

Eclipse as client container for J2EE applications

Hochschule der Medien

University of Applied Sciences

Stuttgart, Germany

Diploma thesis in study course Medieninformatik

Examiners:

1. Prof. Dr. Edmund Ihler, Hochschule der Medien, D-70569 Stuttgart
2. Dr. Christian Wege, DaimlerChrysler AG, D-70567 Stuttgart

Philipp Schill

Stuttgart, February 21, 2005

Statement

Hereby I declare that I have written this diploma thesis independently and without any other resource or literature than indicated throughout this thesis.

date

signature

Acknowledgements

I wish to thank Professor Dr. Edmund Ihler for his supervision during the course of this diploma thesis.

Furthermore, I want to thank Dr. Christian Wege, DaimlerChrysler. During the last six months, Chris gave me excellent supervision and support. I want to thank for the long discussions about the Eclipse technology in general and its application in this diploma thesis in concrete.

I'm also grateful to Jürgen Landwehr for his great support. He helped me starting with the client container and gave me many valuable advices concerning the deep implementation details. Also many thanks to the complete DaimlerChrysler IAP team. Whenever a problem occurred, a team member had a suitable solution.

Stuttgart, February 21, 2005

Abstract

The Eclipse rich client platform as container for component-oriented plug-ins provides a framework to host plug-ins, which -concerning its look and feel- embed well in a client workstation. J2EE client container provide a runtime environment for applications, integrated in a multi-tier architecture and therefore have to access services Java 2 Enterprise Edition (J2EE).

Combining the two container approaches will create a new runtime environment for application clients, which appear in the user interface style of Eclipse and are able to take up the J2EE services.

This diploma thesis discusses concepts of combining Eclipse and the client container.

Contents

1	Introduction	1
1.1	DaimlerChrysler IAP	2
1.2	Structural Overview of this Diploma Thesis	2
2	Basics	5
2.1	Eclipse	5
2.1.1	Evolution	5
2.1.2	Eclipse to Close the Gap	6
2.1.3	Eclipse as Rich Client Platform	7
2.1.4	Eclipse as Plug-in Framework	9
2.2	IAP Client Container	9
2.3	Class Loading Concepts	12
2.3.1	General Aspects	12
2.3.2	Eclipse	13
2.3.3	Client Container	14
2.4	Pattern Format	15
3	The Problem of Container Composition	17
3.1	Own Implementation	17
3.2	Container Composition	17
4	Solution A: Eclipse Inside the Client Container	21
4.1	Context	21
4.2	Problem	21
4.3	Solution	22
4.4	Structure	22
4.5	Participants and Responsibilities	22
4.5.1	Client Container	22
4.5.2	Application Server	23
4.6	Strategies	23
4.6.1	Eclipse Adaptations	23
4.6.2	Client Container Adaptations	24
4.7	Consequences	24

5	Solution B: Client Container Inside Eclipse	27
5.1	Context	27
5.1.1	Interoperability	27
5.1.2	Extendability	27
5.1.3	Security	27
5.1.4	Maintenance	28
5.1.5	Manageability	28
5.2	Problem	28
5.3	Forces	29
5.4	Solution	29
5.5	Structure	30
5.6	Participants and Responsibilities	30
5.6.1	Eclipse	30
5.6.2	Application Server	31
5.6.3	HTTP-Update-Server	32
5.7	Strategies	32
5.7.1	Authorization	32
5.7.2	Deployment Manager	34
5.8	Consequences	34
6	Installation Guide	37
6.1	Introduction	37
6.1.1	Notations in this Chapter	37
6.1.2	Common Directories	37
6.1.3	Prerequisites	38
6.2	Eclipse Client Container Example Installation	38
6.2.1	Client-side Installation of Demo Application	39
6.2.2	Server-side Installation of Demo Application	40
6.2.3	Starting the Demo Application	42
6.3	Setting up an Eclipse IDE	44
6.3.1	Preparations	45
6.3.2	Launch Configuration	45
7	Tutorial: Writing Applications for the Eclipse Client Container	49
7.1	Introduction	49
7.1.1	Structure of this Chapter	49
7.1.2	Prerequisites	49
7.1.3	Overview	49
7.2	Writing a Rich Client Application	50
7.2.1	A "Standalone" RCA	51
7.2.2	Changes to a Traditional RCA	57
7.3	Writing an Application Client Plug-in	59

7.3.1	Implementation	59
7.3.2	Best Practices	61
7.3.3	Packaging to a Feature	62
7.4	Set Up an Eclipse Update Site	63
7.4.1	Preparations	63
7.4.2	Add Features to the Update Site	64
7.4.3	Setting the Update Site in Eclipse Client Container	64
7.5	Closing the Chapter	64
8	Implementation Details	67
8.1	Changes from "Standalone" to Eclipse Client Container	67
8.1.1	No Changes in Eclipse code	67
8.1.2	No Changes on Server-side	67
8.1.3	Only as Many Changes as Necessary in Client Container Code	68
8.2	Porting Swing GUI to SWT and Adding to the Client Container	68
8.3	Implementing a New Plug-in Launcher	72
8.3.1	The Launcher in the Standalone Client Container	73
8.3.2	The Eclipse Plug-in Launcher	74
8.3.3	Changes from Standalone to Eclipse	75
8.4	ECC Deployment Manager	77
8.4.1	Searching for Features	77
8.4.2	Installing Features	78
9	Summary	81
9.1	Approaches	81
9.1.1	Eclipse Runs in Client Container	81
9.1.2	Client Container Runs in Eclipse	81
9.2	Study Results	82
9.3	Conclusions	83
9.3.1	Combining Eclipse and Client Container	83
9.3.2	No Hesitation, but Courage	84
9.4	Future Work	84
9.4.1	User Context with Eclipse 3.x	84
9.4.2	Updating the Client Container Plug-in	85
A	Implementation of the ECC Update Manager	87
B	Accompanying CD-ROM	91
	Glossary	93

List of Figures

2.1	Eclipse architecture [10]	10
2.2	Client container in production environment [21]	11
2.3	Class loader hierarchy in general	12
2.4	Class loader hierarchy in Eclipse	13
2.5	ClassLoader hierarchy of client container	15
4.1	Collaboration: Eclipse runs inside client container	23
5.1	Collaboration: Client container runs inside Eclipse	31
5.2	Sequence diagram of startup	32
6.1	Workbench launch configuration settings	46
6.2	Selection of plug-ins to launch	47
7.1	RCA with and without ECC	50
7.2	Involved classes while creating an Eclipse RCA	51
7.3	Wizard for creating a new plug-in project	51
7.4	A plug-in directory and file structure	52
7.5	Definition of an extension point	52
7.6	Needed plug-ins to run a basic RCA	54
7.7	The created RCA	57
7.8	RCP constellation without and with ECC	58
7.9	Two approaches	62
8.1	Login sequence	69
8.2	Callback references handlers	69
8.3	Class structure after porting	72
8.4	Tasks of the standalone launcher	73
8.5	Tasks of the Eclipse plug-in launcher	75
8.6	The launcher class structure	76

Listings

7.1	Base class of a RCP	53
7.2	Extension point entry in plug-in manifest	53
7.3	A simple perspective	55
7.4	Perspective entry in plug-in manifest	55
7.5	A simple workbench advisor	56
7.6	The final RCP	56
7.7	A rich client for Eclipse client container	59
8.1	A Swing handler method	69
8.2	A SWT handler method	71
8.3	Swing-SWT-Wrapper class	71
8.4	Patch for the InstallCommand	78
A.1	Searching for new features	87
A.2	Installing new features	88

1 Introduction

"With the exception of music, we have been trained to think of patterns as fixed affairs. It's easier and lazier that way, but, of course, all nonsense. The right way to begin to think of the pattern which connects is to think of a dance of interacting parts, pegged down by various sorts of limits."

Gregory Bateson - Cultural Anthropologist

This diploma thesis presents concepts to combine two containers: Eclipse and the J2EE client container.

Nowadays, Eclipse is known as an *Integrated Development Environment* (IDE), especially for the Java programming language, and as a platform for tool integration. The recently released version 3 of Eclipse introduces a new concept which developers from all over the world demand since Eclipse has been turned to an open-source product.

It extracts the benefits of the Eclipse core and provides an IDE-independent general purpose platform to create rich client applications. Those rich client applications run on the client-side and make use of the benefits of an application- and interaction model, based on the Eclipse component architecture. Developers now have the possibility to create applications that profit from the well-known features of Eclipse.

At the same time as the concept of Eclipse, another solution for building client applications has come up.

Complex application infrastructure is typically based on a multi-tier architecture mainly based on the Java 2 Enterprise Edition (J2EE). J2EE client containers form the equivalent to a server-side container, e.g. an Enterprise Java Bean (EJB) container. They provide a runtime environment for application clients, which can benefit from the underlying container framework. Those containers provide a security concept and features for accessing server-side components such as EJB or Java Servlets.

Both approaches provide a client-side container for applications. While the two container approaches compete concerning their concept, they complement each other concerning their features. Combining the two containers will provide a platform for application clients with a native look and feel based on a component architecture on the one hand, and a runtime environment that facilitates the access to J2EE services on the other hand.

This diploma thesis will present concepts, how client applications can profit of both approaches. It shows two patterns to connect Eclipse and the client container.

When combining the two containers, one of them has to open its encapsulation boundaries and subordinate itself to provide its services to the leading container. The thesis will detect the connecting points where opening the encapsulation does not weaken the entire architecture. Furthermore, the thesis assists to balance the forces, that is, it gives a decision guidance which approach suits best in a specified use case.

1.1 DaimlerChrysler IAP

This diploma thesis has been designed and developed at DaimlerChrysler in Stuttgart, Germany. The department ITI/TP develops a platform called *Integrated Application Platform* (IAP), that offers "the basic infrastructure for application projects" [22]. The IAP architecture is the base for applications and solution platforms developed on top of IAP and "allows projects to concentrate on the business application development rather than on infrastructure creation" [23]. IAP differentiates between platforms such as J2EE, Portal, Directory, Security, each containing numerous components such as authorization, logging, directory access and others.

One specific component of the J2EE platform is the IAP client container used in this diploma thesis to practically realize the combining approaches. Differences to the J2EE client container can be found in section 2.2. The practical realization of the thesis, the Eclipse client container (see chapter 5), will become part of the IAP release 3.0 and therefore takes care about the components' reusability developed in the thesis.

1.2 Structural Overview of this Diploma Thesis

Chapter 1

gives an introduction in the diploma thesis.

Chapter 2

describes basics about Eclipse and the client container. It also gives an introduction to the concept of class loading. After presenting the general concept of Java, the class loading concepts of Eclipse and the client container are described.

Chapter 3

discusses the problems when combining two container approaches. It shows the barriers, a developer is confronted with, if he decides to use features of two different containers.

Chapter 4

presents the first approach: Eclipse inside the client container. It points out circumstances when this approach should be applied and describes a concrete solution.

Chapter 5

describes the second approach: client container inside Eclipse, forming the *Eclipse client container* in the next chapters. It defines the starting point when to apply this approach and presents a suitable solution.

Chapter 6

contains an installation guide which describes the steps that have to be done to install the Eclipse client container. Furthermore it contains a tutorial of how to implement concrete application client plug-ins.

Chapter 8

discusses implementation details. It describes how to add a new GUI toolkit to the client container, presents solutions for the concrete launcher problem and introduces into the implementation of the ECC Deployment Manager (the manageability component of the second approach).

Chapter 9

sums up and discusses the results of this diploma thesis. After that, it gives an outlook what can be done in the future.

The appendix contains additional material like some source code listings, an accompanying CD-ROM, the glossary and the bibliography.

2 Basics

This chapter gives an introduction to the two components involved in this diploma thesis. It provides an overview about Eclipse, shows its roots and gives a short introduction of its key features and mechanisms. Then the chapter describes the second key technology involved in this diploma thesis: the client container. Finally, the chapter explains class loading concepts of Java in general and class loading concepts of the client container and Eclipse.

2.1 Eclipse

The Eclipse project [12] defines Eclipse as "an open extensible *Integrated Development Environment* (IDE) for anything and nothing in particular" [12]. Since release 3, Eclipse provides another possible application in addition to the use as IDE: the Eclipse project introduces a platform for hosting rich client applications, based on the Eclipse framework. First, this section describes the circumstances why Eclipse has been developed. Second, it shows application possibilities such as integrated development environment and rich client platform.

2.1.1 Evolution

In the last 10 years a huge number of new technologies were born, especially in the area of the Java programming language. New approaches such as Enterprise Java Beans (EJB), Java Server Pages, Java Servlets, WebServices and others came up.

Each technology is located on different layers of abstraction. Enterprise Java Beans for example, require a large number of configuration parameters and need a generator engine to create the implementation classes automatically. Unlike to EJBs, Java Servlets are more low-level and define an application programming interface (API), which must be implemented by the developer. Each new technology comes along with its own editor to configure configuration files or to create implementation classes. Most software projects are implemented with more than one technology. This heterogenic technology landscape leads to a complex development environment. Numerous editors are involved in the development process and therefore unfortunately prevent collaboration between the different involved members of the development team.

2.1.2 Eclipse to Close the Gap

At this point, Eclipse suits best with its "open platform for tool integration" [13]. The idea of Eclipse is to provide an "open extensible IDE" [12] which "give developers freedom of choice in a multi-language, multi-platform, multi-vendor environment" [13]. Eclipse is open to integrate all kind of editors involved in a software development process to provide one single and integrative platform. This approach promises to "save time and money" [13] as all team members work with the same IDE. The Eclipse IDE introduces a basic collection of tools:

- Java Development Toolkit (JDT)
"The JDT provides tools [...] which support the development of any Java application [...]" [15]. It includes editors for Java classes, code creation wizards and refactoring tools. Additionally, it provides a debugger which supports hot code replacement.
- Plug-in development environment (PDE)
"The PDE project provides a number [...] of editors that make is easier to build plug-ins for Eclipse" [15]. It provides tools to simplify the extension of the Eclipse platform (The plug-in mechanism is explained in detail below).

Eclipse offers a number of advantages:

- Native Look and Feel
Eclipse is implemented with an own graphical user interface (GUI) toolkit. As traditional Java applications make use of AWT or Swing, Eclipse uses its own implementation: the *Standard Widget Toolkit* (SWT). SWT provides a system-near look and feel that resorts to the GUI components of the underlying operating system.
- Performance
Due to the use of a system-near GUI toolkit and numerous performance patterns such as lacy loading, Eclipse is able to operate faster than other Java applications.
- Extensibility
Eclipse itself is written in the Java programming language. The plug-in architecture of Eclipse is the most important characteristic. Eclipse is completely built on plug-ins. It is possible to create hierarchical systems, where plug-ins of a higher level access functionality of a lower level. If a tool or functionality is missing it can be easily added as a new plug-in to Eclipse.
- Large open-source community
Eclipse is published under the open-source common public license (CPL) model. A large community of developers and committers constantly improves the Eclipse IDE by providing additional functionality in form of plug-ins. For almost every technology, a plug-in can be found. Plug-ins exist for the Java programming language and also for C/C++, Perl or PHP.

Over the last two years, Eclipse has become more and more popular. Software teams have discovered, that the application of Eclipse provides an added value with respect to development time and project costs. Furthermore, developers realized that the plug-in mechanism can also be used for non-IDE applications. A lot of feature requests asked for the decoupling of the IDE parts, so that Eclipse can be used for standalone applications and not deal with development environments.

In Eclipse release 2, some IDE specific parts were closely coupled with the Eclipse runtime (e.g. the menu item "Project" was an integral part of the Eclipse core). It was impossible to publish applications based on the Eclipse framework without any useless IDE remains.

2.1.3 Eclipse as Rich Client Platform

Since release 3, the IDE specific parts are decoupled from the Eclipse core and a platform is extracted providing now the proven advantages mentioned above. The result is the rich client platform (RCP) allowing the creation of rich client applications based on Eclipse. Before describing the Eclipse RCP more detailed, the term rich client application has to be defined first.

Characteristics of Rich Client Applications

At the beginning of a project, the development team needs to decide which technologies will be used. Apart from an architectural decision on server-side, the development team has to evaluate the different technologies on the client-side. They have to decide, whether to implement a browser-based thin client, or use standalone rich client. This section points out some basic characteristics of rich client applications which help to come to a decision.

Dirk Bäumer ([6] and [7]) defines the following characteristics for a rich client application:

- Native widgets and toolbars
Rich client applications use the services of the underlying operating system to show the user interface of an application in a native look and feel.
- Integration
A rich client can interact with the platform component model. Especially the mechanism of object linking and embedding (OLE) can be used by those clients to import objects from other vendors.
- Drag&drop
A rich client platform should support the drag&drop functionality.

Applications implemented as rich clients offer some advantages:

- More responsive user experiences
Rich clients can better react to user inputs. Browser-based thin clients have to handle interaction mostly with Java Script.
- Improved integration
Rich clients use own system resources to create the UI. If rich clients rely on the GUI toolkit of the underlying operating system, desktop tools can be integrated in an easier way.
- Offline execution
Rich clients do not need a network connection to the server. Browser-based thin clients need always a connection, while rich clients may operate offline and can execute their jobs when connected to the server.
- Lower server loads
Some operations can be moved from the server to the client while enhancing the server response time.

The Eclipse Rich Client Platform

The RCP generalizes Eclipse as "platform for tool-integration" [9] to "an extensible platform for any application" [10]. The RCP consists of a minimal subset of Eclipse plug-ins not containing any IDE specific parts.

Advantages from implementing applications based on the Eclipse RCP:

- Applications use the plug-in mechanism. Plug-ins can add new functionality or extend already existing functionality.
- Applications use the system-near GUI toolkit SWT to provide a look and feel of the underlying operating system.
- Applications use the workbench, perspectives and views to build a complex but concise user interface.
- Applications provide a good user help with the Eclipse help system.
- Applications are manageable with the help of the Eclipse update manager. The Eclipse update manager is a component which can easily download and install new applications or update already installed applications.

2.1.4 Eclipse as Plug-in Framework

Eclipse is entirely composed of so-called plug-ins. A plug-in can be defined as a component (Dirk Bäumer defines a component as the smallest unit in a system concerning data and code [7]) and "is the fundamental building block of the Eclipse platform" [14]. "Eclipse is the sum of its constituent plug-ins, where each plug-in contributes functionality to the platform and is activated when the functionality it provides is needed." [14]

The Eclipse plug-in architecture defines a life cycle protocol. Each deployed plug-in lives through its own life cycle. It is installed and activated to the system by the deployment manager. It will be started when the user begins to utilize functionalities of the plug-in. If Eclipse is shut down, all installed plug-ins are stopped. The Eclipse runtime uses this life cycle protocol to interact with the installed plug-ins. The runtime is able to start and stop plug-ins automatically (startup and shutdown) and also provides an API to install, start and stop plug-ins programmatically.

Furthermore, the Eclipse plug-in mechanism offers benefits that can simplify the creation of complex applications where own created plug-ins use already existing plug-ins. The key mechanisms are

- **Dependencies**
Access to a functionality of another plug-in is realized by building a dependency. This results in a proof of consistency, as the plug-in will not work if the dependent plug-in is not available. If a dependency does not exist, the access to the plug-in will be prevented. Namespaces are strictly divided.
- **Extension points**
Plug-ins may provide a basic service (e.g. a file browser). Additional functionality can be added to this service (e.g. additional file shortcuts) by extending the plug-in. This mechanism is implemented with the help of extension points.

Figure 2.1 shows a possible topology of a set of plug-ins in Eclipse. This kind of component composition reduces the writing of redundant code and the number of possible bugs in a plug-in. Due to dependencies and extension points, the implementation of glue code, which would make plug-ins able to interact with each other, is not needed.

