An Evaluation of the Eclipse Rich Client Platform for a telecom management application

by

Philip Frising

LIU-IDA/LITH-EX-A--08/047--SE

2008-11-10
Final Thesis

An Evaluation of the Eclipse Rich Client Platform for a telecom management application

by

Philip Frising

LIU-IDA/LITH-EX-A--08/047--SE

2008-11-10

Supervisor: Amir Kadiric
Ericsson AB
Site Linköping

Examiner: Erik Berglund
Linköpings universitet
Department of Computer and Information Science
Abstract

The Software Management Organizer (SMO) application is used by telecom operators for remote software and hardware handling of telecommunication equipment. The graphical user interface (GUI) provided by SMO is called SMO GUI and is costly to maintain, extend and test.

The Eclipse Rich Client Platform (RCP) provides a platform for building component based GUIs with rich functionality. This thesis is to evaluate how the Eclipse RCP can be used for building a new SMO GUI. The evaluation will be performed by building a prototype for a new SMO GUI based on the Eclipse RCP. The thesis shall investigate the difficulty of integrating the prototype in the existing SMO architecture and evaluate how problems that exist in the current SMO GUI can be solved by using the Eclipse RCP.

It was possible to implement a prototype based on the Eclipse RCP that integrates with the existing SMO architecture. The Eclipse RCP provides good support for creating a flexible, extensible and testable application. The drawback of using the Eclipse RCP is the lack of support for detecting configuration errors and the lack of documentation of new features.

The Eclipse Rich Client Platform can be used as a base for building a new SMO GUI. An Eclipse RCP application can be integrated in the existing SMO architecture. The Eclipse RCP provides support for creating an application without the architectural, extendable, testable and quality problems that exists in the current SMO GUI.
Acknowledgement
I would like to thank everyone at SHM for your friendship and support. This thesis wouldn’t be half as good without the support of the following persons:

- Amir – for all your help with getting started and for supporting me during the thesis
- Åsa – for your interest and support throughout the thesis
- Tobias W – for taking the time to discuss Eclipse RCP, especially JFace Data binding and your valuable feedback during the thesis
- Peter L – for sharing your interesting ideas about SMO and your feedback and interest in my work
- Peter F – for your interest and help with my thesis
- Design Team 3 – for your support

I am also grateful for the feedback from my opponent, Malin.

Finally, I would like to thank my family for all your love and support.
# Table of contents

1. **Introduction** ................................................................. 1
   1.1 Background .............................................................................. 1
   1.2 Purpose .................................................................................. 1
   1.3 Method .................................................................................. 1
   1.4 Requirements ........................................................................ 1
   1.5 Limitations ............................................................................ 2
   1.6 Target audience ..................................................................... 2
   1.7 Outline .................................................................................. 2

2. **Problem description** ......................................................... 3
   2.1 Software Management Organizer ............................................ 3
      2.1.1 Architecture ..................................................................... 3
      2.1.2 SMO GUI ......................................................................... 4
   2.2 Problems ................................................................................ 5
      2.2.1 Architecture ..................................................................... 5
      2.2.1.1 Modules ....................................................................... 5
      2.2.1.2 Communication ............................................................. 5
      2.2.1.3 Layers .......................................................................... 5
      2.2.1.4 Thread .......................................................................... 5
      2.2.2 Extendibility ..................................................................... 6
      2.2.3 Testability ......................................................................... 6
      2.2.4 Application quality ............................................................ 6

3. **Eclipse Rich Client Platform** ............................................. 7
   3.1 Background ............................................................................ 7
   3.2 Architectural overview ........................................................... 7
   3.3 OSGi ....................................................................................... 8
      3.3.1 OSGi service platform architecture ................................... 8
      3.3.2 Component model ............................................................. 9
      3.3.3 Service model ................................................................. 11
      3.3.4 Equinox ........................................................................... 11
   3.4 Eclipse extension framework .................................................. 12
      3.4.1 Extension points ............................................................... 12
      3.4.2 Extensions ....................................................................... 13
   3.5 Standard Widget Toolkit ........................................................ 14
   3.6 JFace ....................................................................................... 14
   3.7 Workbench ............................................................................. 14
   3.8 Data binding .......................................................................... 15
      3.8.1 Observable ........................................................................ 15
      3.8.2 Realm .............................................................................. 15
      3.8.3 Binding, Converter and Validator ...................................... 15
      3.8.4 Content and label providers ............................................. 15
   3.9 Threads .................................................................................. 16
   3.10 Testing .................................................................................. 16
   3.11 Development Environment ................................................... 16

4. **Spring Dynamic Modules** ................................................ 19
   4.1 Overview .............................................................................. 19
   4.2 Dependency Injection ............................................................ 19
      4.2.1 Background ...................................................................... 19
      4.2.2 Spring bean configuration ................................................. 20
      4.2.3 Integration with OSGi services ....................................... 20
5 Prototype implementation ................................................................................................ 21
5.1 Architecture ............................................................................................................. 21
5.1.1 Bundles .............................................................................................................. 21
5.1.2 Layers ............................................................................................................... 22
5.2 Use of the extension framework ........................................................................... 24
5.3 View-Models ......................................................................................................... 24
5.4 Use of third party bundles .................................................................................... 24
5.4.1 JFace data binding ............................................................................................ 24
5.4.2 Spring Dynamic Modules ............................................................................... 26
5.4.2.1 Integration with extension points ................................................................. 26
5.4.2.2 Spring bundle ............................................................................................... 26
5.4.2.3 Integration with OSGi services .................................................................... 26
5.5 Testing .................................................................................................................... 27
6 Result .......................................................................................................................... 29
6.1 Overview ................................................................................................................ 29
6.2 Integration with the existing SMO architecture .................................................... 30
6.3 Evaluation of the Eclipse Rich Client Platform .................................................... 33
6.3.1 Architecture ...................................................................................................... 33
6.3.1.1 Modules ........................................................................................................ 33
6.3.1.2 Communication ............................................................................................ 33
6.3.1.3 Layers ........................................................................................................... 34
6.3.1.4 Thread ........................................................................................................... 34
6.3.2 Extendibility ..................................................................................................... 34
6.3.3 Testability ......................................................................................................... 34
6.3.4 Application quality ........................................................................................... 34
6.3.5 General drawbacks of the Eclipse Rich Client Platform .................................. 34
6.4 Summary ................................................................................................................ 35
7 Discussion and Conclusions ....................................................................................... 37
7.1 Discussion .............................................................................................................. 37
7.1.1 Integration with the SMO architecture ............................................................. 37
7.1.2 Eclipse Rich Client Platform ............................................................................ 37
7.2 Conclusions ............................................................................................................ 38
7.3 Future work .......................................................................................................... 38
7.3.1 SMO GUI ......................................................................................................... 38
7.3.2 Eclipse Rich Client Platform ............................................................................ 38
7.3.3 OSGi ............................................................................................................... 38
8 Glossary ..................................................................................................................... 39
9 References .................................................................................................................. 41
List of figures

Figure 1: SMO architecture 3
Figure 2: SMO GUI 4
Figure 3: Eclipse Rich Client Platform Architecture 7
Figure 4: OSGi service platform architecture 9
Figure 5: OSGi bundle 9
Figure 6: Eclipse architecture overview 12
Figure 7: Screenshot of an Eclipse Workbench 15
Figure 8: Plug-in Development Environment perspective 17
Figure 9: Bundle dependencies 23
Figure 10: SMO RCP prototype – Network perspective 29
Figure 11: SMO RCP prototype – Software perspective 30
Figure 12: SMO RCP prototype – File Store Perspective 31
Figure 13: SMO RCP prototype – Network Perspective with jobs represented in tables 32
Figure 14: SMO RCP prototype – Network Perspective with extra views 33
1 Introduction
This report is one component of the author’s master thesis in Computer Science and Engineering at Linköping University. The master thesis has been carried out at Ericsson AB in Linköping, Sweden. The thesis work was performed under supervision of Amir Kadiric at Ericsson and examined by Erik Berglund from the department of Computer and Information Science at Linköping University.

