million tons of cargo. By reason of this permanently increasing values and to shape up well for the future, it is planed to build a new runway in the north west of the airport, as well as a third passenger terminal at the former U.S. airbase in the south.

In an international comparison, Frankfurt is ranked in ninth place in the category "passenger volume" and is on seventh position in "cargo business", while in Europe it is ranked in third and first place in the same categories.

Because of its central position in Germany and its high passenger volume, Frankfurt Airport belongs to one of the most important hubs in Europe besides London-Heathrow and Paris Charles-de-Gaulle. [Fra09, Fra10]

Additionally, more than 71.000 employees, from which over 19.000 are employed directly at Fraport AG and its subsidiary companies, make Frankfurt Airport to the biggest employer in Germany and because of its many experiences in the airport business, Fraport AG also has world-wide investments in several airports. Its intention, amongst financial reasons, is to share its expert knowledge and to finally improve them. Figure 1.1 shows an overview of those global investments, as there are for example investments in Orlando or Cairo.

Figure 1.1.: Fraport's Worldwide Investments [Fra10]

For the provision of IT services and infrastructure on the campus of Frankfurt Airport, Fraport AG is totally responsible by its own. Those services are not only provided for the own concern, but also for all companies located at the airport. For this purpose, an on airport business specialized division called *"Information and Telecommunication"* (IUK) is kept, which is responsible for IT architecture, security, policy and standards, but also for planning, realizations and maintenance.

In addition, IUK is segmented into further divisions due to organisational and strategical reasons. One of those subdivisions is the *AE4*, which is explicitly responsible for business systems.

Business Systems - AE4

The department *AE4* operates within the strategic division of *IUK* and is the abbreviation for *"Enterprise Resource Planning, Dispatching and Management Applications 4"*. The division's main responsibilities are maintaining the *"Business Systems"*, which include amongst others SAP[1]-based supplier and customer relationship management systems, but also data warehouse architectures and simulation systems. Further activity fields can be gathered from figure 1.2.

Figure 1.2.: AE4's Activity Fields

Furthermore, the division also provides different reporting and analysis tools that are stacked on top of the company-wide data warehouse *"BIAF"*[2]. This is a Business Intelligence Platform that provides airport-specific information for goal-oriented statistics and dynamic reporting. Additionally to that, BIAF itself also gets maintained and improved by AE4.

[1]Systems Applications and Products
[2]Business Intelligence Architecture Framework

1.2. Motivation

The Java EE[3]-based client application *AirportMap* is being installed automatically to the users computers via the Java Web Start technology. The advantage for this kind of software distribution at the one hand is that the distribution need not to happen by the inflexible software roll-out processes of the "Software Depot" and on the other hand, the client release cycles can be totally decoupled from the server side. This is advantageous if releasing of new server versions underlies fixed processes, as it is common at Fraport AG. The need for a flexible client roll-out is based among others on the fact that due to permanent building measures at Frankfurt Airport, the cartographic material rapidly becomes obsolete and has to be updated periodically.

Unfortunately, there occurred many problems with this kind of software distribution in the past, which predominantly are caused by incompatibilities between the client's runtime environment and its used frameworks. One reason for those incompatibilities is the fact that the client does not directly run within a Java Runtime Environment (JRE) but a Java EE-based client container. This makes it possible to get access to the server's resources easily by calling its JNDI[4] name. Sadly the installation and starting of the application with that client container is difficult and of bad performance.

To solve those problems, the existing AirportMap client application shall be redeveloped to gain more modularity as well as extensibility. Since the development must agree with requirements of different departments, the client shall have a modular approach on application level instead of code level. This shall bring the ability to develop application components customized to individual departments.

This for example is provided by so-called *"Rich Client Platforms"* that bring own plug-in concepts and update functionality. Therefore, it shall be investigated if the current client application can be realized on base of such a platform.

[3]Java Platform, Enterprise Edition
[4]Java Naming and Directory Interface

1.3. Outline

In chapter 2, the fundamental technologies are illustrated in advance, in order to provide a better understanding of the overall situation and to impart a proper knowledge of the already used technologies. Chapter 3 shorty illuminates the current situation, as it was in the beginning of the bachelor thesis. Thereby, the basic systems are introduced, which are of great importance for the topic of this thesis.

Afterwards, chapter 4 defines the business requirements, which regulates the goals and limitations, as well as the functional and non-functional requirements in focus of software engineering purposes. In the following chapter, possible solutions to the current implementation are discussed, whereas the focus is on web applications and rich clients.

In the sixth chapter, the Eclipse Rich Client Platform gets described in detail, since it is used for building the new application's fundamental groundwork. Thereby especially its assembly and architecture are very much in the foreground.

Chapter 7 introduces the system design, as well as required modifications, which have to be done, in order to integrate used framework and to finally enable a redevelopment of the AirportMap client with Eclipse RCP. The subsequent eighth chapter then puts all requirements into practice, as it reveals the actual steps, which were performed to reach the goals, mentioned in the chapter before.

In the end, the final chapter shortly summarises all performed steps and subsequently provides an outlook to its further usage in the company.

2. Technologies

This chapter gives an introduction to the basic technologies, which are required for understanding this bachelor thesis. Experienced readers with an appropriate prior knowledge could skip this part, since the actual content is handled in successive chapters.
In the first section, the common term "Business Intelligence" gets illustrated and afterwards, the fundamental Java technologies are introduced. Those first of all contain the used Java frameworks, distribution tools, as well as the OSGi Service Platform.

2.1. Business Intelligence

Business intelligence (BI) in general is a broad category of procedures and technologies for gathering systematic analysis of business data, in order to help enterprise users to make better business decisions.

By using analytical concepts and IT-systems, data of the own company are evaluated to improve decisiveness in operational and strategical units. The term *Intelligence* thereby is in context for obtained knowledge and information by analytical investigations.

Additionally, Business Intelligence can be segmented fundamentally in three independent phases: At first the referencing values are captured for data acquisition. This means that it will be decided which data should be gathered and saved or not. The data then is inserted into a so-called *Data Warehouse*. The second phase is recognition of associations and dependencies between the saved data. This is done by several analytical methods. The last phase forwards all gathered information to the management, which makes operational and strategical decisions based on this perceptions.

Hence, Business Intelligence applications include the activities of decision support systems, online analytical processing, statistical analysis, forecasting, as well as data mining, querying and reporting.

2.2. Java Technologies

Java is becoming more and more popular and therefore it's not unusual that it has taken standard at Fraport AG. Its platform-independence and easy to learn syntax makes it to an interesting and powerful programming language. Because Java enables object-oriented programming, developers can produce modern and reusable software components [Ull08]. For this reason, the most important Java technologies will be covered within this section.

Java Frameworks

A framework in general is a set of common software routines that offers a basic structure for developing an application. Frameworks take over the developer's work of writing all the program code from scratch. Today's frameworks, which are mostly object-oriented, are structured as a class library.
Each class library has its way of doing things and although the purpose of a framework is to eliminate a certain amount of code writing, developers must first learn its structure, in order to use it. The *Spring*[1] framework for example is a popular framework for decoupling application components.

The examined application is fairly large and uses many open-source and commercial frameworks for not building up everything from scratch. Therefore, the most important ones will be explained in advance:

1. *Spring Framework*
2. *ILOG JViews Framework*

[1] http://www.springsource.org

The ***Spring Framework*** is an open source application framework, created to simplify the development of complex enterprise applications. The intention of what Spring does, can not be answered easily, but Rod Johnson, one of the founders of Spring, describes it as follows:

> *"Spring's main aim is to make enterprise Java easier to use and to promote good programming practice."* [Joh07]

The complexity of developing Java applications lies in the multitude of Java APIs and their different underlying concepts. Spring solves this problem by providing a simplified and unified API. It also provides a solution for the management of dependencies between objects.

In contrast to enterprise solutions such as *Enterprise Java Beans* (EJB), Spring has a so-called *"non-invasive"* approach. Thereby, the source code gets as few dependencies as possible, because the framework generally does not enforce to implement specific interfaces or to inherit from its classes. Only so-called *"Plain Old Java Objects"* (POJO[2]) are getting put together and additional services like transaction and security controls are configured descriptively.

An application using the Spring Framework underlies the so-called *"Inversion of Control"* (IoC), which means that it enhances loose coupling by letting Spring's lightweight container take the responsibility of establishing dependencies between objects. The IoC pattern is a central principle of a framework, in which they assume the role of the main program, in order to coordinate the flow of application activity. In context of container architecture, the term *IoC* is too common, so that it can be further distinguished to *Dependency Injection*. In this pattern, the dependent objects get injected from the outside instead of finding them by itself. The components express their dependencies by providing setter-methods (setter injection) or specific constructors (constructor injection). [Fow04]

Spring itself consists of several modules, as illustrated in Figure 2.1, which can be used independently from each other and therefore it is called a "lightweight"

[2] Term introduced by Martin Fowler to differ between simple Java objects and objects with multiple external dependencies.

framework. The central concept of Spring is the inversion of control at the one hand as well as the *Aspect Oriented Programming* (AOP) on the other.

AOP complements *Object-Oriented Programming* by providing another way of thinking about the program structure. It is a programming paradigm that enables to develop and to test different logical aspects of an application separately. In the terminology of AOP, aspects like transaction or security are called *Cross-cutting Concerns*, because their implementations are spread over several classes of the application and can not be encapsulated in a single module. [JHK09]

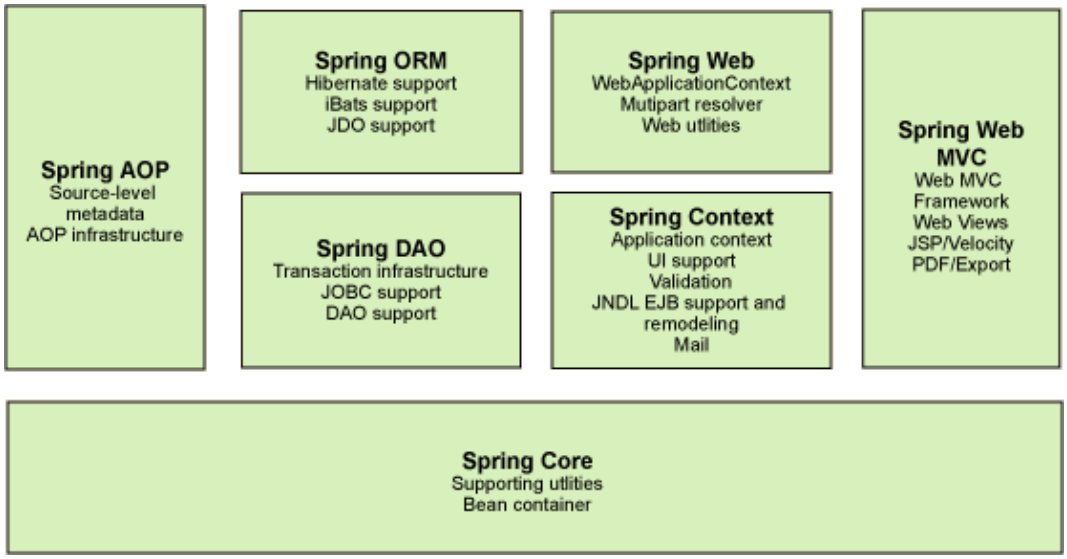

Figure 2.1.: The Spring Framework Modules [JHK09]

Spring offers a consistent approach to integrate existing technologies, as for example for database accesses or web development. [ZLM08]. Furthermore, the combination of IoC and AOP can be used alternatively to EJB for integrating business applications in a Java EE environment.

The ***ILOG JViews Maps Framework*** is a commercial *Geographic Information System* (GIS) of good performance. A Geographic Information System captures, stores, analyses, manages, and presents data that are linked by Cartesian coordinates to locations. In the simplest way, they merge cartographic material with database technology.

The by IBM distributed ILOG product JViews Maps provides excellent zooming and panning functionality and therefore it is a criterion for using it. The framework also brings functionality like a flexible visual representation of background map data. For example, it makes sense to see streets only at a given zoom level and main roads or motorways only when zooming out. Additionally, the framework offers a mechanism for interactive and automatic updating of symbols when the underlying application data changes. This is a basic requirement for air traffic monitoring. [IBM09]

Unfortunately, map data sets can be many gigabytes and specific memory management and display techniques are needed to provide satisfactory performance levels but the ILOG JViews Framework handles those quite problems well. [IBM09]

Java Distribution Tools

Software distribution is a big subject in today's companies. There are many ways to distribute software and one of them is the distribution by a *Package Management System*.

Software quickly becomes obsolete and hence requires constant updates. By using a package management tool, such as the at Fraport used *Software Depot*, developers can bring in the most actual software version and is the most user-friendly way of providing applications. Installation routines could be quite difficult, but this topic is handled by the Software Depot.

Unfortunately the roll-out process of the Software Depot is quite inflexible, complex and bureaucratic. For Java applications, there is an alternative solution: *Java Web Start*. Using the Java Web Start technology, standalone Java applications can be deployed with a single click over the network. This mechanism ensures that the most current version, as well as the correct version of the Java Runtime Environment will be deployed. [SUN09]

By using this mechanism, software must not implicitly rolled out with an inflexible Package Management System, but by calling a network resource. Furthermore, Java Web Start belongs to the category *Java Desktop Technologies*.

Java Platform, Enterprise Edition

Java Platform, Enterprise Edition[3] (Java EE) 6 is the youngest release of a bundle of Java technologies for developing distributed applications with

1. *a lot of server-side logic*
2. *a web or rich client front-end*
3. *access to databases or other back-end systems*

Java EE applications normally run within an application server such as *WebSphere*, *JBoss* or in the easiest way, within a Servlet container such as *Apache Tomcat*. Application Servers provide special services like transactions, security or access to directory services and databases over defined interfaces.

At Fraport AG, the IBM *WebSphere Application Server* (WAS) is standard and therefore used for most applications. To understand this thesis, following Java EE technologies are important to know:

1. *Java Message Service*
2. *Java Naming and Directory Interface*
3. *Enterprise JavaBeans*
4. *Web Technologies*

Distributed systems communicate and coordinate by sending and receiving messages. There is no common memory management like the *"shared memory"* technology and therefore a message system is one of the most important criteria for a distributed system.

The ***Java Message Service*** (JMS) is a messaging standard that allows application components based on Java EE to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous. [SUN02]

The JMS Application Programming Interface (API) specification and its corresponding service provide two different rudiments to send messages. On the one

[3] Java EE or JEE, formerly J2EE

hand there are *message queues* and on the other hand there are *topics*. Message queues and topics provide an asynchronous communication, what means that the sender and receiver do not need to interact with the message queue at the same time. Messages placed onto the queue are stored until the recipient fetches them.

The other communication method is based on topics or channels, where messages are published to. Subscribers (consumers) will receive all messages published to the topics to which they subscribe, and all subscribers to a topic will receive the same messages. The publisher (producer) is responsible for defining the messages to which consumers can subscribe.

For using JMS, a message provider is needed to offer the service and to implement the API. There are commercial as well as open-source solutions. At Frankfurt Airport, the commercial IBM *MQSeries* is used to do that task.

The ***Java Naming and Directory Interface*** (JNDI) is a programming interface for multiple name and directory services and independent of the underlying implementation. It allows depositing data and object references by its unique name and can be accessed by Java software clients. It is mostly used for registering distributed objects in a network and to make them available for *Remote Method Invocations* (RMI). RMI is a Java implementation for *Remote Procedure Calls*, which means that objects running within a different Java Virtual Machine can be accessed by a remote or local application.

If a client wants to obtain a certain remote object via JNDI, it localizes the object by its network name (or additionally by a fitting context in the namespace), which is defined at deployment. By performing those *"lookups"*, the client gets reference to the corresponding remote object. [BG02]

The ***Enterprise JavaBeans*** (EJB) technology is the central server-side component architecture for the Java Platform, Enterprise Edition to represent business logic and its data. Enterprise JavaBeans enables a rapid and simplified development of distributed, transactional, secure and portable applications based on Java technology.

Its components, known as *EJB-Components* or *Enterprise Beans*, appear at runtime as logical units and offer their functionality (the business logic) to clients as Java interfaces. EJB can be further distinguished to the *EJB-Server* and *Client*, as well as the *EJB-Container*. The EJB-Server provides enterprise services and the connection to information infrastructure, whereas the client connects to it by remote or local method invocations (RMI) or just through a message service like JMS. [BG02]

The EJB-Container on the other hand offers a runtime environment to the installed components. For this purpose, it manages especially the bean instances, controls their life cycle and provides them a standardized access to enterprise services. In an EJB based application, the carrier systems consists at least of the combination of an EJB-server and -container. [BG02]

The ***Web Technology*** also plays an important role in many enterprise applications. Using Web Technologies like *Java Servlets* or *Web Services*, clients can easily send requests to the corresponding server that generally returns the processed data. The requesting happens via the *Hypertext Transfer Protocol* (HTTP), because it is a lightweight and reliable protocol.

A Servlet in general is a Java class, whose instances process and reply client's requests within a Servlet container such as *Apache Tomcat*. They form the entry point between HTTP and Java. Figure 2.2 approximately shows, how web services can be realised by using this technology.

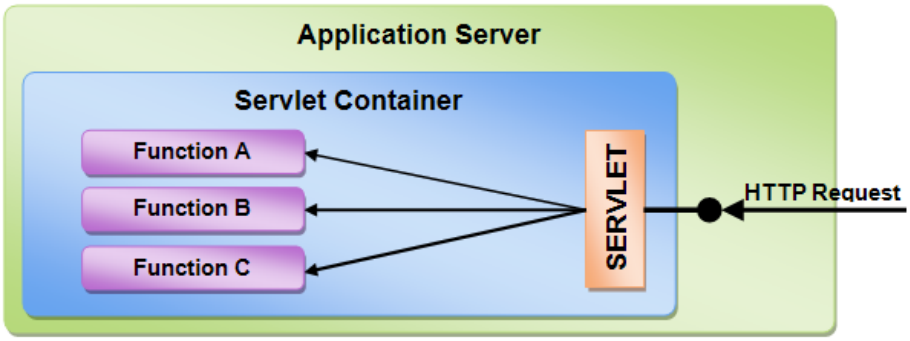

Figure 2.2.: Java Servlet

Servlets normally are bound to a certain URL address, which can include parameters and values. On base of those parameters, the Servlet's logic is able to decide, which function has to be invoked. If for example the address "http://server/service?function=a" is called, the Servlet implementation would access function A, which can perform some server actions.

Web Services are mostly realised using those Java Servlets or other technologies, such as *PHP* for example and can be described as follows:

> *"Web Services are software systems designed to support interoperable machine-to-machine interaction over a network."* [W3C04]

Furthermore, Web Services are generally reachable via an unique address, the so-called *Uniform Resource Locator* (URL) and define their interfaces with the *Simple Object Access Protocol* (SOAP) or other XML artefacts. This makes them quite platform independent, what means that they can also be accessed by other clients than only Java-based applications.

OSGi Service Platform

The development of Java based modular application systems in the past used to be quite difficult, because there were no straight language concepts. This problem is solved by introducing a platform that provides a dynamic module system for Java: *The OSGi Service Platform.*

OSGI, which stands for *"Open Services Gateway Initiative"*, offers a dynamic system, which enables the definition of modules, whose visibilities and dependencies can be defined explicitly. An OSGi module generally consists of several classes located in particular packages, as well as a module description file (Manifest), which defines the dependencies and visibility limitations between single modules. Hence, those modules are also referred to as *"bundles"*.