2.2 IAP Client Container

J2EE Client Container

The specification of Java 2 Enterprise Edition (J2EE) 1.4 [19] defines a client-side container multi-tier infrastructure that provides a runtime environment for J2EE application clients. It encapsulates basic runtime services in a simple API, so that applications can use them without much programming overhead. A J2EE client container must meet the following requirements:

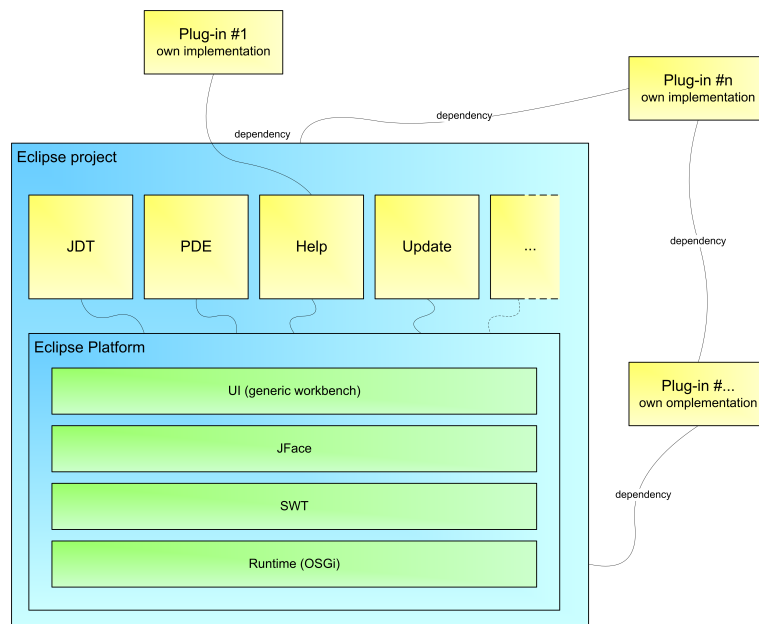


Figure 2.1: Eclipse architecture [10]

- Security
A client container provides a strong security concept with authentication and authorization. Each user, using a deployed application client, will be authenticated first. Based on username and password, he will be able to use the application in case he has the adequate user rights. A client container also provides a role-based authorization concept where actions within an application can be checked.
- Remote calls
J2EE client containers are able to talk with server-side containers. They support remote procedure calls (RPC) by using Enterprise Java Beans (EJB). The communication between client and server is handled via RMI-IIOP.
- Naming directory
Each client container has JNDI (Java Naming and Directory Interface) as naming directory. Inside this directory, runtime (e.g. application settings) and user information (e.g. username, role name, ...) are stored as well as EJB references for RPCs.

IAP Client Container

The IAP client container is a DaimlerChrysler proprietary implementation based on the J2EE specification for client containers. Apart from the features a client container must have according to the specification, the IAP client container provides additional features due to some DaimlerChrysler specific requirements. These additional features are:

- Multiple applications

A conventional J2EE client container is able to host only one single application client. Use cases in the DaimlerChrysler environment prescribe the execution of more than one application client in one instance of the Java Virtual Machine (JVM).

- HTTP(s) instead of RMI-IIOP

Conventional J2EE client containers use RMI-IIOP as transport protocol for EJB access. RMI-IIOP is a binary protocol and is preferably used for intranet communication. As DaimlerChrysler application clients are also installed outside the company network in the internet, communication through firewalls must be possible. Binary protocols are difficult to check with a firewall. Therefore, the IAP client container uses HTTP or HTTPs as transport protocol for RPCs. In order to protect resources on the application server, the authentication mechanisms HTTP BASIC or DIGEST can be used.

- Server-side security

IAP client container does not authenticate users on the client side. It uses the Java Authentication and Authorization Service (JAAS) to receive username and password, and sends them to the server component. The server-side security component does authentication of the user. A UserInformation (Session-) Bean collects user data and provides it to the client container.

- Stateless Session Beans

The J2EE specification describes only Stateful Session Beans as mandatory for a client container. Application clients deployed in the IAP client container can also use Stateless Session Beans.

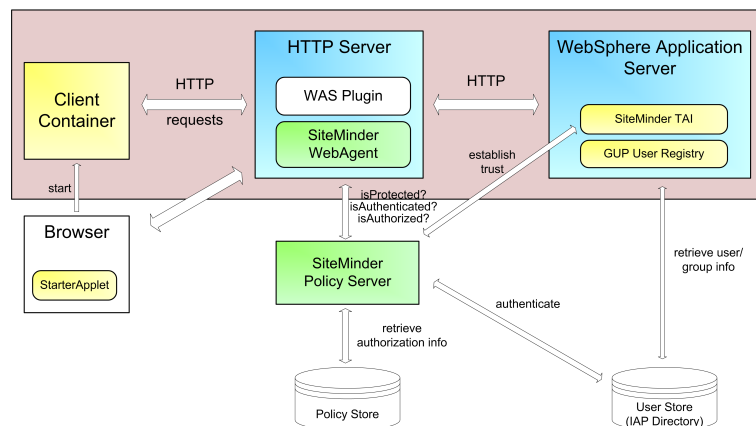


Figure 2.2: Client container in production environment [21]

Figure 2.2 shows a client container infrastructure in a production environment. A client container communicates with an application server. In case of the DaimlerChrysler IAP environment, the IBM Websphere Application Server 5 is used on the server-side. In a production

environment, the web access management software Netegrity Siteminder is used to authenticate and authorize users who want to work with application clients.

2.3 Class Loading Concepts

2.3.1 General Aspects

Applications, written in the Java programming language, use class loaders to reify classes from a physical location (hard disk, database, ...) to the memory and provide them to the Java runtime. Classes are load with the help of three or more different class loaders each with a super-ordinate/subordinate relation each.

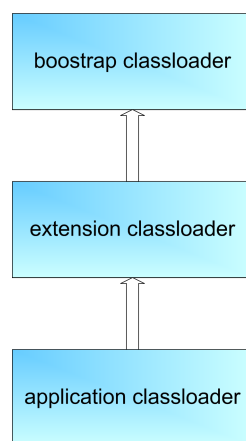


Figure 2.3: Class loader hierarchy in general

The root class loader is called bootstrap class loader and is responsible for loading all Java system classes contained in the library `rt.jar`. This class loader is part of the Java virtual machine and is mainly written in C. It has no parent class loader.

The child of the bootstrap class loader is the extension class loader. It undertakes the task of loading classes or libraries located in the `jre/lib/ext` directory. Java libraries, containing classes that should be globally available in all applications, can be placed there.

Below the extension class loader, the application class loader (also called system class loader) loads classes from the class path. This class loader is responsible to load the current Java application and all its appropriate Java libraries.

By convention, the lowest class loader in a hierarchy has to ask its parent to load the class first. It does not load the class itself until all super-ordinate class loaders are not able to load the class. This convention of asking a parent class loader is not mandatory. Some use cases ask for a flat class loader hierarchy which forces class loaders to load the Java byte code itself.

Figure 2.3 shows default configuration of the class loader hierarchy. Developers can also implement their own class loaders. This concept can be used e.g. to load Java code from a database over a network into the Java virtual machine. Other implementations are used to read encrypted byte code from the hard disk. This approach can be used to block Java code for which a user has not paid.

2.3.2 Eclipse

Eclipse provides a framework for hosting plug-ins. These plug-ins can be developed and published independently by different vendors [11]. All plug-ins are strictly separated from each other to prevent namespace clashes and side effects between two plug-ins. Two plug-ins can contain a class called FooBar.java, which is located in a package structure with same names. Without a strict plug-in separation, the Java virtual machine would find two classes with the same name within the same package. Eclipse has to divide one namespace into different sub-namespaces, one for each plug-in. This concept is implemented with the help of an own class loader hierarchy as shown in figure 2.4.

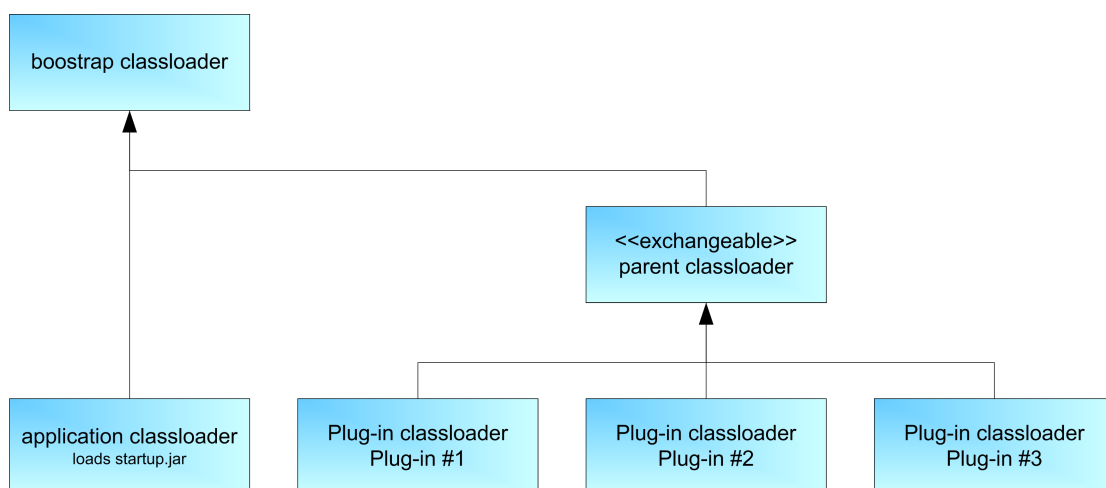


Figure 2.4: Class loader hierarchy in Eclipse

An application (system) class loader is used to load the main class of Eclipse and all classes of the library startup.jar. Furthermore, Eclipse provides a separate class loader for each plug-in, that has a parent, called plug-in boot class loader which is now configurable in release 3 of Eclipse.

In previous releases (v2.x) the class loader hierarchy could only be changed by modifying the source code of the org.eclipse.core Eclipse plug-in. This was a critical and painful way, because developers had to identify the location where to change the class loader hierarchy and they had to rebuild parts of the Eclipse code. Since version 3, Eclipse uses the OSGi framework. The OSGi specifications "define a standardized, component oriented, computing environment". It "adds

the capability to manage the life cycle of software components" [17] (in this case Eclipse plug-ins). The OSGi framework replaces the previously self-developed component framework by a standardized approach. By introducing the OSGi framework, the class loader hierarchy becomes configurable in the OSGi properties file *config.ini*. The parameter called *osgi.parentClassLoader* helps to set the parent class loader of a plug-in class to one of the four class loaders:

- Boot (default)
The plug-in boot class loader has the bootstrap class loader as parent class loader.
- Fwk
The parent class loader can be set to the OSGi framework class loader.
- Ext
The extension class loader which loads classes from the *jre/lib/ext* package can be the parent class loader.
- App
The parent class loader can be set to the application class loader. Due to this, plug-ins have access to the standard class path being defined before starting Eclipse.

The advantage of introducing this parameter is, that no changes in the source code are required. Eclipse can be installed as provided from the Eclipse homepage.

As figure 2.4 shows, plug-ins cannot reference to classes, which are located in the conventional class path, because there is no connection between a plug-in class loader and the application class loader. Indeed, all class loaders have the same super-parent class loader, but the delegation direction is always upwards. The application class loader is not part of the chain up to the bootstrap class loader [1]. In order to be able to reference a class from another plug-in, the two have to be connected via a dependency.

2.3.3 Client Container

The client container implements a related class loader concept like in Eclipse, which is shown in figure 2.5. It provides a separated namespace for each deployed application. An extra class loader called *APPApplicationClassLoader* is provided to prevent namespace clashes. Each application is located inside a sub-directory of the launcher directory. *APPApplicationClassLoader*, child of the application (system) class loader, loads classes (or jar files) from this sub-directory. The class loader structure is more flat than the Eclipse structure. Furthermore, the client container class loader hierarchy does not support connecting two applications, like Eclipse does with its dependency mechanism.

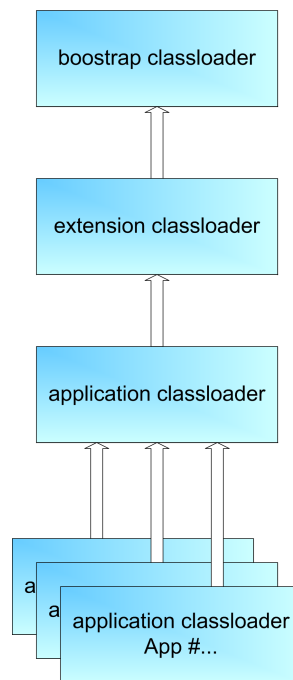


Figure 2.5: Classloader hierarchy of client container

2.4 Pattern Format

This diploma thesis presents two patterns to combine Eclipse and the client container. A pattern generally "describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [26]. Of course, the patterns, discussed here, are tailored to the specific problem, but the ideas are portable to other challenges, so it can be seen as "a description of a solution to a problem found to occur in a specific context" [27].

The two chapters use the pattern format of Gerard Meszaros and Jim Doble [27] to present the two solutions. They are divided into eight sections. The sections *Context*, *Problem*, *Forces*, *Solution* and *Consequences* are defined by Meszaros/Doble. The three other sections *Structure*, *Participants and Responsibilities* and *Strategies* are extensions (inherited of Java's Core J2EE Pattern Catalog [28] and GoF's Design Patterns [29]) that support a better understanding of the proposed solutions.

The meaning of those sections is described in the following:

- **Context**
"The circumstances in which the problem is being solved [...]" [27]. The context describes the situation, a developer is in, when he chooses to apply this pattern.

- Problem
"The specific problem that needs to be solved" [27].
- Forces
"The often contradictory considerations that must be taken into account when choosing a solution to a problem" [27].
- Solution
"The proposed solution to the problem" [27]. It may also "resolve some forces" but can also ignore some.
- Structure
The structure section gives a graphical overview about the proposed solution.
- Participants and Responsibilities
In the case of this diploma thesis, this section presents the components involved in the solution and its functions.
- Strategies
The strategies section discusses abstractly some implementation details without showing any source code.
- Consequences
"The context that we find ourselves in after the pattern has been applied".

3 The Problem of Container Composition

A container presents a runtime environment for components. Based on this runtime environment, developers can create components that may access container features, instead of implementing those features independently.

Sometimes, use cases define requirements, which cannot be fully covered in the application of one container. It is assumed, that the selected container supports remote access to an application server, but does not provide any security concept to protect the server resources against unauthorized access. A software engineer has to differ between two solutions.

3.1 Own Implementation

As a first approach he could decide to implement the missing functionality on his own. He would have to get the container source code and would have to find those locations, where to add the new features. The implementation of the container does not provide any entry point to extend the functionality, that is, he could have to refactor parts of the existing source code. Following this, he would have to provide a product based on an adapted container with the consequence, that he would have to maintain his components and additionally the own implementation of the container.

3.2 Container Composition

In a second approach he could try to find ways to combine the selected container with another one, which provides the missing features. However, this approach is also associated with some problems. Both containers are independent products, which encapsulate themselves strongly against their environment. Furthermore, the two containers are not designed to be extended to communicate with other containers. Due to this non-extensibility, each container "codifies and prescribes the policies and rules for interactions" [25] and therefore claims to be the leading container in the new structure. Fayad, Schmidt and Johnson [24] describe an example in this context: The composition of a measurement and a GUI framework. The measurement framework receives an event and processes it in its well-defined control loop. Also the GUI framework has a thread of control that updates the screen whenever a value changes. These control loops can easily collide with each other "causing the measurement part to miss its real-time deadlines and causing the GUI to present incorrect data due to race conditions between the activities".

To overcome this obstacle, concepts need to be evaluated to minimize the claim to leadership and to provide a possibility to interact with other containers. In any case, connecting two frameworks will cause a programming overhead. However, the complexity of this approach is obviously less than implementing the missing features oneself.

Fayad, Schmidt and Johnson [24] differ between two problems: Problems occur with respect to the "overlapping features" of containers, that means, "both contain a representation [...] of the same real-world entity. If the represented properties are mutually exclusive and do not influence each other, then the integration can be solved by multiple inheritance".

Furthermore, software engineers have to find solutions for integrating the "non-overlapping features". There are different approaches to solve this problem.

The Delegation Approach

Fayad, Schmidt and Johnson [24] propose to run the two containers equally and in parallel and to connect them with a dedicated connection framework, that is to implement a delegation approach. Whenever a state changes in one container, the connection framework delegates the state change to the other.

The delegation approach means a closely coupling of the three frameworks. Developers have to force one framework to provide the state change to the connecting framework, that is, they still have to find an entry point into the framework and to extend it to delegate the change information to the connecting framework. Furthermore, the connection framework handles not only output of one framework, but also has to input the state change information in the other framework.

The Event Approach

D'Souza and Wills [25] do a step beyond the delegation approach. They propose, that two frameworks talk via event-notification with veto possibility. Therein, a framework announces its state changes and asks for any veto. Only if the second framework agrees, the action is performed. The event approach means a more losley coupling of the frameworks. Event notification can happen e.g. via WebService. Developers have to add components on each framework side, providing their change information to the other framework. In contrast to the delegation approach, the implementation of an additional connecting framework unnecessary.

The Containment Approach

This diploma thesis sorts out another concept of combining containers. As the two container approaches complement each other with regard to their features, this approach mainly cares about problems of the non-overlapping features.

The concept discussed in this thesis, uses the containment approach to combine the two frameworks. While delegation and event approach try to run the frameworks in coexistence, the containment approach forces one framework to abandon its claim to leadership and to run inside the other. Developers mainly have to adapt only the inner framework to provide its features to the outer framework. Thus, components running inside the main framework have automatically access to the features of the subordinated framework.

Chapter 4 discusses the solution "Eclipse runs inside the client container". That is, the Eclipse environment is forced to provide the features of the surrounding client container.

Chapter 5 ("client container runs inside Eclipse") presents ways to adapt the client container to provide its features to other Eclipse plug-ins.

4 Solution A: Eclipse Inside the Client Container

This chapter describes the first implementation how the two container approaches can be combined. Section 4.1 describes circumstances, when a developer should think about applying this solution. The following section 'Problem' and 'Forces' (section 4.2) defines the problem and consequences when choosing this approach. The 'Solution' (section 4.3) shows the problem solving and discusses in the following section the implementation details. The section 'Consequences' (4.7) talks about the solution's advantages and disadvantages.

To understand the structure of this chapter, the reader should have read section 2.4.

4.1 Context

The client container hosts already application clients. The clients are based on Swing and use the client container services. It is not intended to migrate old application clients. New applications should base on the Eclipse RCP and be deployed as a plug-in to Eclipse together with all other applications in one single client container. Such plug-ins benefit from Eclipse (e.g. a native GUI) and the client container.

Before starting the client container, a user will need an authorization to use Eclipse and all other application clients. The principle of a single-sign-on to client container (login once and use all applications) should be conserved. Thus, Eclipse and its deployed application client plug-ins should be installed as one whole application into the client container. Eclipse has to be modified in a way that plug-ins can profit from the client container features.

4.2 Problem

The features of both approaches are complementary to each other: the client container provides a security concept with authentication and authorization and a runtime environment for application clients, which can easily access a naming directory or communicate with server-side components like EJBs.

Eclipse provides a framework for applications that profit from a native GUI toolkit and from an extensible framework, based on a plug-in concept.

Concept wise they compete with each other. Each container protects its features with a strong encapsulation against its environment. This encapsulation prohibits other locations, such as subordinated applications, running inside the container to access internal features.

4 Solution A: Eclipse Inside the Client Container

Use cases prescribe a client application with a native look and feel, but require also a strict business logic on the server-side. The challenge is how the two containers can be combined, so that deployed applications clients, implemented as Eclipse plug-ins, can profit from the services of the client container and of Eclipse.

Eclipse as a self-contained system, is deployed into the client container. Eclipse needs an adaptation to enable plug-ins to use client container features. The deployment of Eclipse should not disturb already deployed applications. The client container changes concern only the interaction of Eclipse with the client container.

4.3 Solution

The client container takes the leader role in this solution. Eclipse cannot claim anymore to be the major container. The Eclipse container with its hosted application client plug-ins is deployed as one single application into the client container. Eclipse has to weaken its encapsulation boundaries to enable the client container to provide its features to application client plug-ins.

Eclipse runs in the same namespace to have access to object instances, created by the client container. Additional architectural changes in Eclipse are needed to solve this problem. To provide client container features inside Eclipse, the Eclipse class loading hierarchy has to be changed in a way that Eclipse plug-ins have access to the client container object instances. Details are described in section 4.6.

Consequently, all JAR files, containing the client container components, are included in one Eclipse plug-in. A plug-in dependency of the application client plug-in and the jar-file-plug-in makes the classes available.

4.4 Structure

Figure 4.1 shows the structure of this solution. Eclipse is deployed into the client container as an equal parallel application to other standalone application clients. The client container features are globally available in Eclipse.