1.1 Background
Ericsson is a provider of telecommunications equipment and related services to mobile and fixed network operators globally. In Linköping the Operation Support System (OSS) Common Components department has the mission to develop, verify and deliver products in the area of telecom management.

The applications for software and hardware management are an important part of OSS. One of the major applications is the Software Management Organizer (SMO), which is responsible for remote software and hardware handling of GSM, UMTS and Core networks. SMO provides a graphical user interface (GUI), called SMO GUI, to the operator. The current GUI is based on the Java Swing widget toolkit and is costly to maintain, extend and test.

New OSS products use the Eclipse Rich Client Platform (RCP) for building graphical user interfaces. This thesis is to evaluate how the Eclipse RCP can be used for building a new SMO GUI.

1.2 Purpose
The first purpose of this thesis is to implement a prototype, based on the Eclipse Rich Client Platform, for a new SMO GUI. This involves investigating the difficultness of integrating the new GUI with the existing SMO architecture.

The second purpose is to evaluate how problems that exist in the current SMO GUI can be solved using the Eclipse Rich Client Platform.

1.3 Method
The following activities will be performed iteratively during the project:

- Investigating demand for new functionality
- Literature study of how to implement new functionality
- Implementation of the new functionality
- Demo of the prototype with the new functionality
- Evaluation and documentation of Eclipse RCP and the prototype

New functionality will mainly be requested by the supervisor of this thesis. Small demos will be held frequently for the supervisor and other people interested in SMO.

1.4 Requirements
There are two major requirements of this thesis work:

1. The prototype shall implement the main views of the existing SMO GUI.
2. The prototype shall also implement at least one action that can be performed on a network element.
1.5 Limitations
This master thesis will focus on evaluating the Eclipse Rich Client Platform. Usability aspects of the prototype is out of scope of this report and will not be further discussed.

The prototype implementation is limited to the GUI implementation. The prototype implementation will not include any modifications in the overall SMO architecture.

1.6 Target audience
This report is intended for two audiences:

1. Ericsson employees working with SMO
2. People generally interested in starting to use the Eclipse Rich Client Platform.

The reader is assumed to be familiar with basic computer and software engineering. This includes basic knowledge in software testing and design and architecture patterns.

1.7 Outline

1. Introduction
   This chapter introduces the master thesis.

2. Problem description
   This chapter describes SMO and introduces the problems with the existing SMO GUI.

3. Eclipse Rich Client Platform
   This chapter describes the Eclipse Rich Client Platform.

4. Spring Dynamic Modules
   This chapter describes dependency injection and how it can be implemented using Spring Dynamic Modules.

5. Prototype implementation
   This chapter describes the prototype implementation.

6. Result
   This chapter presents the results of the master thesis.

7. Discussion and Conclusions
   This chapter presents a discussion of the results, conclusions and possible future work.
2 Problem description
This chapter starts with introducing the Software Management Organizer (SMO) system and especially the SMO Graphical User Interface (GUI). The chapter ends with a description of the problems that exists in the current SMO GUI.

2.1 Software Management Organizer
The Software Management Organizer is used by mobile operators for remote software and hardware handling of GSM, UMTS and Core networks.

2.1.1 Architecture
The Software Management Organizer consists of three main parts:

- **SMO GUI**
  The SMO GUI presents software management activities to the operator of a telecom network. Multiple instances of the SMO GUI can cooperate with one SMO Server. The SMO GUI is also referred to as the client.

- **SMO Server**
  The SMO Server is responsible for scheduling of software management activities.

- **SMO Modules**
  The SMO Modules execute software management activities for specific node types.

SMO uses CORBA to communicate between the GUI, Server and Modules. Figure 1 shows an overview of the SMO architecture.

![Figure 1: SMO architecture](image)

*(SMO, Software Management Organizer, System Description)*
2.1.2 SMO GUI

The current GUI is based on Java Swing and consists of the following views:

- **Job**
  The Job view is used to monitor and control jobs that are managed by SMO.

- **Network**
  The Network view is used to perform operations on network nodes and display node software and hardware properties.

- **Software**
  The Software view is used to manage software packages in SMO.

- **SMO File Store**
  The SMO File Store is used to manage files on the SMO Server.

- **License**
  The License view is used to manage licenses in SMO.

Figure 2 shows the Job view in the current SMO GUI.
2.2 Problems
The following section describes the problems with respect to architecture, extendibility, testability and quality in the current SMO GUI.

2.2.1 Architecture
Most problems in the SMO GUI are related to architectural aspects of the application. This section will describe those problems in detail.

2.2.1.1 Modules
The SMO Modules are reflected in the architecture of the SMO GUI. The GUI is composed of modules that correspond to the SMO Modules. This is problematic because in terms of functionality the modules in the GUI are overlapping and code is thereby duplicated across the composing modules.

2.2.1.2 Communication
The SMO GUI and server communicate via CORBA. A pull mechanism is used by the GUI to retrieve most of the data from the server. The GUI can however, subscribe to notifications about jobs and network topology changes. This communication model is not efficient because every SMO GUI that is connected to a given SMO server has to take into consideration that some data may have been updated by another instance of the SMO GUI. Therefore all clients have to check if something has changed instead of one server notifying the interested clients.

The SMO File Store enables the operator to access files in the native file system on the SMO Server. The content in the file store can change in two ways:
- The content is changed outside SMO with for example a UNIX shell.
- The content is changed via SMO.

The problem with this is that each client has to make sure that the file store content they are displaying is up to date by pulling the server for new data.

The SMO Server uses CORBA call backs to notify clients of updates. This solution is fragile when the system gets stressed since messages may get lost in between the server and client.

2.2.1.3 Layers
The architecture of the SMO GUI lacks layers. There is no separation of communication related functionality and the GUI itself. The data received via CORBA is used directly in the GUI without any transformation. This creates a problem as a small change in the communication interfaces could trigger changes in many GUI classes.

In some cases the GUI contacts network nodes directly. This was not intended in the original design. This is a problem because it adds responsibilities to the GUI that should be centralized to the SMO Modules.

2.2.1.4 Thread
There are three different thread models in the current SMO GUI. These models are not compatible with each other and dead locks may occur if threads from different thread models block each other.
2.2.2 Extendibility
When SMO needs to support a new node type the GUI needs to be extended as well. The current SMO GUI is often extended by copying existing classes and then modifying the necessary parts to fit the new node type. This is quite time consuming since the GUI has to be extended for every new node type that is introduced. This way of extending the GUI is also a problem with respect to code duplication.

The SMO GUI contains logic for specific node types. This makes it hard to extend SMO with more node types, since the GUI has to know about each node type and their relationships.