According to [CHL08, p.12], the OSGi Service Platform is responsible for managing the life cycle of each bundle, as the state diagram of Figure 2.3 shows.

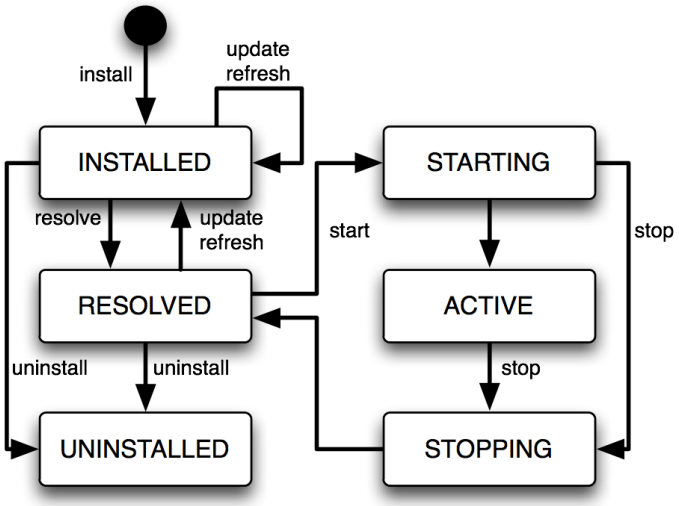

Figure 2.3.: The OSGi Bundle Life Cycle [CHL08]

At first, every bundle gets installed and therefore is available in the *INSTALLED* state. If a request for starting the bundle comes up, OSGi will successively resolve the bundle and if this was successful, it will be put into the *ACTIVE* state. When a request is done to stop the bundle, the OSGi framework will move it back to the *INSTALLED* state. A request may then be made to uninstall the bundle and while it is in *INSTALLED* or *ACTIVE* state, it could be updated. Finally, if a bundles is in INSTALLED state, it can be uninstalled if wanted. This design offers the possibility to implement the idea of modularisation to larger units than only classes. Due to the OSGi service architecture, bundles offer particular services and use services provided by others.

The OSGi Service Platform is not a direct implementation, but a clear specification of how such a platform must be look like. The *OSGi Alliance*, a world-wide consortium of trusts like *IBM* or *Sun*, specifies those guidelines and hence, there is a multiplicity of commercial and free OSGI frameworks implementations.

3. Current Situation

This chapter reflects the current situation, as it was before the bachelor thesis has been started. It provides an overview of the company-wide data warehouse "BIAF" as well as its basic systems and environments. Then the chapter introduces the AirportMap, which forms the main issue to this thesis.

3.1. BIAF Data Warehouse

The *Business Intelligence Architecture Framework* (BIAF) is a Business Intelligence Platform that provides airport-specific information for goal-oriented statistic or dynamic reporting. The information is conditioned to the different information needs of users and groups. Since distinct departments of the Airport also have different views on the airport processes, the information is prepared and provided with regard to different stakeholders.

Before BIAF was introduced in 2005, each system at Fraport AG had its own reporting engine and the analysis of productive data was error-prone and inconsistent. At present, BIAF delivers a central information platform that enables a global and consistent view to company data and replaces the isolated solutions of other departments. Thereby it is important that BIAF makes daily as well as historical data available for instant evaluation. Also fast and flexible reporting with standard tools by individual analysts is now possible. [SD10]

BIAF combines both, a company-wide Business Intelligence Platform as well as a *Data Warehouse*. This architecture includes a defined infrastructure and a consistent data model. Its comprehensive data pool allows statistics, histor-

ical analyses and forecasts based on SAS[1] technologies. With help of a web application, known as *BIAF portal* or additional software, the users have the possibility to display, create and individualize reports.

> *"A data warehouse is a repository of accurate, time related information that can be used to better understand your company."* [Bou09]

In general, a data warehouse is an information system that consists of a staging area, a central database and several data marts, as shown in Figure 3.1.

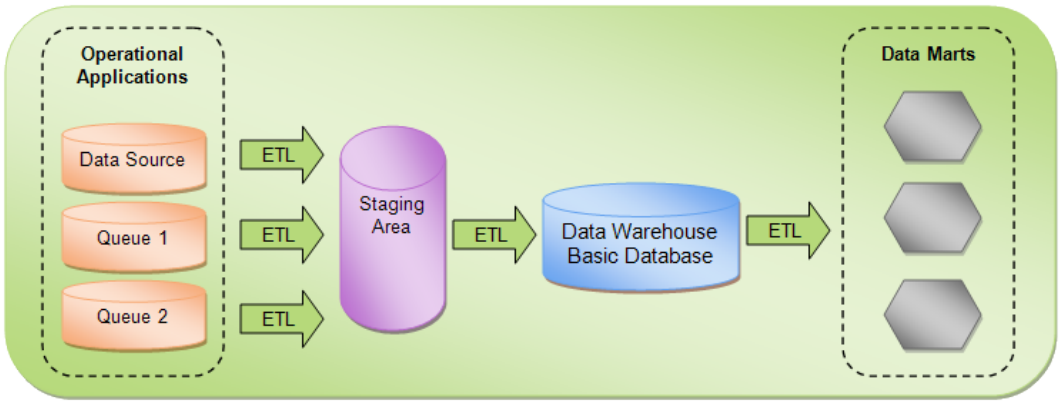

Figure 3.1.: Basic Data Warehouse Architecture

In the staging area, all data sources are stored by ETL[2] processes. It forms a data pool for temporary storage, in order to decrease the stress of the central database. In general, queries are send to this database and therefore the ETL processes use the provided staging area to not compete with them about performance. ETL ensures that data gets extracted, cleaned, transformed and loaded into the central data warehouse database. For performance reasons, the database is only filled at night. All information is available in one single data pool and can be stored separately in the data marts. Data marts are redundant storages of high performance because they contain only data that is customized for the departments.

[1] major producer of business intelligence solutions
[2] Extract, Transform, Load

Furthermore, there is an important *Airport Operational Database* (AODB), called INFOplus, which gathers all airport specific data like flight information, ground handling, passenger lists and so on. This AODB is a core system of Frankfurt Airport and required for business operations. All typical airport participants like apron tower, baggage driver, fire fighter or load masters depend on the information provided by INFOplus. Additionally to that, it is one of the data sources for BIAF, as it can be seen on figure 3.2. Amongst others, the weather service and baggage system are connected to it.

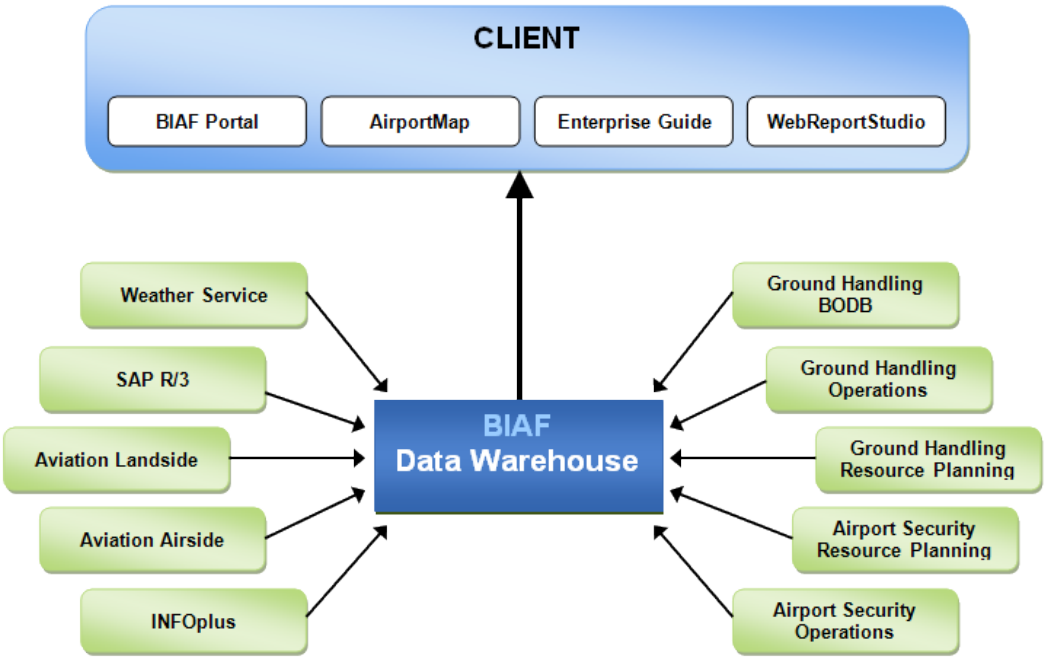

Figure 3.2.: BIAF Structure and Sources

Since launching BIAF in 2005, many different systems could be connected and the corresponding department analysts can create reports via diverse tools like *BIAF Portal*, *Enterprise Guide* or *WebReportStudio*. [SD10]

In recent years, the *Business Intelligence Architecture Framework* has become one of the most important systems at the Frankfurt Airport and its user group increases rapidly, as there are currently more than 750 users at Frankfurt Airport. Furthermore, BIAF is the underlying information provider for the visualising *AirportMap* application.

3.2. AirportMap

In certain situations, reports that consist only of data, figures and facts, are not the right instrument for representing information. In some cases, it requires a visualization to understand and interpret the widely complex processes.

AirportMap is such an information system for air-side processes, which visualizes the current airport situation and potential problems that can occur. In comparison to other productive systems like INFOplus, the AirportMap is not considered a critical system and airport business will not be influenced if it crashes. Rather, it is an information system that provides additional and valuable information to support airport domain experts in their decisions. The main goal for the application was to support and serve only for information and indication purposes.

The AirportMap at first represents the geographical map of Frankfurt Airport with all of its areas such as runways, landing strips and buildings. Those areas, which form the map's background, are segmented into single layers and can be hidden or shown if wanted. On top of this backgrounds, which are based on Cartesian coordinates, a representation of the current airport situation is drawn, as it can be seen in figure A.1 of the appendix.

Every aircraft broadcasts its current position, which gets stored constantly in a movement system called *FAST-MS*. The AirportMap server prepares those and other relevant information from the centralized data warehouse and regularly pushes them into a certain topic channel, from which connected clients can receive those information. Based on this constant receipt of data, the current aircraft positions get updated, so that the impression of a radar occurs.

But AirportMap also offers different process views, which have their focus on special business processes. For example, the process view *Punctuality* has its focus on delayed aircraft and standby positions. Irrelevant processes get dwarfed and only delays come to the fore. Since every department has its own requirement of airport information and views, the sectioning to different views enables a customized usage of the whole application. Furthermore, AirportMap indicates potential delays and conflicts by highlighting corresponding aircraft or standby positions depending on the particular process view. For

instance, the process view *Aviation* highlights current or upcoming standby position conflicts in red, what means that an aircraft is going to use a still occupied position. This conflict and delay calculation is done by interpreting events and so the event management is an important component of the AirportMap. Events like standby or departure are also displayed separately in an own column.

3.2.1. Architecture

The AirportMap has a so-called "*client-server*" architecture. This structure classifies the whole system to a back- and front-end part: the server and client. In a client-server architecture, the server provides services that are accessible and consumed by one or more remote clients. Resources and data security are controlled by the server, which guarantees that only clients with appropriate permissions can access and manipulate data. In general, clients and servers communicate over a network, but accessing services can vary in different technological ways.

The simplest method to access a service is to invoke a procedure on the remote server, which is known as remote procedure call. In context of Java EE, remote procedure calls are implemented on the server side with Enterprise JavaBeans (EJB). There are three different types of EJBs: Entity, Stateless and Stateful Session Beans. [IHH07]

Since the AirportMap is only offering a simple service for authentication and data initialization, a Stateless Session Bean is used to expose these services for remote connection.

In contrast to Stateless Session Beans, which expose server functionality in a synchronous request-reply manner, there are sometimes requirements where an asynchronous communication model is more appropriate. The last one comes up in event-based systems, where applications are informed by other applications through messaging. In such a communication model, a producer sends a message to a messaging system and consumers, which are interested in this information, subscribe to this message system, in order to receive the message. This publish-subscribe procedure is called *topic*.

Topics are one of the communication channels provided by the *Java Message Service* (JMS), which in fact is used to realise the asynchronous communication for the AirportMap system. They are needed for pushing event notifications and aircraft movements to all clients, so that they all receive the same messages. JMS furthermore promotes loose coupling, what means that publisher and subscriber do not have to be online at the same time. Messages put into the topic are available for a certain amount of time and therefore can be received even if the publisher is not connected any more. This brings a more appropriate approach, than calling server functions remotely.

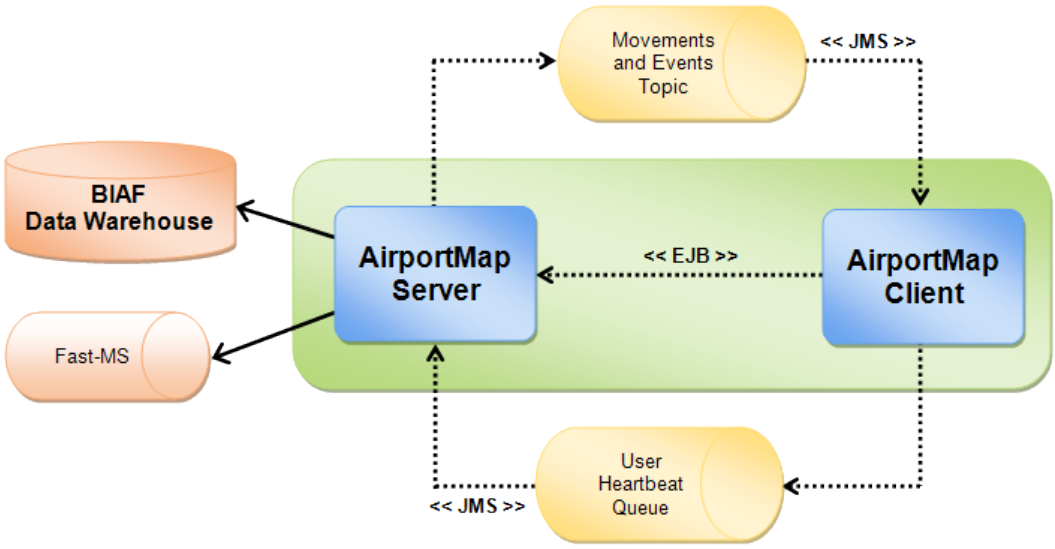

Figure 3.3.: AirportMap Architecture

Figure 3.3 shows an overview of the server-client communication of the AirportMap system. The server requests all required data by polling the BIAF data warehouse every 50 seconds. Those data amongst others get turned into events and are published to a topic. Additionally, the individual aircraft coordinates provided by the *Fast-MS* queue are also published to this topic. On start-up, clients contact the server's EJB in order to validate authentication and to initialize all relevant events. Afterwards, movements and events are received every second by being connected to the topic.

Another used communication channel of JMS is the so-called *message queue*. By using this communication medium, messages are placed onto a stack and are stored until the recipient retrieves them. After a message is being delivered, it gets removed from the queue, what is the main difference to topics.

In the AirportMap, message queues are used to transmit *heartbeats*, which in fact are short notifications of appropriate clients to inform the server of still being connected. This feature is only needed for business purposes, such as generating reports of the average usage or member counts and does not influence the actual AirportMap processes.

Furthermore, the client application also contains basic data like the cartographic material[3] of Frankfurt Airport, which is segmented into several layers. This enables an easier handling, because individual layers can be shown or hidden on demand. On top of these maps, the received events get displayed as objects or notifications to the active application window. Depending on the type of events, as there are for example *Arrival, Delayed* or *Standby* events, the visualisation of aircraft, positions and their highlighting is done.

Currently, the AirportMap front-end is a Swing-based Java EE application, which means that its visual components like backgrounds, buttons or labels are based on the Java Swing technology and that the application itself runs within a Java EE environment. Swing is a widget toolkit and API for developing graphical user interfaces and is an improvement of the *Abstract Window Toolkit*. It also provides platform-independent graphical elements, which emulate the look and feel of several platforms. It is modular as well as object-oriented and therefore well-suited for developing complex graphical applications. Because the components are just emulated rather than of using the native graphic libraries of the underlying operating system, the performance in fact is worse than other graphic technologies.

[3] In average, the map gets updated every 4 weeks due to building measures

3.2.2. Used Frameworks

Besides architecture, the AirportMap client application also uses a variety of frameworks and technologies. Frameworks offer basic structures and allow to reuse program segments. To avoid unnecessary work and developing existing components from scratch, frameworks are used to found the application's groundwork.

For displaying Frankfurt Airport's cartographic material as individual layers, the commercial *JViews Maps* framework is used. There is an own shape file for each layer, which gets rendered and assembled by the internal engine. The graphical modules are provided as Java Swing components for becoming platform independent. Furthermore JViews Maps originally offers additional features like zooming, panning and magnifying and since it is quite reliable and of good performance, it was selected for the displaying engine.

Another very important part of the AirportMap is the *Spring Framework*. Spring simplifies developing enterprise applications by offering among others a mechanism to integrate familiar frameworks. Besides the provided Inversion of Control and Dependency Injection, which is used widely in the AirportMap, in order to manage the object's life cycles, Spring promotes good programming practices.

As the AirportMap uses Enterprise JavaBeans technology, Spring is not intended to replace the EJB-container, but to simplify the usage of its components, their implementations as well as their functionalities. For example, the lookups for the server's EJB-container can be completely taken out of the code into special spring configurations, so that different implementations of a component can be used if required. In context of the AirportMap, there are three configurations, which are used: a productive, a testing and a quality control configuration. One of the key benefits of Spring is that testing of those components is much more facile because they are just normal Java components.

Furthermore, Spring is used to integrate message queues, since it brings special containers and classes that simplify the connection to certain queues or topics. By the use of provided handlers, an action can be performed whenever an incoming message occurs.

Those reasons altogether make the Spring and JViews Maps framework to important and helpful components of the AirportMap.

3.3. Prospects

The AirportMap was created out of the necessity to visually represent textual reports and analysis of the BIAF data warehouse. Analysing only facts and figures is often unclear and could lead to confusions. Having an application, which enables visualizing those reports in a graphical way would simplify the overall view and gives users the ability to recognize problems immediately.

The head of the Frankfurt traffic control, which for example could close the airport in extreme situations, constantly uses this application, by its own statements. Due to the visualisation of the present airport situation, the possibility is given to identify current and potential conflicts immediately and he therefore can initiate countermeasures instantly.

This circumstance makes the AirportMap to one of the most important and helpful utilities of the Frankfurt Airport and therefore it is important to constantly keep the application up to date. Because the present implementation is hard to maintain and difficult to update, the client application requires a fundamental reorganization and revision.

4. Business Requirements

This chapter clarifies the scope and requirements of this bachelor thesis. In the first part, the actual problem is defined, which was the decisive factor for initiating this issue. In the second part, the goals as well as limitations get introduced and finally, the requirements are distinguished.