4.5 Participants and Responsibilities

4.5.1 Client Container

The client container is the major container in this context. It hosts traditional standalone application clients (e.g. based on a self-programmed Swing GUI) as well as the Eclipse Rich Client

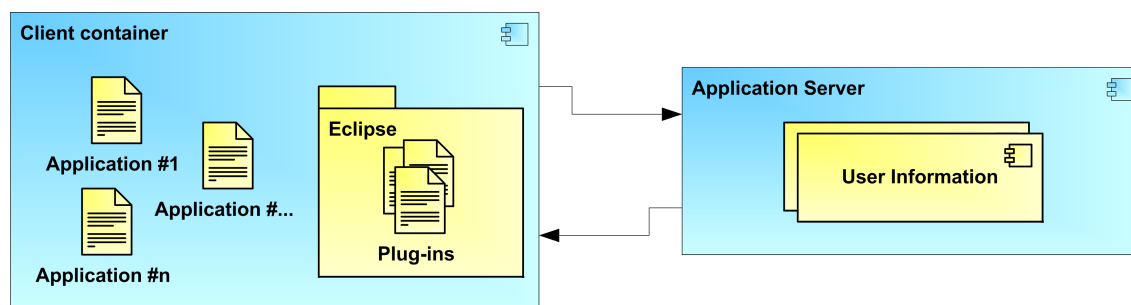


Figure 4.1: Collaboration: Eclipse runs inside client container

Platform. Due to the global availability of the client container, services can be accessed from all areas in Eclipse.

4.5.2 Application Server

The application server is the server-side container for client applications. It is responsible for user authentication and authorization. It provides user information to an application client, sessions and roles for a specific user. The application server hosts also Enterprise Java Beans providing the business logic for the application clients.

4.6 Strategies

As mentioned above, adaptations are needed to make client container services available inside an Eclipse application client plug-in. Therefore, the Eclipse configuration needs to be adapted before the client container services can be used.

4.6.1 Eclipse Adaptations

Client container classes are loaded with an application (system) class loader (details see chapter 2.3.3). Eclipse plug-ins must get access to the client container object instances. The current Eclipse class loader structure prevents however such an access. The solution is to replace the plug-in parent class loader with the application class loader which is part of the common class loader hierarchy. With Eclipse release 3.x, the class loader hierarchy became configurable. This means for this concept, the parent class loader is being moved from the plug-in class loader to the application (system) class loader (*osgi.parentClassloader=app*). The client container uses also an application class loader as parent. Both are located on the same hierarchy level. Access to the client container classes is now possible.

4.6.2 Client Container Adaptations

Due to the changed Eclipse class loader structure, the session data, provided by the client container, can be accessed by every plug-in.

Class loading conflicts appear only, if an application client plug-in wants to access a server-side Enterprise Java Bean. During the startup, the client container loads all available EJB stubs of the application into the JNDI directory. Applications can access the stubs to communicate with the server. The application class loader of the client container creates the stub instances.

If an Eclipse application client plug-in requires access to a server object (EJB), it narrows a home object reference from the JNDI directory. Despite this, all application client plug-in classes are loaded with the Eclipse plug-in class loader (see 2.3.2). If a plug-in wants to instantiate an EJB home object reference, a class cast exception is thrown. A class (in an application client plug-in), loaded by the Eclipse plug-in class loader, retrieves a remote object reference, loaded by the client container application class loader. Using the Eclipse plug-in class loader for providing the remote object reference to the plug-in can solve the problem.

A context class loader can be set (*setContextClassLoader()*) during a running thread. "The context class loader is provided by the creator of the thread for use by code running in this thread when loading classes and resources. If not set, the default is the class loader context of the parent thread. The context class loader of the primordial thread is typically set to the class loader used to load the application." [20]

From every point inside the code the context class loader can be retrieved with a static method (*getContextClassLoader()*) of the thread class.

The client container provides an opportunity to use a context class loader for creating EJB references. A VM parameter needs to be set before the client container can use the context class loader. The parameter *-Ddcx.integration.service.useContextClassloader* can be used without any assignment. The context class loader will generate from now on all stubs.

The client container starts Eclipse as an application client. Consequently, Eclipse runs within the same thread as the client container itself. However, before the client container can use the Eclipse application client plug-in class loader, the context class loader must be set. If this context class loader is set to the Eclipse plug-in class loader, the client container uses this class loader to provide an EJB reference. The result is, that the client container and Eclipse make use of the same (Eclipse) class loader and therefore no class cast exception is thrown anymore. Now plug-ins can have access to all features of the client container as all other applications, too.

4.7 Consequences

Eclipse and its application client plug-ins will be deployed commonly into the client container. All application client plug-ins inside Eclipse are equal: on this layer (plug-ins) authorization is not performed. Only an authorized user can handle security issues on application layer (Eclipse). Users are only authorized on the application layer (Eclipse), and not on the plug-in layer.

Eclipse runs inside the client container among other applications. Some infrastructures use Java WebStart or the IAP StarterApplet to distribute the client container and its applications. Whenever applications are downloaded with WebStart or StarterApplet the entire application, including the Eclipse core framework, is downloaded. This results in an overhead of 8 to 10 megabytes. However, there is no possibility to limit the download to the application client plug-ins.

The only way to run Eclipse inside the client container is an architectural change in Eclipse by changing the default class loader hierarchy. Eclipse was formerly known as a class loading framework. With these changes the primary idea of a strict separation of plug-in namespaces has been withdrawn. Instead of the plug-in boot class loader, the application class loader is now the parent class loader of each plug-in class loader. A consequence of this change is that the dependency mechanism has been abolished.

5 Solution B: Client Container Inside Eclipse

This chapter describes the second possibility to combine the two containers approaches. Section 5.1 describes circumstances, when a developer should think about applying this solution. Next, it will be described, what the problem (5.2) is, that needs to be solved and which contradictory considerations (5.3) must be taken into account. Section 5.4 describes the real approach. The last section shows the consequences when implementing this solution (5.8).

To understand the structure of this chapter, the reader should have read section 2.4.

As solution B is suitable for most application cases, the following chapters will talk about the *Eclipse client container* (ECC), which means this solution.

5.1 Context

5.1.1 Interoperability

The Eclipse client container will be introduced to an existing J2EE environment. To achieve a satisfactory result, applications should profit from a security infrastructure and from an already existing GUI framework. Therefore, a J2EE application client will be developed, based on a combination of the client container and Eclipse - the Eclipse client container.

5.1.2 Extendability

Several application clients have to run on the same platform. Furthermore, additional applications will be added to the existing client installation in the future. Therefore, the infrastructure must be able to host multiple applications.

5.1.3 Security

Several users might share one workstation. These users have different privileges to run applications and to execute jobs. A security infrastructure, has to protect applications against unauthorized access.

5.1.4 Maintenance

The Eclipse client container consists of two containers: the client container and Eclipse. As Eclipse is an externally developed product and the client container is a proprietary product of DaimlerChrysler, changes should be made only to the client container. No source code modifications should be made in the Eclipse distribution..

5.1.5 Manageability

The customers' support team continues evolving the application. Several revisions will be released over the time. Bugs and fix-packs will be packed and published. The customer requests a way to publish and install bug-fixes or new versions of its applications on the client workstations. Therefore, the system must be able to download and install updates and patches automatically.

5.2 Problem

The problem to be solved is how to couple the two container implementations. As the two approaches complement each other, only the combination of the two containers results in an environment, where developers do not have to implement missing framework services. If the developer would choose to base his product e.g. only on Eclipse, he would have to add the missing features usually provided by the client container. This could mean additional expenses and an increased error rate.

A well designed multi-tier architecture needs an established server-side framework and a stable application on client-side. It is important to establish a client container infrastructure to guarantee functional and non-functional requirements for a long-term application. Development costs will increase if developers do not care about a stable underlying framework.

Some use cases request a client environment where multiple applications can run in parallel. These applications must be deployed without disturbing the operation of already installed ones. A coexistence of the applications without interference and therefore a strictly separated runtime environment must be one of the key features of the Eclipse client container. Furthermore, it must fulfill non-functional requirements like reliability or availability. A single application should not be able to stop the whole container operation by throwing an exception. Clearly separated name spaces prevent security attacks.

5.3 Forces

Many client container approaches provide a runtime environment containing standardized components such as security, GUI toolkit or server access. Every container protects itself with a strong encapsulation against its environment. If two client container approaches should be combined, both will compete for the leading role. An interaction of two containers will force at least one to weaken its encapsulation. The container, which abandons its leading role, has to serve its features across its encapsulation boundaries to the other container.

Eclipse plug-ins do not have a user context. All plug-ins run as equivalent applications inside Eclipse. As security is mandatory in a J2EE environment, especially while using a client container, a user context is needed to authorize users to use plug-ins.

5.4 Solution

Eclipse becomes the major container in this configuration. The client container abandons its leading role and runs as peer Eclipse plug-in amongst others. It is adapted to use Eclipse GUI components to implement user interaction. It provides its components like security or remote server access to other Eclipse plug-ins which request these features.

Client container applications are no longer standalone applications. They are implemented as Eclipse plug-ins. From now on, these plug-ins are called *application client plug-ins*. All other plug-ins can access client container features via Eclipse plug-in dependencies to the client container plug-in.

Other plug-ins can also run inside the Eclipse client container. Not all installed plug-ins must be an application client plug-in. The Eclipse *Java Development Toolkit* (JDT) for example, can be installed amongst others in the container. Plug-ins, which should act as application client plug-ins, have to extend an extension point. This extension point marks the plug-in as application client plug-in. The plug-in receives additional runtime information like the application ID. Thus, the Eclipse client container is able to identify the deployed application client plug-ins.

Different to solution A (Eclipse runs inside the client container, chapter 4), Eclipse is installed only once on every workstation. Eclipse is responsible to start the client container plug-in. The amount of disk space is reduced to only one Eclipse installation (unlike in solution A where each application needs its own Eclipse installation). Standard components like the Eclipse runtime are reused for every application client plug-in.

Eclipse does not provide any security component to its plug-in architecture, that is, users cannot be authorized to use an application client plug-in. Therefore, the client container has to add its authentication component. To separate between user configurations, each login has access to a protected Eclipse extension site where all application client plug-ins for this login are located.

A user is able to utilize only those plug-ins he is allowed to start. The client container security component also permits user actions within a plug-in. For more details see section 5.7.1.

An HTTP server hosts the Eclipse client container plug-ins to which all users can have access. This so-called update site can also provide updates and patches for already installed application client plug-ins. New plug-ins, updates and patches are downloaded and installed automatically by the ECC Deployment Manager during startup of the Eclipse client container.

The *ECC Deployment Manager* keeps the Eclipse installation up-to-date. Every time a user logs on, the ECC Deployment Manager searches for new application client plug-ins, or for updates and patches for already installed plug-ins. Without such an update process, traditional update mechanisms like Java WebStart or IAP StarterApplet would have been used to distribute updates. Java WebStart and IAP StarterApplet are used to distribute whole applications. In the Eclipse client container concept, Eclipse is installed once (see above) and following that, only application client plug-ins should have to be installed. Therefore Eclipse introduces its own concept of keeping an installation up-to-date. This approach is included as a new component in the Eclipse client container.

5.5 Structure

Figure 5.1 shows the solution structure. The client container is integrated in Eclipse as plug-in. This plug-in runs, as all other applications, inside Eclipse and provides its feature via APIs and extension points.

5.6 Participants and Responsibilities

5.6.1 Eclipse

Three components are involved during the Eclipse platform startup.

- Client container core component
It represents the main component of the client container which is adapted to run inside Eclipse now. The main task is to handle the authentication process on client side. It provides user dialogs to handle the user login and manages authentication to the application server. After login, the component manages the update process including downloading, installing and starting new or updated plug-ins.
- User Extension Site
An extension site is a feature of the Eclipse update manager and provides a mechanism to store features and plug-ins in any section of the hard disk other than the Eclipse installation directory. In this solution, extension sites represent the user domains of the application

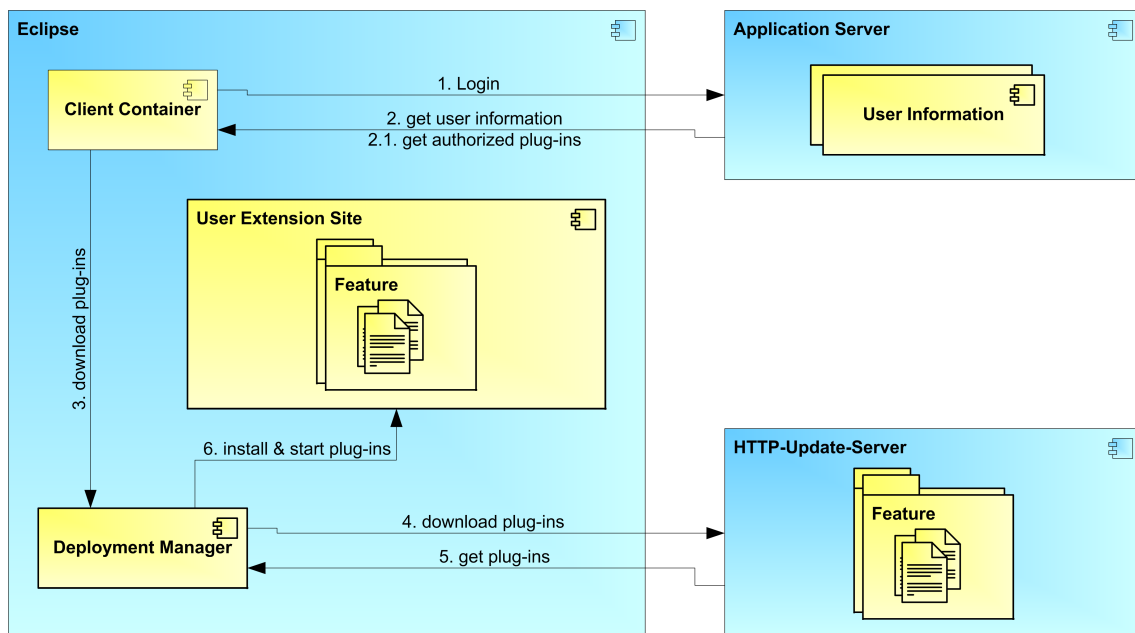


Figure 5.1: Collaboration: Client container runs inside Eclipse

client plug-ins. Each user is allowed to start and to use different plug-ins. Extension sites strictly separate the various application client plug-ins from each other. Due to this, it can be assured, that a user can only start those plug-ins he is allowed to use. The concrete process of separating application client plug-ins is described in section 5.7.2.

- **ECC Deployment Manager**

The ECC Deployment Manager is responsible to search for application client plug-ins on the HTTP update server. It uses the features of the Eclipse update manager component. If the ECC Deployment Manager has found plug-ins (packed as features) for a specific user, it will download and install them in the corresponding extension site. After a successful installation, it tries to start the application client plug-ins. Details about the update mechanism are outlined in section 5.7.2.

5.6.2 Application Server

The application server hosts the server-side components (like EJBs) of client applications. It provides user information, sessions and security data for a specific user to an application client.

5.6.3 HTTP-Update-Server

The update server can be a standalone component or can be integrated into an application server infrastructure. A conventional HTTP server is used to provide application client plug-ins. The update server hosts application client plug-ins for all users. The ECC Deployment Manager accesses the update server and downloads newest versions or patches for the currently logged in user.

5.7 Strategies

This section describes two main strategies newly implemented in the Eclipse client container.

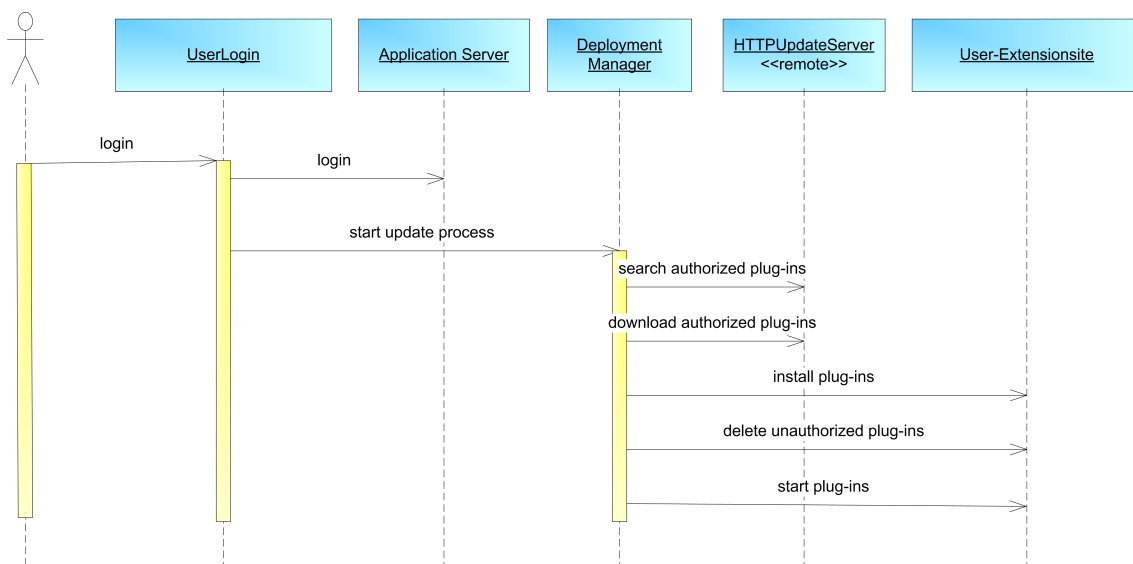


Figure 5.2: Sequence diagram of startup

Figure 5.2 shows the complete startup process of the Eclipse client container. Section 5.7.1 describes the mechanism of authorization. Components, involved in this mechanism, are the UserLogin component, the application server and finally the Eclipse user extension sites. Section 5.7.2 describes how to keep an installation up-to-date. Components, involved in this process, are the ECC Deployment Manager, the HTTP-Update-Server and the Eclipse user extension site.

5.7.1 Authorization

The client container has a dedicated security concept. Running standalone, it authenticates users to use the deployed application clients. Additionally, actions inside an application client like server access or query data structures are protected by role-based authorization.

In case of the Eclipse client container, the concept standalone application client is replaced by the concept of an Eclipse application client plug-in. Such plug-ins represent an integrated application that runs inside Eclipse, using the features of the client container.

The Eclipse 3.0.1 does not know a user context. Eclipse is not able to distinguish between different users. Thus, installing or starting plug-ins can't be authorized by Eclipse.

Therefore, the client container adds its security concept to Eclipse. Before starting up Eclipse, a user dialog identifies the user. However, application authorization is still missing. All users have access to all installed plug-ins. Therefore, a new component is introduced to authorized users to use only the allowed plug-ins.

In previous versions of Eclipse, the installation of new plug-ins had to be performed manually by copying them into the Eclipse installation directory. Due to the recently introduced OSGi framework [17], plug-ins can be installed during runtime of Eclipse without restarting the workbench.

The Eclipse update manager component provides a feature called *extension site*. An extension site is a structural image of the Eclipse home directory. It contains the main directories 'features' and 'plug-ins' where plug-ins can be deployed. An extension site can be programmatically added to a running Eclipse instance.

The Eclipse client container authorization concept is implemented with the help of extension sites. In a base installation, only those plug-ins which are needed to run the rich client application (including the client container plug-ins) are located in the main Eclipse plug-in directory.

All application client plug-ins which have to be authorized to users are located in strictly separated extension sites. Each user has his own extension site where his application client plug-ins are located. Depending on the user login, the corresponding extension site will be added and started after a successful user login. If a new user logs in, which is not known yet, the security component creates a new extension site for this user.

Example: a user starts the rich client application and logs in with his username 'j2eecct11'. After confirming the login the security component searches for an extension site called 'j2eecct11'. If the login name is new to the Eclipse client container, a new extension site will be created. If the user has already logged in, the security component adds the extension site to Eclipse and installs all plug-ins located in this extension site to the running Eclipse instance. A reset of the workbench is not necessary, because all plug-ins contribute their UI components before the workbench is displayed.

A user could place a plug-in that he is not allowed to start into his extension site before starting the Eclipse client container. After a successful user login, the Eclipse client container loads all plug-ins within the extension site of this user, including the newly added, prohibited plug-in. To prevent this bypass, the security component compares the list of authorized plug-ins, retrieved by the application server, with the currently installed plug-ins in the extension site. The security

component will remove all plug-ins from the extension site which the user is not allowed to utilize.

This concept is closely related to the authentication and authorization mechanism of Lotus Workplace Client Technology *Lotus Workplace Client Technology* [18].

5.7.2 Deployment Manager

The Deployment Manager guarantees manageability. Administrators can keep already installed client installations up-to-date by installing new applications and updating or patching already installed ones. The component consists of two components: a client-side and server-side component.

The server-side component is implemented as Eclipse update site. This is the standard mechanism of Eclipse to implement a managed environment. The update site is located on an HTTP server. This server hosts all available application client plug-ins, updates or patches. The content of the server is described in a site map file called `site.xml`.