Future telecom networks are expected to contain huge number of nodes. This creates new challenges when presenting the network topology to the operator. It will probably be necessary to group and filter nodes by personal preference in order for the operator to manage the large networks. The existing SMO GUI has no support for grouping and filtering nodes today and it would be quite time consuming to accomplish.

2.2.3 Testability
The SMO GUI is hard to unit test mainly because the code wasn’t designed to be testable from the beginning. Classes have too many responsibilities and are quite big. It is also common with inner and anonymous classes, which is not possible to unit test.

The GUI doesn’t generally use the model view controller pattern. Views in the GUI contain logics, which make it impossible to unit test the functionality of the GUI without running the actual views.

2.2.4 Application quality
The current SMO GUI has problems with synchronization of view updates. For example if the user selects some entry in one view then other depending views may not be updated. This is because a poorly implemented event system is used instead of the standard event system that exists in Java Swing.

The SMO GUI contains too many quick and dirty fixes. These fixes were introduced in stressed situations where time was very limited. However, the code hasn’t been cleaned up and refactored afterwards. This is a problem with respect to application quality.
3 Eclipse Rich Client Platform

This chapter describes the Eclipse Rich Client Platform (RCP). The chapter starts with presenting the background of Eclipse RCP. Then the chapter continues with describing an overview of the Eclipse RCP architecture. After that the following sections of the chapter describe in detail the components that make up the Eclipse RCP. The chapter ends with describing the thread support, how unit testing can be performed with Eclipse RCP applications and the RCP development environment provided by Eclipse.

3.1 Background

Eclipse is an open source community that runs projects that focus on building an open development platform. (About the Eclipse Foundation)

The Eclipse platform was from the beginning a component based system (Szyperski). A proprietary runtime system was used to run the components, which were called plug-ins. This system enabled third party developers to create components that could extend the Eclipse platform with new features. The basic Eclipse components were created generally and could have been used to build any GUI.

The proprietary runtime system was replaced during the development of Eclipse 3.0 with an OSGi implementation called Equinox, which is a spin off project from the Eclipse project. The Eclipse platform consists of components that are called bundles or plug-ins. All components are since Eclipse 3.0 OSGi bundles and runs on top of the Equinox run-time system.

The Eclipse platform was during development of version 3.0 refactored in order to separate the IDE and the underlying platform. This was done in order to be able to use the Eclipse platform as Rich Client Platform (RCP) for building any rich client application. The Eclipse Rich Client Platform contains a minimal set of OSGi bundles that enables development of rich client applications. (Gruber et al)

3.2 Architectural overview

The Eclipse Rich Client Platform consists of several components; the OSGi run-time system, the Eclipse Core run-time system, the Standard Widget Toolkit (SWT) and JFace widget libraries and the workbench. This is shown in Figure 3.

The Eclipse RCP enables an application developer to structure an application into components; OSGi bundles and/or Eclipse plug-ins. These components together with the Rich Client Platform and possible third party components form an Eclipse RCP application.

<table>
<thead>
<tr>
<th>Eclipse Rich Client Platform Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse Workbench</td>
</tr>
<tr>
<td>Eclipse plug-ins</td>
</tr>
<tr>
<td>JFace</td>
</tr>
<tr>
<td>Eclipse Core Runtime</td>
</tr>
<tr>
<td>OSGi bundles</td>
</tr>
<tr>
<td>SWT</td>
</tr>
<tr>
<td>OSGi run-time system (Equinox)</td>
</tr>
<tr>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>Operating System</td>
</tr>
</tbody>
</table>

Figure 3: Eclipse Rich Client Platform Architecture
3.3 OSGi

The OSGi alliance was founded in March 1999 and the original specifications were targeted at residential Internet gateways with Home Automation applications. However, the standards found their way to other areas such as smart phones, cars, desktop and server applications. (About the OSGi Service Platform)

OSGi was originally a shortening for *Open Service Gateway initiative*. However, as the standards found their ways from the home automation applications to other areas the OSGi shortening lost its meaning and is today used as a name.

The OSGi Alliance describes OSGi as follows in *The OSGi Architecture* article:

> The OSGi technology is a set of specifications that define a dynamic component system for Java.

The OSGi specifications define the OSGi Service Platform, which add the concept of components to the Java platform. OSGi runs on top of a single Java virtual machine and can handle thousands of components and applications at the same time. (About the OSGi Service Platform)

An OSGi component can be designed to run on everything ranging from an embedded device in for example a car to a mainframe server as long as there is an OSGi run-time system available. Components are often referred to as bundles or modules in the OSGi world. (About the OSGi Service Platform)

3.3.1 OSGi service platform architecture

The functionality of the OSGi Service Platform is divided into five layers as shown in Figure 4. The layers in the figure are briefly explained below (About the OSGi Service Platform):

- **Services layer**
  The service layer contains the services exported by the bundles.

- **Service registry layer**
  The service registry layer keeps track of coming and going services.

- **Life cycle layer**
  The life cycle layer provides support for installation, uninstallation, updating, starting and stopping bundles.

- **Modules layer**
  This module layer defines how bundles can import and export Java packages.

- **Security layer**
  This security layer handles security aspects.

The OSGi service platform provides a component and service model, which are described in 3.3.2 and 3.3.3 respectively.
3.3.2 Component model

OSGi bundles consist of an external and an internal part as shown in Figure 5. Bundles can contain the following items:

- **Internal packages**
  These packages are visible inside the bundle but not to other bundles. (Gruber et al)

- **Exported packages**
  These packages are exported to other bundles. (Gruber et al)

- **Metadata**
  Bundles include a MANIFEST.MF file that contains metadata that describe the bundle and its relations with other bundles. (Gruber et al)

- **Optional data**
  Bundles may contain icons, documentation, jar libraries and etcetera.

A bundle must contain metadata. However the exported and internal packages and optional data are not required.

OSGi bundles are defined in the MANIFEST.MF file. The file is located in the META-INF folder in the bundle’s root directory. The name, location and format of the manifest file are equivalent with an ordinary jar library. This is convenient because a bundle may therefore be used as a normal jar lib, without access restrictions to internal parts, as well as an OSGi component. (Gruber et al)
Example 1 shows an example of a MANIFEST.MF file.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: ExampleService Plug-in
Bundle-SymbolicName: ExampleService
Bundle-Version: 1.0.0
Bundle-Activator: exampleservice.Activator
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: org.osgi.framework; version="1.3.0",
                  org.osgi.util.tracker; version="1.3.1"
```

**Example 1: MANIFEST.MF**

When the OSGi runtime system starts it reads the manifest files for each bundle. The classes are not loaded until the bundle is started. If a specific bundle is required to start the run-time system will then first resolve the bundle’s dependencies. (Gruber et al)

A bundle can import classes from other bundles in two ways:

- **Import package** `com.example.package1`
  The runtime system will try to find the requested package in other bundle’s export declarations.

- **Require bundle** `com.example.bundle1`
  The bundle imports all packages in the specified bundle.

The import package declaration is more flexible since it doesn’t require the importing bundle to have any knowledge about which bundle that are to provide the required packages. Packages can be moved between bundles without having to update the consuming bundles. (Gruber et al)

The require bundle declaration are used by Eclipse bundles when using the Extension/Extension points described in 3.4. The require bundle declaration makes sure that the OSGi run-time system loads the required bundle before the bundle that declared the requirement.

Each bundle has its own class loader and can only access classes defined by itself and the packages imported via the import package and require bundle declarations in the manifest (The OSGi Architecture). However, the classes in the standard java packages are available as in standard Java development.

A bundle can specify an activator in the manifest, which then will be used in the life cycle management of the bundle. When the bundle is started the activator’s start method is run and when the bundle stops the activator’s stop method is run. The activator can for example create java objects and publish them as OSGi services as described in the next section. (Gruber et al)

Bundles can have the following different states in an OSGi system:

- **Installed**
  The bundle is successfully installed, but not resolved.

- **Uninstalled**
  The bundle is uninstalled and cannot be used.

- **Resolved**
  A bundle is resolved when all dependencies are met. It can now be used.
• **Starting**  
The bundle is in the process of starting.

• **Stopping**  
The bundle is in the process of stopping.

• **Active**  
The bundle is running.

Bundle code can only execute in the starting, stopping and active state. In the other states no classes are loaded and only the manifest is accessible. (*Service Platform Release 4 API*)

## 3.3.3 Service model

OSGi services are Plain Old Java Objects (POJOs). Services are registered and accessed via the service registry and with respect to a specific contract – a Java interface. (Gruber et al)

Example 2 shows how a service is created and exported.