4.1. Problem Definition

The *AirportMap* application has a high complexity on the technical side, as well as on the domain side. The technical complexity is due to the fact that there is a versatile but required mixture of technologies such as *Enterprise JavaBeans*, *Spring* but also other Open-Source frameworks. The domain complexity is caused by the inherent intricacy of airport processes, which comes along with a lot of information to be evaluated, in order to visualize them. Calculating position conflicts for example requires a multitude of information, which at first must be brought together, in order to evaluate them afterwards. This all together makes its development to a huge challenge.

Another problem applies to the software distribution process with the currently used Java Web Start technology. While this technology enables a roll-out independently from the complex processes of the *Software Depot*, it reveals in the present version an incompatibility with the used Spring framework. Since Spring is an important core component of the application, it is not an option to replace it by another technology.

Furthermore, the maintainability and extensibility is quite difficult because the source code testifies to a complex structure and insufficient modularity. There is no strict separation of components like models and views, which makes the application hard to maintain.

In order to overcome these mentioned problems, the existing AirportMap client shall be realised on base of a *Rich Client Platform*, which brings a modular approach on application level. The advantage of having the AirportMap as a rich client is the fact that it provides a rich user interface, which is needed for the multitude of user interactions. Furthermore, Rich Client Platforms promote modular development, in which the source code get segmented into models, views and controller classes.

4.2. Goals and Limitations

Goal of this bachelor thesis is to prototypically redevelop the *AirportMap* client, based on the modular architecture provided by Rich Client Platforms. Thereby, it is not intended to completely migrate the client application, but to analyse and implement basic functionalities, in order to show its feasibility. Therefore, the current source code must be customized to fit the plug-in concept for providing a well-arranged design.

Currently implemented frameworks of the AirportMap client must also be added to the prototype, where it especially shall be considered, how the visualising framework *"JViews Maps"* plays together with the architecture and how the remote communication between client and application server can be realised without using the client container.

Additionally, a new plug-in based update mechanism shall be implemented, in order to achieve a more fine-grained software distribution. To provide customized modules for individual departments, the application also needs to be adjusted as far as possible.

4.3. Requirements

In order to redevelop the AirportMap as a rich client application, several requirements must be distinguished. Thereby, existing requirements of former development stages must be adopted, but also new ones must be fulfilled.

Since the redevelopment in context of this bachelor thesis is implemented as a prototype, not all predefined requirements must be adopted, of course. Important are those, which are needed to obtain basic functionalities. If all basic functions work properly, it can be assumed that functions, which are built on them, will work as well.

In general, software requirements are categorized into functional and non-functional requirements. Functional requirements capture the intended behaviour of a software solution, as it is for example expressed as services and mostly are predefined by the department. Non-functional requirements just describe typical characteristics and attributes such as reliability or scalability of an application.

Within the scope of the AirportMap system, basic functional requirements are specified by particular departments. Before the necessity of redeveloping came up, the department's functional requirements were as follows:

- *Representing the current airport situation*
- *Providing a graphical user interface for interaction*
- *Near-time remote communication*
- *Real-time movement interception*

In return to those, non-functional requirements must cover areas such as performance-related issues, reliability issues, and availability issues. Former non-functional requirements were for example:

- *Availability*
- *Performance*
- *Usability*
- *Security*

Since the current client application underlies the necessity of a completely new redesign, the department brings up new requirements. These requirements are rather of technical nature, because the actual business logic and specifications remain the same. The desired prototypical implementation shall have a new fundamental and modular approach and therefore it is of great importance to specify new requirements, which are listed as follows:

- *Server communication*
- *JViews Maps integration*
- *Update and extension functionality (Deployment)*
- *Modularity*

Since it is necessary to rework the current client-server communication, a new requirement therefore is to find alternative solutions and to implement them. The visualising framework and geographic information system JViews Maps, which has proved itself to be well-suited for this purpose, again must be an important component of the new prototypical application and replacing it is not intended.

Furthermore, the current deployment is very complex and hence another requirement is to implement an own update and extension mechanism. Thereby the current software distribution shall be reviewed.

The last criteria, which comes up newly, is modularity on application level. The new application shall have a modular approach, where individual software components can be added or removed easily at runtime.

All those mentioned aspects must be fulfilled, in order to accept the new prototypical implementation.

5. Solution Approaches

In this chapter, solution approaches to the current AirportMap client application are revealed. Thereby it introduces alternative technologies such as web applications and rich clients, which can be used for replacing the current implementation. Finally, this chapter gives reasons for using rich clients and provides information about generating them.

5.1. Alternative Technologies

Currently, the AirportMap client is realized as a Swing-based desktop application, as it is described in chapter 3. Due to the window emulation mechanism, Swing could lead to loss of performance and hence should be replaced by a more efficient graphics library. But this would not omit the complex roll-out processes of a package management system, nor would it bring a more modular structure. To cover all those issues, an alternative solution must be found.

5.1.1. Web Applications

One alternative solution to the current approach would be a web application. Thereby, the entire roll-out process gets omitted completely, because the client application is deployed in a special container at the server-side. This container gives communications and multithreading support, life cycle management, and declarative security to the web application, so that developers can be totally concentrated on the business logic [BSB08, p. 58].

Web applications require only a web browser, which is already in place on almost every computer. Unlike traditional client-server applications such as the AirportMap, no further installation of software is necessary, except for certain

browser plug-ins. Web applications in general achieve a high degree of platform independence and mobility, because browsers are available for almost every operating system and offer the possibility to access the corresponding server from every workstation in the network. This mobility allows deployers to bring in the most actual version, whenever wanted, because the whole application is hosted on server-side. Thereby, the browser, which actually is a so-called *Thin Client*, is focused only on data input and representation. As many as possible data are transmitted from the back-end server system to the client, so that it just displays the content. In general, the additional task of interacting with the user is done by showing input masks or forms and the whole business logic is done by the server system. This architecture showed to be quite reliable in the past and one of its major advantages is that developers can focus their work completely on the server.

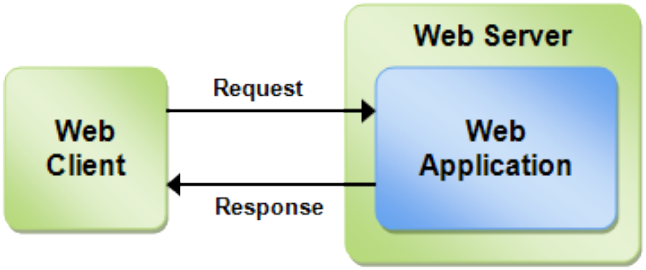

Figure 5.1.: Synchronous Request-Reply Pattern

Since web applications underlie a synchronous *request-reply* pattern, the performance would not be as good as the present solution. As Figure 5.1 illustrates, a client sends a request message to a replier system, such as a web application deployed on a Servlet container, which receives and processes the request and finally returns the results in response. Additionally to this, web applications communicate via HTTP, which is a stateless protocol. This means that several requests basically get handled as transactions, which are independent of each other. In particular, requests are treated without reference to previous requests, which means that current information is lost and hence must be transferred constantly anew. Since the majority of business logic is executed on the server side, this is not a significant issue.

Furthermore, the source code can also be structured into models, views and controller classes, which brings modularity on code level. This design pattern is

also known as the *Model-View-Controller* principle (MVC). Aim of this pattern is to provide a flexible program design, which simplifies a subsequent change or extension of software components and thereby offers reusability. By having the whole application bound at one single point, modularity on application level can be ignored.

Because web browsers would take on the responsibility in representing the actual AirportMap client, the included user interface functionality therefore is also correspondingly low. To fill this gap, web applications can be equipped for example with *Flash*[1] technology, which enables creating interactive and multimedia-based web contents. Since this approach was made famous by multimedia portals like *youtube.com*, the number of distributed Flash plug-ins increased enormously and hence there is a huge community and user group. Flash for example could take on the role of providing user interface functionalities such as customized buttons or drag-and-drop.

Developing web applications provides a broad variety of frameworks - in no other typology, the developers can choose between a wide range of technologies. An alternative to user interface representation by Flash could be realized with JavaScript. This technology has amongst others the advantage that its interpreter is already included in most browsers.

The biggest challenge of migrating AirportMap to web applications is the fact that windows of the included Geographic Information System must be displayed. This can be realized by both technologies. A famous representative of such applications is *Google Maps*[2], whose front-end has been developed exclusively by JavaScript technology.

In summary, the combination of web applications together with Flash or JavaScript technology would be an alternative solution to the existing AirportMap client. Its performance in fact is directly worse than the current solution, but it therefore offers mobility by being accessible from any workstation. Realizing the AirportMap as web application in fact is possible but requires much efforts to get it executable in the known way.

[1] http://www.adobe.com/products/flash
[2] http://maps.google.com

5.1.2. Rich Clients

One disadvantage of web applications is the fact that they are unavailable if their server is offline. Furthermore, user interactivity is only realizable via complex frameworks and technologies, whereby browser differences must be considered and brought into development.

Required real-time graphic representation is hard to implement and the probably biggest decision against using web applications is their medium of communication. Since HTTP is the underlying foundation of every web-based product, individual clients could not be explicitly addressed. All needed information must be provided in the current connection and hence mechanisms like message queues and topics, which are an important prerequisite for the AirportMap, can not be implemented easily. To meet all required criteria, another approach must be found.

A more effective alternative to web applications, which in fact have a thin client approach, are so-called *Rich Clients*. Rich clients are not browser-based client applications and can, additionally to data representation, process operations autonomously of corresponding servers. Thereby, it is irrelevant if a rich client is just a purely local or client-server application. The business logic mostly happens completely on the client-side and only required tasks will be performed, in order to relieve the back-end server system as far as possible. The data that has been processed does not need to be stored directly on a server system. In a rich client, this data can also be stored partially or completely in the client application.

The main characteristic of a rich client is its rich user interface representation and interaction, which is also a basic requirement. Rich clients are mostly developed using a *Rich Client Platform* and designed for providing typical desktop application functions, which can be extended by optional and individual features, such as update mechanisms or automatic server synchronisation. They also offer a rich window-, editor- and view management.

Rich clients support a high-quality end-user experience for a particular domain by providing a rich native user interface (UI) as well as high-speed local processing. Rich UI's support native desktop metaphors such as drag-and-drop, system clipboard, navigation, and customization. When done well, a rich client

is almost transparent between end-user and their work - fostering focus on the work and not the system. The term *rich client* was used to differentiate such clients from terminal client applications, or simple clients, which they have replaced. [ML08, p. 3]

Considering all named issues, rich clients are used all over the place, where applications with fast reactions and a good performance are needed. They provide a good platform integration but also have a higher amount of maintenance in general. Therefore, rich clients make a good alternative to the current AirportMap solution.

5.2. Rich Client Development

Web applications run centrally, and therefore offer advantages in roll-out, management and maintenance. On the client-side, only a browser is needed for accessing the server, which increases mobility. But redeveloping the AirportMap as web application is not applicable since curtailments of usability and visualization capabilities must be made, which can not be accepted. Every interaction establishes a new request to the server, which causes loss of performance, which is certainly needed for real-time processing, as aircraft and their positions are constantly refreshed in real-time. Furthermore, rich user interfaces are complex to realize and current technologies can not be adopted entirely. Besides complexity, web applications can only provide modularity on code level, which does not meet the requirements.

Therefore, the decision fell on rich clients, which bring native and rich user interfaces, as well as modularity on application level. Furthermore, they give an impression of using classic desktop applications with native UI and allow to adopt current frameworks and technologies.

Rich clients are mostly developed using a *Rich Client Platform* (RCP), which allows developers to focus on their core function: the development of business and application logic. The developer does not need to explicitly implement basic widgets, since they are managed and provided by the rich client platform and can be build together like a construction kit. By using such RCPs, a robust

client application with native UI can be developed quickly. Those rich clients can be extended at any time and consist of a multitude of modules, the so-called plug-ins.

To finally create rich clients, several platforms exists, which can be used for generating powerful and modular rich clients, as there are for example:

1. *Spring RCP*
2. *NetBeans RCP*
3. *Eclipse RCP*

Since the Eclipse development environment and other products based on it are already known and widely used at Frankfurt Airport, it is supposed to use its platform for rich client development. For this reason, the other platforms are not dwelt on any further.

Eclipse in general is a programming tool for developing software of different kinds. Primarily, Eclipse was thought to serve as an *Integrated Development Environment* (IDE), but today it is used also for other tasks due to its extensibility provided by the underlying OSGi framework implementation *Equinox*. There is a multitude of *Plug-ins* that can be used to extend the originally platform, which are mostly open-source. That is why the development of Eclipse happens so rapidly. Eclipse in addition also provides its own environment for developing rich clients: the *Rich Client Platform* (RCP).

Furthermore, Eclipse RCP has a huge community and is more than adequate as a starting point for creating complex rich clients, as well-known implementations such as the *IBM Lotus Workplace* show [ML08, p. 9]. By this reasons, Eclipse can be used eminently for rich client development.

6. Eclipse RCP

In the last few years, the Eclipse development environment has gained great popularity and reputation. Therefore, Eclipse would be an ideal occasion to develop rich client applications, as illuminated in chapter 5. By this reasons, this chapter introduces the basic knowledge about the *Eclipse Rich Client Platform* and outlines its usage for rich client development, before the concrete acquisition to RCP is handled in subsequent chapters. Furthermore, this chapter covers technical details of the modular components as well as the plug-in architecture in following sections.

6.1. Eclipse as Rich Client Platform

Eclipse is much more than just being an extensible development environment, like assumed by the general population. It also has established itself to one of the most famous platforms for developing rich client applications and shows imposingly that Java based clients can also be effective by all meanings.

Java based clients have always had the reputation of being slow and inefficient. This was due to the fact that graphical elements were mostly provided by the *Abstract Window Toolkit* (AWT), the first Java based graphics library. This however does not offer enough widgets for rich user interfaces. To solve this problem and to increase further platform independence, Java Swing was developed. Swing in fact provides more widgets and a better look-and-feel, but is quite inert, because its components get emulated and rendered by the Java Runtime Environment.

Therefore, the most Java-based client applications were unsuccessful, but this deficiency changed basically by using Eclipse and its underlying graphics library, which is known as the *Standard Widget Toolkit* (SWT). Since SWT uses

native widgets of the operating system, it is of good performance and offers a multitude of graphical elements.

Together with Eclipse 3.0, the *Rich Client Platform* was introduced to eliminate those aforementioned disadvantages. It is a collection of basic plug-ins to realise fundamental concepts of Eclipse-based applications and allows it to use its standard user interface concepts directly for own rich client developments. The developer team at its official web site describes it as follows:

> *"Eclipse RCP is a platform for building and deploying rich client applications. It includes Equinox, a component framework based on the OSGi standard, the ability to deploy native GUI applications to a variety of desktop operating systems, such as Windows, Linux and Mac OSX and an integrated update mechanism for deploying desktop applications from a central server."* [Ecl10]

In summary, Eclipse can be used outstandingly as a foundation for Java-based rich clients. Due to its underlying graphics library, the application's user interface is of good performance and can keep up well with other client technologies.

6.2. Bundles

The great flexibility of Eclipse RCP lies in its modular architecture and the possibility to change software components at runtime, without compiling and starting the whole application again. Since Eclipse was released in version 3.0, it is totally based on the modular approach of the OSGi service platform. The hereby used Java classes and resources are combined to so-called *"Bundles"*. In context of Eclipse RCP, bundles and plug-ins have both the same meaning and underlie a strict component specification. This is justified by the fact that present Eclipse components are plug-ins, as well as OSGi bundles.
According to [HC01], a component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. By this reason, bundles (or plug-ins) are categorized to this component definition.

Equinox, the basic OSGi implementation of the Eclipse platform, provides a mechanism for dynamic class loading by managing bundles, as well as their dependent components due to their so-called *Bundle Manifest*. The Bundle Manifest is a certain file (*manifest.mf*) in the bundle package, which contains descriptive meta information about the bundle itself, but also about their dependencies. Required bundles then successively are loaded and managed by the OSGi implementation on base of their unique identifier name. This is a huge advantage over normal desktop applications, because this mechanism works classpath comprehensive and today, every Eclipse plug-in contains such a manifest file.

The individual bundles furthermore differ in their designated usage and hence can be divided into several categories:

1. *Code Bundles*
2. *Library Bundles*
3. *Resource Bundles*
4. *Language Bundles*

Code bundles in general contain code of business logic, but also resources such as pictures or property files. Library bundles contain exclusively code of third party libraries and in contrast to those, resource bundles do not contain any source code, but only configurations, build- or property files. A language bundle is a special type of resource bundle, which contains property files for the language conversion of code bundles.

Those differentiations help to fit own rich client applications in a reasonable way and to allow an easier replacement of individual software components.

6.3. Extension Registry

One of the most important differences to classic desktop applications is the fact that the Eclipse Rich Client Platform and its underlying OSGi implementation together provide a mechanism for defining and running separate components. The Eclipse runtime thereby adds a mechanism for declaring relationships between individual plug-ins: the *Extension Registry*.

The Extension Registry actually is a component repository, which is accessible and modifiable by individual plug-ins. Plug-ins can open themselves for extension or configuration by declaring a so-called *Extension Point* [ML08]. Extension points are certain defined spots that are provided to other plug-ins for extending their functionalities. *Extensions* then subsequently are able to connect to those well-formed sockets, in order to implement new content or functions to them.

Though the use of this *Extension Point* mechanism, user interface elements like editors can be added declaratively and the internal architecture makes it possible that plug-ins are only loaded when they are needed - this will avoid increasing code bulk and long startup times.

This mechanism is an elegant and easy to use component technology, which allows extending a system at predefined locations, without customizing the whole system again. This can happen either on graphical, but also on beneath layers. Thereby, the *plugin.xml* defines, which extensions as well as its according extension points are implemented in the corresponding bundle. This furthermore allows loose coupling among each others. [HS09]

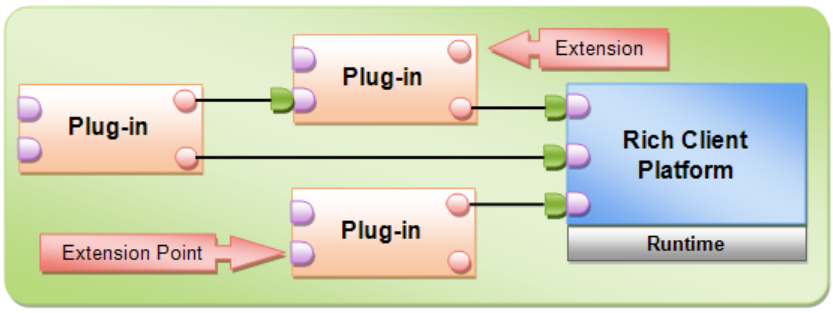

Figure 6.1.: Extensions and Extension Points

As Figure 6.1 approximately shows, the extension mechanism can be compared with connectors that get plugged into certain sockets. A plug-in thereby provides one or more sockets (extension points), which subsequently can be filled by several connectors (extensions). Thereby, dependencies are inverted, since the consuming extension point does not know anything about its extender. Only the extension has a proper knowledge of the used extensions points.

According to [Lip07], plug-ins that provide an extension, must have a *required-bundle* dependency. Conversely, this means that the providers of the extension point does not have any dependencies to the plug-in, which docks at these sockets. This is a big advantage over monolithic systems, as for example the application's main window does not know which menu items should appear in its menu bar. Only the provider of these items specifies, where to show it.