The client-side component called Deployment Manager is responsible to search for new applications or updates on the server component. It uses features of the Eclipse update manager component to implement the update process. It receives the user login name from the security component. The Deployment Manager accesses the Eclipse update site and searches for new plug-ins or updates. The search process compares the version numbers of installed plug-ins with the corresponding ones on the HTTP server. If a newer version is found, the Deployment Manager uses the Eclipse update manager API to download and install the new plug-in.

An application client plug-in might contribute UI components to the Eclipse workbench. If such a plug-in is added to the workbench programmatically, the Eclipse framework normally has to perform a workbench reset to display the added UI components properly. The user has to confirm manually this reset (with a user dialog).

In case of the Eclipse client container, workbench UI contribution happens in the background. The workbench will not be shown until the process of plug-in installation and starting is finished.

5.8 Consequences

The Deployment Manager guarantees manageability. Traditional approaches like Java WebStart or the DaimlerChrysler IAP StarterApplet do not fit in the Eclipse client container architecture as they provide an update mechanism for complete applications. However, the Deployment Manager has only to manage application client plug-ins, leaving the Eclipse client container untouched. Therefore, it uses the Eclipse update manager component to implement the plug-in

update process. It searches only for new application client plug-ins, for updates, and for bug-fixes for already installed ones.

Eclipse in the current version 3.0.1 does not implement a user context. The client container component adds authentication and authorization to the Eclipse client container. Authentication in form of user login and re-login after a certain time is completely implemented.

Beside authorization of actions inside an application client plug-in, the client container component adds authorization on the plug-in layer. Users can only start and use those plug-ins their user rights allow them. As Eclipse does not provide this concept of plug-in authorization, integrated security cannot be guaranteed. A user, knowing Eclipse well, is able to manually import plug-ins in his extension site and start them (via the OSGi console) after client container login process is finished and Eclipse is started. In this case, a user can only see the UI components of a compromised plug-in after its installation. Each server access is protected by an application server or an access control server (like Netegrity SiteMinder). If a user enforces access to such a plug-in, he is not able to use business logic of this plug-in.

An already running server infrastructure has to be extended by adding a new HTTP server. This server hosts all application client plug-ins and their updates and patches. Due to a potential high number of users in a production environment, this update server has to guarantee non-functional requirements like security, reliability and availability. Hardware components have to stand high load during peak-periods. A cluster of HTTP servers would reduce this load but increases manageability of hosted plug-ins on all involved servers.

6 Installation Guide

This chapter describes at first the basic installation of the Eclipse client container including the Echodemo application, implemented as Eclipse application client plug-in. Second, this chapter will guide the developer through the preparation of an Eclipse IDE, allowing to develop application client plug-ins for the Eclipse client container.

6.1 Introduction

6.1.1 Notations in this Chapter

- All commands which have to be execute are notated in italics, e.g. *cd %TMP%*.
- This guide is based completely (client- and server-side) on a Windows system. All path names are notated with backslashes (\).
- For place holders the term *<value>* is used. The place holder has to be replaced by an appropriate value. E.g. the places holder *<fully-qualified-servername>* has to be replaced by *c960100009294.str.daimlerchrysler.com*.

6.1.2 Common Directories

Common directories like the Eclipse client container installation directory or the home directory of the IBM Websphere application server are represented by special place holders. The directories used in this chapter are listed below:

Name	Description
%IAP%	The base directory of IAP J2EE application package software
%TMP%	A directory located on your local hard disk where to install demo applications
%IAP_CFG%	The IAP configuration directory for an application server e.g. <i>C:\IBM\WebSphere\AppServer\properties\iap\server1</i>
%HT_DOCS%	The document root of the HTTP server e.g. <i>C:\ibm\IBMHttpServer\htdocs</i>
%ECLIPSE_HOME%	The home directory of an Eclipse IDE e.g. <i>C:\Program Files\Eclipse</i>

6.1.3 Prerequisites

It is assumed, that the reader of this document has a basic knowledge concerning a traditional client container infrastructure. He should know about the features and the advantages an application client has, compared to a conventional application. A developer should have read the IAP J2EE Development Installation Guide to understand the basics of an IAP installation.

On the technical side, this chapter assumes, that the following software components are successfully installed in the development environment and completely configured:

- A traditional client container installation
- The Echodemo application (at least on server-side) as standalone application for the traditional client container
- IBM Websphere Application Server 5
- An HTTP server (supported by IBM WAS 5, e.g. Apache or IBM HTTP Server) which includes the WAS plug-in and which is also able to host web sites independently from the IAP installation.

This installation guide assumes, that the reader has performed the following steps of the J2EE Development Installation Guide for IAP 2.5:

- Steps 1 - 12 of unit A
- Steps 1 - 9, 10 - 15, 16 - 25 of unit B
- Steps 32 - 36 of unit D

6.2 Eclipse Client Container Example Installation


First, section [6.2.1](#) describes the installation of the Eclipse client container and its Echodemo application client plug-in on the client-side. Second, section [6.2.2](#) will guide through the steps that have to be done on the server-side. The last section ([6.2.3](#)) describes how to start and test the Echodemo application client plug-in.

6.2.1 Client-side Installation of Demo Application

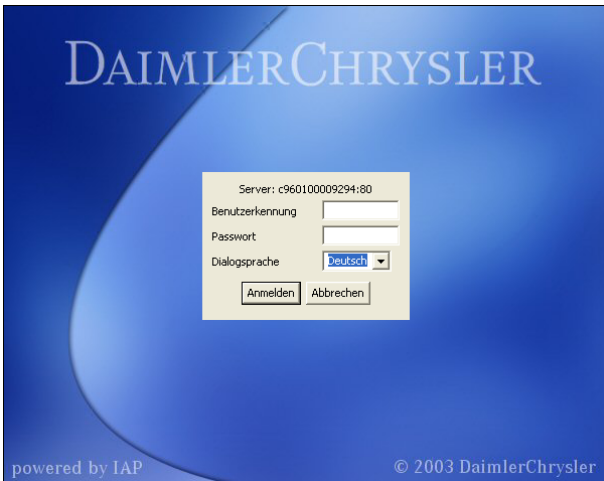
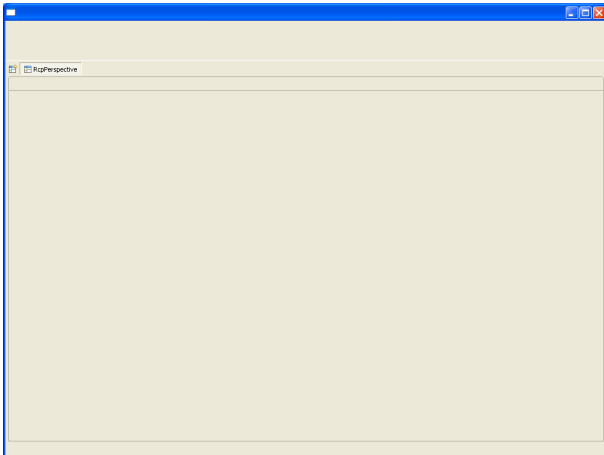
No.	Action
1	<p>Install the Eclipse client container demo application on the client side</p> <pre data-bbox="475 517 1209 591">unzip %IAP%\eclipseclientcontainer\example_ \eclipseclientcontainer\eclipseclientcontainer.zip %TMP%</pre> <p>You should have a structure like this in your %TMP% directory:</p> <pre data-bbox="360 667 576 801"> - eclipsecc + features + plugins + workspace</pre> <p>This structure corresponds to a standard Eclipse installation directory. The features directory contains the feature-related parts of an application. Eclipse plug-ins (application client plug-ins as well as core plug-ins) are located in the plug-ins directory. The workspace directory must exist before starting up the Eclipse client container. After startup, the Eclipse client container will create the OSGi <i>configuration</i> and a <i>.dcxuserextensions</i> directory within the installation directory.</p>


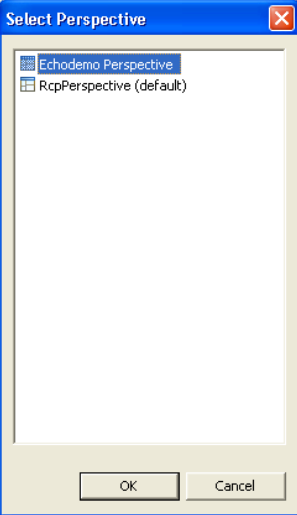
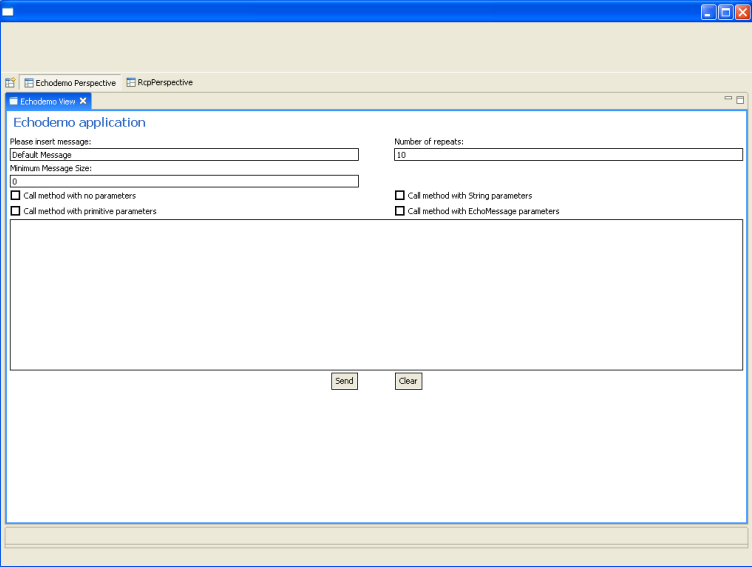
6.2.2 Server-side Installation of Demo Application

No.	Action
2	<p>Append the security definitions of the Eclipse client container demo application to the IAP properties file in the Websphere <i>classes</i> directory.</p> <pre data-bbox="515 589 1098 701">type %IAP%\eclipseclientcontainer\example_ \res\configuration\IAPECC_iap.properties » %WAS_HOME%\classes\iap.properties</pre> <p>NOTE: It is assumed, that a standalone version of the IAP client container and its demo applications is already installed in the development environment. All demo, tutorial and administration users are already added to the IAP properties file. Otherwise please perform Step No. 28 of the J2EE Development Installation Guide.</p>
3	<p>Add the application client element for the Eclipse client container to the server-side Client container configuration file: Open a text editor and copy the content of</p> <pre data-bbox="437 1115 1174 1189">%IAP%\eclipseclientcontainer\example\configuration_ \dcx.appclient.container.clientcontainer-additional-ecc.xml</pre> <p>into the XML element <code><client-type></code> of</p> <pre data-bbox="437 1283 1174 1317">%IAP_CFG%\dcx.appclient.container.clientcontainer.xml</pre> <p>NOTE: As mentioned in step 2, it is assumed that a version of the standalone client container is already installed. If not, please copy the server-side client container configuration file into the IAP configuration directory:</p> <pre data-bbox="437 1574 1174 1686">copy %IAP%\example\clientcontainer\res\configuration_ \dcx.appclient.container.clientcontainer-development.xml %IAP_CFG%\dcx.appclient.container.clientcontainer.xml</pre> <p>After that, please perform step 40 and 41 of the J2EE Development Installation Guide.</p>
4	<p>Install the server-side components of the Eclipse client container demo application. Follow the steps 42 - 50 of the J2EE Development Installation Guide.</p>

No.	Action
5	<p>Unzip the Eclipse HTTP-Update-Server contents into the htdocs-directory of your web server.</p> <pre>unzip %IAP%\eclipseclientcontainer\example\dist\httpum\eum.zip %HT_DOCS%</pre> <p>You should have a directory structure like this in the httpus-subdirectory:</p>  <pre>graph TD httpus[httpus] --- sitebuild[.sitebuild] httpus --- features[features] httpus --- plugins[plugins]</pre> <p>The <i>.sitebuild</i> directory is an internal directory of Eclipse which is used to build the feature-/plug-in-packages. The <i>features</i> directory contains the feature-related parts of all users which could logon to the update server. The <i>plug-ins</i> directory contains the real plug-ins which can be downloaded. Inside the root directory of the update manager you can find a <i>site.xml</i>. This XML document describes the Eclipse update site and will be read by the Eclipse update manager.</p>

6.2.3 Starting the Demo Application

No.	Action
6	Change to the <i>%TMP%</i> directory.
7	Start the Eclipse client container using the MS-DOS Batch file <i>startup.bat -CCserver <fully-qualified-server-name></i>
8	<p>If IBM Websphere security is enabled the following user login dialog will be shown:</p>  <p>The login dialog box has a blue background with the DaimlerChrysler logo at the top. It contains a text box for 'Benutzerkennung', a text box for 'Passwort', and a dropdown menu for 'Dialogsprache' set to 'Deutsch'. There are 'Anmelden' and 'Abbrechen' buttons at the bottom. The server address 'c960100009294:80' is displayed at the top of the dialog. The background also says 'powered by IAP' and '© 2003 DaimlerChrysler'.</p> <p>Login with user <i>j2eecct11</i> and password <i>start</i></p>
9	<p>If the login was successful, the login window should be closed and the Eclipse RCP with a primitive workbench should open. It should look like this:</p>  <p>The image shows a standard Eclipse RCP window with a title bar and a menu bar. The main area is empty, showing a primitive workbench.</p>

No.	Action
10	<p>Click the -Button in the perspective bar and select the menu item 'Other' Select the Echodemo perspective in the following dialog and choose OK.</p> 
11	<p>A view within the Echodemo perspective like the following should open:</p> 

No.	Action
12	<p>Input a message or keep the default message. Select at least one of the four checkboxes. 'Call method with no parameters' will send a blank message to the EJB on the application server. 'Call method with primitive parameters' will generate a message with integer, double, long and float parameters. 'Call method with String parameters' will send a message with the input string as message. 'Call method with EchoMessage parameters' will send a complete object instance to the server.</p> <p>The server should answer this request and the result should be shown in the large text box:</p> <div><pre>Received: 10 messages Duration: 120 ms, per Loop: 12 ms. Call echoPrimitives() method... Received: 10 messages Duration: 270 ms, per Loop: 27 ms. Call echoString() EJB method with the string: [15]: "Default Message" Received: 10 messages Duration: 90 ms, per Loop: 9 ms. Call echoMessage() method with the message: [15]: "Default Message" Received: 10 messages Duration: 231 ms, per Loop: 23 ms.</pre></div>
13	Close the Eclipse client container.
14	Now you could login with another user than <i>j2eecct11</i> (e.g. <i>j2eecct21</i>). You should be able to start the Eclipse RCA, but you should not be allowed to see the Echodemo perspective.

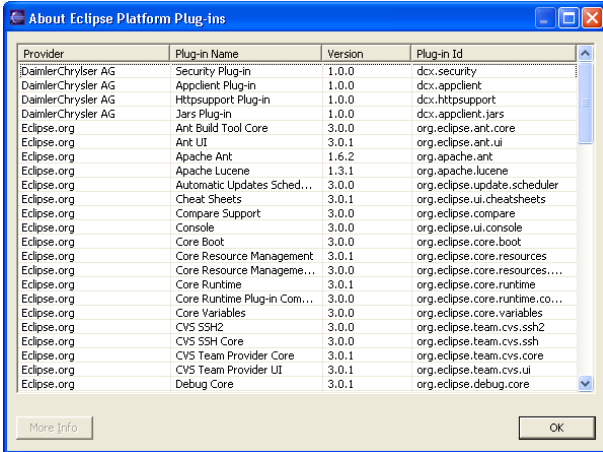
6.3 Setting up an Eclipse IDE

This section describes, which steps have to be done, to be able to develop application client plug-ins for the Eclipse client container. It is assumed, that the reader has a basic knowledge how to develop a plug-in for Eclipse. Please note, that only those steps are described, which are needed for deploying, debugging and testing plug-ins without using the ECC Deployment Manager of the Eclipse client container.

NOTE: If these steps are not performed, a developer will have to set up an Eclipse update site. Each time he wants to test the application client plug-in, he has to build the plug-in, deploy it as a feature to the update site and delete the currently deployed plug-ins from the user extension sites.

Section 6.3.1 will describe how to prepare an Eclipse IDE to develop application client plug-ins. Section 6.3.2 will show which properties have to be set in a launch configuration to deploy and run an application client plug-in inside the Eclipse client container.

6.3.1 Preparations

No.	Action
1	<p>Close your Eclipse IDE if it is already running. Unzip the Eclipse client container feature (with its incorporated plug-ins) to the Eclipse installation directory:</p> <pre>unzip %IAP%\eclipseclientcontainer\res\eclipseclientcontainer.zip %ECLIPSE_HOME%</pre> <p>The client container feature should now be allocated to the <i>features</i> directory, the corresponding plug-ins should be found in the <i>plug-ins</i> directory of your Eclipse IDE installation.</p>
2	<p>Now start your Eclipse IDE and check the correct installation of the Eclipse client container plug-ins:</p> <p>To check the configuration click <i>Help > About Eclipse platform > Plug-In Details</i></p> <p>You should now get a window like this.</p>  <p>If the four Eclipse client container plug-ins (Security Plug-In, Appclient Plug-In, Httpsupport Plug-In and Jars Plug-In) will appear in the list, the plug-ins are correctly installed.</p>

6.3.2 Launch Configuration

To avoid the application of the ECC Deployment Manager and thus to simplify the development of application client plug-ins, some properties have to be set in the launch configuration inside the Eclipse IDE. This section will describe the settings which have to be done to get an Eclipse RCP for the Eclipse client container work properly.

The ECC Deployment Manager will use the HTTP update site. After having made changes in the code of an application client plug-in, a developer has to rebuild the update site and to delete the currently installed plug-in on client-side (installed in the `.dcxuserextensions` directory). The Eclipse client container can be configured for a development environment such, not to utilize the ECC Deployment Manager.

The implemented rich client application can be run and debugged inside the Eclipse IDE. To achieve this, you first have to add a new launch configuration. Click *Run* in the menubar and select *Debug....* Select *Run-time Workbench* on the left and click in *New*.

Two configuration steps have to be performed, to run the Eclipse client container:

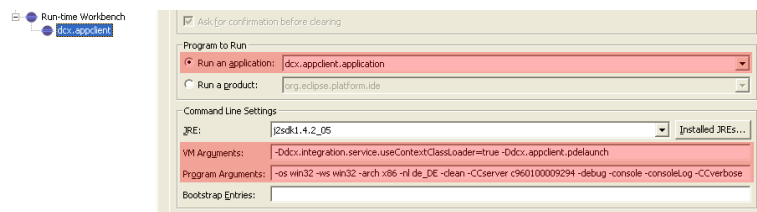


Figure 6.1: Workbench launch configuration settings

- 'Program to run'
The Eclipse client container starts the rich client application for you. Details can be read in chapter 7.2.2. To run the Eclipse RCA, you have to select the radio button *Run an application* and search for the application *dcx.appclient.application* (see figure 6.1).
- 'Command Line Settings'
The Eclipse client container needs runtime settings to run properly inside an Eclipse IDE. First be sure, that you have selected a Java virtual machine, at least version 1.4.2_05. You must insert the VM parameter *-Ddcx.integration.service.useContextClassLoader=true*. Additionally, if you want to debug application client plug-ins and if you want to avoid using the ECC Deployment Manager (as addressed at the beginning of this section), you have to insert the parameter *-Ddcx.appclient.pdelaunch*.
Apart from the general *Program arguments*, the Eclipse client container needs as parameter its server (as also needed in the standalone client container). Therefore, insert the property *-CCserver <fully-qualified-server-name>*. You can additionally add other program arguments to get more debug output of the client container component (e.g. *-CCverbose*) on the one hand and of Eclipse (e.g. *-debug* or *-console*) on the other.

To run only those plug-ins belonging to your rich client application, you have to set some properties on the *Plug-ins* page.