```java
// Create the service
HelloService service = new HelloServiceImpl();

// Register the service
context.registerService(  
    HelloService.class.getName(),  
    service,  
    new Hashtable<Object, Object>());
```

**Example 2: Creating OSGi service**

Example 3 shows how a service can be accessed and used.

```java
// Create a tracker and track the service
ServiceTracker helloServiceTracker = new ServiceTracker(  
    context,  
    HelloService.class.getName(),  
    null);

helloServiceTracker.open();

// Grab the service
service = (HelloService) helloServiceTracker.getService();

// Use the service
service.speak();
```

**Example 3: Using OSGi service**

Services can register and unregister at any time. The service consumer must take this into account.

## 3.3.4 Equinox

Equinox implements a subset of the OSGi specification and the current version of Equinox is based on the OSGi R4.1 specification. Equinox does not implement the full set of OSGi specifications. Instead Equinox implements the parts needed by Eclipse. This includes the parts of the specification that concerns the module, life cycle and service concepts (Gruber et al).

There are some features in Equinox that didn’t exist in the OSGi specification from the beginning (Gruber et al). These features were implemented to be fully compatible with OSGi specifications and have later been added to the OSGi R4.1 specifications.
Equinox was designed as a run-time system for Eclipse. However, it’s possible to run any kind of application on top of Equinox. Server side applications as well as rich client applications can take advantage of the Equinox component system.

3.4 Eclipse extension framework

The Eclipse extension framework is build on top of OSGi and provides support for run-time extensibility. This is one of the core concepts of Eclipse. (Gruber et al)

In the Eclipse world components are referred to as plug-ins. Plug-ins are OSGi bundles that uses the extension framework and are managed by the Eclipse core runtime. The extension framework enables plug-ins to use the concepts of extension points and extensions. These concepts are explained further on in this section. (Gruber et al)

Plug-ins are defined in the Eclipse online help as follows:

Plug-ins are structured bundles of code and/or data that contribute function to the system. Function can be contributed in the form of code libraries (Java classes with public [application program interfaces] APIs), platform extensions, or even documentation. Plug-ins can define extension points, well-defined places where other plug-ins can add functionality.

The Eclipse architecture is shown in Figure 6.

As all OSGi bundles the Eclipse plug-ins contains a manifest file. In addition the plug-ins also contains another meta-data file called plug-in.xml. This file specifies in XML the extension points and extensions provided by a plug-in. (Gruber et al)

3.4.1 Extension points

Extension points are used to let future plug-ins add functionality to an existing context (Gruber et al).

The full name of the extension point is the id of the plug-in plus the id of the extension point. Example 4 shows an extract from org.eclipse.ui/plug-in.xml that defines an extension-point called views. The full name of the extension point will therefore be org.eclipse.ui.views. (Gruber et al).
Example 4: Creating extension point

An extension point definition may supply a schema. An extension point schema is defined as follows in the Eclipse online help.

An extension point schema is an XML schema that defines a grammar that formally expresses elements, attributes, and types. This information can be used by tools to validate extensions or offer assistance during the creation of extensions.

It is highly recommended to supply a schema to be able to validate that extension points are used in the intended way to avoid run-time problems.

3.4.2 Extensions

Extensions are the implementation of new functionality that will be added to an existing context (Gruber et al). Example 5 shows an example of a plugin.xml file. The corresponding plug-in contributes with a view, a perspective and a help extension.

Example 5: Creating extensions
3.5 Standard Widget Toolkit

The Standard Widget Toolkit (SWT) is a Java API that enables access to native widgets on the platform that it runs on. Applications that use SWT can run on any platform supported by SWT and will look and feel like native applications on the platform. SWT is available on a wide variety of window systems and operative systems including Microsoft Windows, Solaris and Linux.

SWT can be used as ordinary Java libraries and as OSGi bundles.

(McAffer & Lemieux)

3.6 JFace

JFace is a higher level widget toolkit based on SWT. The JFace implementation is platform independent, but requires an implementation of SWT for the platform used. JFace components don’t hide the fact that they use SWT components underneath, but supplies methods for accessing the SWT components.

JFace contain among others dialogs, framework for preferences and wizards, progress reporting facilities and viewers for tables and trees. The JFace components are the basis of the Eclipse UI.

JFace tree and table viewers use content and label providers to map view elements to the input model elements. Content providers are responsible of supplying the model elements to the label provider which is responsible of providing the view with the appropriate label text.

(McAffer & Lemieux)

3.7 Workbench

The Workbench is an important part of an Eclipse RCP application. Every window of an Eclipse RCP application contains a workbench. The workbench can hold the following visible items:

- Views
- Editors
- Menus
- Documentation and help information
- Perspectives (can hold views, editors and menus)

The workbench can be extended with new perspectives, views, editors and actions during runtime by starting a new plug-in that contribute with an extension to the application. Views and editors can easily be rearranged, minimized and closed in the workbench.

(McAffer & Lemieux)

Figure 7 shows a workbench with a menu, a toolbar with a button, a status line with a message, two perspective shortcuts and a folder in which views and editors can be placed.
3.8 Data binding
Data binding is essentially about connecting components without having to implement and manage listeners. Eclipse support data binding through the JFace data binding framework. This section will describe a number of core concepts of the JFace data binding framework. (JFace Data Binding)

3.8.1 Observable
Observables are data holders whose state can be observed. JFace provides support for observable values, collections, sets and maps. (JFace Data Binding)

3.8.2 Realm
A realm is used to synchronize access from different threads to an observable. Each observable belongs to a realm and can only be changed in that realm. The realm itself belongs to a Java thread, for example the GUI thread.

It’s possible to initiate modification of an observable from a different thread than the realm belongs to by using the Realm.asyncExec(Runnable) method. Then the modification will run in the thread that the realm belongs to. (JFace Data Binding)

3.8.3 Binding, Converter and Validator
Bindings are used to synchronize two observables. The synchronization consists of two phases – validation and conversion. Converters are used to convert one value type to another and validators are used to validate the synchronization. If an update isn’t valid it can be denied. (JFace Data Binding)

3.8.4 Content and label providers
JFace has content and label providers that support data binding. An observable list can be set as input to a table view. Then a content provider that support data binding will add and
remove elements from the table when the list changes. A label provider that supports data binding will create a binding between the element in the table rows and the properties in the model. So, if objects in an observable list will change one of its properties, the table would update the corresponding cell.

### 3.9 Threads
Eclipse RCP provides support for running background tasks and provides facilities to monitor and manage background tasks. Background tasks are called jobs in Eclipse RCP. It’s possible to set priorities and scheduling properties of jobs.

Modification of GUI objects can be initiated from other threads than the GUI thread. This is done by running the Display.asyncExec(Runnable) method. The provided runnable will then run in the first free time slot of the GUI thread. As mentioned in 3.8.2 observables can also be modified from other threads by using the Realm.asyncExec(Runnable) method.

(McAffer & Lemieux)

### 3.10 Testing
Eclipse RCP supports both ordinary standalone unit testing of classes and Plug-in Development Environment (PDE) unit testing. PDE unit testing enables access to the Rich Client Platform components during the test suites (Eclipse online help). PDE unit tests are written exactly as normal Junit tests. The only difference in the working with PDE unit is that another launch configuration is used to run the test suites. For both standalone unit testing and PDE unit testing libraries as JMock can be used to create mock objects.

In Eclipse, fragment projects can be used to contain test cases. Fragment projects are projects that belong to a plug-in project, but are not required by the plug-in project. Fragment projects can access all packages in the plug-in project, both internal and exported packages.

(McAffer & Lemieux)

### 3.11 Development Environment
Eclipse provides support simplifying development of OSGi bundles and Eclipse bundles via the Plug-in Development Environment (PDE). The PDE is similar to the Eclipse Java development tools that are used when developing standard Java applications in Eclipse. For example manifest and plug-in.xml files can be edited by using a graphical editor. An example of the Eclipse PDE is shown in Figure 8.

(Eclipse online help)
Figure 8: Plug-in Development Environment perspective
4 Spring Dynamic Modules

The purpose of this chapter is to provide background information about dependency injection and Spring Dynamic Modules in order to support the rest of the report. The chapter starts with an overview of Spring Dynamic Modules. Then the chapter continues with describing a background of dependency injection and how it can be done for an OSGi and Eclipse RCP project using Spring Dynamic Modules.

4.1 Overview

The Spring Framework is a popular open source application framework for the Java platform. Spring consists of a collection of smaller framework divided in common aspects of complex applications such as:

- Inversion of control
- Data access
- Remote access

Spring Dynamic Modules is a porting of the Spring Framework to the OSGi platform with some extensions for the OSGi platform.

When using Spring Dynamic Modules an application context is added to the bundles that are managed by Spring. An application context can be seen as a container that manages Spring beans. Spring beans are Plain Old Java Objects (POJOs) managed by the application context. The application context is responsible for creating the beans and injects their requested dependencies.

Spring Dynamic modules provide an OSGi bundle called org.springframework.osgi.bundle.extender that is responsible for instantiating the Spring application contexts for each bundle that uses Spring. The application context for each bundle is then by default published as an OSGi service.

(Colyer et al)

4.2 Dependency Injection

This section describes some background information about dependency injection, how to use dependency injection in Spring and how Spring can be integrated with OSGi services.

4.2.1 Background

Dependency injection is used to control dependencies between classes. By injecting dependencies to a class, instead of letting the class create the dependencies, the class can be unit tested. When the class is unit tested a mock version of the dependency is injected by the test case.

Dependencies can be injected in three ways (Johnson, Rod et al):

- **Constructor Injection**
  The dependency is injected via the constructor when the object is created.

- **Setter Injection**
  The dependency is injected after the object is created, but before it’s used.

- **Parameter Injection**
  The dependency is injected as a parameter in method calls.
4.2.2 Spring bean configuration

In Spring dependencies are often managed in XML configuration files. However it’s possible to configure dependencies with annotations as well. (Colyer et al)

Example 6 shows how a service dependency can be injected into a bean called View via setter injection. Spring will use the corresponding setter method for the service property in the view class to inject the dependency. Setter methods must be named according to the Java Beans standard.