6.4. Architecture

The Eclipse Rich Client Platform is an environment, which offers a common infrastructure for developing rich client applications. It provides a minimal set of bundles, which can be extended individually by own application modules. An RCP application is a composition of different plug-ins and the runtime, on which they are executed.

Figure 6.2 approximately shows, how the Eclipse Platform and RCP belong together. As it can be seen, the Eclipse Platform provides the Rich Client Platform that consists of a minimal collection of certain fundamental components, as for example the Runtime and OSGi implementation.

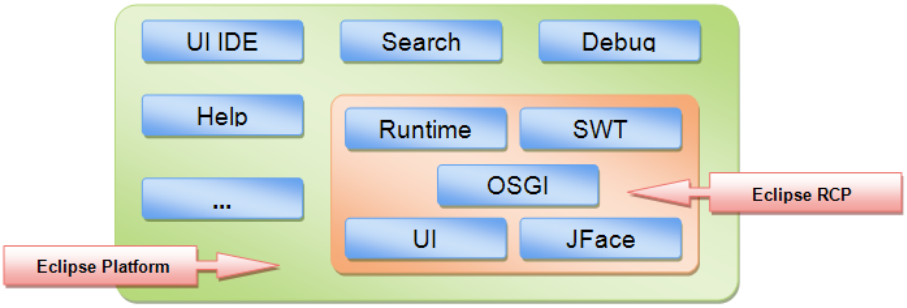

Figure 6.2.: The Eclipse Plug-in Architecture

Since Eclipse was released in version 3.0, it totally embraces the modular approach provided by the OSGi implementation *Eclipse Equinox*. One of the core functionalities of OSGi is the control of the life cycle of individual bundles. This core concept forms the foundation of Eclipse.

The assembly of RCP client applications is essentially determined by the bundle concept of their underlying OSGi implementations. In Eclipse, everything

is a plug-in. Even the OSGi framework and the runtime show up as plug-ins. All plug-ins interact via the extension registry and public API classes. These facilities are available to all plug-ins and there are no secret back doors or exclusive interfaces. [ML08, p. 25]

Furthermore, Eclipse RCP again is a small set of bundles, located on top of a Java Runtime Environment and provides an infrastructure, similarly to native Operating Systems, which are waiting for applications to be installed. Even the classic Eclipse IDE platform, which is used as a development environment, is just a highly functional RCP application. The general frame of Eclipse RCP and the developed clients based on it, is the same - they are both just sets of plug-ins that make up a coherent whole. These terms of consistency and uniformity recur throughout Eclipse. [ML08, p. 14f]

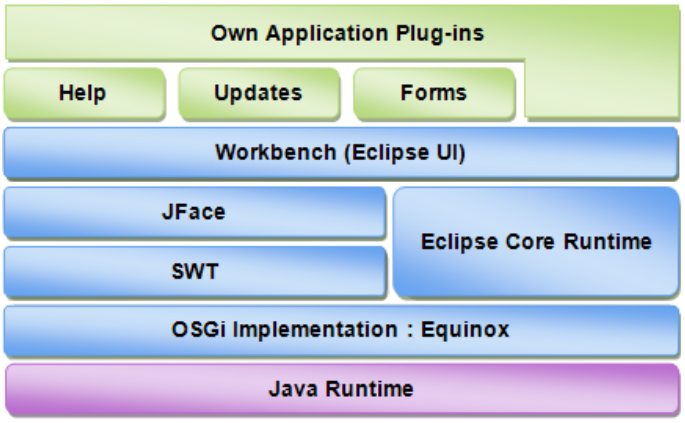

Figure 6.3.: The Eclipse RCP Stack [Ebe09]

The internal details for the Eclipse RCP collection stack is illustrated in Figure 6.3. The basis is formed by the OSGi implementation Eclipse Equinox that uses functionality of the underlying Java runtime. All following components are plug-ins that are managed by it. Those plug-ins form the basis of every Eclipse-based rich client application and can be divided into two main areas: the core runtime environment for providing common rich client functionality and the so-called *Workbench* - a collection of modular user interfaces, which use functions of SWT and JFace. Those components will be explained in advance.

6.4.1. Eclipse Core Runtime

The Eclipse Core Runtime is a very important component of the Rich Client Platform. It provides the common but non-graphical rich client functionality as for example maintaining the bundle's defined dependencies at runtime and therefore is responsible for bootstrapping and initialising the application.

Compiling classic desktop applications is quite straight forward because they are started via a main method and work independently of other classes. From this entry point, individual methods of other instances are called, which for example let Java Swing paint a window with some labels and buttons. Using Eclipse RCP, this simple programming model is not possible any longer, since launching an Eclipse-based application involves many other components.

For recognizing different bundles at runtime, extensions as well as their required extension points are described in the plug-in.xml file. Also the bundle manifest configuration of the OSGi service platform is interpreted and resolves the defined bundle dependencies. Thereby, OSGi provides a container, from where those dependencies can be called at runtime.

Concerning only this mechanism, it reveals fundamental differences between RCP and classic desktop applications. This makes its development to a difficult task with a steep learning curve.

Furthermore, there is a difference between plug-in and rich client development. For developing rich client applications, at least following two components must be included:

1. *org.eclipse.ui*
2. *org.eclipse.core.runtime*

Both plug-ins have transitive dependencies to other plug-ins, which are resolved by the OSGi implementation. But if the plug-in **org.eclipse.ui** is missing, graphical components like views or editors can not be located and the application would be non-graphical therefore. If **org.eclipse.core.runtime** is missing admittedly, the whole application would not even start, because the component contains important modules, which are needed to finally run RCP applications. If an application does not contain those two bundles, it is just a normal plug-in.

The AirportMap Core application in fact represents a stand-alone executable environment, which actually provides the real rich client functionality and therefore must include the *Eclipse RCP core plug-ins*. The application thereby contains special classes, which manage and control the workbench's life cycle, as it can be seen on Figure 6.4.

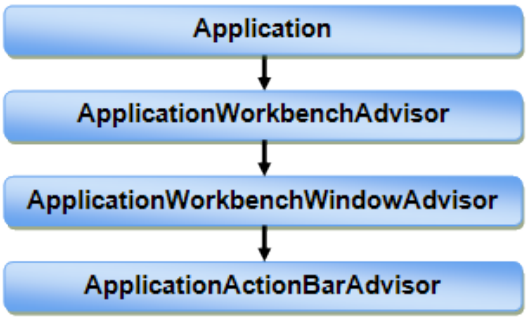

Figure 6.4.: The Initial RCP Classes

The entry point for every rich client application is the class *Application*. In the Eclipse rich client platform, this represents what is the main method for desktop applications. This class creates and runs the workbench and calls the *ApplicationWorkbenchAdvisor*, which is responsible for setting the initial perspective and for window position attributes. It furthermore uses the *ApplicationWorkbenchWindowAdvisor*.
This class handles visual components such as perspectives and status bars by interpreting predefined extensions from the plug-in.xml and invokes the *ApplicationActionBarAdvisor*, which controls the initial actions of the application. From this point, the workspace is created completely and will be displayed.

This circumstance makes the runtime to one of the most important components of the Rich Client Platform. Without this mechanism, rich client applications are not able to start, because it provides the basic rich client functionality.

6.4.2. SWT and JFace

The RCP approach with AWT and Swing for graphical user interfaces did not obtain much acceptance in the Java community. There are two main reasons for this circumstance. On the one side, AWT does not offer many widgets, as it is required for rich client applications. On the other side, also Swing, the advancement of AWT, could not get rid of its bad reputation of being slow. Therefore, both technologies are inappropriate for this purpose. With the emergence of the Eclipse platform, Java based rich clients obtain more and more acceptance.

The Standard Widget Toolkit that is not only used by Eclipse, is a low level graphics library that provides the standard widgets of the underlying Operating System. It offers an efficient and portable access to the user interface of the Operating System, on which it is implemented and exposes it through a consistent and portable Java API.

In principle, it is possible to redevelop Swing or AWT based applications with Eclipse RCP by using a software bridge that allows to embed the desired components directly in SWT. These graphic bridges offer the possibility to display and execute certain Swing or AWT widgets in an own graphics environment, but this is not an optimal solution, because different graphic technologies are mixed together and used simultaneously. Hence it is recommended to use plain SWT technology, in order to obtain a cleaner code and to avoid the aforementioned disadvantages.

JFace complements SWT by additional structure and facilities such as the MVC pattern for common UI notions. It is window-system-independent in its API, as well as implementation, and designed to abstract from SWT. Furthermore, it includes a multitude of user interface components such as different views or image registries. These and other structures, such as actions and editors, together with the Eclipse Core Runtime, form the basis of the Eclipse user interface, which is also known as *"Workbench"*. [ML08, p. 23]

Additionally, the JFace library can also be used in environments, other than Eclipse. The Workbench in particular makes an extensive use of it. JFace provides a lot of functionality out-of-the-box, which are frequently used by developers. There is for example the resource management of the graphical

user interface, the design of wizards or the strict separation between individual components and their involved actions.

Based on this foundation, the Rich Client Platform is able to be configured with further components, such as a help system, an update manager or reporting tools. JFace also allows customized branding of the actual rich client application.

6.4.3. Workbench

Eclipse UI furthermore provides the so-called *"Workbench"* on top of its modular plug-in architecture. As JFace adds structure to SWT, the Workbench adds presentation and coordination to JFace. For the user, the Workbench consists of views and editors, arranged in a particular layout. [ML08]

This can be imagined like the *Eclipse IDE* just without any content: an empty, graphical application that provides the familiar concepts like menu structures and editor views, as it can be seen on figure 6.5.

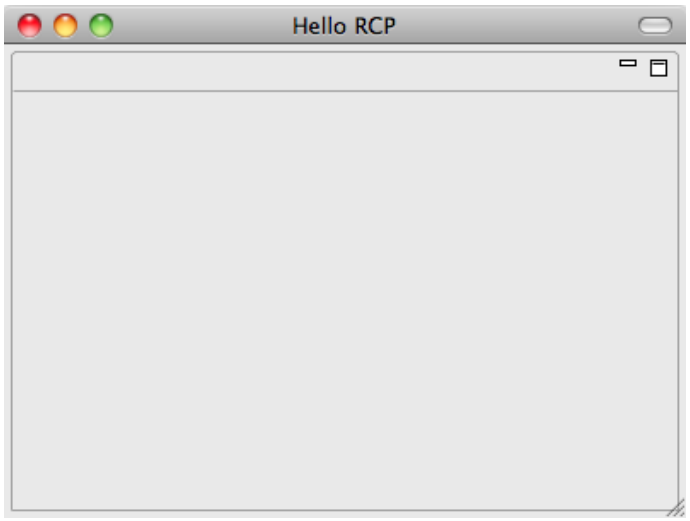

Figure 6.5.: An Empty Workbench

The workbench appears as a collection of different windows and provides further UI elements. The *perspective* is a container for several views and editors including their positions and sizes. Everything that is made visible to the user,

is finally laid out by a certain perspective. A perspective supports a particular set of tasks by providing a restricted set of views and provides action sets as well as shortcuts to other related views or perspectives. Users can switch between perspectives to for example change between developing code, trading stocks or instant messaging. Each of these tasks may have unique layouts and content [ML08]. Therefore, the workbench is a fundamental and important element of the Eclipse Rich Client Platform.

6.5. Eclipse RCP and AirportMap

Since the Eclipse Rich Client Platform has a clear and proven architecture, as well as a large collection of robust components, it can be used outstandingly for redeveloping the Swing-based AirportMap client. Thereby, all important Operating Systems are supported, so that they also can be exported to other platforms. As Eclipse RCP provides a native look-and-feel, the new prototype application will have a more beautiful user interface.

Furthermore, there is a huge number of documentation and literature about Eclipse RCP development, which would support developers in their work. This is, why Eclipse RCP is so attractive for rich client development and finally for redeveloping the AirportMap.

7. Design

This chapter introduces the system design for the required modifications of redeveloping the existing *AirportMap* client application based on Java Swing to Eclipse RCP. The fundamental approaches of both architectures differ strongly from each other, so that many changes in source code and design must be made. Those modifications are discussed in the individual sections.

7.1. Concept

The current *AirportMap* client architecture differs fundamentally from a rich client application and porting it to Eclipse RCP is not a trivial task. This is due to the fact that it comes up with new design concepts that differ strongly from former graphical solutions based on Java Swing technology and classic desktop application development.

Another challenge results from the fact that the current business logic is not very well separated from the presentation as well as authorisation logic. In order to be able to redevelop the AirportMap client in a modular way, the business logic has to be identified, separated and extracted from them.

Furthermore, the integration of the currently used frameworks could also be problematical. The integration and connection of different frameworks together is sometimes challenging, since individual frameworks can be incompatible in combination and may not work as expected any longer. It also has to be considered that they must be able to work in the modular approach, provided by Eclipse RCP.

Another major challenge of redeveloping the AirportMap is its remote communication. Since the current implementation uses a client container, it must be evaluated and replaced, in order to use Eclipse RCP. This may require

code changes on server-side and is one of the most important steps of the redevelopment.

In addition, graphical elements, which are currently realised by Java Swing, must be replaced as far as possible by components of the *Standard Widget Toolkit*. Because SWT is the default and required graphics library of Eclipse based applications, it must be used for creating the graphical user interface. Thereby a solution must be found to implement components, which can not be replaced by SWT.

Furthermore, an own update mechanism shall be integrated, which enables application internal software updates and extensions independently from the complex roll-out routines of the *Software Depot*. Thereby the application at first has to be initially deployed to the user's computers.

To finally redevelop the AirportMap as an Eclipse-based application, all those important subjects must be considered.

7.2. Modularity

Modularity is a basic part of object-oriented programming and can be described as functional or logical components of an application that belong together. Those parts are grouped in conjunction and can be used by other modules as one unit that brings, amongst others, the important advantage of reuse and extensibility of individual modules.

According to [Mey99, p.40], a design method, which wants to obtain modularity, should satisfy the following five fundamental requirements:

1. *Decomposability*
2. *Composability*
3. *Understandability*
4. *Continuity*
5. *Protection*

Thereby, decomposability in general means the possibility to divide one single complex unit into several, coherent parts. Among specialists, this procedure is also known as *division of labour*. The opposite to this requirement is

composability and addresses the reverse process: extracting existing software components from the context for which they were originally designed, in order to use them again in different contexts. Both requirements are amongst others provided by the OSGi Service Platform, as it allows to bundle individual components to one single application unit and vice-versa. Unlike the normal Java Platform, OSGi provides a bundle oriented approach, in which the individual components can not obtain access to implementation classes of other bundles, as it is widely common in Java applications. Bundles can provide access to their packages by exporting them for others. This procedure allows to hide information and to make only chosen interfaces (contracts) public for other bundles. OSGi thereby guarantees that only exported packages can be seen by other bundles and that the actual implementations are hidden. Using the Java Platform, this in fact is not possible.

Another important requirement is the modular understandability. A design method obtains understandability, if it creates software components in a human-readable way, without having to know other components in the same environment. If for example bundle A can be understood completely without knowledge of bundle B, then bundle A supplies modular understandability. This immensely simplifies the basic understanding of certain components.

A software product yields modular continuity, if small changes in the specification also results in a small number of modification in the software modules. OSGi also provides this requirement innately. Every bundle is responsible for a certain number of functions and therefore only affected bundles must be customized in order to change functionality. Since only exported packages are visible to others, the impact brought by changes is minimized.

The last important requirement of modularity is the modular protection. As described in [Mey99, p.45], a design method satisfies modular protection if it yields architectures in which the effect of an abnormal condition occurring at runtime in a module will remain confined to that module, or at worst will only propagate to a few adjacent modules.

All five criteria are provided by both, the Java Platform as well as the OSGi Service Platform and hence Eclipse RCP promotes and requires modular programming. Thereby OSGi offers better modularity, as it would be possible with

plain Java development and so the question arises in how deep the modularisation must be done, to obtain a good design for redeveloping the AirportMap client application.

At present, the AirportMap client is not developed very modular and its source code is wired strongly in many parts. The current implementation furthermore injures the required modular criteria as for example it is developed in a monolithic and strongly coupled way, which contradicts the first four requirements. The business, authentication and presentation logic are not clearly separated from each other, what contradicts the decomposability requirement. In addition to this, it furthermore does not provide clearly defined interfaces, as it is wished for object oriented software development. Therefore, a fundamental rework of the AirportMap architecture as well as its corresponding software components must be done in order to obtain full modularity, as it is assumed by Eclipse-based rich client applications.

7.3. Framework Integration

In classic desktop applications, frameworks are integrated by adding their library file to the application's class path. The class path can be imagined as a place, where the Java class loader looks up for classes and other resource files, when programs are compiled or executed. Thereby, class loaders are used to retrieve Java classes from a physical location, such as hard disks and provide them to the Java Runtime Environment.

Eclipse RCP follows a different approach. Due to its underlying OSGi Service Platform and bundle architecture, it uses an own dynamic class loading mechanism for retrieving bundles and their included Java classes. This takes on significant changes on the technical level, because bundles are handled independently of each other and hence have no common class path. Instead, every bundle defines its own class path and class loader, which brings the new opportunity to include different versions of the same framework in one single application. Systems are composited by a set of plug-ins, which are interconnected at certain well-formed extension points and explicitly describe their dependencies on OSGi level. Thereby, certain version numbers of individual dependencies can be specified.

```
1  Bundle bundle = Platform.getBundle("de.example.bundle");
2  Class myClass = bundle.loadClass("de.example.classes.MyClass");
```

Listing 7.1: Manual Class Loading

But Eclipse RCP also provides the possibility to reference certain bundles and classes manually, as listing 7.1 shows. To manually load them, the Platform must be called to return a specific bundle by searching for its unique identifier name. In this example, the bundle's identifier is "de.example.bundle". Afterwards it can be requested to return a certain class, which can be used subsequently.

According to [Rei09, p.41 ff], there are different solutions of integrating frameworks in context of Eclipse RCP. In the first variant, the required libraries are integrated in the appropriate bundle. They are thus part of it and can not be managed separately, what abandons the strict separation of components. The problem using this approach is the fact that the libraries have to be copied manually to all bundles, but this in turn creates unnecessary redundancy. Via the bundle manifest, the included libraries get added to the bundle's class path so they can be used within the bundle and others can not gain access to them. Unfortunately, in case of an library update, all bundles containing the libraries have to be maintained individually. Therefore, this approach is not recommended.

Another variant is to use *Library Bundles*, which allow to extend a bundle's existing class path for other bundles. Thereby, the framework libraries are coupled within an own bundle that provides the required classes and which exports their packages. If other bundles want to access them, they just can import those packages individually. Unfortunately, this approach is only unidirectional. To finally enable bidirectional communication, the so-called *Buddy Classloading* can be applied. Thereby, the library bundle needs to add the entry *"Eclipse-BuddyPolicy: registered"* to its bundle manifest. Other bundles then can register themselves as *buddies* to the library bundle by including the *"Eclipse- RegisterBuddy"* entry in their manifest file. [Rei09, p.47]

In certain cases, there is also a third variant. To use this solution, the framework must be already available as an OSGi bundle. This bundle then can be copied directly to the *Target Platform*, which in fact is the set of plug-ins used for developing RCP applications. By default, this target is set to the whole plug-in collection of the Eclipse IDE and since the default target is used, the bundles can also be integrated by using the internal update manager of Eclipse. Adding bundles to the target is the most recommended way, because the framework thereby can be managed completely independent from any source bundles.