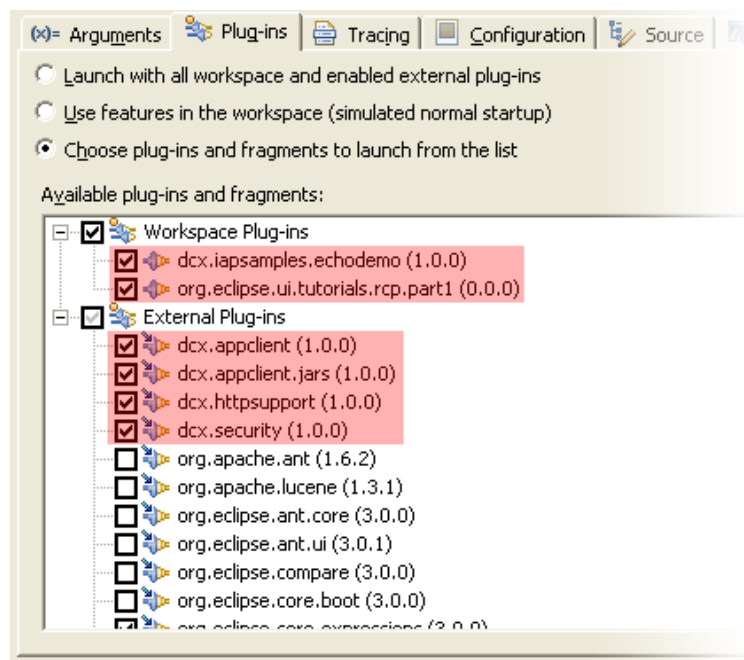


Figure 6.2: Selection of plug-ins to launch

First choose *Choose plug-ins and fragments to launch from the list*. After that, you should get a complete list of all installed Eclipse plug-ins and features as well as your plug-ins, developed by yourself, in your current workspace.

1. Click on the button *Deselect all* on the right side.
2. Select your Eclipse project which contains the Eclipse rich client application. When running the example, select *org.eclipse.ui.tutorials.rcp.part1*.
3. If you have chosen to avoid running the ECC Deployment Manager, select the application client plug-in from your current workspace. If running the example, select *dcx.iapsamples.echodemo*
4. Click the button *Add Required Plug-ins*. All plug-ins, that are needed to run the RCA and the examples, should now be selected automatically.

Your launch configuration is now prepared to run properly. If you have finished the configuration, you can click on *Debug* to run the Eclipse RCA.

7 Tutorial: Writing Applications for the Eclipse Client Container

The chapter explains how to create a rich client application (RCA) based on the Eclipse rich client platform (RCP) in a traditional way, and how to implement a rich client application for the Eclipse client container. Furthermore, it shows, how developers can implement application client plug-ins for the Eclipse client container.

7.1 Introduction

7.1.1 Structure of this Chapter

First, section [7.2](#) introduces how to build a rich client application with the Eclipse RCP in general. After that, the section shows what changes have to be made, if the implemented rich client application with the Eclipse client container shell be used.

Section [7.3](#) shows implementation details of writing application client plug-ins for the Eclipse client container. If the system architecture envisions manageability with the help of the ECC Deployment Manager, section [7.4](#) describes how to set up an Eclipse update site, to prepare it for the use with the Eclipse client container and how to deploy it to an HTTP server.

7.1.2 Prerequisites

It is highly recommended, that the reader has already some basic understanding of the plug-in concept of Eclipse (see section [2.1.3](#)) and has some know-how about common mechanisms, like Eclipse plug-in dependencies or extension points (see section [2.1.4](#)). He should have also read chapter [6](#) in order to ensure a proper installation of the Eclipse client container and he should know how to run and test an Eclipse client container RCA and its application client plug-ins.

7.1.3 Overview

Previous versions of Eclipse (2.1 and older) represented only an integrated development environment. Developers discovered quickly, that the Eclipse plug-in mechanism was a phantastic tool to extend the IDE by adding new functionality and to use Eclipse for non-IDE purposes [\[5\]](#). Version 3 detaches the IDE part of Eclipse and introduces the rich client platform (RCP). Eclipse

is now able to host applications that are not IDE-specific. Developers can use the Eclipse RCP to build applications, which make use of those features, for which Eclipse is popular: a native look and feel with perspectives and views or the help system. The Eclipse plug-in mechanism helps developers to create extensible applications. More facts about Eclipse can be found in [chapter 2](#).

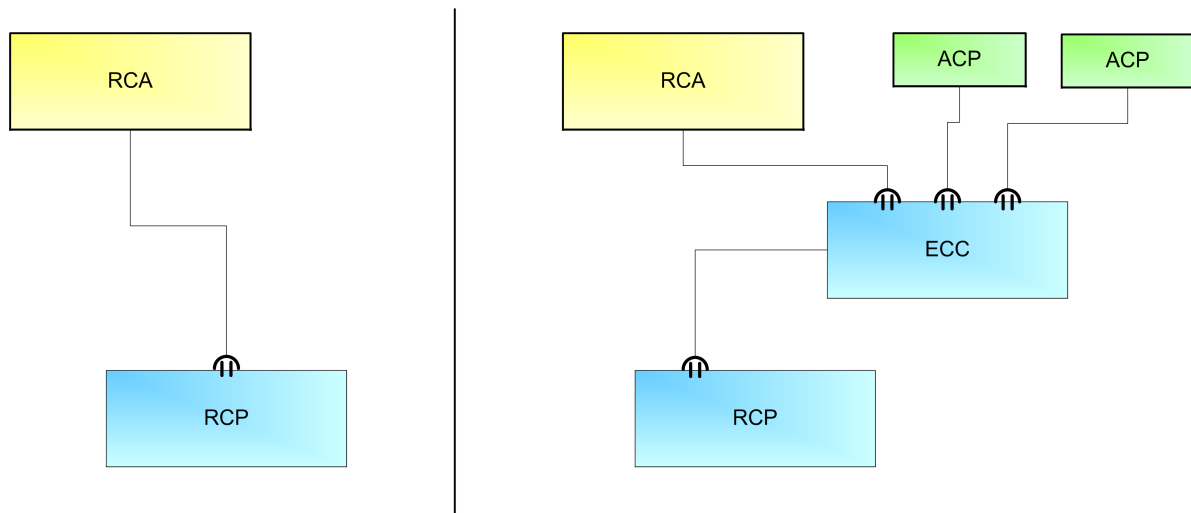


Figure 7.1: RCA with and without ECC

The Eclipse client container uses the RCP functionality (see [figure 7.1](#)). It needs a rich client application as a 'main program' to provide features like authentication, authorization or remote server access with its API to application client plug-ins.

A conventional Eclipse plug-in is no application client plug-in for the Eclipse client container per se. This metamorphosis happens by extending an Eclipse client container extension point. If a plug-in extends an extension point, it extends an already existing plug-in by adding new functionality. In this case, a plug-in extends the Eclipse client container and adds another 'program' using the Eclipse client container. [Section 7.3](#) shows the steps how to implement an application client plug-in.

7.2 Writing a Rich Client Application

This section describes how to set up a rich client application. The section 'Changes to a traditional RCA' ([7.2.2](#)) shows, which steps are needed to merge a conventional RCA to a one, which interacts closely with the Eclipse client container.

7.2.1 A "Standalone" RCA

Figure 7.2 illustrates the necessary steps to create an Eclipse rich client application.

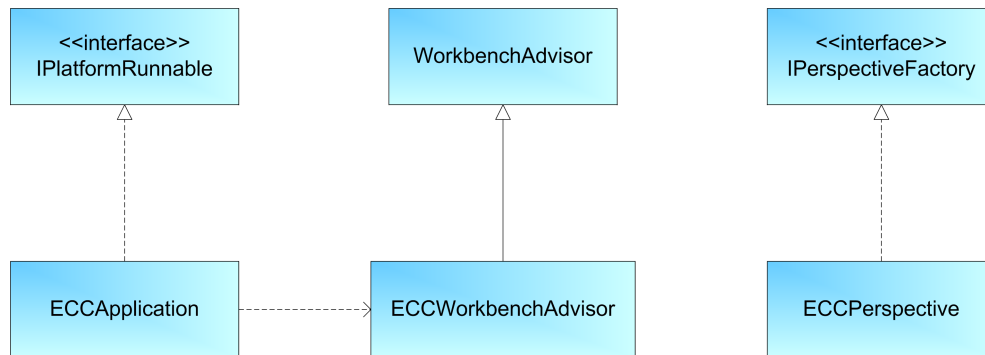


Figure 7.2: Involved classes while creating an Eclipse RCA

An Eclipse rich client application consists basically of three classes. The result of this tutorial will be a RCA, which only pops up the Eclipse workbench without any menu or tool bars.

Creating a Plug-in Project

Before a rich client application can be implemented in Eclipse a new plug-in project has to be created. To start a new project in Eclipse, click *File > New > Plug-in Project*. Define a project name, e.g. *dcx.iapsamples.mysample*. By convention, plug-in projects in Eclipse follow the naming conventions for packages in the Java programming language.

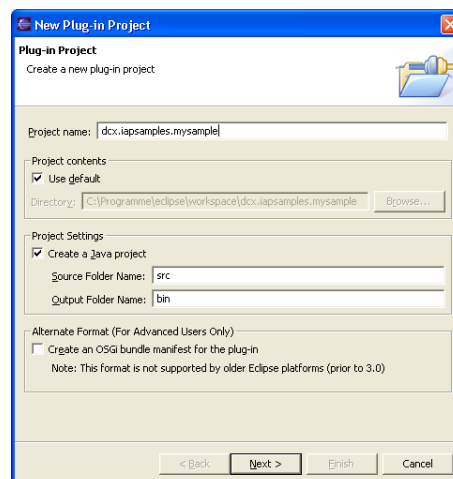


Figure 7.3: Wizard for creating a new plug-in project

If *Create a Java project* is not selected, choose it in the project settings and type *src* as source folder for Java files and *bin* for the compiled Java classes. Make sure, that the checkbox *Create an OSGi bundle manifest for the plug-in* is not selected. It is unusual to create a manifest file. It is suggested to create only a OSGi MANIFEST.MF, if features of the underlying OSGi framework are needed [2]. If you want to change descriptive statements or enter a provider name, you can alternatively click *Next*. After completion of these steps, click *Finish*.

The plug-in project wizard creates a new Eclipse project in your workspace. If you have followed the instructions above, it will create the following directory and file structure:

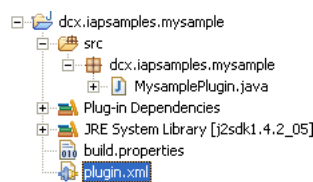


Figure 7.4: A plug-in directory and file structure

The 'Main' Application

Traditional applications, written in the Java programming language, have got an entry point into the program [3]. This concept is realized with the help of a main method, executed by the Java virtual machine. Starting a program in Eclipse, is replaced by a more flexible mechanism: Eclipse uses its extension point mechanism.

Instead of calling the application's main method, the Java virtual machine starts first the Eclipse runtime [3]. The Eclipse runtime itself provides an extension point called *org.eclipse.core.runtime.-applications*. A rich client application being started by the Eclipse runtime, has to extend this extension point. This extension point defines an interface called *IPlatformRunnable* which must be implemented by the application.

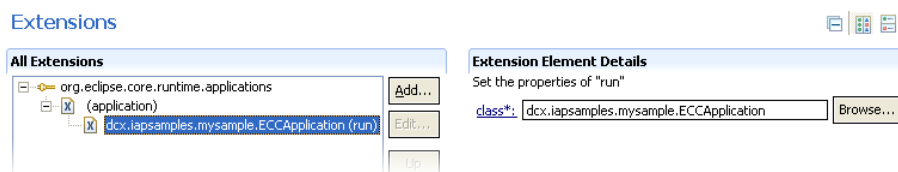


Figure 7.5: Definition of an extension point

To implement a rich client application, open the plug-in manifest file *plugin.xml*. Switch to the tab *Extensions*. After that, click on the button *Add* (figure 7.5). Select the extension point *org.eclipse.core.runtime.applications* from the list and confirm by clicking on *Finish*. Choose the

new entry in the new list *All Extensions* and enter *id* and *name* *mysample* in the right box. Right-click on the entry and choose *New > application* to add a new application. Right-click on the added application child and choose again *New > run*.

In the section *Extension Element Details* you can now select a class that implements the interface *IPlatformRunnable* as already mentioned. If you want to create a new class, click on the underlined item *class**. The common new class wizard will popup with some default entries, e.g. the interface to implement. Select a package and define a class name (e.g. *ECCApplication*) and click *Finish*. After that, the wizard creates the implementation class with its run method. If it does not open automatically, open this class with the package explorer. Replace the *return null* statement by the two lines (no. 7 and 8) in the following code sample:

Listing 7.1: Base class of a RCP

```

1 package dcx.iapsamples.mysample;
2
3 import org.eclipse.core.runtime.IPlatformRunnable;
4
5 public class ECCApplication implements IPlatformRunnable {
6     public Object run(Object args) throws Exception {
7         System.out.println("My first RCP application!");
8         return IPlatformRunnable.EXIT_OK;
9     }
10 }
```

You have got now the most primitive rich client application based on the Eclipse RCP, but with no UI yet. During startup of the Eclipse runtime, the implementation class, defined by the extension point, will be instantiated and the *run*-method will be called. The parameter *args* stores the program arguments given from the command line. To access those arguments, cast this object to a string array.

The extension point wizard has added the following lines to the plug-in manifest file. Compare the lines of listing 7.2 with those in your *plugin.xml*.

Listing 7.2: Extension point entry in plug-in manifest

```

1 <extension
2     id="mysample"
3     name="mysample"
4     point="org.eclipse.core.runtime.applications">
5     <application>
6         <run class="dcx.iapsamples.mysample.ECCApplication"/>
7     </application>
8 </extension>
```

Launching From Eclipse

Open a new Run-time Workbench launch configuration to start the new rich client application with the Eclipse launcher,. In the section *Program to run* select the ID from the created application (*dcx.iapsamples.mysample.mysample*). Switch to the *Plug-ins* tab and select *Choose plug-ins and fragments to launch from the list*. Deselect all plug-ins and select the created plug-in *dcx.iapsamples.mysample*. Click on *Add Required Plug-ins* to add those plug-ins to the launch configuration, required to run the rich client application. When launching this configuration, you will see the output *My first RCP application!* in the console window.

Launching Outside Eclipse

Alternatively, you can launch the rich client application from the command line. The first step is to assemble the plug-ins, which are needed to run a rich client application. They form the least common denominator of a RCA. Create a new directory on your hard disk, create a subdirectory called *plug-ins* and copy the plug-in directories (shown in figure 7.6) from the plug-in directory of your Eclipse installation into the new plug-in directory.

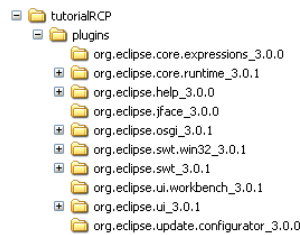


Figure 7.6: Needed plug-ins to run a basic RCA

Second step is to copy the *startup.jar* from your Eclipse installation home to the newly created RCA directory.

Then export your rich client plug-in into the previously created plug-in directory. Click *File > Export* and select *Deployable plug-ins and fragments* from the list. Select your rich client plug-in from the list of available plug-ins in your workspace. Deploy the plug-in into a *Directory*. Select the previously created directory of your new RCA. Pay attention that the root directory (not the plug-in directory) is chosen.

You can now start your rich client application as a standalone client from the command line. Use the following command:

```
%JAVA_EXE% -cp startup.jar org.eclipse.core.launcher.Main  
-application dcx.iapsamples.mysample.mysample -consoleLog %*
```

Replace the place holder `%JAVA_EXE%` by the path to your Java virtual machine. Alternatively, you can use the `eclipse.exe` program to launch your RCA. by copying this file to the RCA root directory. Define this program with the following parameter:

```
eclipse.exe -application dcx.iapsamples.mysample.mysample
```

Now that we have a executable rich client application we can add more features like the common workbench to our example.

A Default Perspective

Each opened workbench starts with an initial perspective. This perspective layouts the workbench and locates the involved views on the screen. In our example, a default perspective with no content will be created. Extending the extension point `org.eclipse.ui.perspectives` will create a new perspective. This is done in the same way as you have extended `org.eclipse.core.runtime.applications`. Create a new implementation class called `ECCPerspective`. No changes on the basic implementation of this class are needed. It will look similar to the example:

Listing 7.3: A simple perspective

```
1 package dcx.iapsamples.mysample;
2
3 import org.eclipse.ui.IPageLayout;
4 import org.eclipse.ui.IPerspectiveFactory;
5
6 public class ECCPerspective implements IPerspectiveFactory {
7     public void createInitialLayout(IPageLayout layout) {
8     }
9 }
```

The plug-in extension wizard should add the perspective description to the plug-in manifest:

Listing 7.4: Perspective entry in plug-in manifest

```
1 <extension
2     point="org.eclipse.ui.perspectives">
3     <perspective
4         class="dcx.iapsamples.mysample.ECCPerspective"
5         name="dcx.iapsamples.mysample.perspective"
6         id="dcx.iapsamples.mysample.perspective"/>
7 </extension>
```

The Workbench Advisor

Each rich client application needs a workbench advisor. This advisor configures the workbench by adding e.g. tool bars or new perspectives. Furthermore, the Eclipse runtime calls methods of this class (e.g. *preStartup()*, *postStartup()*, *postShutdown()*) during the lifecycle of a RCA. Developers can override these methods to execute the code before, during or after starting or stopping the Eclipse runtime. In our example, we create a very simple workbench advisor:

Listing 7.5: A simple workbench advisor

```
1 package dcx.iapsamples.mysample;
2
3 import org.eclipse.ui.application.WorkbenchAdvisor;
4
5 public class ECCWorkbenchAdvisor extends WorkbenchAdvisor {
6     public String getInitialWindowPerspectiveId() {
7         return "dcx.iapsamples.mysample.perspective";
8     }
9 }
```

The method *getInitialWindowPerspectiveId()* is mandatory. In our case, it returns the perspective ID created before. Properties, like the initial window size or the title of the rich client application, can be set in the *preWindowOpen()* method.

Extending the Rich Client

The last step creating a basic rich client application, is to combine the units created before:

Listing 7.6: The final RCP

```
1 package dcx.iapsamples.mysample;
2
3 import org.eclipse.core.runtime.IPlatformRunnable;
4 import org.eclipse.swt.widgets.Display;
5 import org.eclipse.ui.PlatformUI;
6 import org.eclipse.ui.application.WorkbenchAdvisor;
7
8 public class ECCApplication implements IPlatformRunnable {
9     public Object run(Object args) throws Exception {
10         WorkbenchAdvisor workbenchAdvisor = new ECCWorkbenchAdvisor();
11         Display display = PlatformUI.createDisplay();
12         try {
13             int returnCode = PlatformUI.createAndRunWorkbench(display,
14                 workbenchAdvisor);
15             if (returnCode == PlatformUI.RETURN_RESTART) {
16                 return IPlatformRunnable.EXIT_RESTART;
17             } else {
18                 return IPlatformRunnable.EXIT_OK;
19             }
20         }
21     }
22 }
```



```
19     }  
20     } finally {  
21         display.dispose();  
22     }  
23 }  
24 }
```

First, an instance of our workbench advisor is created. To show our rich client application on the screen, we need to create a display instance. After that, the PlatformUI class starts and shows the workbench with the help of our display and advisor instance. This method starts and controls the main event loop of the Eclipse runtime.

If you launch the rich client, a very simple workbench should pop up (see figure 7.7).

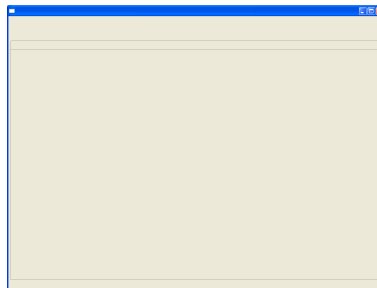


Figure 7.7: The created RCA

7.2.2 Changes to a Traditional RCA

This section describes how to transfer a conventional RCA to an RCA using the Eclipse client container. It is assumed that you have created an RCA, based on the section before. First, make sure that you have installed the Eclipse client container plug-ins in your development environment. If you want to launch your rich client application outside Eclipse, copy the ECC plug-ins into the plug-ins directory of your RCA.

The following steps have to be done to get a RCA interacting with the Eclipse client container:

1. Change the extension point *org.eclipse.core.runtime.applications* to *dcx.appclient.clientcontainer*
2. Change the interface *IPlatformRunnable* to *IAPClientContainer*
3. Use an already created instance of the Display class

Adding Eclipse client container functionality to an existing RCA is very simple. A conventional rich client application extends the extension point *org.eclipse.core.runtime.applications* to

get started by the Eclipse runtime. This concept is changed when using the Eclipse client container (see again figure 7.1). The Eclipse client container runtime implements this extension point and therefore is itself a rich client application.

The ECC runtime provides an extension point called *dcx.appclient.clientcontainer* which has to be extended by rich client applications now. The Eclipse runtime starts the RCA of the Eclipse client container instead of starting the rich client application. That means, if starting the container from the command line, replace *-application dcx.iapsamples.mysample.mysample* by *-application dcx.appclient.application*. The Eclipse client container runtime searches for the RCA which extends its extension point and calls the *run*-method of the implementation class. Figure 7.8 shows the starting order.