```xml
<bean id="View"
    class="myPackage.OneView"
    scope="prototype">
    <property name="service" ref="Service"/>
</bean>

<bean id="Service"
    class="myPackage.OneService">
</bean>

Example 6: Spring dependency injection XML configuration file

4.2.3 Integration with OSGi services

Spring beans can easily be integrated with OSGi services. Spring beans can be exported as OSGi services and OSGi services can be imported and used as Spring beans (Colyer et al). This is very powerful since it makes it possible to control dependencies between OSGi bundles in a similar way to how dependency between classes are managed using Spring.

Example 7 shows how a bean Service is exported as an OSGi service with interface IService.

```xml
<osgi:service ref="Service"
    interface="IService"/>

Example 7: Exporting Spring bean as OSGi service

Example 8 shows how an OSGi service with interface IService is imported and then handled as local bean Service.

```xml
<osgi:reference id="Service"
    interface="IService"/>

Example 8: Importing OSGi service to Spring context
5 Prototype implementation

This chapter describes the implementation of the prototype, based on the Eclipse Rich Client Platform, for a new SMO GUI. The chapter starts by describing the architecture of the prototype. The chapter continues with a description of how the Eclipse Extension Framework has been used, how the prototype uses the concept of view-models, which and how third party plug-ins that is used in the prototype and finally about testing the prototype.

5.1 Architecture

The prototype is composed by several bundles. Some bundles are standard third party components while others were developed as a part of the master thesis. The bundles that were developed during the project are shown in Figure 9 and are described in 5.1.1.

5.1.1 Bundles

The bundles that were developed during this thesis are divided into six groups; application, perspective, service, common, common GUI and test. These are described below. Figure 9 shows the dependencies between the bundles. Note that the bundles related to unit testing are not shown. These bundles are fragment bundles to the bundle in which they test and are therefore just extensions of those bundles.

Application
The application group contains one bundle and that is the Eclipse RCP application bundle. An application bundle starts up the Eclipse workbench and let other bundles contribute with extensions.

- **com.ericsson.nms.bss.sm.smo.rcp.client.client**
  The client bundle is responsible of loading the dependencies in the correct sequence and starting up the bundles that contribute to extension framework.

Perspective
The perspective group contain bundles that contribute with a perspective extension to the Eclipse RCP application.

- **com.ericsson.nms.bss.sm.smo.rcp.client.filestore**
  The filestore bundle contributes the SMO File Store perspective to the application.

- **com.ericsson.nms.bss.sm.smo.rcp.client.job**
  The job bundle contributes the job perspective to the application.

- **com.ericsson.nms.bss.sm.smo.rcp.client.license**
  The license bundle contributes the license perspective to the application.

- **com.ericsson.nms.bss.sm.smo.rcp.client.network**
  The network bundle contributes the network perspective to the application.

- **com.ericsson.nms.bss.sm.smo.rcp.client.software**
  The software bundle contributes the software perspective to the application.

Service
The service group contains bundles that define or implement services.

- **com.ericsson.nms.bss.sm.smo.rcp.client.iservice**
  The iservice bundle provides interfaces for the services provided by the service bundle. The bundles that use the services provided by the service bundle don’t know where the services come from. They only refer to an interface defined in the iservice bundle.
• **com.ericsson.nms.bss.sm.smo.rcp.client.service**  
The service bundle wraps CORBA communication by providing service oriented interfaces.

**Common**  
The common group contains bundles that are common to the other bundles and that doesn’t depend on any other bundle developed in this project.