Based on this fundamental difference between classic desktop applications and Eclipse-based rich clients, a suitable solution must be chosen for redeveloping the AirportMap client. The traditional way in adding libraries to the applications class path can not be applied any longer and a structure must be considered, of how the frameworks get integrated, since several plug-ins may need to gain access to them.

Another problem of integrating the used frameworks is the fact that the current AirportMap client implementation is completely based on Swing components. For migrating it to Eclipse RCP, all Swing components must be replaced by their corresponding SWT widgets. For example, the widget *javax.swing.JButton* can be completely put in place by the SWT pendant *org.eclipse.swt.widgets.Button*, which in fact is not a difficult task. The problem thereby are Swing components, which can not be replaced that easy.

The JViews Maps framework for example provides only Swing-based components, which can not be replaced, because it is a commercial and hence closed-source framework. Therefore, graphic bridges must be used to enable their adoption. The only opportunity of using those widgets without Swing, is to develop them completely in SWT, which is impossible because the source code is not public.

For integrating Swing in Eclipse RCP, there are several graphic bridges available. On the one hand, Eclipse itself includes the *SWT_AWT* class, which provides a bridge between SWT and AWT, so that it is possible to embed AWT components in SWT and vice versa. Because Swing is based on AWT, this bridge can be used for embedding the JViews Maps elements.

On the other hand, JViews Maps itself provides such a graphic bridge for displaying its own components. This bridge is known under the class name *IlvSwingControl*.

Both bridges can be potentially used for migration purposes, since they both offer a mechanism to embed Swing components in Eclipse RCP. All other standard widgets are replaced by their corresponding SWT pendants.

7.4. Remote Communication

The current AirportMap implementation has a client-server architecture and therefore consists of a front- and back-end part. The Client application need a communication protocol to interact with the server and for the remote communication two common technologies are currently used:

1. *Java Message Service*
2. *Enterprise JavaBeans*

Since both technologies are part of the *Java Platform, Enterprise Edition*, the AirportMap server application is running within an application server, which provides the application with Java EE functionality and services.

In the AirportMap, a message queue is used to retrieve information about how long individual users were online. Further, a topic is used to transmit movement and airport-specific data like off-block and arrival times. This service in fact offers loosely coupled communication, but for invoking server methods, this technology is not appropriate.

For this reason, the *Enterprise JavaBeans* technology is used to provide a login procedure and data initialisation. Java EE compliant servers provide an EJB container that holds amongst others Sessions Beans, which offer their services by local and remote interfaces. Those remote interfaces can be accessed by JNDI look-ups to get their stub references. This enables clients to call remote functions that run in a different Java Runtime Environment and works in a synchronous request-reply manner. Thereby, the EJB container is used as network layer between server and client.

Furthermore, the container is an element of an Java EE compliant application server, as it can be seen on Figure 7.1.

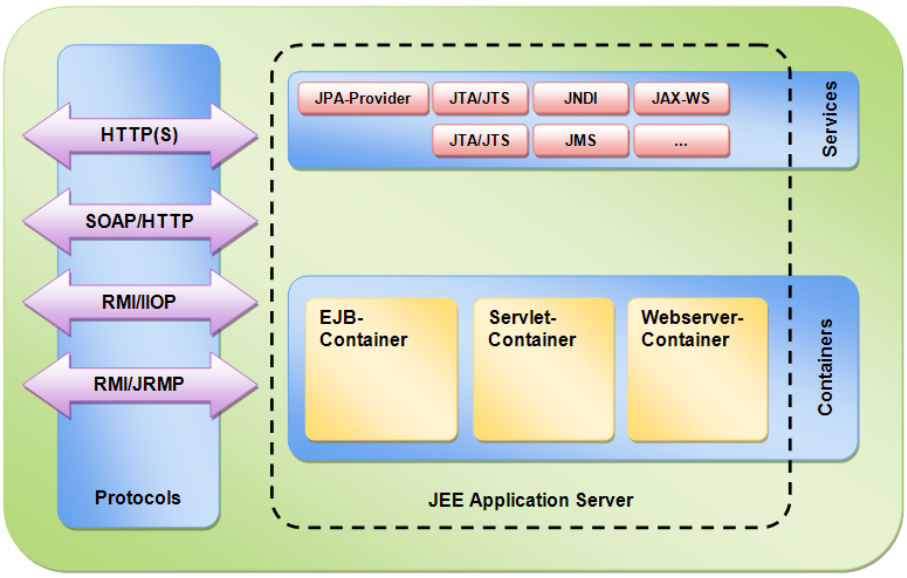

Figure 7.1.: Java EE Compliant Application Server

First of all, an application server is a container, which again contains several other containers and services. It defines the runtime environment, in which Java based software artefacts such as *EAR*[1] are deployed, administrated and executed. The EJB-Container represents the runtime environment for EJB components and therefore contains their instances. It also allows and monitors their execution. The therefore required functions are either provided by itself or demanded by its corresponding application server. When remote communication between client and EJB-Container happens, the client binds itself to the *JNDI* naming service via *Remote Method Invocation*, in order to obtain reference to a specific EJB component from it [IHH07].
This can either be done by managing this connection manually or as it is realized in the current Airportmap client, by the use of a runtime environment that performs this task: *The Java EE client container*.

[1]Enterprise Application Archive - file format for saving enterprise applications

The client container is a reduced counterpart of the server's EJB-Container and distributed together with the client application. It provides the runtime environment, in which the application is running and offers services and features like a simple JNDI lookup by picking up initial context properties from the EJB-Container or the usage of Java EE security, which includes authentication and server-specific functions. At present, the AirportMap requires just a minimum of the those provided features and services. Only the connection to the server's container is needed, in order to invoke the remote functions for authentication and data initialisation.

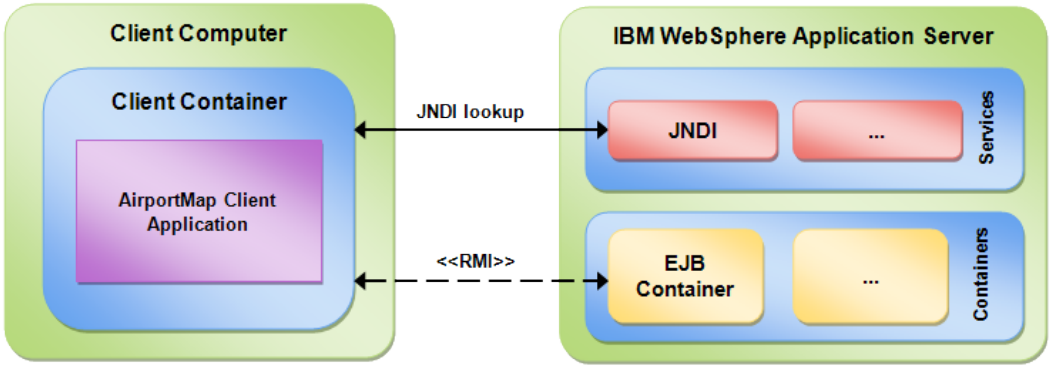

Figure 7.2.: Java EE Client Container

As figure 7.2 shows, the client container binds itself on the *JNDI* name service provided by the application server. It looks up for the qualified JNDI name of the server's EJB-Container, in order to get a reference to it. In the figure, this logical connection is indicated as a dashed arrow. Furthermore, JNDI lookup and EJB referencing is automatically managed by the client container. This has the advantage that physical resource names can be changed on the application server without readjustment of the client application, because their logical name needs not to be modified and hence no client source code must be customized explicitly.

Unfortunately, the client container turned out to be quite problematic in development, deployment and maintenance. This is because it requires a special Java runtime environment provided by IBM, which also has to be included for distribution, instead of using Sun's standard JRE. This circumstance blows up the total application size to approximately 180 Megabytes, because the IBM runtime alone has a current size of about 150 MB.

Although, it is possible to develop rich client applications using those client containers, it is not recommended due to its complexity and known issues. Further, in the past, a lot of problems and incompatibilities came up in connection with the usage of the client container in the AirportMap.

In the last development iteration for example, the IBM Java runtime environment was upgraded from version 1.4 to 1.5. On the one hand new features like enumerations and generics were introduced and on the other hand, this version is pretended by the current development environment. Despite of the improvements, this upgrade turned out to be problematic because incompatibilities with the existing frameworks were introduced. For example, the currently used release of the Spring Framework (2.5.6) revealed to be incompatible with the client container and due to its complexity[2], this issue could not be resolved until today. Furthermore, its deployment is tricky and complex, as the client container has to be included into the deployment process.

Based on the already existing problems with the AirportMap client and the client container and with the potential risk in mind of introducing further incompatibilities with the usage of Eclipse RCP, the department decided to redevelop the AirportMap client completely without it. Therefore, an alternative solution has to be found to access resources and services of the application server. At present, those services consist of accessing to the EJB session facade for authentication and initialisation, which can be replaced by *Java Servlets*, for example.

The second communication way offered by the client container is realised by the JMS messaging infrastructure, which can be alternatively resolved by using the messaging libraries directly. Thereby, the Spring Framework can be used, which amongst others provides special connection factories and listener containers that simplify the JMS communication.

[2]The complexity lies in the incompatibility of the client container and the classloading mechanism of the OSGi Service Platform of Rational Application Developer

7.5. Software Distribution

In contrast to web applications, which do not have to be rolled-out explicitly, a rich client is not usually pre-installed on computers and therefore requires distribution. The process of rolling-out software on different computers is generally known as *Software Distribution*. This term implies especially the installation of new software, but also the update of existing components and the adding of new functionalities. Eclipse RCP thereby allows new possibilities regarding to roll-out and update functionalities, which are covered in advance.

7.5.1. Software Updates

Every application becomes outdated at some point in time. In order to counteract, a mechanism is required, which brings applications to newer versions. This mechanism is called *Update*. Updates mostly fix bugs or include program changes and can only be performed if an installed version is available.

Applications, which are based on Eclipse RCP, provide possibilities to update themselves. In Eclipse RCP, so-called *features* represent updatable units. Features collect sets of plug-ins that go together to form some logical unit of function. They have a very simple structure because they are just a list of plug-ins and other features [ML08]. Furthermore, they contain a version number for unique identification. This concept also allows adding new software components that are not yet installed in the existing application.

To avoid time-consuming and complex service request of the order management, it is useful to have an application that is able to update itself. Due to settled operations, the average period for handling service requests at Frankfurt Airport is at about 10 days, which in fact is too long. Having an own update mechanism from where deployers can publish the most actual release instantly, would dispose those bureaucratic steps.

A simple way to perform update and installations with Eclipse-based applications would be placing the selected features into the appropriate plug-in directory. With the help of this folder, the underlying OSGi framework looks for the included software components. If a new plug-in gets copied or removed from that location, changes can be found immediately. Although this

mechanism is quite simple and easy to understand, it is not very user-friendly, because plug-ins and their dependencies must be known, obtained and copied to that location.

Thereby users can make many mistakes as for example to forget important plug-ins. If required plug-ins are deleted, the whole application may not start any longer. In addition, updates and new functionalities must be held as features, which can quickly confuse the user while using the update routine, as they are just lists of individual plug-ins. So it would be nice and of great usability to have an own installation routine, which is responsible for the plug-in management of the AirportMap client.

Implementing update and install managers within Eclipse-based applications in fact is not a difficult task. In principle, those components have to search for available features at certain locations, such as remote file servers and need to copy them together with their corresponding plug-ins to the appropriate plug-in and feature directories of the rich client application.

```
1  BundleContext.installBundle();
2  Bundle.update();
3  Bundle.uninstall();
```

Listing 7.2: Bundle Management API

As listing 7.2 shows, the OSGi framework itself offers an API for installing, updating and removing bundles, which can be used for the implementation of effective and useful update managers. As an alternative therefore, the Eclipse internal update mechanism can be integrated to the prototype applications.

7.5.2. Initial Distribution

Applications, which are based on Eclipse RCP, bring advantages and improvements over classic desktop applications, since they can use internal mechanisms to update themselves. But therefore, they at first have to be distributed initially.

A classic Java application is normally distributed containing batch scripts that are used to launch the application. But Eclipse-based applications bring in their own launching functionality by providing an executable file, which is customized for the particular operating systems.

This circumstance allows Eclipse-based applications to be rolled-out by a package management tool such as the *Software Depot*. This process in fact is very complex and time-consuming, but due to the independent update functionality provided by Eclipse RCP, the application can be responsible for future updates and extensions by its own. Software Depot is only used to get the basic application onto the client computers and it only needs to be rolled out newly in the rarest situations, as for example when serious changes must be done. Complex roll-out processes are therefore avoided as far as possible.

Furthermore, an Eclipse-based AirportMap can also be rolled-out by the currently used Java Web Start technology. Thereby, the integration of a Java runtime of course gets avoided, because Java Web Start must be already installed on the target system, which assumes an already installed Java Runtime Environment. With some limitations and new settings in the configuration file of Java Web Start, also RCP applications could be distributed in this way.

7.6. Review

In summary, the current AirportMap client can be completely redesigned as an Eclipse-based application. To reach modularity, the source code must be refactored, so that business, authentication and presentation logic get separated from each other, which can be performed by the new design concepts of Eclipse RCP. Existing frameworks can be integrated to Eclipse RCP by including them to bundles or by adding them to the target platform. Furthermore, almost all existing widgets can be replaced by their corresponding SWT pendants and the use of a graphic bridge. The remote communication can be realized without the client container by implementing Java Servlets, instead of EJB technology for example. The deployment can be done by the already used Java Web Start technology or by a package management software such as the Software Depot. Due to the plug-in architecture of Eclipse RCP, a mechanism can be implemented, which is responsible for updating and inserting new features.

All those design choices remove the last obstacles for the implementation of an Eclipse-based AirportMap client prototype.

8. Implementation

This chapter introduces the prototypical implementation of the AirportMap client application. Thereby, all design requirements are put into practice. In the beginning, the chapter illustrates the strategies, which were done for obtaining remote communication without a client container. Afterwards, it explains how the required technologies are integrated into Eclipse RCP and what thereby should be considered. The third section covers the important strategies of how the application components must be structured, in order to reach a good design. The final section then covers the update functionality and distribution of the prototype, where it especially is focused on the update mechanism.

8.1. Removing the Client Container

Developing a rich client application with Eclipse RCP alone has a steep learning curve. Therefore it is advisable to develop completely without the complex and problematic Java EE client container. To relieve the client container and to gain integration into the Java EE environment, the decision fell on using web services.

By using a web service, requests can be sent to the server via *HTTP*. The server receives those requests, processes them and optionally returns them in response. This among others has the advantage that the server can also be accessed by other technologies than only Java applications, which makes it more interoperable among other technologies. But this in fact is not the primary focus, since only an alternate solution to the client container is needed. Another topic that argues for web services is that they are build easier than EJB and therefore are better to maintain.

In general, web services are implemented using the Java Servlet technology and are connected to the clients via SOAP-based remote procedure calls. This approach requires to specify a WSDL[1] file that is used to compile the stubs on the client-side. By using these stubs, the actual services can be addressed and invoked. This requires the usage of further frameworks, which would make the application more complex.

Another way of realising web services is to apply an architectural style, which is known as *Representational State Transfer* (REST). REST embraces a stateless client-server architecture in which the entities of a system are viewed as resources and hence can be identified by their URL's. This offers interoperability, since only the widespread HTTP protocol is needed [Bay02]. In context of the prototypical AirportMap development, those so-called *RESTful Web Services* are used for representing an alternative solution to the current remote EJB communication.

The Hypertext Transfer Protocol is widely used for transporting data over a network, such as the internet. Its main goal is to load pages from the Web into a browser. Certainly, this transfer protocol is quite more, since it can also be used as an application protocol. Its HTTP methods thereby are used to describe the necessary create, read, update, and delete (CRUD) actions. Even SQL, the query language for data bases only applies those four basic actions. By this reason, the HTTP methods GET, PUT, POST and DELETE can be used to perform actions at the server-side. Additionally, the Hypertext Transfer Protocol assigns several status codes that represent status information such as "Bad Request" or "Unauthorized" [Bay02, p.3]. Because they have different meanings, they can be applied for user authentication for example.

In REST, objects and events are managed as resources, which are identified by a certain URL. A resource can be essentially any coherent and meaningful concept that may be addressed by for example a database identifier. Since RESTful web services are only used for replacing the current EJB communication, the objects provided by the server are not needed to become such

[1] Web Service Description Language

resources. Only services have to be defined, which in fact allow a remote communication to a Java Servlet that in turn delegates to the local EJB.

To finally reach that goal, the AirportMap server needs a Java Servlet that gets mapped to a specified address, by applying a procedure, which is known as *Servlet Mapping*. Servlet mapping instructs the web container of which Servlet should be invoked for a certain URL. It maps URL-specific patterns to Servlets and when there is a request from a client, the servlet container decides to which application it should forward to. This is the first step, which has to be done for realizing web services in the AirportMap environment.

Furthermore, RESTful web services allow to explicitly specify the return type of every response, which can be of any mime-type. Mime-types in general define the structure of internet messages such as text, videos, or byte-streams. Since the focus in context of the prototype application was more on removing the client container than on reaching interoperability, the return type is set to byte-stream, which allows to return serialized Java objects. By using the de-serializing method, the original Java Object can be restored from the available byte array.

Now, there is a Servlet, which can be reached via certain URL's and which is able to respond serialized Java objects in return. The remote interfaces of the stateless session bean are not used any longer and can be replaced by local interfaces. Since Servlet- and EJB container are both on the server-side, they can be easily connected using the internal container mechanism.

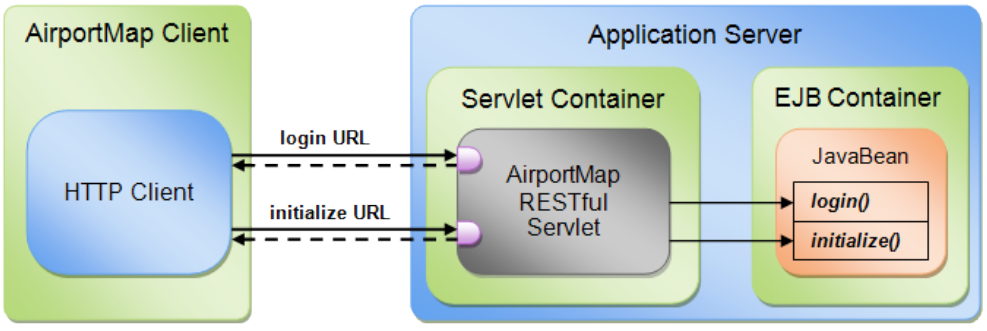

Figure 8.1.: The AirportMap Servlet

The Servlet now at once can access the Enterprise JavaBean directly, as Figure 8.1 approximately illustrates. The integrated library of the Apache commons

package (HTTP Client) connects via a predefined resource URL to the RESTful Servlet, which again would call the local services of the corresponding Stateless Session Bean. The results are further processed by the Servlet and respond as serialized Java Objects to the client. The byte-stream then gets delivered to the AirportMap application, in order to obtain the transferred Java Object.