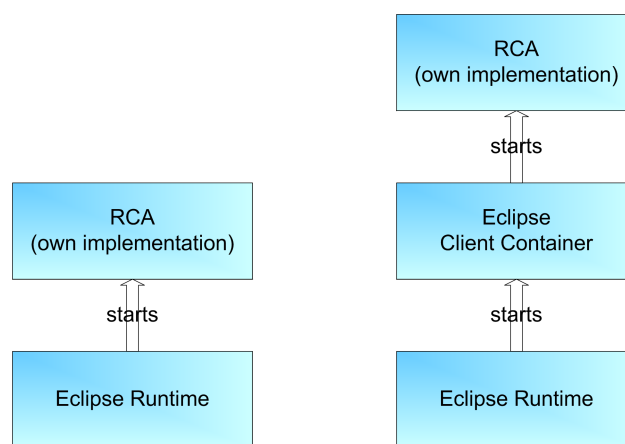


Figure 7.8: RCP constellation without and with ECC

As rich client applications now use an alternative extension point, the implemented interface must be changed from *IPlatformRunnable* to *IAPClientContainer*. Just simply change the *implements*-statement. The class body remains the same.

A conventional rich client application has to create a *Display* instance before opening the Eclipse workbench. The creation of this instance is done in the Eclipse client container runtime. Calling the statement

```
Display display=PlatformUI.createDisplay()
```

will throw an exception, because the creation of the instance is already done by the Eclipse client container. Replace this line of code by

```
Display display=Display.getCurrent()
```

This static method returns the currently used display instance.

The simple rich client application created in the section before will now look like this:

Listing 7.7: A rich client for Eclipse client container

```
1 package dcx.iapsamples.mysample;
2
3 import org.eclipse.core.runtime.IPlatformRunnable;
4 import org.eclipse.swt.widgets.Display;
5 import org.eclipse.ui.PlatformUI;
6 import org.eclipse.ui.application.WorkbenchAdvisor;
7
8 import dcx.appclient.eclipse.IAPClientContainer;
9
10 public class ECCApplication implements IAPClientContainer {
11
12     public Object run(Object args) {
13         WorkbenchAdvisor workbenchAdvisor = new ECCWorkbenchAdvisor();
14         Display display = Display.getCurrent();
15         try {
16             int returnCode = PlatformUI.createAndRunWorkbench(display,
17                 workbenchAdvisor);
18             if (returnCode == PlatformUI.RETURN_RESTART) {
19                 return IPlatformRunnable.EXIT_RESTART;
20             } else {
21                 return IPlatformRunnable.EXIT_OK;
22             }
23         } finally {
24             display.dispose();
25         }
26     }
27 }
```

The result when following the steps described in this chapter so far, is a rich client application, based on the Eclipse client container. You have seen, that the modifications from a conventional RCP to an Eclipse client container RCP are marginal.

7.3 Writing an Application Client Plug-in

This section of the tutorial describes how to implement an application client plug-in (ACP) for the Eclipse client container. Section 7.3.2 deals with the best practices, that is tips and tricks, of writing an application client plug-in that fits well in the Eclipse environment. After that, a section (7.3.3) discusses shortly, which steps have to be performed, to merge plug-ins to a feature that can be deployed to the HTTP update server.

7.3.1 Implementation

The following steps have to be performed to create an application client plug-in:

1. Create an Eclipse plug-in
2. Add a dependency to *dcx.appclient*
3. Implement the Eclipse plug-in
4. Extend the Eclipse client container with *dcx.appclient.applicationPlugin*
5. Implement the Interface *IApplicationClient*
6. Assign the application client plug-in to a user

Creating an Eclipse plug-in

An application client plug-in is a conventional Eclipse plug-in enriched with the Eclipse client container functionalities. To start developing such an ACP, create a new Eclipse plug-in project. The Eclipse client container provides different features, e.g. authentication and authorization, accessing the JNDI tree and making remote calls to Enterprise Java Beans. These features must be imported by adding a dependency to the ECC plug-in *dcx.appclient*. Connecting the two plug-ins will merge the namespaces of the Eclipse client container and the ACP. E.g. the common JNDI directory can now be read and written or EJB base classes like *EJBHome.class* are available.

Extending the Eclipse client container

A plug-in, which requests to act as an application client plug-in, can be seen as an extension of the Eclipse client container. This metamorphism is implemented by extending the extension point *dcx.appclient.applicationPlugin* provided by the Eclipse client container. An interface called *IApplicationClient* has to be implemented, which is associated with the extension mechanism. The *run(...)* method is called at startup of the application client plug-in - the *shutdown()* method is called before shutdown of the Eclipse runtime.

The run method receives an *IAPApplicationContext*. This context contains basic runtime information for the application client plug-in like command line arguments or the application ID. On startup of the platform, the user's rights are checked by the Eclipse client container. If a user is allowed to utilize the application client plug-in, the ECC will start it. After that, the implemented run method is called.

You may recognize, that two classes will be instantiated on startup of a plug-in. A conventional plug-in may contain a plug-in class, which is called by the Eclipse runtime on startup. Additionally, the Eclipse client container calls its own run method during starting a plug-in. On the one hand, an application client plug-in needs additional runtime information, which cannot be distributed by the conventional plug-in class. On the other hand, implementing a plug-in class isn't mandatory in Eclipse. Section [7.3.2](#) deals with this problem.

The result of the steps before is an application client plug-in for the Eclipse client container. It can be packed now in a feature (section 7.3.3) and deployed (section 7.4.2) to the HTTP update server to get downloaded by installed rich clients.

Assign an ACP to a user

If a user wants to be able to use the developed application client plug-in, he must get the user rights. Therefore the application ID of the ACP must be assigned to the the different usernames. The client container component (independent if it runs in standalone mode or in Eclipse) prescribes to define an application ID for each application. If using Eclipse client container, there is a naming convention for the application ID (see section 7.3.2).

First, the application must be defined in the `clientcontainer.xml` server configuration. Insert a new application client in the launch configuration *ECLIPSECC*. Define a security (application) ID like described in section 7.3.2. If the application uses EJB references, additionally define the RPC URLs, as it is known from the standalone client container.

NOTE: The element `<main-class>` must be defined to null (`<main-class>null</main-class>`). Otherwise the Eclipse client container won't be able to find and start the application (defining a main class is mandatory according to the `clientcontainer.dtd`. As Eclipse does not know any main class, the element has to be set to null).

In a development environment, authorization is done with the help of a configuration file *iap.properties*. Follow the instructions for defining user rights in a standalone client container environment. No extra properties have to be set.

7.3.2 Best Practices

Naming Conventions

Application IDs are used to implement the security concept in the standalone IAP client container. Each application gets such an identifier. This ID can be logically assigned to a user. If the user's list contains this application ID, he is allowed to use the application. The choice of naming is free in the standalone client container.

The Eclipse client container introduces a naming convention for the application ID. Each application client plug-in must have an application ID allowing to be identified by the security component of the Eclipse client container. The naming of an Eclipse plug-in follows the naming convention of packages in the Java programming language. The example application *Echodemo* has the plug-in name *dcx.iapsamples.echodemo*. As dots (.) are used to separate attributes in the *iap.properties* configuration file, they must be replaced by underscores (_) to prevent errors while interpreting the properties file. The application ID of the Echodemo application

is now *dcx_iapsamples_echodemo*. All application client plug-ins should follow this naming convention.

Application Client Plug-in Implementation

An Eclipse plug-in development rule advises to strictly separate GUI elements from business logic. This can be assigned to the development of an application client plug-in, too. It is not recommended to put together Eclipse specific code and Eclipse client container code.

The *Echodemo* application for the Eclipse client container is taken as example here. The main class of the example is the *FormView* class. This class is instantiated by Eclipse when opening the perspective (the constructor is called when creating this view). If this class also implements the interface *IApplicationClient*, the class would be instantiated twice: the first time on startup of the ACP and second time when creating the view. If the *run()* method (the first instantiation) of the implemented interface would create instances of Enterprise Java Beans, they won't be accessible for the view (the second instantiation). Figure 7.9 shows the two approaches.

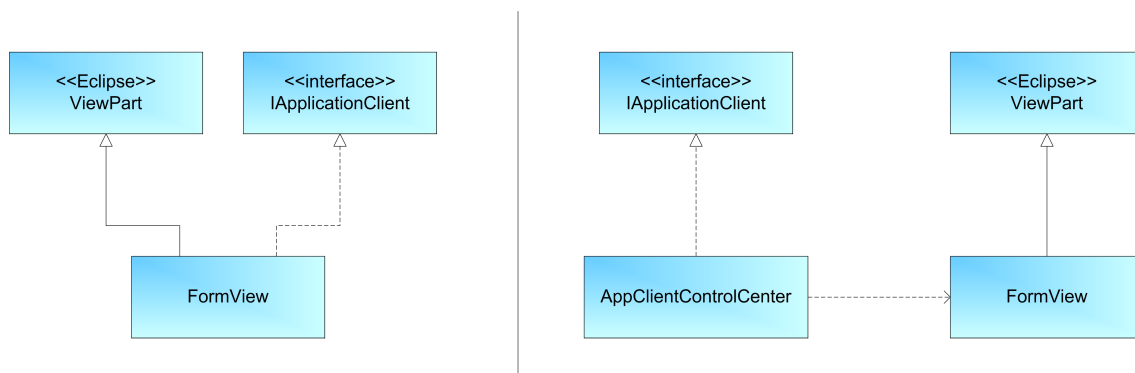


Figure 7.9: Two approaches

The recommendation is to extract the implementation of the *IApplicationClient* into a separated class. When creating the *FormView* (instantiation by Eclipse), the *AppClientControlCenter* receives a reference of the *FormView* instance. When shutting down the Eclipse platform, the *shutdown()* method of the control center will be called. This method delegates the shutdown request to the *FormView* class which releases EJB references and closes the view.

7.3.3 Packaging to a Feature

Before an application can be deployed in the Eclipse client container, the involved plug-ins, which contain the application client plug-in, have to be packaged to a feature. A feature is a collection of plug-ins which belong together in the context of an application. Furthermore, it can include

other features which belong to this feature. It can contain product branding or licence terms of its provider.

In the context of the Eclipse client container, application client plug-ins have to be packaged to features, because they are handled by the ECC Deployment Manager using the Eclipse update manager. Features can be defined and placed on an update site, so that the Eclipse update manager is able to find and to install them [4].

Packaging one or more plug-ins to a deployable feature is very simple in Eclipse. To create a new feature, click *File > New > Feature Project*. Define a project name (e.g. *dcx.iapsamples*) and click *Next >*. On the next page, feature details like the name of the JAR file or the provider name can be edited. On the next page (click *Next >*), you can select those plug-ins belong to the application client. After having selected the corresponding plug-ins, click *Finish* and the new feature project will be created in your current workspace.

NOTE: The feature creation wizard checks the dependencies, each plug-in needs to run in an installed environment. A plug-in, which poses as application client plug-in, has a dependency to the Eclipse client container plug-in *dcx.appclient*. This dependency is taken over into the feature configuration. The Eclipse update manager component checks the local installation before downloading and installing new features from the update site. The update manager recognizes that *dcx.appclient* is not available on the HTTP update server and aborts the installation mechanism without throwing any exception (this is a bug of Eclipse 3.0.1). Therefore, **delete the dependency to *dcx.appclient* in the feature manifest.**

7.4 Set Up an Eclipse Update Site

This section will describe the last step of a complete development circle. Application client plug-ins are distributed with the help of an HTTP update server. The ECC Deployment Manager connects (via the Eclipse update manager component) to the update server, downloads and installs the latest releases of plug-ins for the user who is currently logged in. This section will show the steps to perform, to get a working update environment: it shows how to set up an Eclipse update site.

7.4.1 Preparations

As already addressed in section 7.3.3, plug-ins must be logically aggregated to a feature to get distributed to the installed rich clients. Therefore it is mandatory that you have completed the steps, which have packaged the involved plug-ins to a deployable feature.

An Eclipse update site consists of two parts: two directories, named 'plugins' and 'features', which contain the distributable features and their plug-ins. The other part is the so-called site

map - a XML file, which describes the structure of the update site and gives information about downloadable features.

For the next steps, it is assumed that a development environment is set up as described in chapter 6 and that Eclipse IDE is installed on the same machine where the HTTP server is located. To get started, a new update site must be created. Click *New > Project > Plug-in Development > Update Site Project*. Type in a project name, e.g. *iap-update-site*. Deselect the checkbox *Use default* and select the document root directory of the HTTP server (e.g. C:\ibm\IBMHttpServer\htdocs\de_DE) and create a new subdirectory (e.g. C:\ibm\IBMHttpServer\htdocs\de_DE\iap-update-site). Eclipse update sites can be located outside the Eclipse workspace and therefore, the update site project will be directly deployed to the server.

7.4.2 Add Features to the Update Site

The features, which could be downloaded, have to be located on the update site, before the update site can be used by the Eclipse client container installations. It can be seen as a repository of features.

To add features click *Add* in the left section and select the features you want to deploy with the update site. In this case, select *dcx.iapsamples* (if you have chosen this name in section 7.3.3 when packaging your plug-in to a feature) and click *Finish*. At this point, only the entries have been made in the site map. The feature doesn't exist physically in the update site directory yet. To export the feature to the update site, click *Build All*. Now, the feature will be packaged (as it is done by the Eclipse feature export wizard) and exported.

The available features in this update site must be published. To do so, select the feature and drag and drop the feature to the right section. The feature should appear as an entry of the list now.

7.4.3 Setting the Update Site in Eclipse Client Container

The Eclipse client container doesn't know the update site yet. To make it known to the ECC, open the binary JAR file of your ECC distribution and edit the *settings.properties* file in the package *dcx.appclient.container*. Set the parameter *clientcontainer.eclipse.updatesite.url* to the HTTP URL of your update site (if you have followed the steps as described before, the URL can be *http://<fully-qualified-servername>/iap-update-site*).

7.5 Closing the Chapter

The different sections above have shown the steps, which are needed to develop an application client plug-in for the Eclipse client container. It has been described, how to create a basic rich client application and how to distribute the application client plug-ins with the help of an HTTP

update server. You should now have a better understanding of how the different involved components interact with each other.

8 Implementation Details

This chapter presents extracts of the practical realization of solution B (client container runs inside Eclipse, chapter 5). Solution A will not be discussed here, as there were no programming issues. The chapter especially focuses on the modifications, which had to be done, to embed the client container as a plug-in in Eclipse. Another section discusses how to port a user interface, implemented in Swing, to the Eclipse proprietary implementation SWT. The last section presents ECC Deployment Manager implementation details.

8.1 Changes from "Standalone" to Eclipse Client Container

Three base requirements for implementing solution B were established at the beginning of the diploma thesis. These requirements had to be considered during the whole implementation phase.

8.1.1 No Changes in Eclipse code

Eclipse is a standardized software product. The Eclipse source code should be left as provided by the Eclipse project. The Eclipse client container implementation has to be independent from the Eclipse distribution. A complete manual Eclipse distribution rebuild would be necessary, if changing the source code. Thus, developers would not be able to download Eclipse from the Eclipse homepage and install and run the client container properly with this version.

8.1.2 No Changes on Server-side

The server-side components, mostly Enterprise Java Beans, provide business logic for the client applications. These components are independent from the implementation of the client-side (standalone mode of client container or operation inside Eclipse). Therefore, no changes on server-side installed components should be made.

8.1.3 Only as Many Changes as Necessary in Client Container Code

The current IAP client container release is a stable product already running properly in several customer installations. The client container implementation should essentially be left as provided in the current version. Changes should only be made, where the client container has to operate with Eclipse or application client plug-ins. An example is the client container launcher - for details see section 8.3. If changes are necessary in the client container source code, the new Eclipse-specific implementation should strongly follow the already used patterns. A consequence of this requirement is, that adaptations are made with respect to the independence of the run-time environment: components have to detect whether they currently run inside Eclipse or as a standalone application. An example for this requirement is the use of Swing or SWT. The client container detects that it is currently used in standalone mode and therefore uses Swing for presenting the user dialogs. Otherwise, if it runs inside Eclipse, the SWT implementation of the UI dialogs must be used. For details of this implementation, see section 8.2.

Following this introduction to the implementation details, this chapter will concentrate on some realization scenarios. At first, it will be discussed how to port Swing GUIs to SWT and adding them to the client container implementation. After that, a section describes the main issue of this diploma thesis: the implementation of a new client container launcher, which is responsible for the complete plug-in management process.

8.2 Porting Swing GUI to SWT and Adding to the Client Container

The client container provides a security concept to protect application clients against unauthorized access. A user has to authenticate first before he is able to use application clients.

The complete security scenario is realized as an extra component, using the Java Authentication and Authorization Service (JAAS). A short introduction in this component is given first to understand what changes have to be done in the UI implementation. The login implementation is much more complex than described here.

The client container uses the standard mechanism of the SUN Corp. for HTTP BASIC authentication. Authentication is performed in a non-preemptive way (non-preemptive: an activity is executed until the server requests an authentication on client-side - preemptive: authentication details like username and password are sent with every HTTP request).

Figure 8.1 shows a sequence diagram of a user login. During startup, the client container requests an own JAAS-LoginModule implementation (*AppHttpLoginModule*) to perform the login on server-side. Therefore, the LoginModule tries to get the user information as a JAAS-Principal from the application server (remote call to an EJB). The server answers with error 401 (authentication required), because the resource is protected and access has to be granted with HTTP BASIC authentication. The LoginModule creates the login dialog on client-side. The dialog

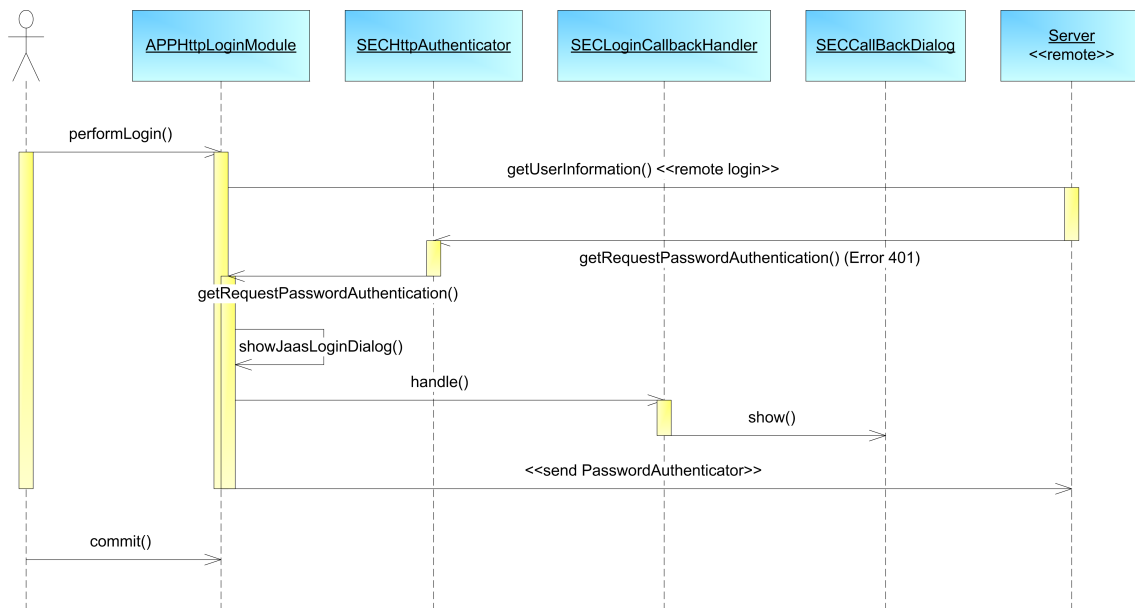


Figure 8.1: Login sequence

requests the users' username and password and sends back a *PasswordAuthenticator* object (with username and password) to the server.

The login dialog is implemented as JAAS callback handler and is created dynamically during runtime by the LoginModule. Each dialog component (e.g. background image, text field and buttons) is also encapsulated in a callback handler, that is, the callback dialog consists of several sub-callback handlers. As Figure 8.2 shows, the callback handler references a callback dialog class providing handler methods to create UI representatives of the different sub-callback handlers.