• **com.ericsson.nms.bss.sm.smo.rcp.client.model**  
The model bundle contains domain specific models.

• **com.ericsson.nms.bss.sm.smo.rcp.client.spring**  
The Spring bundle is responsible for Spring related features, like finding beans.

**Common GUI**  
The common GUI group contains bundles that contain GUI related functionality that is used by bundles in the perspective group. These bundles depend on the model and the iservice bundle.

• **com.ericsson.nms.bss.sm.smo.rcp.client.job.tree**  
The job.tree bundle contains a tree viewer for showing jobs.

• **com.ericsson.nms.bss.sm.smo.rcp.client.topology**  
The topology bundle is responsible for showing the topology view and providing a view-model of the topology view.

• **com.ericsson.nms.bss.sm.smo.rcp.client.ui.common**  
This bundle contains common features that are used by the other GUI bundles.

**Test**  
The test group contain a bundle related to unit testing.

• **com.ericsson.nms.bss.sm.smo.rcp.client.software.test**  
The software.test bundle is a fragment bundle and contains test cases for the software bundle.

**5.1.2 Layers**  
As shown in Figure 9 the prototype consists of the three layers; RCP Application layer, Perspective layer and Service layer. These layers are described below:

• **RCP Application layer**  
The RCP application layer contains the RCP application bundle called client. The client bundle starts the Eclipse workbench, the Spring Dynamic Modules framework and the five bundles from the perspective group that contribute with perspective extensions.

• **Perspective layer**  
The Perspective layer contains bundles that contribute with perspective and view extensions and some common bundles. The bundles in the perspective layer use OSGi services provided by the service layer to populate the views and perform actions towards SMO Server and Modules.

• **Service layer**  
The service layer is responsible for handling communication with the SMO Server and Modules. The service layer provides higher level interfaces for accessing features on the SMO Server and Modules. These interfaces are service oriented in contrast to the module oriented interfaces provided by the SMO Server and Modules.
Figure 9: Bundle dependencies
5.2 Use of the extension framework

Each perspective is located in a separate bundle, which provides the corresponding perspective to the prototype application by extending the org.eclipse.ui.perspectives extension point. These bundles are also responsible of creating views, menus and actions on their own by extending predefined extension points. These extensions are defined in each bundle’s plugin.xml file.

The perspective bundles are completely separated and there is no dependency between each perspective bundle. A new perspective can easily be added, preferably by creating a new bundle that contributes with a perspective extension.

The topology bundle provides the topology view by extending the org.eclipse.ui.views extension point. The perspective bundles can use this view by adding the view to their perspective. This is done by referring to the id of the topology view and declaring the topology bundle as a dependency in the manifest file.

5.3 View-Models

Most of the views in the prototype have a corresponding view-model that stores the state of the view. The state of the view-model can be used by other view-models that correspond to views that are dependent on the state of the first view-model. These view-models can be unit tested without running the actual widgets.

Example 9 shows how the topology view-model is specified. There is a method for accessing an observable list of topology selections, a method for toggling the filtering property and a method for accessing the observable filtering property. These methods are used to read and modify the state of the view-model.

```java
public interface ITopologyViewModel {
    public abstract IObservableList getSelected();
    public abstract void toggleLinkEnabled();
    public abstract IObservableValue getLinkEnabled();
}
```

Example 9: ITopologyViewModel – a view-model interface

5.4 Use of third party bundles

This section describes the use of JFace data binding and Spring Dynamic Modules.

5.4.1 JFace data binding

Data binding is used frequently in the prototype. The services provide data in models and lists that support data bindings. If data provided by a service is updated the views that use that data will also be updated automatically. This is used when the service bundle receives events about job updates from the SMO Server. The service bundle updates the job model and the job views will automatically be updated.

Data binding is also used between views and view-models and between a view-model and another view-model. The TopologyViewModel class provides a model of among others selections in the topology view. These selections are used by other view-models such as the SoftwareViewModel. The SoftwareViewModel is notified when the selections has changed and will then update its list of relevant software packages. The SoftwareView uses the list of
relevant software packages in the SoftwareViewModel as its input and will then automatically update if the list is changed.

Example 10 shows how a table view with support for data binding is created.

```java
public class NodeSelectionView extends ViewPart {
    private TableViewer viewer;

    @Override
    public void createPartControl(Composite parent) {
        // Some code initializing table

        initDataBindings();
        IObservableList input = nodeViewModel.getDataList();
        viewer.setInput(input);
    }

    private void initDataBindings() {
        ObservableListContentProvider contentProvider =
            new ObservableListContentProvider();

        LabelProvider labelProvider =
            new ObservableMapLabelProvider(
                BeansObservables.observeMaps(
                    contentProvider.getKnownElements(),
                    SelectionInfo.class,
                    new String[] { "name",
                        "platform",
                        "type",
                        "release",
                        "description",
                        "lastSWAdjust",
                        "lastHWAdjust",
                        "lastLicenseAdjust" }));

        viewer.setContentProvider(contentProvider);
        viewer.setLabelProvider(labelProvider);
    }

    @Override
    public void setFocus() {
        viewer.getControl().setFocus();
    }
}
```

Example 10: Table viewer with data binding

The table viewer takes an observable list as input and has a label and content provider that supports data binding. The content provider will update the table view when an object in the observable list is added or removed and the label provider will map the properties supplied for each instance of SelectionInfo with the corresponding table cells. If for example the lastSWAdjust property for an object in the input list would change, the table view would automatically be updated.

All table and tree viewers in the prototype are created similar to the viewer in Example 10.
5.4.2 Spring Dynamic Modules
The prototype uses Spring Dynamic Modules to manage dependencies between components both on class and bundle level.

5.4.2.1 Integration with extension points
An Eclipse plug-in defines view extensions in its plugin.xml file. Example 11 shows an example of such an extension.

```
<extension
  point="org.eclipse.ui.views">
  <view
    class="com.ericsson.nms.bss.sm.smo.rcp.client.network.TopologyView"
    icon="icons/extern_lib/tree.gif"
    id="com.ericsson.nms.bss.sm.smo.rcp.client.network.TopologyView"
    name="Topology">
  </view>
</extension>
```

Example 11: View extension

However, when using Spring Dynamic Modules the view has to be created by the Spring application context if dependencies are to be injected. This can be done by specifying a factory class instead of the view class in the view definition. This is shown in Example 12.

```
<extension
  point="org.eclipse.ui.views">
  <view
    class="org.eclipse.springframework.util.
             SpringExtensionFactory:TopologyView"
    icon="icons/extern_lib/tree.gif"
    id="com.ericsson.nms.bss.sm.smo.rcp.client.network.TopologyView"
    name="Topology">
  </view>
</extension>
```

Example 12: View extension with Spring extension factory

The factory takes the bean name as an argument and will return the Spring bean if it exists in the bundle’s application context.

5.4.2.2 Spring bundle
Spring beans can be accessed by any class from the corresponding bundle by using the Spring bundle. Example 13 illustrates how this is done.

```
BeanFinder finder = new BeanFinder();
IJobService jobService = (IJobService) finder.findBean(IJobService.ID, Activator.PLUGIN_ID);
```

Example 13: Bean finder

The findBean method in the BeanFinder class takes two arguments, the bean id and the id of the bundle that contains the bean.