The most important reason, why RESTful web services are used for this purpose, is the fact that the IBM Java Runtime Environment is not compatible with the SUN Runtime Environment. RESTful allows to access services only via the HTTP protocol, which makes the remote communication independent of any runtime environment and even other technologies can access them. Listing 8.1 accordingly shows, how the authentication is implemented for the RESTful Servlet.

```
1   /** Spring injected EJB facade for local EJB communication. */
2   private ServiceFacade amapService;
3
4   @GET
5   @Path("/username/{username}/password/{password}")
6   public Response login(@PathParam("username") String username,
        @PathParam("password") String password)  {
7      UserRoleEnum[] userRoles = null;
8      try {
9         userRoles = amapService.login(username, password);
10     } catch (UserAuthenticationDeniedException e) {
11        return Response.status(Status.FORBIDDEN).build();
12     } catch (UserAuthenticationException e) {
13        return Response.status(Status.UNAUTHORIZED).build();
14     } catch (RemoteException e) {
15        return Response.status(Status.SERVICE_UNAVAILABLE).build();
16     }
17     byte[] bytes = SerializationUtils.serialize(userRoles);
18     return Response.ok(bytes, "application/octet-stream").build();
19  }
20 }
```

Listing 8.1: AirportMap Authentication Procedure

When users try to authenticate, they have to enter their username, as well as their password. If they fit together and authentication is accepted, the user roles are returned to the client. On base of them, further authorization can be done. As the previous listing shows, the user roles are returned by calling the login function of the via Spring injected EJB. If authentication fails, a certain status code is returned, which shows the actual reason. Thereby, different status codes can be distinguished, as for example the *Forbidden* state, which indicates that the user has no access. The exception handling thereby differs the individual cases. If authentication was accepted and user roles were set, they get serialized to an byte array and respond to the client by using the mime-type *octet-stream*. Furthermore, the status code 200 (OK) also gets transmitted. Afterwards, everything the clients have to do, is to deserialize the object, in order to obtain the actual user role enumeration.

Furthermore, the resource URL's are set via specified annotations. For example, the login function is reachable under the URI-template pattern */username/{username}/password/{password}*, where as the parameters username and password are bound by *Jersey*, the reference implementation of the RESTful web service API. The Servlet thereby is a special *SpringServerServlet*, which enables Dependency Injection within it. The procedure for data initialization is quite similar except that it needs no parameters.

In summary, the remote communication without a JEE Client Container can be realised using the RESTful architectural style and the SpringServerServlet. Because Servlet- and EJB-Container are both on the server-side, they can be connected easily. This shows, how the client container is replaced entirely.

8.2. Integrating Technologies

One important requirement of migrating an application is the condition that afterwards it must work exactly as the original version. For this reason, all present technologies and frameworks must be adopted in order to reach that goal. Because the current prototype handles only the most important frameworks, irrelevant technologies are ignored.

Eclipse RCP currently offers different approaches of integrating libraries. The particular integration thereby depends on the individual use situation. Of course, since Eclipse RCP provides a modular environment, it is recommended to use the most modular approach, but in some cases, this is not applicable. In the following, the integration of key technologies is described explicitly.

8.2.1. JViews Maps

JViews Maps is the core component for displaying the airport situation, as well as its corresponding cartographic material. It is part of the commercial ILOG Framework provided by IBM and therefore closed-source as well. Its integration to the Eclipse Rich Client Platform is particularly difficult since its components are based on the Swing graphics library. Developing an own Geographic Information System on the base of SWT would be too complex and hence a solution must be found that allows to use the JViews Maps widgets within the Eclipse environment.

The JViews Maps framework at first has to be added to the rich client application, in order to access its classes. The ILOG products normally are distributed containing normal Java libraries, as well as OSGi bundles that can be installed by the integrated installation manager of the Eclipse IDE. The official recommended way of integrating JViews Maps to Eclipse RCP is installing those bundles. By performing this step, the OSGi bundles are added to the default target platform and made available for development. The integration of the library now is not that complicated. The actual difficulty is in using the Swing components within the rich client's workbench, but this can be realised using the two bridges *SWT_AWT* and *IlvSwingControl*.

Developing with the aforementioned libraries revealed that both bridges work quite similar and provide an effective environment for implementing Swing widgets within RCP applications. Thereby, the Eclipse-own SWT_AWT bridge can be implemented as follows:

```
1 IlvJManagerViewPanel panel = new IlvJManagerViewPanel();
2 Composite bridge = new Composite(parent,SWT.EMBEDDED);
3 SWT_AWT.new_Frame(bridge).add(panel);
```

Listing 8.2: SWT_AWT Bridge

Listing 8.2 in the beginning shows that for adding the *ILvManagerViewPanel*, which represents the main panel for other JViews Maps widgets, a wrapping composite must be created. Afterwards, the SWT_AWT frame is added to it, where finally the Swing widget is included.
This variant unfortunately showed some misbehaviour while painting its content. Sometimes the background started to freeze and the rendering of individual JViews widgets was faulty in some situations. Since ILOG offers its own solution anyway and because it is tailored directly to the framework, it is finally used to do that task. The following listing 8.3 shows, how it can be realized using the IlvSwingControl bridge.

```
1 IlvJManagerViewPanel panel = new IlvJManagerViewPanel();
2 IlvSwingControl panelBridge = new IlvSwingControl(parent,SWT.NONE);
3 panelBridge.setSwingComponent(panel);
```

Listing 8.3: ILOG Bridge

This listing is quite similar to listing 8.2 since IlvSwingControl also represents a wrapping composite, in which the actual Swing widget is added. By using this bridge, all required JViews Maps components work as usual, without great loss of performance. Therefore it is used as the actual software solution for integrating Swing components.

Another problem that came up during development, was the ILOG licence verification of the actual prototype application. Thereby a certain valid license key is needed, which gets verified within an internal procedure, in order to

deploy and execute its software components. In classic Java applications, the license verification is done at deployment level. There is a special Java tool (JLMDeploy), which converts the actual license file to a signed property of the exported application. This tool is applied using build tools such as *Ant* or *Maven*. Thereby, the whole application gets signed, which in case of Eclipse RCP destroys the OSGI manifest file, where dependencies to other bundles are defined. The application or corresponding plug-ins then becomes corrupt and inoperative.

To finally export Eclipse-based applications containing JViews Maps, another procedure must be used. Certainly, the ILOG framework itself provides a solution to this problem. During the OSGi bundle installation, a special plug-in is installed to the environment that is responsible for automatic license deployment and which does not destroy the OSGi manifest. This approach finally enables using the framework for rich client development and developers do not have to care about this issue any longer. The detailed manual for installing the bundles can be found in section A.2 of the appendix.

8.2.2. Queues And Topics

The IBM MQSeries is the actual message provider at Fraport AG. It implements the JMS API specification and is needed for supplying topics and message queues. Since the client container used to connect to them, the appropriate MQSeries libraries were integrated within it. Because the client container now is removed, the libraries must be integrated directly to the prototypical client application, in order to gain topic and message queue connectivity.

Unfortunately, MQSeries is not available as OSGi bundles and therefore must be copied directly to the appropriate source bundle. However, this variant couples the libraries too strong to the individual bundle, so that every project that would need its API has to contain a single copy. This would lead to redundancy and every project must be customized, in case the library needs update. Therefore, the integration is done by adding the MQSeries libraries in an own library bundle, where the coupling is not that tight. Buddy Classloading does not need to be implemented, as the library bundle has no bidirectional access to other bundles.

This bundle contains not only the actual libraries but also their dependencies and exports all of its packages, as it can be seen on Figure 8.2. Other bundles then can include those exported packages, in order to use their supported classes. The library bundles thereby specifies the library files, which are added to the classpath, as it can be seen in the lower right corner.

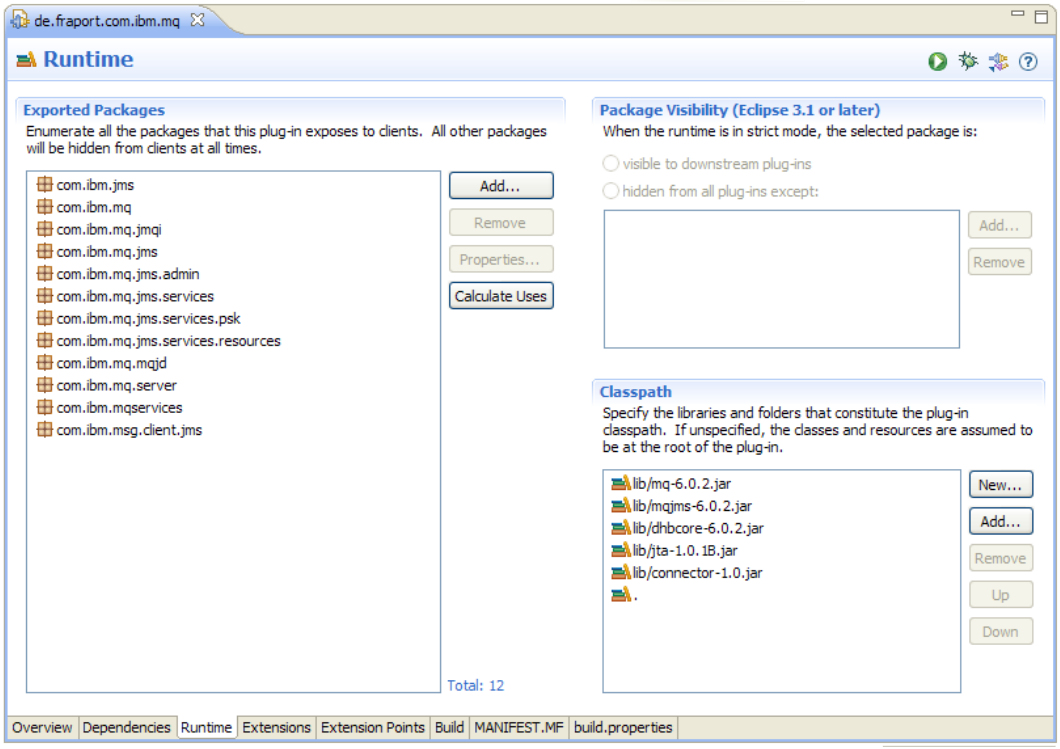

Figure 8.2.: MQSeries Library Bundle

In context of the AirportMap prototype, the libraries could also be added directly to the particular bundle, since only one module is responsible for topic and queue connectivity. But to keep up with the modularity, the variant using library bundles was chosen. This approach would avoid design problems, which can occur when several projects contain copies of one certain library. They all have to be maintained individually, what would be an unclean solution. Having the libraries in a bundle, which exports its packages that can be referenced by others, avoids this problem.

8.2.3. Spring Dynamic Modules

The Spring Framework belongs to one of the central technologies within the AirportMap client. It has a huge user community and constantly improves quite fast. Since Spring showed to be a powerful framework for simplifying the development of Java EE applications on the server side, as well as integrating familiar frameworks, it is also attractive for the use in client applications.

In classic desktop applications, Spring can be integrated easily by adding its libraries directly to the class path. The Spring Container thereby can be start up manually. Since Eclipse RCP follows a modular approach on bundle layer, Spring therefore has to be modularized. Having a single application context is not enough, since every bundle must contain its own. The application context is a file, in which amongst others *Spring Beans* are declared. As different bundles can be added and removed on demand, a mechanism has to be implemented, which will search for an available application context within all bundles at start-up. Furthermore it must be able to export and import Spring Beans, as it is common with the library bundle approach. By this reasons, an own Spring component was developed, which is customized especially to OSGi and Eclipse RCP: The *Spring Dynamic Modules* (Spring DM).

Spring Dynamic Modules, formerly known as *Spring OSGi*, enables the usage of Spring functionality within Eclipse-based rich client applications and integrating it to Eclipse RCP is quite simple. There is an online repository[2] that provides all Spring components, as well as their dependencies as OSGi bundles. In order to use them, they only have to be copied to the target platform.

Furthermore, this approach allows to install different versions simultaneously, since they are distinguished by their individual version numbers. If a newer release of Spring is published, it can just be added to the target platform. Source bundles then can specify a certain version number by adding an entry in their bundle manifest, which enables the opportunity of interconnecting technologies with different release dependencies. Of course, this mechanism can also be applied to all other frameworks, which are available as OSGi bundles.

[2] http://www.springsource.com/repository/app/

By this reason, integrating the Spring Framework in Eclipse RCP is no problem at all. One of the more difficult tasks is to start up the container, which manages every Spring Bean. This has to be done on the lowest level, since Spring Beans must be imported and exported as it can be done with packages. Thereby the *Buddy Classloading* approach is used, because the individual bundles need to access the Spring container. To finally bring up this container, Spring DM provides an own bundle, which is known as the *Spring Extender*. The Spring extender will check every bundle for an available application context and compiles the Spring container, before the actual application gets started. In order to launch this bundle, it must be added to the auto-start configuration of the rich client application, as it can be seen on Figure 8.3.

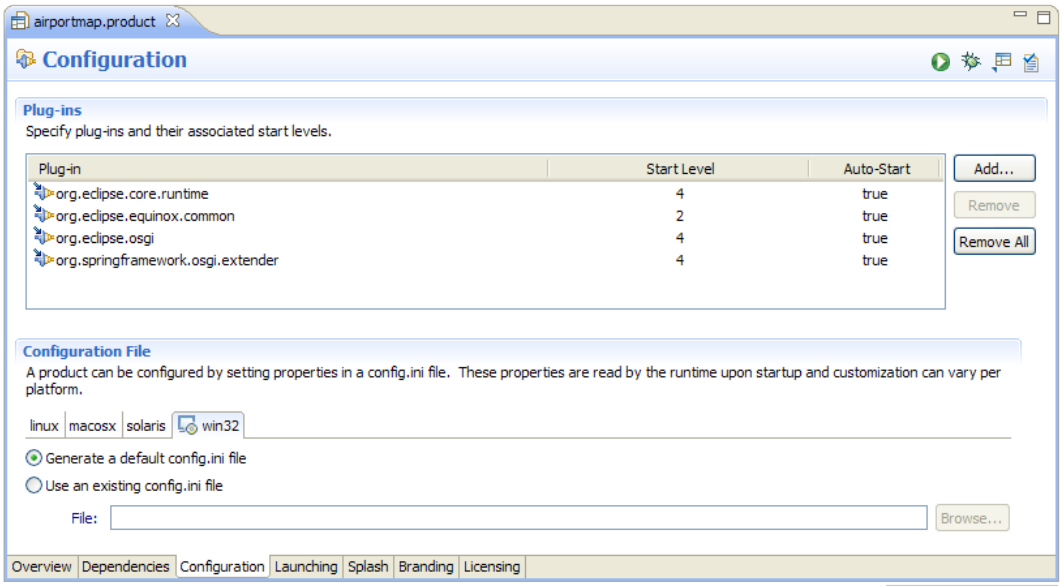

Figure 8.3.: The AirportMap Start Levels

The Spring extender is placed on the same start-level as the core runtime and the OSGi platform, which means that all three components are loaded synchronously. Thereby the default run-level is 4. This configuration ensures that Spring is loaded right in time, since it is important for dependency injection. If individual bundles and their classes are loaded before the spring container is started up, corresponding Spring Beans are not set, what accordingly would cause *NullPointerExceptions*.

Since the rich client application now contains a Spring container that manages the life-cycles of all corresponding Spring Beans, this measure also has to be applied within Eclipse-based applications. At this point, a strategy must be considered, because Eclipse RCP is working mainly with the extension point mechanism, which instantiates and manages its components automatically. Therefore, a factory is required, which resolves Spring Beans out of the container and injects them in distinct extensions. Until today, there is no official solution provided by the Spring or Eclipse communities and hence, this mechanism must be implemented by one's own hands.

```
1  <plugin>
2       <extension point="org.eclipse.ui.views">
3          <view
4             class="de.fraport.airportmap.views.PunctualityView"
5             id="de.fraport.airportmap.views.punctualityMap"
6             icon="icons/clock.png"
7             name="Punctuality View">
8          </view>
9       </extension>
10 </plugin>
```

Listing 8.4: Normal Extension Point Declaration

Listing 8.4 accordingly shows the basic declaration of the AirportMap client's punctuality view in the plug-in.xml of its appropriate module. This extension point contains meta information such as id or class name, due to that Eclipse RCP then automatically is responsible for instantiating and managing it. If the users for example click at the view icon, the corresponding class *"de.fraport.airportmap.views.PunctualityView"* gets instantiated and the predefined function of its interface is executed. Because this class now is compiled outside of the Spring container, the included Spring Beans are not injected. This is the situation, where the factory must come into action.

The factory must be designed using static methods and needs to implement the interface *"IExecutableExtensionFactory"*. This enables to obtain the actual class out of an object pool, instead of letting the extension registry instantiate them by itself. Furthermore, the factory needs to access the Spring Container, which in fact represents the required object pool and therefore can return the appropriate Spring Beans, according to their specified bean id.

The *SpringExtensionFactory*[3] implemented by the developer team around Martin Lippert, exactly performs that task in a reliable and effective way and hence is used in the AirportMap prototype. The actual class just needs to be replaced by the SpringExtensionFactory, as it can be seen on listing 8.5.

```
1  <plugin>
2       <extension point="org.eclipse.ui.views">
3          <view class="org.eclipse.springframework.util.
               SpringExtensionFactory"
4             id="de.fraport.airportmap.views.punctualityMap"
5             icon="icons/clock.png"
6             name="Punctuality View">
7          </view>
8       </extension>
9  </plugin>
```

Listing 8.5: Extension Point Declaration with Spring DM

Since all extensions must implement the *IExecutable* interface, the SpringExtensionFactory can be used for every single one. The bean allocation happens via comparing the ids of the appropriate Spring Beans with the one defined in the plug-in.xml file. This increases the reusability of every single extension, because they can be hold as Spring Beans and hence the same instance can be used several times.

[3] http://martinlippert.blogspot.com

In summary, the Spring Framework can be integrated easily, but needs a little effort in configuration. Via the Spring Extender, the OSGi framework calls up the container right in time, so that Spring Beans are available before the actual workbench is created. By integrating the SpringExtensionFactory in the plugin.xml, also Spring Beans can be injected into extensions such as views and editors.

8.3. Component Structure

Modularity plays a significant role within Eclipse RCP applications, since it has a modular plug-in architecture. Therefore, it is of great importance to modularize the entire application into individual parts. Each module thereby is responsible for a certain task and especially business logic must be separated from presentation logic.

The Java Swing based AirportMap client is not very modular in its structure, because the source code is wired strongly within its presentation logic. This must be changed basically, in order to obtain a modular design.

The cartographic material for example is constructed as an own resource bundle, since it becomes quickly outdated due to continuous building activities at the airport. Updating the whole application therefore is inefficient and very time-consuming. With the cartographic material located in one single plug-in, it can be replaced selectively and the application must be deployed only if its really necessary.

Furthermore, it is also intended to structure the individual process views as own components, so that they can be installed or removed at runtime, without affecting the whole application. Process views are customized views, which show the map specific to individual business processes. Typical processes are for example *Aviation* or *Punctuality* that are focused on different points of view. If those individual process views could be designed as perspectives within a separate bundle, modularity will be improved, because coherent components can be centralized in one single module.

To merge all these modules to a working application, some kind of initial point has to be defined, which bundles and supplies them with basic functions and

services, as for example the server communication. This is important, because otherwise every module has to manage this connection on its own. Instead, each module now can access the services offered by the central component and hence needs only to be maintained at one single place.