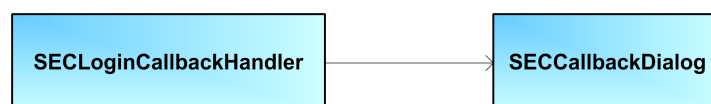


Figure 8.2: Callback references handlers

These handler methods depend on the UI implementation. They were able only to handle Swing components in the client container standalone mode.

Original handler methods look like this:

Listing 8.1: A Swing handler method

```

1 public JTextField addNameField( String prompt, String defaultName, final
  int maxCharacters, boolean isEditable) {
2     GridBagConstraints constraints = new GridBagConstraints();
3     JLabel label = new JLabel( getResourceString( prompt));
    
```

8 Implementation Details

```
4     inputPanel.add( label);
5     [...]
6     layout.addLayoutComponent( label, constraints);
7
8     final JTextField textField = new JTextField();
9     textField.setPreferredSize( new java.awt.Dimension( 300,20));
10    if (!isEditable) {
11        textField.setEnabled( false);
12    }
13    if (defaultName != null) {
14        textField.setText( defaultName);
15    }
16    [...]
17    inputPanel.add( textField);
18    [...]
19    return textField;
20 }
```

This example of such a method creates a name input component inside the login dialog. It consists of a label ("Name :") and a text field prompting for a username. The dialog part is fully implemented in Swing. The method itself returns a Swing-specific *JTextField*.

Porting this handler method to SWT, there are several problems to solve.

1. Each GUI toolkit requires an own implementation of the handler methods
The SWT class hierarchy (e.g. container hierarchy) differs completely from the Swing implementation. Furthermore, method names and method headers are different in SWT and Swing.
2. The handler methods must be generalized
A method may not return a Swing class, but return a class containing the component data from Swing as well as from SWT.
3. The callback handler has to use the generalized object
The callback handler class must be changed to get UI component data from the generalized (UI implementation independent) object.
4. Adding a factory
A factory creates a new UI implementation instance and returns it to the callback handler. This factory has to detect whether it has to create a Swing or a SWT instance.

The first step to solve these problems, was to implement all GUI components (name input field, password input field, background image, ...) with the SWT toolkit. Therefore, the user dialog structure had to be analyzed and the suitable SWT correspondences for the UI components had to be found. The SWT developers created a GUI toolkit providing nearly the same features in SWT. The method names differ in some cases from the Swing method names. SWT uses another

concept of adding a component to a parent container: while Swing provides "add-methods", SWT uses the component constructor.

Listing 8.2 shows the SWT implementation of the same handler method above (listing 8.1):

Listing 8.2: A SWT handler method

```
1 public TextFieldWrapper addNameField(String prompt, String defaultName,
2   int maxCharacters, boolean isEditable) {
3     Label label=new Label(inputPanel, SWT.NONE);
4     label.setText(getResourceString(prompt));
5
6     GridData gd=new GridData(GridData.HORIZONTAL_ALIGN_FILL);
7
8     Text textField=new Text(inputPanel, SWT.SINGLE | SWT.BORDER);
9     textField.setTextLimit(maxCharacters);
10    textField.setLayoutData(gd);
11    if (!isEditable)
12        textField.setEnabled(false);
13    if (defaultName!=null)
14        textField.setText(defaultName);
15
16    return new TextFieldWrapper(textField);
17 }
```

Next, the method header (line 1 in listing 8.2) had to be changed to a GUI toolkit independent implementation. New handler methods return wrapper objects to the callback handler instead of the real GUI component objects. This wrapper class is able to host a Swing *JTextField* component as well as a SWT *Text* component. It provides a *getText()* method to retrieve the content of a text field. Depending on the wrapped text field implementation object, the wrapper returns the corresponding content as *String*.

Listing 8.3: Swing-SWT-Wrapper class

```
1 public class TextFieldWrapper {
2     private JTextField jTextField;
3     private Text swtTextField;
4
5     public TextFieldWrapper(JTextField textfield) {
6         jTextField=textfield;
7     }
8
9     public TextFieldWrapper(Text textfield) {
10        swtTextField=textfield;
11    }
12
13    public String getText() {
14        if (jTextField!=null && swtTextField==null) {
15            return jTextField.getText();
16        }
17    }
18 }
```

8 Implementation Details

```
17         else if (jTextField==null && swtTextField!=null) {
18             return swtTextField.getText();
19         }
20         else return null;
21     }
22
23     public char[] getChar() {
24         return getText().toCharArray();
25     }
26 }
```

Wrapper classes are implemented for text fields, combo boxes and confirmation buttons. If new GUI toolkits are added to the security component, the corresponding GUI component implementations must be added to the wrapper classes.

Adding a new GUI toolkit to the client container in the future, should be easy. To achieve this, a new interface (*SECILoginDialog*) is created and located above the two existing Swing and SWT implementation classes. The interface contains all method headers. A UI implementation class (such as the new SWT class) has to implement this interface. Figure 8.3 shows a class diagram after porting Swing implementation to SWT (compare to figure 8.2).

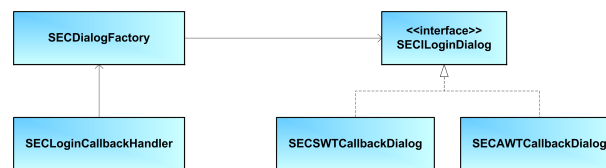


Figure 8.3: Class structure after porting

A newly created factory decides which UI implementation must be created. The factory checks the runtime environment. If an instance of the SWT Display class is available, the factory returns a SWT implementation object, otherwise a Swing implementation object.

8.3 Implementing a New Plug-in Launcher

The launcher is one of the central components of the client container. It is responsible to perform three tasks: the launcher first of all creates the JNDI namespace. Second, within that, it stores all configuration data. Finally, the launcher is responsible to start and run the application clients deployed in the client container. The next sub-sections introduce the traditional standalone launcher and discuss the newly implemented Eclipse plug-in launcher.

8.3.1 The Launcher in the Standalone Client Container

A standalone client container uses the *AppLauncher* to manage and start application clients. Figure 8.4 shows the sequence diagram which demonstrates the tasks, which the launcher has to perform during client container startup.

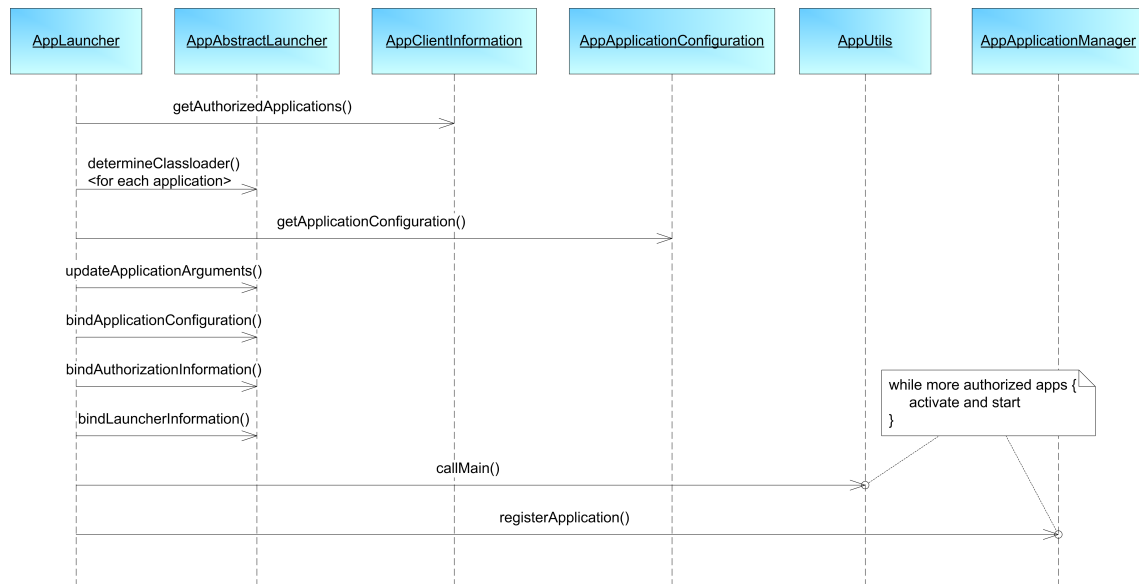


Figure 8.4: Tasks of the standalone launcher

The client container asks the remote site (application server), via an *AppClientInformation* instance, for the authorized application clients. It returns a list of all application clients which the user, who is currently logged in, is allowed to start and utilize.

After a new class loader is created for each application, the launcher component calls for the configuration data of each application. This contains

- the application ID
- the URL to the RPC servlet of this application
- the security roles the application contains.

The command line arguments will be updated now. Each application client receives the command line arguments. The launcher inserts the application ID into the first position of the command line arguments array.

The launcher binds all collected data (containing application and security information) into the JNDI namespace.

All application clients will now be started by calling the `main()`-method of each. The main method receives the array of command line arguments. Having started an application successfully, it will be registered at an *AppApplicationManager* which controls the proper shutdown of the client container when all applications are terminated.

8.3.2 The Eclipse Plug-in Launcher

Implementing the client container as a plug-in for Eclipse, changes the concept of application clients completely. Therefore a new launcher has to be implemented according to the following requirements:

No applications, but plug-ins

Eclipse consists of plug-ins. The traditional concept of a main method called by the Java virtual machine does not exist in Eclipse. Plug-ins can be installed and activated during runtime. They will be started when the user starts a plug-in functionality. The new launcher has to install and activate Eclipse plug-ins further on.

No client container class loader, but Eclipse class loader

Eclipse implements its own class loading concept. Traditional applications in a standalone client container use a class loader concept provided by the client container. The new Eclipse client container launcher has to adapt to the class loading concept of Eclipse.

No launcher directory, but plug-in directory

Traditional applications in a standalone client container are located in the launcher directory of the client container home directory. The class loader searches for compiled Java classes and executes them from this launcher directory. Eclipse uses a similar concept. A sub-directory of the Eclipse home directory hosts all installed plug-ins. Due to an incorporated security concept, the application client plug-ins are stored in directories (extension sites) other than the Eclipse plug-in directory. The new launcher has to activate plug-ins from such an extension site.

The following sequence diagram (figure 8.5) shows the process of starting application client plug-ins with the new Eclipse plug-in launcher.

Like the traditional launcher, the Eclipse plug-in launcher also retrieves a list of all authorized applications. In this case applications represent Eclipse plug-ins.

The creation of an application class loader is not needed, due to the own class loading concept of Eclipse. A new Eclipse class loader will be created later, when starting the plug-in.

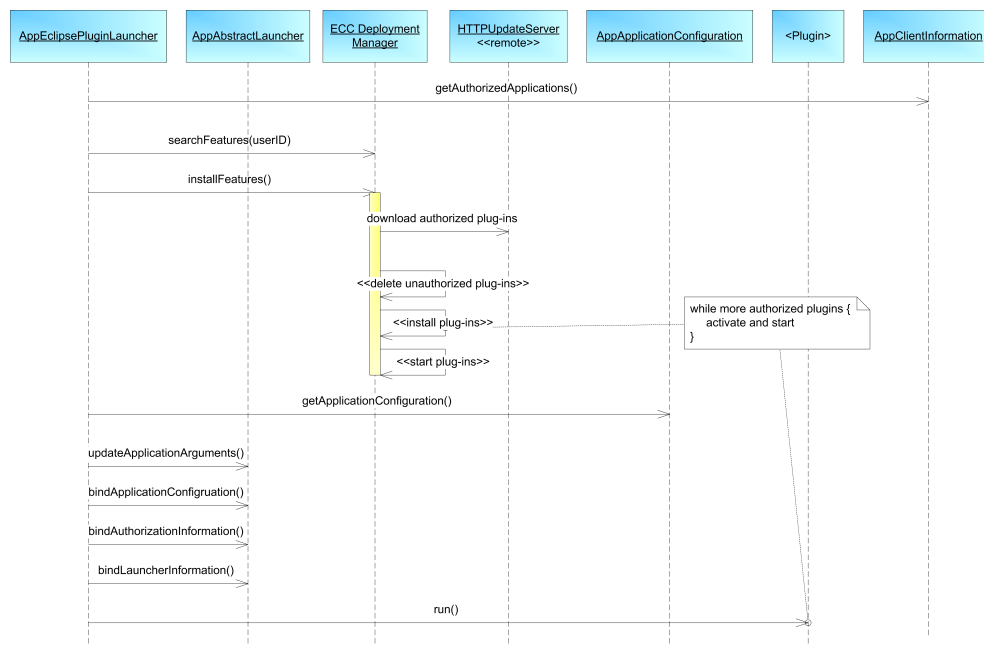


Figure 8.5: Tasks of the Eclipse plug-in launcher

The Eclipse plug-in launcher asks the ECC Deployment Manager to search for new plug-ins or for updates of already installed plug-ins instead. In a second step, the ECC Deployment Manager will install these plug-ins and updates (for details of this process, see section 8.4).

After having installed and updated the system successfully, the launcher binds the plug-in configuration data to the JNDI namespace in the same way as the traditional launcher does. Finally, the launcher fills an *ApplicationContext* object for each plug-in. This context contains runtime information for a plug-in such as command line arguments or application ID. Remember that all application client plug-ins have to extend an extension point *dcx.appclient.applicationPlugin* that requires implementing an interface *IApplicationClient*. This context is delivered to the *run()*-method of the implemented interface of each application client plug-in.

8.3.3 Changes from Standalone to Eclipse

Porting the client container as a standalone container to an Eclipse plug-in requires some changes in the client container source code:

Extract methods and pull them up

A process analysis identified the functionality that is not changed in the Eclipse client container environment. This mainly relates to the update mechanism of the command line arguments and

the JNDI binding mechanism. All application client plug-ins receive the command line arguments with the application ID as first parameter. Furthermore, all configuration data about an application client plug-in, including security and launcher data, are saved in the JNDI namespace. All these functionalities, which were originally located in the *AppLauncher* class, are extracted and pulled up in a super class *AppAbstractLauncher*. The two launchers *AppLauncher* and *AppEclipsePluginLauncher* extend this super class. Figure 8.6 shows a class diagram with the new launcher structure.

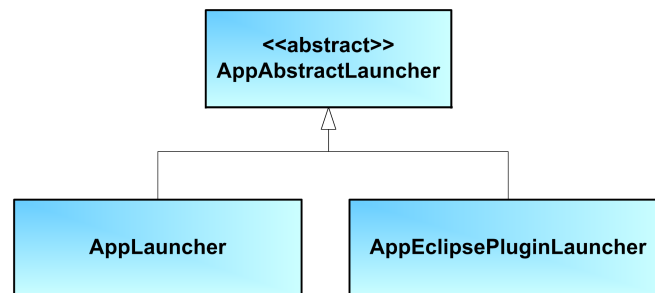


Figure 8.6: The launcher class structure

Implementing a new launcher for Eclipse plug-ins

Having created this super class, any new launcher can be inserted into the client container in the future. A launcher has to implement a "launch()" -method. The launch process is divided into two sub-methods "prepareStartup()" and "startup()". The implementation cannot be dictated by the abstract super class, due to the different method headers.

However, the Eclipse plug-in launcher follows this convention as well. The "prepareStartup()" method of the Eclipse plug-in launcher interacts with the ECC Deployment Manager and fills the JNDI namespace. The "startup()" -method instantiates the extension point implementation and calls the run() -method of each application client plug-in.

Selecting a launcher before startup

The launcher class name must be set in the settings.properties file of the client container. A new check mechanism is implemented, analysing the runtime environment. It utilizes the traditional *AppLauncher*, if the client container is started standalone. In case of running inside Eclipse, the *AppEclipsePluginLauncher* is instantiated.

8.4 ECC Deployment Manager

As already discussed in chapter 5.1.5, manageability of a client installation is one of the key requirements in a production environment. Administrators must be able to distribute function updates, bug fixes and patches to all Eclipse client container installations. While Java Web-Start or IAP StarterApplet distribute application client update of a standalone client container, a mechanism had to be found to find an equivalent distribution functionality for the Eclipse client container.

The Eclipse client container introduces a new component: the *ECC Deployment Manager*. This update component uses features of the *Eclipse update manager* (EUM). Developers can use this update component with a UI, but also programmatically with its open API.

This section addresses the process of finding features on the update server and installing them on the client side. Due to the simple EUM API, the component is represented by one class (*UpdateManager*) with two main methods "searchFeatures(...)" and "installFeatures(...)" (see listings A.1 and A.2 in the appendix).

8.4.1 Searching for Features

Chapter 5.7.2 introduces an HTTP update server where all application client plug-ins and its updates and patches are hosted. Each client installation can ask this server for available features and can download them via HTTP. Such an update site has an object representation in the EUM-API named *org.eclipse.update.core.ISite*. A helper class (*org.eclipse.update.core.SiteManager*) returns such an *ISite* instance on the basis of a *java.net.URL* (line 4 in listing A.1) representing the path to the update server.

The returned *ISite* instance can be asked to return an array of *org.eclipse.update.core.ISiteFeatureReferences*. An *ISiteFeatureReference* maps the physical feature to a Java object. This reference saves information about version, feature id, download size, etc.

Having found all available features on the update server, the search algorithm compares the identified features with a list of authorized application client features. The process compares the list of authorized application IDs with each feature ID, found on the server. If the application IDs correspond (that is if a user is allowed to utilize a feature that is located on the update server), the *ISiteFeatureReference* is saved in an array.

After having analysed all features, the array of downloadable features is returned to the *APP-EclipsePluginLauncher* which starts the download and installation process which is discussed in the next section.

8.4.2 Installing Features

The ECC Deployment Manager provides a method "installFeatures(...)" to download and install the list of features it has received from the search algorithm. It iterates the array an *ISizeFeatureReferences* and instructs a *DCXInstallCommand* instance to download and install the current feature.

The *DCXInstallCommand* class is a bug-fixed copy of the Eclipse *org.eclipse.update.standalone.-InstallCommand* class.

By copying this class, the implementation violates the convention defined in section 8.1.1. This convention states not to change any Eclipse code. However, in this case, the application of an implementation class other than *InstallCommand* is possible, because the choice of an install process is configurable.

The *InstallCommand* class installs a feature, downloaded from an update server, to an Eclipse extension site. It tries to create a new extension site each time a new feature has to be installed. An exception is thrown, if the extension site already exists. This diploma thesis discovered the malfunction of the *InstallCommand* the first time (for details see [16]). Therefore, no patch or workaround existed in the Eclipse Bugzilla and an own workaround implementation had to be found. The class *DCXInstallCommand* is an exact copy of the *InstallCommand*, but contains a patch to be able to install features to an extension site which already exists.

Listing 8.4: Patch for the *InstallCommand*

```
1 //DCX-Patch for installing more than one feature at once
2 try {
3     targetSite = getConfiguration().createConfiguredSite(sitePath);
4 } catch (CoreException e) {
5     IConfiguredSite[] confSites=getConfiguration().getConfiguredSites();
6     File extensionPath = new File(toSite+"/eclipse");
7     for (int i = 0; i < confSites.length; i++) {
8         IConfiguredSite ics=confSites[i];
9         if (ics.getSite().getURL().sameFile(extensionPath.toURL()))
10             targetSite=ics;
11     }
12 }
```

Listing 8.4 shows the implementation of the corresponding patch. Initially, *DCXInstallCommand* will try to create the new extension site. If a *CoreException* is thrown, an array of all known extension sites is returned. An iteration through this array searches the currently needed extension site and returns it when found. In the next versions of Eclipse (from M4 on), this bug is fixed due to the contribution of this diploma thesis.

Before starting the installed features, the "organizeDirectories()" -method deletes all plug-ins in a user extension site, which the user is not allowed to utilize. It compares the list of authorized

plug-ins with the directory entries in the extension site. If a plug-in is installed, which must not be started, this method will delete it physically.

The result is an extension site which contains only those features, the user is allowed to utilize and which are up-to-date. Finally, the "installFeatures(...)"-method extracts all plug-ins of each feature and start them.

Using the Eclipse update manager via the UI inside Eclipse could cause a workbench restart or refresh, because features that contribute to the Eclipse UI have to insert their components to the workbench. In case of the ECC Deployment Manager, the update and installation process takes place before showing the workbench. All features can contribute their UI components in the background. When all features are started by the ECC Deployment Manager, the workbench is presented to the user with all contributions of the application client plug-ins.

9 Summary

This chapter examines and concludes on the presented two solutions. A brief outlook on future developments will follow.