5.4.2.3 Integration with OSGi services
The service bundle provides five services as shown in Figure 9. These services are exported as OSGi services by using Spring Dynamic Modules. The service consumers import OSGi services by using Spring Dynamic Modules and inject the services into the classes that use the corresponding service. This is very flexible since the service bundle can be replaced without
having to modify any of the bundles that consume the services. The replacing service bundle could either publish the services directly as OSGi services or by using for example Spring.

Example 14 shows how the service plug-in publishes the JobService as an OSGi service.

```
<osgi:service ref="JobService"
  interface="com.ericsson.nms.bss.sm.smo.rcp.client.service.IJobService"
  context-class-loader="service-provider"/>

<bean id="JobService"
  class="com.ericsson.nms.bss.sm.smo.rcp.client.service.imp.JobService"
  destroy-method="destroy"/>
```

**Example 14: Spring bean published as an OSGi service**

Example 15 shows how the JobService is imported and then injected into the JobViewModel class.

```
<osgi:reference id="JobService"
  interface="com.ericsson.nms.bss.sm.smo.rcp.client.service.IJobService"/>

<bean id="JobViewModel"
  class="com.ericsson.nms.bss.sm.smo.rcp.client.job.perspective.internal.JobViewModel">
  <property name="jobService" ref="JobService"/>
</bean>
```

**Example 15: Injecting OSGi service to Spring bean**

### 5.5 Testing

The prototype has not been fully tested, but rather evaluated as to how it can be tested. The software bundle was selected for evaluation and a fragment project to the software bundle was created to store test cases.

As described in 5.3 the prototype uses View-Models to extract state and logic from GUI classes. The Software Package Repository View uses a View-Model called SoftwareViewModel. That class uses the network service and the TopologyViewModel class. These dependencies are injected by Spring when the prototype starts. The SoftwareViewModel class was unit tested by injecting mock versions of its dependencies.

The SoftwareViewModel uses data binding to synchronize changes in the TopologyViewModel with the list of software packages. This requires access to the Eclipse RCP while running the test cases. The test cases were therefore run by PDE-junit.

The non GUI classes in the software bundle were successfully unit tested using junit or PDE-junit and JMock. GUI classes where just informally and manually tested, since it was out of the scope of this master thesis to create automated GUI tests.
6 Result
This chapter presents the result of integrating the prototype with the existing SMO system and the result of evaluating the Eclipse Rich Client Platform.

6.1 Overview
The thesis resulted in a prototype application with a subset of features from the existing SMO GUI. The main views in the current SMO GUI are represented by five perspectives:

- Jobs
- Network
- Software
- SMO File Store
- License

These perspectives show most of the data from the corresponding views in the existing SMO GUI. The Network, Software and License perspectives can also show jobs that are relevant to the workflow performed in the corresponding perspective. Figure 10 shows a screenshot of the prototype when the Network perspective is active.

The prototype is also capable of performing certain operations. It can import software packages to the SMO Server, update the information regarding installed software, hardware and licenses on nodes and copy files between SMO File Store and file stores on nodes.

The requirements stated in the Introduction chapter are thereby fulfilled.

![Figure 10: SMO RCP prototype – Network perspective](image-url)
6.2 Integration with the existing SMO architecture

The prototype was successfully integrated with the existing SMO architecture. It was possible to implement all of the selected features from the current SMO GUI. The prototype communicates with the SMO Server and the SMO Modules via the same CORBA interfaces as the current SMO GUI.

It was possible to add some new functionality in the prototype that doesn’t exist in the current SMO GUI. For example, the prototype is able to filter the software packages and license keys based on selections in the topology. When the toggle button is enabled, the software package repository view is filtered based on the selection in the topology. This is shown in Figure 11.

![Figure 11: SMO RCP prototype – Software perspective](image)

The prototype has also some new functionality for managing files between the SMO File Store and file stores on nodes. In the current SMO GUI, files are transferred by executing download and upload commands. The prototype provides a more intuitive way of transferring files. In the prototype, files can be dragged and drop between SMO File Store and a node file store. This is shown in Figure 12.

![Figure 12: SMO RCP prototype – Software perspective](image)
The prototype is able of showing jobs in both tree views as in Figure 11 and in table views as in Figure 13. A user can easily change the presentation of jobs by adding and removing views from the workbench. The job views can also be moved and resized inside the workbench by a user to fit his/her needs.

The job tree views show only jobs that are relevant to the selected perspective. The job table views shows currently all jobs. However, it’s possible to add filtering to the job table views as well.
Any view in the prototype can easily be added to any perspective by a user. For example the software package repository view from the software perspective can be added to the network perspective. This is an advantage from an end user perspective because it enables customization according to individual preferences. It’s also an advantage from a development perspective since it enables easy and fast prototyping.

Figure 14 shows the network perspective with the software package repository view and the license repository view added. By adding these views to the network perspective it’s possible to perform all actions, except for file store actions, in one perspective.
It was not possible to implement some new features such as filtering of jobs based on topology selections because the communication interfaces lacked support for efficient queries about the relationship between jobs and related network nodes. However, it would only require small changes in the communication interfaces to add support for that functionality.

6.3 Evaluation of the Eclipse Rich Client Platform

This section presents the result concerning how problems that exist in the current SMO GUI can be solved by using Eclipse RCP.

6.3.1 Architecture

Eclipse RCP provides a good platform for creating a flexible architecture. This is described in detail in the following sections.

6.3.1.1 Modules

The SMO GUI can be designed in terms of functionality instead of in terms of modules as the current GUI is. This can easily be done using plain Java classes. The Eclipse RCP extends this by adding support for grouping functionality into reusable independent components.

6.3.1.2 Communication

The Eclipse RCP provides the possibility to wrap the existing poor communication interfaces with the SMO Server and Modules into a component that exposes flexible service oriented interfaces and hides the actual implementation to other components. By abstracting the communication interfaces the rest of the GUI will be unaffected by future changes in the communication interfaces.
6.3.1.3 Layers
The Eclipse RCP provides good support for creating a layered architecture. Bundles can be used to divide the code base into components with internal implementation invisible to other components.

The Eclipse RCP bundles can use OSGi services as complement to the extension/extension point mechanism. Spring Dynamic Modules and OSGi services integrate seamlessly. A bundle can export Spring beans with Spring Dynamic Modules as OSGi services for other bundles to consume. This decouples the service provider from the service consumer since there is no need for the service provider and consumer to know about each other. The service provider and consumer will only have to refer to a common service interface that can be located in a separate bundle.

6.3.1.4 Thread
The Eclipse RCP has good thread support. It’s possible to monitor the progress of threads using functionality in the platform. The platform has also well defined support for initiating modifications of GUI related objects from other threads than the GUI thread.

6.3.2 Extendibility
An application based on the Eclipse RCP can be designed to be easily extended by new functionality. The extension/extension point mechanism is flexible for creating an application that can be extended by future bundles. It is very flexible to let future bundles add new functionality to an existing context without having to alter the existing context.

It is easy to add views that show models that support data binding because of the high abstraction of the GUI components and the integrated event handling.

Menus, toolbars and popup menus can be defined in XML and added during runtime. Bundles can contribute to the application by adding new entries and menus. This is very flexible and makes it easy to extend an application based on the Eclipse RCP.

6.3.3 Testability
The prototype is easier to unit test than the current SMO GUI since it was designed to be testable from the beginning. The prototype is composed of bundles that can be tested separately.

By moving state from the views to view-models the application can be unit tested with high coverage without the need of running the views. The views can then be tested by using automated GUI test tools.