Figure 8.4 approximately shows, how the bundle structure of the actual prototype application is be designed. The central starting point is found by the *AirportMap Core* application that in fact provides rich client functionality and basic services, such as the remote connection for authentication and initialisation, but also access to the Spring Container and JViews Maps components.

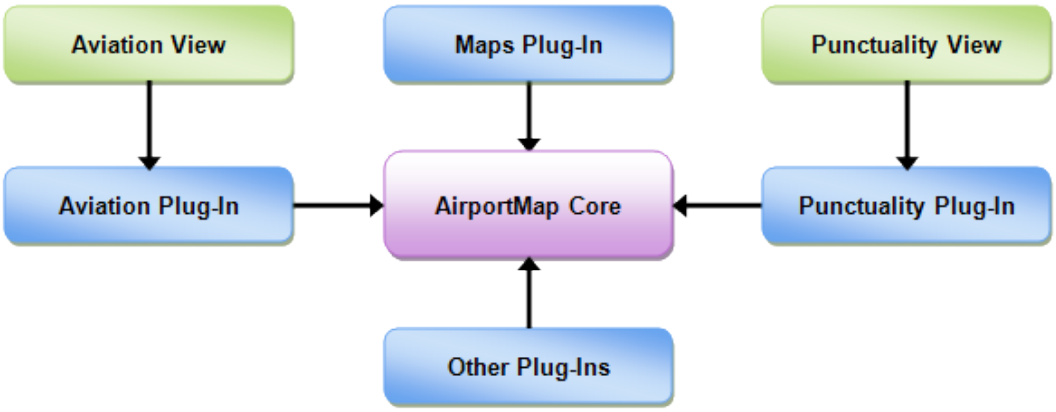

Figure 8.4.: The AirportMap's Bundle Structure

The individual process views are available as additional plug-ins, which can be installed or removed at will, as for example for different user roles. Each process view plug-in contains at least one own perspective and a corresponding set of views or editors. Furthermore the cartographic material is also available as plug-in, even though it makes neither contribution to user interface nor brings any business logic. Therefore it is designed as pure resource bundle, which just contains the files used for displaying the map. Due to this segmentation, the map files can be easily replaced and updated.

```
1  Bundle bundle = Platform.getBundle("de.fraport.airportmap.maps");
2  Path path = new Path("maps/apron.shp");
3  URL url = FileLocator.find(bundle, path, null);
```

Listing 8.6: Declarative File Loading

Listing 8.6 shows, how the content of the map bundle is being accessed. At first, the bundle is resolved using its unique identifier. Afterwards, the path to the actual required file is set, whereas the URL is created by applying the find method of the *FileLocator* class.

In addition, rich client development allows to specify dependencies to individual plug-ins, which must be integrated at start-up. In context of the AirportMap, the core application will not work properly if the cartographic material is missing and therefore it depends on this plug-in. Furthermore, the core application must have an initial perspective, which gets used at start-up. Initial perspectives are set by defining their unique identifier in the ApplicationWorkbenchAdvisor class, as Listing 8.7 shows.

```
1  public String getInitialWindowPerspectiveId(){
2     if(Platform.getBundle("de.fraport.airportmap.aviation")!=null){
3        return "de.fraport.airportmap.perspectives.aviation";
4     }
5     return DefaultPerspective.ID;
6  }
```

Listing 8.7: Initial Perspective Declaration

The function at first checks, if the aviation bundle is available in the application and will return its identifier, in case it is installed. Otherwise the identifier of the default perspective is returned. Since the Aviation process view is default in the current release, the AirportMap Core also depends on the Aviation plug-in, which provides this perspective.

Based on this modular approach, it is possible to simply extend the application by further plug-ins. For example, it is planned to develop a *position allocation* module that provides two perspectives for current and forecast assignment of aircraft to their positions. Adding or removing plug-ins at runtime would not be possible with the current implementation, since it has no underlying OSGi framework that would manage its components and hence this is a huge step towards extensibility.

8.4. Packages and Classes

The prototypical implementation of the AirportMap client application actually is segmented into several source bundles, as showed in Figure 8.4. The source code thereby is totally refactored to separate business, authentication and presentation logic. By reasons of the new design concepts provided by Eclipse RCP, the separation is simple, as for example the presentation logic is placed in views and editors. To finally separate business from authentication logic, they are just developed in different classes. Figure 8.5 shortly gives an overview of all used classes in the core application.

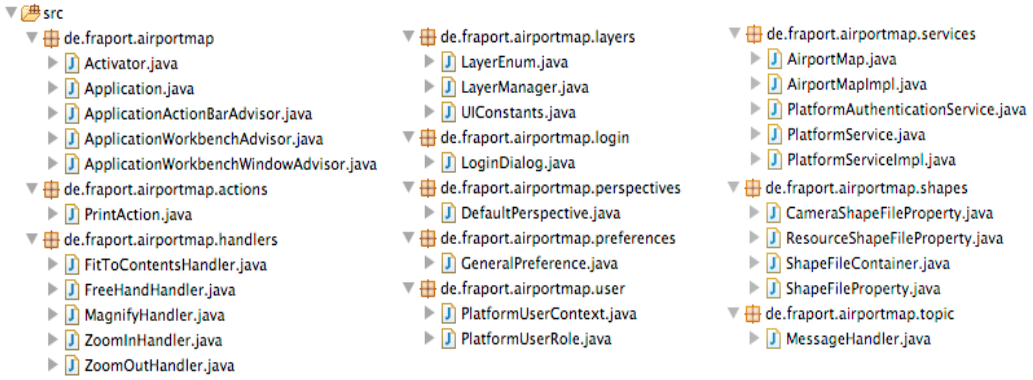

Figure 8.5.: Package Listing of the AirportMap Core Application

In the main package (de.fraport.airportmap), the fundamental classes for RCP functionality are located, which are used to finally create the workbench. Furthermore, there are several packages containing actions and handlers. They are used to execute commands such as zooming and panning of the map. In addition, the layers and shapes packages are used to finally create the map layers. The actual cartographic material thereby is located in a special resource bundle, which gets loaded within the *LayerManager* class.
Another important package is the service package that amongst others contains service interfaces to perform the authentication procedure as well as message queue and topic connectivity. All other packages and classes are just used by aforementioned classes.

In the following, Figure 8.6 shows the package structure of the resource bundle. The bundle contains nothing except the map files, which are needed for displaying the map. They are located in the folder *"maps"*, which can be referenced by the file loading procedure, shown in listing 8.6.

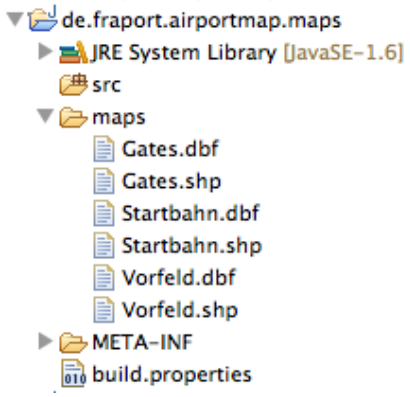

Figure 8.6.: Package Structure of the Maps Resource Bundle

Figure 8.7 shows the package listing for the aviation plug-in, which extends the core application with a certain perspective and view, focused on aircraft business.

Figure 8.7.: Package Listing of the Aviation Plug-In

As the focus of redeveloping the current AirportMap implementation is more to show its feasibility, the aviation plug-in does not contain any business logic. Therefore, the bundle only contains the Aviation Perspective class and corresponding views. The package listing for the punctuality plug-in is quite similar and therefore not pictured.

A detailed description of the functionality of individual classes, as well as UML diagrams are disclaimed, as this is not the primary focus of this bachelor thesis.

8.5. Spring Services

Services are a key element of the OSGi architecture. In short, the service mechanism is a way for one bundle to register a service provider and other bundles discover and use those services [ML08, p.457]. If several code bundles have to work together, then this can be realised by consuming those defined OSGi services.

But every bundle has its own separated application context and so they are unable to access Spring Beans of others. Therefore, the Spring Dynamic Modules brings the opportunity of defining own Spring-based OSGi services. This allows exporting services for other bundles, which can import them by adding references to their own application context. Thereby, those references are managed by the Spring Framework.

In some situations, it is important to concentrate certain tasks to one distinct application component, which defines all basic functions in one or several offered service interfaces. In context of AirportMap redevelopment, the core application offers OSGi services, as for example the resolution of user roles, which can be accessed by all other source bundles.

Listing 8.8 accordingly shows the export configuration of the AirportMap's platform service. Thereby, the Spring Bean with the id *"service"* is referenced to the OSGi service, which provides its implementation by casting it to the corresponding interface.

```
1  <beans>
2     <bean id="service"
3        class="de.fraport.airportmap.services.PlatformServiceImpl"/>
4     <osgi:service ref="service"
5        interface="de.fraport.airportmap.services.PlatformService"/>
6  </beans>
```

Listing 8.8: OSGi Service Export

As Figure 8.8 approximately shows, the core application is the central service provider for its corresponding plug-ins. It accesses the server directly and obtains results from it. The individual plug-ins can access the offered Spring service and let the core application perform central tasks. This provides a more modular and independent programming paradigm, since it would be of a bad design, to leave central actions like server communication to each bundle individually.

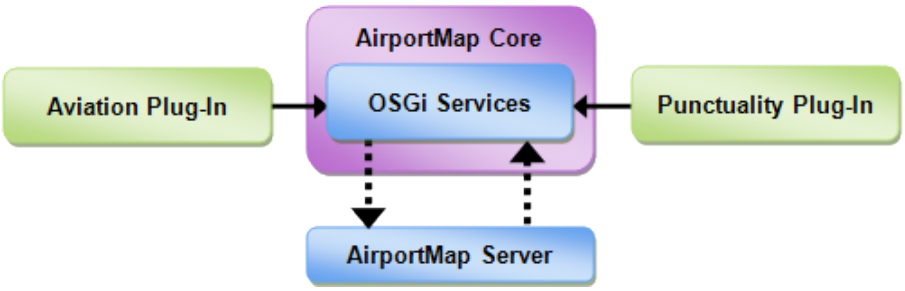

Figure 8.8.: Services in the AirportMap Prototype

If a plug-in wants to import that provided platform service, it must specify its unique identifier, as well as interfaces to the application context, as listing 8.9 shows. Subsequently, it can be injected into appropriate classes like any common Spring Bean as well.

```
1  <beans>
2     <osgi:reference id="service"
3        interface="de.fraport.airportmap.services.PlatformService"/>
4  </beans>
```

Listing 8.9: OSGi Service Import

This in fact allows the sharing of exported Spring services in various bundles, but to finally use them, the appropriate bundles must also import the packages containing the interfaces. In the AirportMap prototype, this is realised by exporting the appropriate packages in the core application, as Figure 8.9 shows.

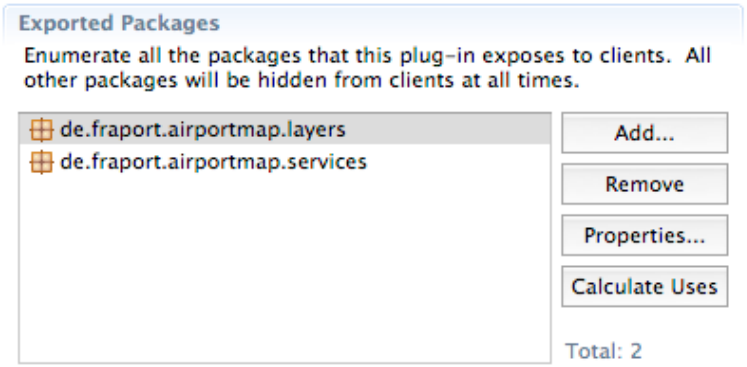

Figure 8.9.: Exported Packages of the AirportMap Core Application

By reasons of exporting those packages, the interfaces located in *"de.fraport.airportmap.services"* can be accessed and therefore used in the Spring services. Furthermore, the classes needed for map compiling also must be exported, since further plug-ins need to access them, as for example the *Punctuality* perspective would manipulate the map for a special view on delayed aircraft.

In summary, exporting OSGi services with Spring DM is not really difficult and allows a bundle comprehensive usage of separated Spring Beans. This increases modularity on application level and simplifies the usage of individual services. The core application manages the server communication and thereby provides a platform service, which can be accessed by other bundles using Spring Services.

8.6. Update and Distribution

The Eclipse Rich Client Platform enables a fine-grained and individual update and extension mechanism, since the hereby compiled rich clients have a deep modular approach. This allows adding and replacing individual components at runtime and opens the possibility of writing effective update and install managers. Further, there are already prefabricated solutions, which are directly provided by Eclipse RCP. In the following, the update functionality as well as distribution of the prototype is described explicitly.

8.6.1. Update Site

For being able to update and extend existing applications in an user-friendly way, an update server is needed, which provides all actual updates and extensions. In context of Eclipse RCP, such a server hosts one or more *Update Sites*, which actually contain features and their corresponding plug-ins, but also meta information about their version numbers, file sizes and checksums. All files are provided via HTTP and packaged with the same folder structure as the rich client application. An Update servers in fact is comparable to normal web servers that additionally host meta-data and artefacts, such as plug-ins and features.

The Eclipse IDE provides particular tools that allows to create such Update Sites in a comfortable way. Individual features can be added and grouped to certain categories, which give them a logical structure. Afterwards, the update site can be exported, so that all features and plug-ins are built. The whole project folder can then be copied to the actual update server. A detailed manual is given in section A.2 of the appendix.

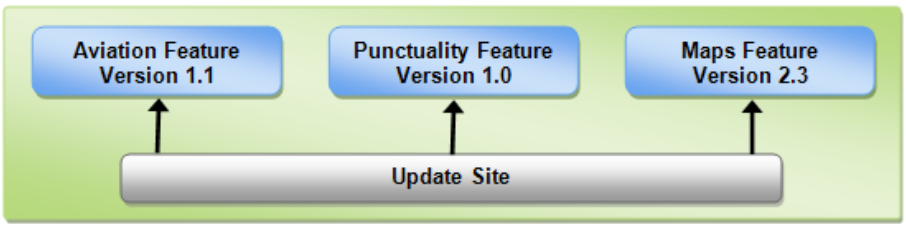

Figure 8.10.: The AirportMap Update Site

Figure 8.10 approximately shows, how the update server is realised in the prototypical AirportMap environment. The Update Site hosts three different features, as well as their meta information. The idea behind this architecture is that those components only have to be maintained at one single place in the system environment and can be accessed individually by each client. Deployers could publish new features and plug-ins immediately, which then can be downloaded and installed by the client applications.

8.6.2. Equinox p2

For performing update and installation procedures, Eclipse RCP has its own update and installation manager. This software components is able to access individual update sites and interprets their meta information for calculating available updates and dependencies.

Until Eclipse was released in version 3.4, Eclipse RCP provided the *Update Manager*, which was responsible for downloading, installing and upgrading features and their corresponding plug-ins. But this technology has become outdated and is in version 3.4 replaced by the provisioning tool *Equinox p2*. Update Manager and Equinox p2 have similar user interfaces, so that the replacement did non attract great attention. Before Equinox p2 was introduced, many users avoided the Update Manager and installed new plug-ins by inserting them into the plug-ins directory, since with this approach, local plug-ins need not to be available as an update site. P2 also allows this procedure and additionally provides a new dropins folder, which is much more powerful and allows separation of content managed by p2 from content managed by the users. Since p2 is the recommended way for the update mechanism, it is also used in the prototype of the AirportMap client.

In order to integrate Equinox p2 to the prototype application, certain changes must be done. At first, the application has to be feature based, because Equinox p2 requires them for its functionality and hence a new feature must be created, which contains the actual application core plug-in and rcp functionalities (*org.eclipse.rcp*). To finally use the actual Equinox p2 components, the *org.eclipse.equinox.p2.ui* feature must be added to the application. This in fact includes all the required plug-ins, but they are not loaded until one of

its menu-bar entries is clicked. Therefore, the menu-bar must contain a "help" field, in that Equinox p2 could insert its menu-bar entries. The detailed installation manual can be extracted from section A.2 of the appendix.

This procedure indeed would integrate Equinox p2 to the application, but it still needs further configuration, which is extracted from the *"p2.inf"* file located in the root directory of the source bundle. This file contains the URL's of available update sites. Listing 8.10 shows the actual configuration for the prototype application.

```
1 instructions.configure=\
2   addRepository(type:0,location:http${#58}//localhost/updates);\
3   addRepository(type:1,location:http${#58}//localhost/updates);
```

Listing 8.10: Equinox p2 Repository Configuration

The configuration file contains appropriate defined repositories (update sites), which in this case are located on the local development machine, from where Equinox p2 will search for updates as well as extensions. Of course, several repositories can be registered, whereby type 0 and 1 represent meta-data and artefact repositories.

By interpreting those meta information, Equinox p2 is able to calculate available software updates and installable features. Furthermore, it supplies a graphical user interface, which accompanies the user through the whole installation process and makes installation details visible, as it can bee seen on Figure A.3 of the appendix. For usability purposes, it fades out already included features, but can deinstall them again at will. This is of great importance, if for example individual plug-ins needs to be reinstalled. Thereby, p2 offers a listed history of installations, so that a former state can be restored.

Figure 8.11 approximately shows, how the update mechanism works in the AirportMap environment.

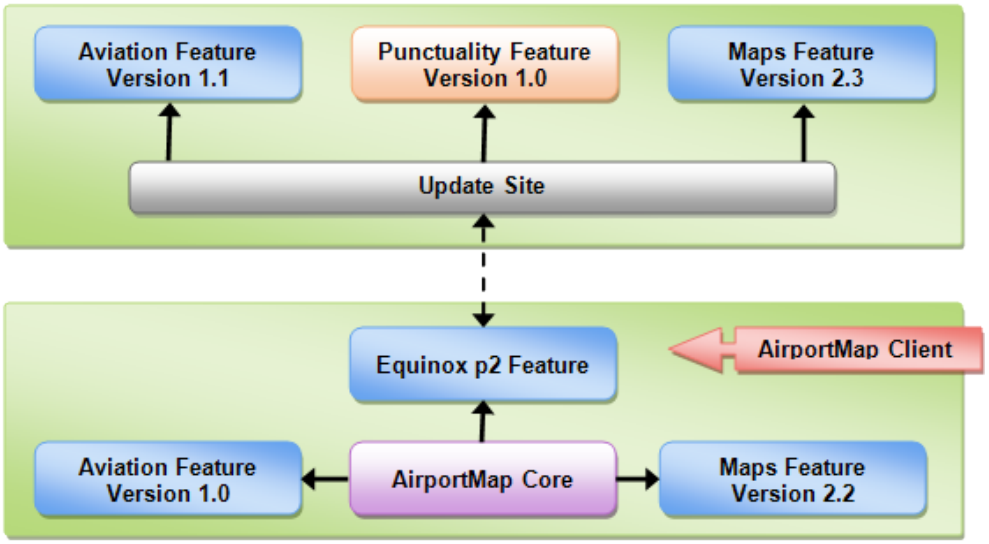

Figure 8.11.: The AirportMap Update Mechanism

The core application contains the Equinox p2 feature, which is responsible for accessing the update site, defined in its configuration file. Since there are newer versions of the already installed components available on the update site, Equinox p2 will mark them as updatable features and offers them for update. Furthermore, the feature *Punctuality*, which is not installed currently, is also offered for installation. This shows that it is hard to define the scope of updates and extensions, as they are pretty similar in context of Eclipse RCP.