9.1 Approaches

This diploma thesis is dealing with two different container approaches: Eclipse as container for standalone applications and the client container as container for J2EE applications, which communicates with application servers. Both containers provide features that simplify the application development. However, in some cases it would be useful to build J2EE applications (features of the client container) with a native UI (feature of Eclipse), by building applications based on the combination of both approaches. The diploma thesis discusses two possibilities how to combine Eclipse and client container to a powerful consortium:

9.1.1 Eclipse Runs in Client Container

In this solution Eclipse, with its plug-ins, is deployed as a complete application into the client container. In order to make client container services available in Eclipse, the Eclipse class loader hierarchy had to be adapted. While this change took place in the source code in Eclipse 2, it is now possible to adapt the hierarchy with an OSGi configuration parameter in Eclipse 3.x. As the client container has to provide its services to Eclipse, and Eclipse has a strong encapsulation, the changes in the class loader hierarchy weaken the strong separation of plug-ins. The fundamental framework concept, in which plug-in namespaces are strictly encapsulated from their environment, is debilitated.

9.1.2 Client Container Runs in Eclipse

The second solution is more complex. It takes into account that the client container runs inside Eclipse. In this case, the client container is implemented as plug-in and is deployed equally together with all other plug-ins in Eclipse. It provides its services with the traditional Eclipse dependency mechanism.

All UI client container components, especially the login dialog, have been converted to SWT. As Eclipse plug-ins do not provide a user context for user authentication or authorization, a security

concept had to be introduced in Eclipse. The using of different user extension sites, featured by the Eclipse update manager, implements this concept. Furthermore, application client plug-ins are distributed by the newly implemented ECC Deployment Manager. Client installations are manageable and replace the traditional distribution method with Java WebStart or IAP Starter-Applet.

In this solution, the client container has to weaken its encapsulation in order to provide its features to Eclipse. Eclipse as a component framework is not weakened like in solution A. Additionally, this solution allows using Eclipse components (update mechanism, client container as plug-in) in an advanced way.

All feature requirements, which were discussed and established in the preliminary stages, have been implemented and tested in both approaches. The two implementations delivered the expected results.

9.2 Study Results

The diploma thesis is based on two already existing container frameworks. It was the idea to combine the container approaches to a new powerful application client framework. Eclipse and client container offer many features, which have been used to implement this new infrastructure while combining already existing functionalities.

This diploma thesis uses commercially available products in combination with newly implemented components. This section defines the split between existing and newly developed components.

At the beginning nearly all components already existed

- A runnable implementation of the client container
 - A launcher for standalone application clients
- A rich-client-ready Eclipse distribution with its sub-components
 - Update manager (API, Extension site, HTTP update server)
 - OSGi framework to manage the lifecycle of plug-ins

The diploma thesis objective was, to combine all involved components in an intelligent way.

The client container has been wrapped into an Eclipse plug-in. Its implementation has been adapted to manage the life cycle of Eclipse plug-ins in the context of the Eclipse client container (starting a plug-in if authorized). Therefore, a new launcher was created (section 8.3), which is capable to install plug-ins in a user extension site and start them afterwards.

All Swing components had to be ported to SWT (section 8.2) to be able presenting UI dialogs in an integrated manner. The client container had to be prepared for the use of a new UI implementation.

A client container environment defines a security concept with authentication and authorization. The client container component contributes with a role-based authorization concept. Eclipse has no user context, that means, users cannot be authenticated to utilize any installed plug-ins.

The main part of this diploma thesis consists of finding a way to introduce a proper authentication concept in Eclipse.

Eclipse provides an update mechanism for keeping Eclipse installations up-to-date. Consequently, the Eclipse update manager component provides an open API for programmatic use. The ECC Deployment Manager uses this API to provide the Eclipse client container security (section 8.4).

The diploma thesis required to use the update mechanism of Eclipse for the implementation of an inexistent user context in Eclipse.

9.3 Conclusions

9.3.1 Combining Eclipse and Client Container

The current releases and implementations of Eclipse and the client container allow combining both in an almost comfortable way. Especially solution B delivers a satisfying result.

From my point of view, solution A (Eclipse deployed in the client container - chapter 4) is not useful for a long-term application. Due to the class loader changes, developers have a substantial programming overhead. Additionally, Eclipse is losing its strong plug-in encapsulation. The approach can be used, however, as an interim solution, which will introduce Eclipse only to new developed applications. "Traditional" applications remain Swing-based until their expiration. When all applications are Eclipse-based, developers can switch over from solution A to solution B. For new projects, it is highly recommended, to develop applications, based on the solution B (Client container as plug-in in Eclipse - chapter 5) - although developers have to spend more time to build up the complete and complex infrastructure (e.g. setting up the HTTP update server). Applications can be created easier and quicker rather than using solution A.

Developers can just start creating application client plug-ins and need not to care about programming overhead concerning class loader problems of solution A.

In fact, security functionality is implemented by using different extension sites for each user. However, there is still a security lack remaining due to a missing plug-in user context: experts will be able to start plug-ins in areas where they have no permission. This security lack concerns only the UI specific parts of the application client plug-in. All other server-actions are checked by the server-side security component. The user can only view the client side UI components, the unauthorized plug-in utilization is prevented.

9.3.2 No Hesitation, but Courage

From my point of view, the approach of connecting two containers to a new formation has key advantages. The brain work and the effort to implement a theoretical approach will pay off: new application clients can be quickly implemented by utilizing the features of both containers, once this infrastructure has been created.

In some cases, a container implementation does not offer its services to its environment, due to its strong encapsulation. However, patterns can be found, to bridge or adapt the container in a way, that services can be accessed also from outside its encapsulation.

Due to missing documentation and lack of deep implementation details concerning the different technologies, it is often tedious to find a suitable solution. Forums and newsgroups can assist in retrieving information, but will not be able to provide the needed details. Very often only debugging and copy analysing of source code will help to find a solution for a specific problem. Developers should not be afraid of these barriers. The result of a combined container infrastructure will pay off soon.

9.4 Future Work

9.4.1 User Context with Eclipse 3.x

As already mentioned, the current Eclipse release 3.0.1 does not provide a plug-in user context which would be however important to provide a consistent security concept. Currently, the concepts which are presented in this diploma thesis, implement security with the help of user extension sites and the Eclipse update manager. Each user has his own location, where his plug-ins are stored. If a user is not allowed to utilize a plug-in, it will be deleted physically from the hard disc. New authorized plug-ins are downloaded and installed during runtime.

A plug-in user context, however, would make this mechanism obsolete. With additional configuration parameters a developer could set the execution rights of a plug-in and could control whether a user will be allowed to see the corresponding perspective or view. Furthermore, Eclipse should contribute an own configurable login screen implementation to authenticate users with a username and a password. With this information a role-based security concept could be realized. Dirk Bäumer told me during a session in Zurich [8] that the Eclipse developers plan the introduction of such a user context with a thorough authentication and authorization concept, allowing the control of user access to a plug-in. This user context is forecasted to be available in Eclipse release 3.1. A consequence of the new release will be a code review and a proof of the concept of the discussed solutions in this diploma thesis.

9.4.2 Updating the Client Container Plug-in

Currently, the Eclipse update manager component allows only the update of application client plug-ins. An administrator has to install Eclipse and its client container plug-in manually or automatically with IAP StarterApplet or Java WebStart. In most cases, however, it could make sense to install Eclipse and the client container plug-in only once. This would prevent the possibility to update the client container plug-in itself. Patches or bugfixes of the client container are necessary. In the case of existence of an update mechanism for the client container plug-in with the ECC Deployment Manager, the update and installation process via StarterApplet or WebStart could become obsolete. Consequently, the update manager component must be reengineered.

A Implementation of the ECC Update Manager

Listing A.1: Searching for new features

```
1 public ISiteFeatureReference[] searchFeatures(List authorizedApps) {
2     HashSet featureReferences=new HashSet();
3     try {
4         ISite site=SiteManager.getSite(new URL(dcxSearchSite), false, null);
5         ISiteFeatureReference[] remoteFeatures=site.getFeatureReferences();
6
7         for (int i = 0; i < remoteFeatures.length; i++) {
8             ISiteFeatureReference ref=remoteFeatures[i];
9             IFeature currentSearchedFeature=ref.getFeature(null);
10            String securityId=getFeatureIdAsSecurityId(currentSearchedFeature
11                );
12
13            for (Iterator iter = authorizedApps.iterator(); iter.hasNext();)
14            {
15                APPApplicationConfigData app = (APPApplicationConfigData) iter
16                    .next();
17                if (app.getSecurityId().equals(securityId)) {
18                    featureReferences.add(ref);
19                    APPSplashWindow.getSplashWindow().setMessageText("New
20                        feature found: "+currentSearchedFeature.getLabel()+ "
21                        for user "+APPSession.getSession().getUserPrincipal().
22                        getUserId());
23                    logger.log(SMISStatus.INFO, APPLog.LIDFNE, "New feature
24                        found {0} for user {1}", currentSearchedFeature.
25                        getLabel(), APPSession.getSession().getUserPrincipal()
26                        .getUserId());
27                    break;
28                }
29            }
30        }
31    } catch (CoreException e) {
32        e.printStackTrace();
33    } catch (MalformedURLException e) {
34        e.printStackTrace();
35    }
36
37    ISiteFeatureReference[] returnRefs=new ISiteFeatureReference[
38        featureReferences.size()];
39    return (ISiteFeatureReference[]) featureReferences.toArray(returnRefs);
40 }
```

Listing A.2: Installing new features

```
1 public boolean installFeatures(ISiteFeatureReference[] featureRefs, File
   extensionLocation) {
2     boolean result=true;
3     HashSet newPlugins=new HashSet();
4
5     for (int i = 0; i < featureRefs.length; i++) {
6         ISiteFeatureReference featureRef = featureRefs[i];
7
8         try {
9             APPSplashWindow.getSplashWindow().setMessageText("downloading: "+
               featureRef.getVersionedIdentifier());
10            logger.log(SMISStatus.INFO, APPLog.LIDFNE, "downloading {0}",
               featureRef.getVersionedIdentifier());
11
12            //create extension site (if needed/not already created)
13            DCXInstallCommand ic = new DCXInstallCommand(featureRef
               .getVersionedIdentifier().getIdentifier(),
14                featureRef
               .getVersionedIdentifier().getVersion().
               toString(),
15                dcxSearchSite, extensionLocation.
               getAbsolutePath(),
16                "false"); //$NON-NLS-1$
17
18            String pluginLoc=extensionLocation.getAbsolutePath()+"/eclipse/
               plugins/"; //$NON-NLS-1$
19            //extract all (new) plug-ins contained in this feature to install
               later
20            IPluginEntry[] pes=featureRef.getFeature(null).getPluginEntries()
               ;
21            for (int j = 0; j < pes.length; j++) {
22                IPluginEntry pe=pes[j];
23                String storeLoc=pluginLoc+pe.getVersionedIdentifier();
24                newPlugins.add(new File(storeLoc).toURL());
25            }
26
27            //download if needed
28            result = ic.run();
29            logger.log(SMISStatus.INFO, APPLog.LIDFNE, "download successful:
               {0} ", result+"");
30        } catch (Exception e) {
31            e.printStackTrace();
32        }
33    }
34
35    //delete unauthorized plug-ins physically
36    organizeDirectories(featureRefs, newPlugins, extensionLocation);
37
38    //Installation of all new plug-ins
39
```

```

40     if (newPlugins.size()>0) {
41         BundleContext bCtx=AppclientPlugin.getBundleContext();
42         String[] bundlesToInstall = getBundlesToInstall(bCtx.getBundles(), (
            URL[])newPlugins.toArray(new URL[newPlugins.size()]));
43
44         for (int i = 0; i < bundlesToInstall.length; i++) {
45             boolean installable=true;
46             String pluginDir=new File(bundlesToInstall[i]).getName();
47             String pluginId=(pluginDir.substring(0, pluginDir.indexOf("_")).
                replace('.', '_'));
48
49             try {
50                 if (installable) {
51                     //install the bundle/plugin
52                     logger.log(SMISStatus.INFO, APPLog.LIDFNE, "Now starting
                        plug-in {0}", FILE_PREFIX+bundlesToInstall[i]);
53                     Bundle b = bCtx.installBundle(FILE_PREFIX+bundlesToInstall[
                        i]);
54                     b.start();
55                 }
56             } catch (BundleException e) {
57                 e.printStackTrace();
58             }
59         }
60     }
61
62     return result;
63 }
```

B Accompanying CD-ROM

This CD-ROM contains this elaboration in PDF format and a German article about this diploma thesis.

Glossary

ACP	Application Client Plug-in; an Eclipse plug-in adapted to be used in the Eclipse client container.
API	Application Programming Interface; a formalized set of software calls and routines that can be referenced by an application program in order to access supporting services.
Application Server	A platform to run interoperable applications.
AWT	Abstract Windowing Toolkit; A class library that provides the standard API for building GUIs for Java programs.
Bytecode	The output of compiling a Java source program.
Component	A software object, meant to interact with other components, encapsulating certain functionality or a set of functionalities.
Container	An entity that provides life cycle management, security, deployment, and runtime services to components. Each type of container (EJB, web, JSP, servlet, applet, and application client) also provides component specific services.
CPL	Common Public License; a software license for open source projects.
Design Pattern	A solution to a problem in context; that is, it represents a high-quality solution to a recurring problem in design.
ECC	Eclipse client container; the container combination, resulting of the second approach of this diploma thesis. See chapter 5
ECC Deployment Manager	The update manager component of the Eclipse client container. The component is responsible to manage Eclipse installations. See chapter 5.7.2
EJB	Enterprise Java Bean; reusable and portable software components that model business objects and processes.

Framework	A body of software designed for high reuse, with specific plugpoints for the functionality required for a particular system. Once the plugpoints are supplied, the system will exhibit behavior that is centered around the plugpoints.
GUI	Graphical User Interface ; a computer terminal interface, such as Windows, that is based on graphics instead of text.
HTTP	Hypertext Transfer Protocol ; HTTP is the protocol used between a Web browser and a server to request a document and transfer its contents.
IAP	Integrated Application Platform ; the product of DaimlerChrysler ITI/TP. It offers the basic infrastructure for DaimlerChrysler internal application projects.
IAP StarterApplet	A IAP component to distribute Java applications over the internet. It extends functionality of Java Webstart.
IBM Websphere	IBM WebSphere is the leading software for on demand business, delivering business integration, application and transaction infrastructure, portals, application transformation, mobile/speech middleware, product information management and e-commerce.
IDE	Integrated Development Environment ; an application or set of tools that allows a programmer to write, compile, edit, and in some cases test and debug within an integrated, interactive environment.
J2EE	Java 2 Enterprise Edition ; a Java-based, runtime platform created by Sun Microsystems used for developing, deploying, and managing multi-tier server-centric applications on an enterprise-wide scale. J2EE builds on the features of J2SE and adds distributed communication, threading control, scalable architecture, and transaction management.
JAAS	Java Authentication and Authorization Service ; a framework for different authentication and authorization services built on top of the Java platform.
JAR file	Java Archive ; a platform-independent file format that permits many (Java) files to be aggregated into one file.
Java Webstart	A technology to deploy Java applications with a single click over the network.

JSP	Java Server Pages ; a scripting language based on Java for developing dynamic Web pages and sites.
JVM	Java Virtual Machine ; a computing machine for interpreting compiled Java bytecode.
Netegrity Siteminder	An access control platform enabling companies to administer and consistently enforce user access to web applications abd by providing single sign-on services to users.
OSGi	Open Services Gateway Interface ; a specification that defines a standardized, component oriented computing environment for networked services.
plugin.xml	Manifest file of an Eclipse plug-in, where the behaviour of the plug-in is configured.
RCA	Rich Client Application ; an application running on a client workstation that uses features of the OS (e.g. the native look and feel).
RCP	Rich Client Platform ; a basic platform to build rich client applications.
rt.jar	A Java library containing all classes of the Java specification.
SSL	Secure Socket Layer ; a security protocol that provides communications privacy over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery.
SSO	Single Sign On ; a system that enables a user to access multiple computer platforms or application systems after being authenticated only once.
Swing	A GUI toolkit that provides a further set of components that extend the capabilities of AWT.
SWT	Standard Widget Toolkit ; A GUI toolkit delivers native widget functionality for the Eclipse platform in an operating system independent manner.
XML	EXtensible Markup Language ; a widely used standard from the World Wide Web Consortium (W3C) that facilitates the interchange of data between computer applications.

Bibliography

- [1] Erich Gamma and Kent Beck: Contributing to Eclipse,
Addison Wesley, 2004, page 70
- [2] Ed Burnette: Rich Client Tutorial, Part 1
<http://www.eclipse.org/articles/Article-RCP-1/tutorial1.html>
- [3] Martin Lippert and Markus Völter: Rich clients mit Eclipse 3
<http://www.voelter.de/data/articles/jmEclipseRCP.pdf>
- [4] Sherry Shavor, Jim D'Anjou, Scott Fairborth, Dan Kehn, John Kellerman, Pat McCarthy:
Eclipse, Anwendungen und Plug-Ins mit Java entwickeln
Addison Wesley, 2004
- [5] Todd Williams: The Case for Using Eclipse Technology in General Purpose Applications,
01/2005
<http://www.genuitec.com/products/eclipseapplicationframework.pdf>
- [6] Dirk Bäumer and André Weinand: Mehr als eine Plattform
Article in Eclipse Special, Special Edition of JavaMagazin, page 25 ff., Software and Support Verlag, 2004
- [7] Dirk Bäumer: Die Eclipse Rich Client Platform
Session of Java User Group Switzerland, Zurich, 12.10.2004
- [8] Personal communication with Dirk Bäumer
Session of Java User Group Switzerland, Zurich, 12.10.2004
- [9] Mike Milinkovich, Executive Director Eclipse Foundation *Article in Eclipse Special, Special Edition of JavaMagazin, page 7., Software and Support Verlag, 2004*
- [10] Nick Edgar: Eclipse Rich Client Applications *Session on EclipseCon 2004*
[www.eclipsecon.org/2004/ ...](http://www.eclipsecon.org/2004/...)
[EclipseCon_2004_TechnicalTrackPresentations/11_Edgar.pdf](http://www.eclipsecon.org/2004/TechnicalTrackPresentations/11_Edgar.pdf)
- [11] Martin Lippert: Classloading in Eclipse
Article in JavaSpektrum, page 69, SIGS-DATACOM GmbH, 2005
- [12] The Eclipse project, 01/2005

- <http://www.eclipse.org>
- [13] Eclipse Project Description, 01/2005
<http://www.eclipse.org/org/main.html>
- [14] Article PDE does Plug-ins, 01/2005
[http://www.eclipse.org/articles/ ...
Article-PDE-does-plug-ins/PDE-intro.html](http://www.eclipse.org/articles/...Article-PDE-does-plug-ins/PDE-intro.html)
- [15] The Eclipse projects, 01/2005
<http://www.eclipse.org/eclipse/main.html>
- [16] Eclipse Bugzilla Bug 77100: InstallCommand fails with existing extension site, 01/2005
http://bugs.eclipse.org/bugs/show_bug?id=77100
- [17] The OSGi technology, 01/2005
http://www.osgi.org/osgi_technology/index.asp?section=2
- [18] Harish Grama, Keith Attenborough, John Banks-Binici, Jim Marsden, Carl Kraenzel, Jeff Calow, Shankar Ramaswamy, Mary Ellen Zurko: IBM Workplace Client Technology - Technical Overview
<http://www.redbooks.ibm.com/abstracts/redp3884.html?Open>
- [19] Java™2 Platform Enterprise Edition Specification, v1.4, 12/2004
http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf
- [20] J2SE1.4.2 API specification, 12/2004
[http://java.sun.com/j2se/1.4.2/docs/api
/java/lang/Thread.html#getContextClassLoader\(\)](http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Thread.html#getContextClassLoader())
- [21] WAS Security Integration User Guide: Client container in a production environment
DaimlerChrysler AG, 2004
- [22] IAP High Level IT Architecture: The scope of the IAP Architecture
DaimlerChrysler AG, 2004
- [23] IAP - Integrated Application Platform: Architecture of IAP 2.5 Platforms
DaimlerChrysler AG, 2004
- [24] Mohamed E. Fayad, Douglas C. Schmidt, Ralf E. Johnson: Building Application Frameworks
John Wiley & Sons, 1999

- [25] Desmond Francis D'souza, Alan Cameron Wills: Objects, components and frameworks with UML - The Catalysis Approach
Addison Wesley, 1998
- [26] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel: A Pattern Language
Oxford University Press, New York, 1977
- [27] Jim Doble: A Pattern Language for Pattern Writing
<http://www.hillside.net/patterns/writing/patterns.htm>
- [28] Java J2EE BluePrint: Core J2EE Patterns
<http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>
- [29] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns
Addison Wesley, 1995