Spring dependency injection is useful for controlling dependencies between classes and bundles. By using Spring it’s possible to unit test classes that use external services by supplying a mock class.

6.3.4 Application quality
By using JFace widgets and data binding all event handling are already taken care of. It is therefore easier for a developer to create a high quality product because of the higher abstraction level.

6.3.5 General drawbacks of the Eclipse Rich Client Platform
The main problems encountered were concerned with errors in various configuration files. The Eclipse Plug-in Development Environment has good support for detecting some errors in
the plug-in.xml files. However, these errors are by default just marked as warnings and some kinds of errors are not found at all. For example mismatching IDs of views where not detected. Errors in the Spring configuration files were not detected by Eclipse and resulted in run-time errors. However, solutions to this are discussed in the following chapter.

Some initial problems were also concerned with finding enough information about how to use the latest version of Eclipse RCP. For example it was initially hard to find enough information about latest features of the JFace data binding framework. However, it was easier to find information in the end of the master thesis because of the gained knowledge about the Eclipse RCP during the project. It was thereby easier to search for information by entering more precise keywords and look for information in the right places.

6.4 Summary

- It was possible to implement a prototype based on the Eclipse RCP that integrates with the existing SMO architecture.

- The Eclipse RCP provides good support for creating a flexible, extensible and testable application.

- The drawback of using the Eclipse RCP is the lack of support for detecting configuration errors and the lack of documentation of new features.
7 Discussion and Conclusions

This chapter presents a discussion about the results of integrating the prototype with the existing SMO architecture and the evaluation of how Eclipse RCP can solve problems that exists in the current SMO GUI. The chapter also presents the conclusion of the thesis and suggested future work.

7.1 Discussion

7.1.1 Integration with the SMO architecture

This thesis resulted in a prototype application that was successfully integrated with the existing SMO architecture. Even though the prototype was successfully integrated, it was harder and more time consuming than in the ideal case. The CORBA communication interfaces were complicated to use, slow and too low level.

Much of the functionality that was implemented in the prototype’s service bundle could be moved from the client to the server side. The SMO Server could publish the same services that the service bundle does and the service bundle would then only be responsible of connecting the services to local proxy objects. This could be done using for example Spring remoting in the Spring Dynamic Modules framework.

By moving intelligence from the client to the server the total complexity and size of SMO would decrease. The speed probably could also be improved by using another transport layer than CORBA. If CORBA are to be used in the future it would be beneficial to use the dedicated CORBA event channels instead of call back objects that have no delivery guarantees.

7.1.2 Eclipse Rich Client Platform

The Eclipse Rich Client Platform has been very interesting to work with. The OSGi bundles are a component abstraction above Java classes. The possibility to control which packages that are visible to other bundles is useful since it forces the developers to use the intended public interfaces for accessing the component.

Eclipse RCP lacks support for managing dependencies between classes. It is often necessary to use dependency injection to make classes that use external services testable. The Spring Dynamic Modules framework adds support for dependency injection to the Eclipse RCP. The framework integrates well with the Eclipse RCP and provides the necessary tools for managing dependencies between classes and bundles.

The JFace data binding framework was also helpful and practical to work with after getting into the concepts. New features where added faster to the prototype than in traditional GUI development. This was because of the high abstraction level and integrated support for event handling in the framework.

Non GUI classes where successfully tested without much trouble. The use of dependency injection both between classes and bundles made the classes testable. GUI classes where just informally and manually tested, since it was out of the scope of this master thesis to create automated GUI tests. However, other Ericsson projects have reported successfully use of automated GUI testing tools with Eclipse RCP applications.
The OSGi component model, the extension framework provided by Eclipse RCP, the dependency injection with Spring Dynamic Modules and the JFace data binding enables a designer to create a high quality application with a well designed architecture that can easily be extended and tested.

However, there are also some general drawbacks of the Eclipse RCP. Perspective, View, Menus and etcetera are specified in XML configuration files. This is quite flexible, but the downside of this is that some errors are not detected until run-time. The IDE is not able to find certain kinds of faults such as mismatching IDs. However these kinds of faults could be found using static analysis and most errors would be quite easy to detect. This will probably be supported by Eclipse in the future.

It was initially hard to find information about the latest features in Eclipse RCP. However, when the area was more familiar it was easier to find documentation because of the gained knowledge about the technology itself and about the Eclipse community. To take advantage of all the benefits of using the Eclipse RCP it requires a good overall picture of the OSGi component system and the Eclipse RCP. It would probably be very beneficial to create some kind of internal forum for developers, where experiences can be shared and evaluated.

7.2 Conclusions

The Eclipse Rich Client Platform can be used as a base for building a new SMO GUI. An Eclipse RCP application can be integrated in the existing SMO architecture.

The Eclipse RCP provides support for creating an application without the architectural, extendable, testable and quality problems that exists in the current SMO GUI.

7.3 Future work

7.3.1 SMO GUI

The architecture of the prototype is flexibly designed and can easily be extended. If the current SMO GUI is to be replaced by an Eclipse RCP based application the architecture of the prototype could be reused. The composing prototype bundles could be replaced one by one with bundles intended for customer use.

7.3.2 Eclipse Rich Client Platform

The Eclipse RCP is an exciting evolving platform. There are many interesting frameworks that could be used together with the platform. The Eclipse Modelling Framework, EMF, looks very promising and is used to create domain models. Future work could include investigating the possibilities and benefits of using EMF.

Static analysis could be used to find run-time errors in an Eclipse RCP based application. It would be very interesting to investigate the available tools for doing static analysis on Eclipse RCP applications.

Future work could also include investigating the possibilities and benefits of creating general Eclipse RCP components as for example network topology components that could be used by other Ericsson products.

7.3.3 OSGi

Another interesting area to explore is the benefits of using OSGi bundles and Spring Dynamic Modules on the SMO Server and Modules.
## Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Requesting Broker Architecture</td>
</tr>
<tr>
<td>JFace</td>
<td>An API that is built upon SWT. Provides powerful widgets.</td>
</tr>
<tr>
<td>RCP</td>
<td>Rich Client Platform</td>
</tr>
<tr>
<td>SMO</td>
<td>Software Management Organizer</td>
</tr>
<tr>
<td>Spring</td>
<td>An open source application framework</td>
</tr>
<tr>
<td>SWT</td>
<td>Standard Widget Toolkit. API used to access native widgets.</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
9 References

About the Eclipse Foundation [www]
<http://www.eclipse.org/org/>

About the OSGi Service Platform (2007) [www]

Colyer, Adrian et al. Spring Dynamic Modules Reference Guide [www]
<http://static.springframework.org/osgi/docs/1.1.1/reference/html/>

Eclipse Online Help [www]
<http://help.eclipse.org>

Gruber, Oliver et al (2005). The Eclipse 3.0 platform: Adopting OSGi technology [www]

JFace Data Binding [www]
<http://wiki.eclipse.org/index.php/JFace_Data_Binding>

Johnson, Rod et al. The Spring Framework - Reference Documentation [www]
<http://static.springframework.org/spring/docs/2.5.x/reference/index.html>


Service Platform Release 4 API [www]
<http://www.osgi.org/javadoc/r4v41/org/osgi/framework/Bundle.html>

SMO, Software Management Organizer, System Description (2007) [www]
<Ericsson Internal Network>


The OSGi Architecture [www]
<http://www.osgi.org/About/WhatIsOSGi>
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Philip Frising