Due to the usage of Update Sites and Equinox p2, the opportunity is given to let the AirportMap update and extend itself independently. Previously, this was hardly possible, but now this opens a new way for its development and distribution.

8.6.3. Software Depot

Once the AirportMap application is installed on the client machine it can be updated by its own through the Eclipse update mechanism. But in order to profit from this update mechanism, it is necessary that the Eclipse based application has been already distributed to the client machine. The following section deals with the remaining question of how the application can be initially distributed to the client machines.

Currently, the distribution of the AirportMap is solved by the Java Web Start technology and therefore it is nearby to distribute the prototypical client application again with this technology. Furthermore, it provides a handling of authorization for copying the application into the user directory, what would a positive argument for using this technology. Using Web Start within Eclipse RCP applications is quite possible actually, but further examinations found out, that according to [Rei09, p.250], installed plug-ins can not be updated by the integrated update managers and some configuration parameters for OSGi will not work any longer. By this reason, Java Web Start is unsuitable for the AirportMap RCP application.

Due to that reason, the company-wide Software Depot could be used for initial software distribution the AirportMap client. This package management application allows the comfortable management of software on a computer system, where it is responsible for installing, updating and uninstalling individual software in form of packaged applications.

By using the Software Depot, every kind of software can be packaged, managed and distributed. For classic Java applications that are executed via special batch files, this procedure is of no relevance and hence insignificant. But for Eclipse-based applications, this package management would make sense.

Since the Fraport AG already has this system in use, it is recommended to benefit from this infrastructure by using it also for the initial AirportMAP client distribution.

9. Conclusion

In this bachelor thesis, the AirportMap client application was developed by using the Eclipse Rich Client Platform. Thereby, many requirements were declared, as well as needed changes were performed. This chapter shortly summarizes the measures, accompanied by the redevelopment and furthermore gives an outlook about the usage of RCP applications in the corporation.

9.1. Summary

In conclusion, the current Swing and Java EE-based client application could be successfully redeveloped as a prototype with a more fine-grained modular approach, by using Eclipse RCP as its basic foundation. Certainly, several adjustments had to be performed and its fundamental assembly was revised totally.

The first important step in this context was to replace the remote connection, which was currently solved using a Java EE client container. As alternative solution therefore, a RESTful web service was implemented on the server side, in order to enable a reliable server communication. The libraries for accessing the topic and message queues are directly implemented to the new prototype application.

Because the AirportMap client is intended to be an Eclipse-based RCP application, the fundamental assembly for this reason had to be redesigned in a more modular way. The cartographic material, as well as each process view are implemented as individual plug-ins for a core application, what in fact enables the opportunity to update only certain parts of the whole AirportMap client.

Furthermore, all existing technologies such as the Spring or the JViews Maps framework must have been integrated to the new prototype application, in order to obtain the full functionality, as it is given in the current implementation. The methods of how to integrate those individual technologies differ strongly, since Eclipse RCP in fact comes up with new design concepts. Thereby, the existing technologies are integrated either as library- or direct OSGi bundles.

Another important point of redeveloping an existing application to Eclipse RCP is the fact that all graphical widgets have to be replaced by a corresponding SWT component. Therefore, remaining widgets, such as the graphical elements of the JViews Maps framework are emulated using a graphical software bridge provided by ILOG itself.

In order to get detached from the complex and time-consuming roll-out processes of the management, the AirportMap prototype now contains an own update mechanism, which can be used for constantly updating the client application on demand. Furthermore, an update site is constructed, which will provide all upcoming and available plug-ins. Finally, the application gets distributed by the package management system *Software Depot*, as it is needed for an initial roll-out of the core application.

To summarize all aforementioned topics, the prototypical AirportMap client runs fluently and meets all predefined requirements. By this reasons, the application can be used as a template for future development to Eclipse RCP. Because the source code is more modular and testifies of a good design, it is well-prepared for later maintenance. For this reason, the prototype was successfully implemented.

9.2. Outlook

In general, building robust and maintainable applications with a rich user interface is a challenge to any software developer. With Eclipse RCP, a framework is introduced that simplifies the development to a minimum, as it brings up new design concepts. The developer thereby can focus directly on its work, instead of working out graphical widgets by its own.

At Fraport AG, rich client development with Eclipse RCP will be further persecuted, since it brings many advantages and simplifications for building client applications. For example, there are prospective plans of implementing an Eclipse-based *BIAF Platform* application, which will be used to provide special data analysis and interpretation plug-ins. Thereby it should keep on the design structure of the AirportMap rich client. Because the users have different roles, the Java Authentication and Authorization Service (JAAS) can be adopted, which would be responsible for authentication. Users then can only use extensions, for that they are authorized.

Therefore, Eclipse RCP becomes more and more interesting as a powerful alternative to Swing development at the Fraport AG.

A. Appendix

A.1. Screenshots

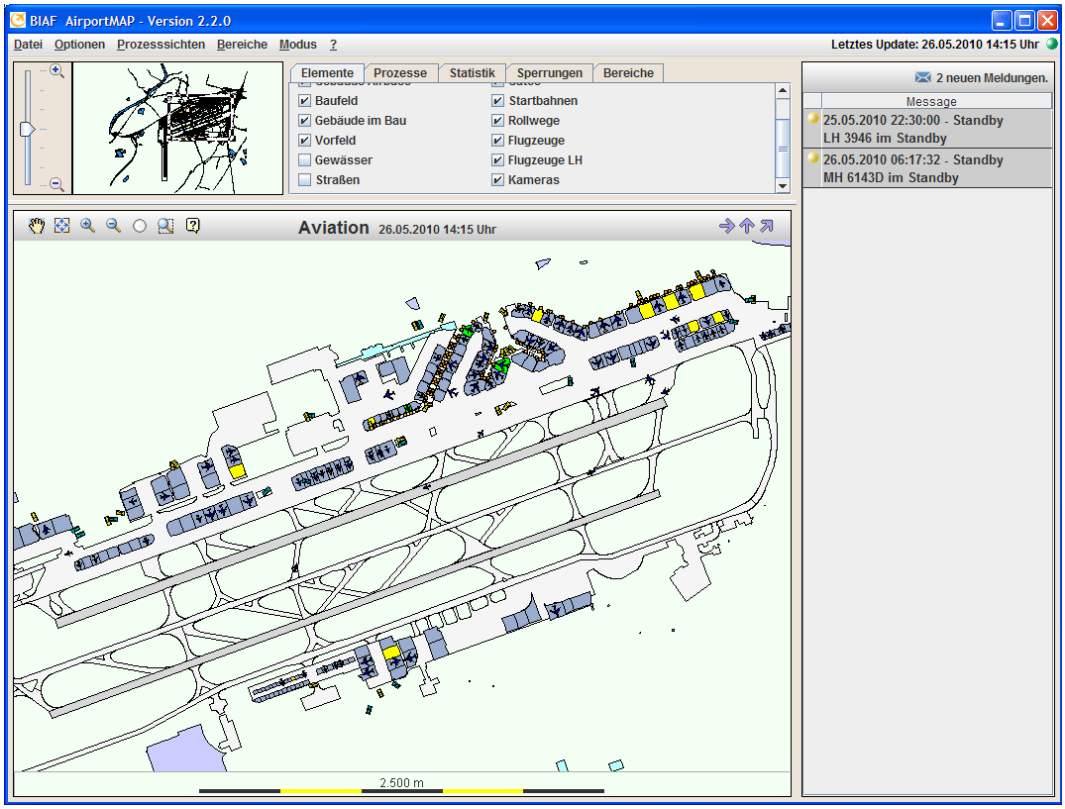

Figure A.1.: *Screenshot:* AirportMap Swing Client

As Figure A.1 shows, the graphical user interface does not fit the native look-and-feel of the Operating System. The default Swing widgets are emulated, what let the application looks quite antiquated. Furthermore, the picture illustrates the *"Aviation View"*, as well as the event notification list and map-layers.

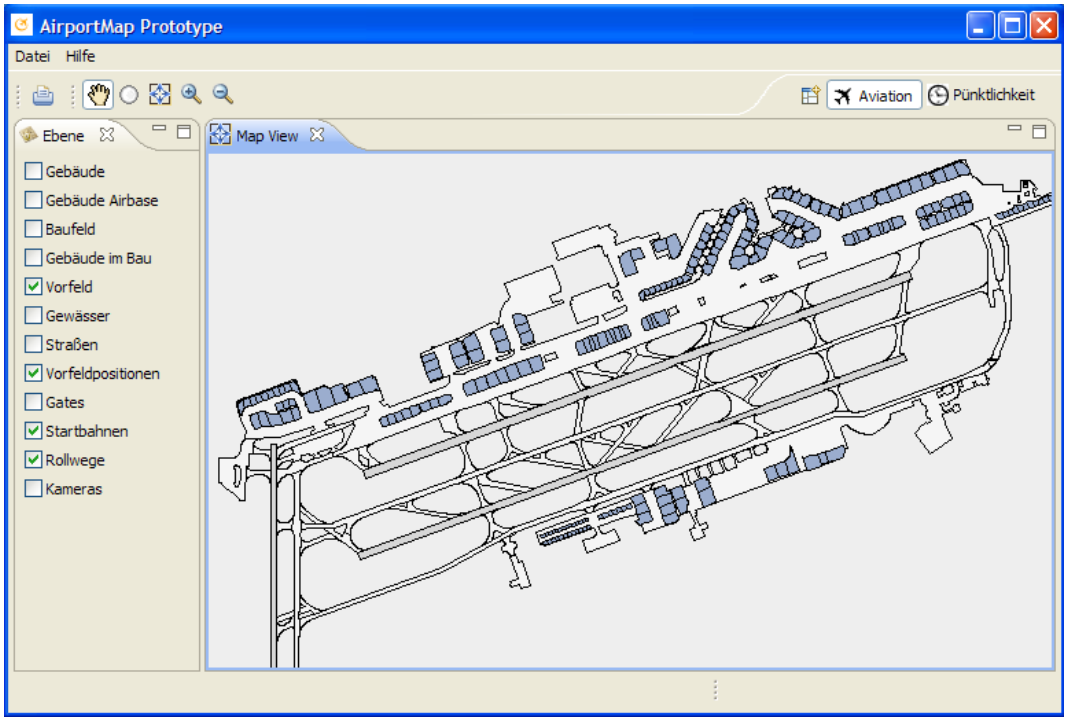

Figure A.2.: *Screenshot:* AirportMap RCP Client

Figure A.2 shows the prototypical redevelopment of the actual AirportMap client application. It thereby shows the different implemented perspectives, which in fact represent the individual process views, but also a view for (de)selecting certain map-layers. Furthermore, map-specific tools are embedded into the toolbar of the RCP application and in conclusion, the new prototype has a native look-and-feel.

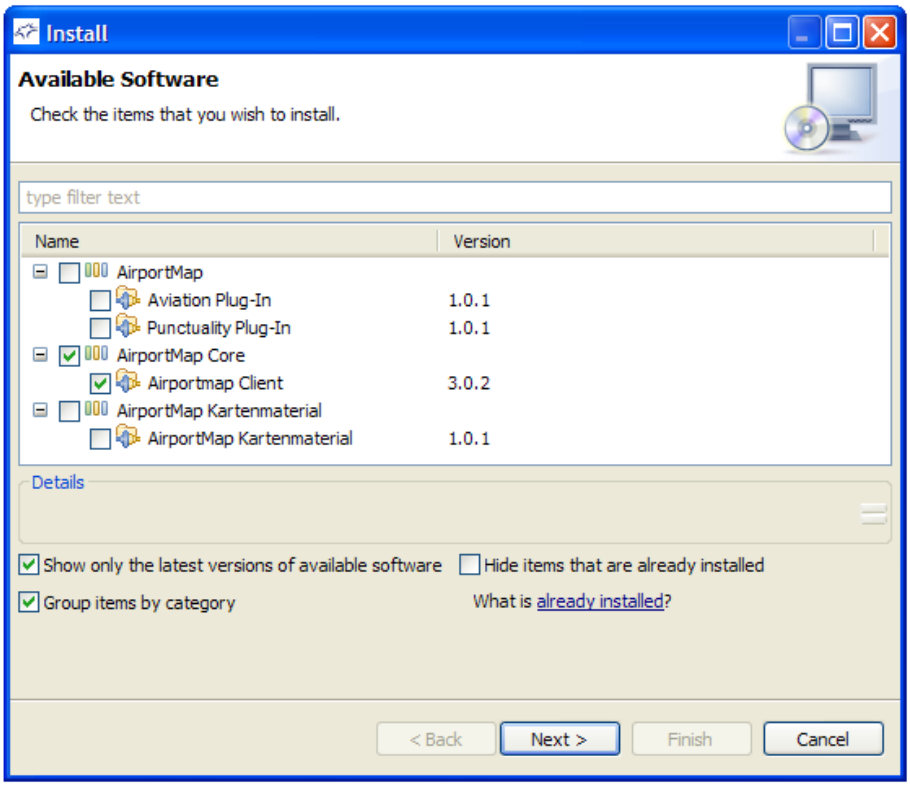

Figure A.3.: *Screenshot:* AirportMap Update Installation

The Equinox p2 provisioning tool provides certain predefined windows and update routines. Figure A.3 shows the selection of available updates and extensions. The Update Site thereby offers categories, in which the individual features are categorized. Furthermore, a distinct version number can be chosen, in case of several equal features.

A.2. Manuals

ILOG Installation

1. Install *jviewsmaps86.exe* to the local drive
2. Start up Eclipse IDE
3. Menubar: Help - Install New Software...
4. Add the local ILOG update site[1] as repository
5. Select the new repository and install the features provided by it
6. Follow instructions of the Installation Manager
7. ILOG now has been installed to Eclipse IDE

Equinox p2 Installation

1. The application's product configuration must be feature-based.
2. A new feature must be created, which contains amongst others
   a) the rcp application plug-in
   b) "org.eclipse.rcp"
   c) "org.eclipse.equinox.p2.ui"
3. The p2 configuration file (p2.inf) must be created and added to the root directory of the corresponding RCP application
4. A new help menu containing *IWorkbenchActionConstants.M_HELP* has to be added to the ApplicationActionBarAdvisor class.
5. The RCP application must be exported including a meta-data repository, in order to use Equinox p2.

[1] by default located in ../IBM/ILOG/jviews-framework86/tools/ilog.views.eclipse.update.site

Update Site Compilation

1. Install a Web Server such as Apache HTTP Server[2]
2. All wanted update plug-ins must be available as features
3. In Eclipse IDE: File - New - Other - Update Site Project
4. Assign a project name and optionally the web page listing - Finish
5. Add new categories including ID and descriptions ("New Category")
6. Attach the wanted features to the specified categories ("Add Feature...")
7. Specify a name and description in the "Archives" tab editor.
8. Build the Update Site by performing "Build All"
9. Copy the project from the workspace into HTTP server

[2]http://httpd.apache.org/

Bibliography

[Bay02] Thomas Bayer. Rest web services. Technical report, Orientation in Objects GmbH, 2002.

[BG02] Martin Backschat and Otto Gardon. *Enterprise JavaBeans - Grundlagen, Konzepte, Praxis*. Spektrum Akademischer Verlag, 2002.

[Bou09] Martin P. Bourque. The sas data warehouse: A real world example. 2009. Available from: `http://www2.sas.com/proceedings/sugi22/DATAWARE/PAPER126.PDF`.

[BSB08] Bryan Basham, Kathy Sierra, and Bert Bates. *Head First - Servlets and JSP*. O'Reilly, 2nd edition, 2008.

[CHL08] Adrian Colyer, Hal Hildebrand, and Costin Leau. Spring dynamic modules reference guide. Technical report, SpringSource, 2008. Available from: `http://static.springsource.org/osgi/docs/current/reference/pdf/spring-dm-reference.pdf`.

[Ebe09] Ralf Ebert. Ueberblick Eclipse RCP. Technical report, 2009. Available from: `http://www.ralfebert.de/rcpbuch/overview/`.

[Ecl10] Eclipse Foundation. Rich client platform. Technical report, 2010. Available from: `http://www.eclipse.org/home/categories/rcp.php`.

[Fow04] Martin Fowler. Inversion of control containers and the dependency injection pattern. 2004. Available from: `http://martinfowler.com/articles/injection.html`.

[Fra09] Fraport AG. Zahlen, Daten, Fakten zum Flughafen Frankfurt, 2009. Available from: `http://www.fraport.de/cms/presse/dokbin/391/391285.zahlen_daten_fakten_2009@de.pdf`.

[Fra10] Fraport AG. Portrait der Fraport AG. 2010. Available

from: http://www.fraport.de/cms/presse/dokbin/403/403512.portrait_fraport_ag_05_03_2010.pdf.

[HC01] George T. Heinemann and William T. Councill. *Component-Based Software Engineering*. ACM Press, 2001.

[HS09] Manfred Hennig and Heiko Seeberger. Einfuehrung in den "Extension Point"-Mechanismus von Eclipse. *Java Spektrum*, 2009. Available from: http://www.sigs.de/publications/js/2008/01/hennig_seeberger_JS_01_08.pdf.

[IBM09] IBM Corporation. *ILOG JViews 8.6 Documentation*, 2009.

[IHH07] Oliver Ihns, Dierk Harbeck, and Stefan M Heldt. *EJB3 professionell*. dpunkt.verlag, 1st edition, 2007.

[JHK09] Rod Johnson, Juergen Hoeller, and Donald Keys. Spring reference documentation. Technical report, 2009. Available from: http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/pdf/spring-framework-reference.pdf.

[Joh07] Rod Johnson. Introduction to the spring framework 2.5. *TheServerSide.com*, October 2007. Available from: http://www.theserverside.com/tt/articles/article.tss?l=Introtospring25.

[Lip07] Martin Lippert. Von Steckdosen und Steckern. Technical report, Eclipse Magazin, 2007. Available from: http://www.martinlippert.org/publications/EclipseMagazin9-06-UnterDerHaube-Teil4.pdf.

[Mey99] Bertrand Meyer. *Object Oriented Software Construction*. Prentice Hall, 2nd edition, 1999.

[ML08] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform - Designing, Coding, and Packaging Java Applications*. Addison-Wesley, 2008.

[Rei09] Stefan Reichert. *Eclipse RCP im Unternehmenseinsatz*. dpunkt.verlag, 1st edition, 2009.

[SD10] Dieter Steinmann and Jaqueline Dechamps. BIAF: Die unternehmensweite Berichtsplattform. Februar 2010.

[SUN02] SUN Microsystems. Java message service specification. Technical report, 2002. Available from: `https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_Developer-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=7195-jms-1.1-fr-spec-oth-JSpec@CDS-CDS_Developer`.

[SUN09] SUN Microsystems. Java web start guide. Technical report, SUN Microsystems, 2009. Available from: `http://java.sun.com/j2se/1.5.0/docs/guide/javaws/developersguide/contents.html`.

[Ull08] Christian Ullenboom. *Java ist auch eine Insel*. Galileo Computing, 7th edition, 2008.

[W3C04] W3C. Web services architecture. Technical report, 2004. Available from: `http://www.w3.org/TR/ws-arch/`.

[ZLM08] Alfred Zeitner, Birgit Linner, and Martin Maier. *Spring 2.5 - Eine pragmatische Einfuehrung*. Addison-Wesley, 2008.