Development of LifeSocial Plug-ins

Report



TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Electrical Engineering
and Information Technology
Department of Computer Science
(Adjunct Professor)

Multimedia Communications Lab - KOM
Prof. Dr.-Ing. Ralf Steinmetz

Submitted by
Luciana Alvite
on 08.07.2009

Advisor
Prof. Dr.-Ing. Ralf Steinmetz

Tutor
Kalman Graffi

Contents

1 Updating the non-OSGi™ library (.jar) 1

2 Setting up the OSGi Framework 5
2.1 Prerequisites . . . 5
2.2 Obtaining the core LifeSocial plug-ins . . . 5
2.3 Run configuration for the Equinox Console Extension . . . 6
2.4 Troubleshooting . . . 8

3 Development of new Plugins in the LifeSocial Platform 9
3.1 The core plug-in implementation . . . 9
3.1.1 Creating the basic structure of the plug-in . . . 9
3.1.2 Dependency Declaration . . . 10
3.1.3 The Plugin Interface . . . 10
3.1.4 Plugin Data types . . . 10
3.1.5 Plugin Commands . . . 15
3.1.6 Plugin Properties . . . 19
3.2 Extending the Plugin on DS . . . 19
3.3 Interacting with the platform - The Commands Bundle . . . 34
3.4 Linking the GUI . . . 35
3.5 Run configurations for the GUI . . . 40

A MyPlugin Example 45
A.1 Complete Source Code for the MyPlugin Example . . . 56
A.2 Visual Guide . . . 66
A.3 Run Configurations . . . 77

List of Figures

1.1 Jar Creation Step 1 . . . 1
1.2 Jar Creation Step 2 . . . 2
1.3 Jar Creation Step 3 . . . 2
1.4 Jar Creation Step 4 . . . 3
2.1 Run configuration for the Equinox Console Extension in Eclipse . . . 7
3.1 Example of the command architecture . . . 15
3.2 OSGI-INF Folder . . . 21
3.3 Component Definition - The main plug-in component . . . 22
3.4 Component Definition - Commands . . . 22
3.5 Component Definition - My Command . . . 23
3.6 OSGI-INF folder - all components . . . 23
3.7 Component Definition - Exposing the plug-in interface . . . 24
3.8 Component Definition - Exposing the command aggregator . . . 25
3.9 Component Definition - Exposing a command . . . 25
3.10 Manifest File - Exporting a Package . . . 26
3.11 Component Definition - Declaring a Dependency . . . 28
3.12 Component Definition - Dependency Options . . . 28
3.13 Component Definition - Command Aggregator Dependencies . . . 29
3.14 Component Definition - Command Handlers Dependencies . . . 30
3.15 Manifest File - Imported Packages . . . 32
3.16 Creating a Service Component (GUI) . . . 36
3.17 Creating a Service Component - Component Definition (GUI) . . . 36
3.18 Service component - Adding a dependency (GUI) . . . 37
3.19 Service component - Editing a dependency(GUI) . . . 38
3.20 Eclipse Application Run Configuration for LifeSocial (GUI) . . . 41
3.21 Eclipse Application Run Configuration for LifeSocial (GUI) . . . 42
A.1 OSGI-INF Folder . . . 66
A.2 Component Definition - The main plug-in component . . . 67
A.3 Component Definition - Commands . . . 67
A.4 Component Definition - My Command . . . 68
A.5 OSGI-INF folder - all components . . . 68
A.6 Component Definition - Exposing the plug-in interface . . . 69
A.7 Component Definition - Exposing the command aggregator . . . 69
A.8 Component Definition - Exposing a command . . . 70
A.9 Manifest File - Exporting a Package . . . 70
A.10 Component Definition - Declaring a Dependency . . . 71
A.11 Component Definition - Dependency Options . . . 71
A.12 Component Definition - Command Aggregator Dependencies . . . 72
A.13 Component Definition - Command Handlers Dependencies . . . 72
A.14 Manifest File - Imported Packages . . . 73
A.15 Creating a Service Component (GUI) . . . 73

A.16 Creating a Service Component - Component Definition (GUI) 74
A.17 Service component - Adding a dependency (GUI) 75
A.18 Service component - Editing a dependency(GUI) 76
A.19 Run configuration for the Equinox Console Extension in Eclipse 77
A.20 Eclipse Application Run Configuration for LifeSocial (GUI) 78
A.21 Eclipse Application Run Configuration for LifeSocial (GUI) 79

List of Tables

2.1 Equinox Bundles 5
3.1 Plug-in packages 10

Listings

3.1 IMyPlugin . . . 11
3.2 MyPluginComponent . . . 12
3.3 MyPluginItem . . . 13
3.4 MyPluginItemPayload . . . 14
3.5 MyPluginMyCommand . . . 16
3.6 MyPluginCommands . . . 17
3.7 AbstractMyPluginCommandHandler . . . 18
3.8 Command Naming Convention . . . 19
3.9 PluginPropertiesContainer . . . 20
3.10 Example of Service Binding . . . 31
3.11 Binding Methods - MyPluginComponent . . . 32
3.12 Binding Methods - AbstractMyPluginCommandHandler . . . 33
3.13 Manifest File - Service Component Declaration . . . 34
3.14 Service Binding - Commands Component(GUI) . . . 38
3.15 Activator - getCommands() (GUI) . . . 39
3.16 Manifest File - Service Component Declaration (GUI) . . . 39
3.17 Using the Commands component (GUI) . . . 39
A.1 IMyPlugin . . . 45
A.2 MyPluginComponent . . . 46
A.3 MyPluginItem . . . 47
A.4 MyPluginItemPayload . . . 48
A.5 MyPluginMyCommand . . . 49
A.6 MyPluginCommands . . . 50
A.7 AbstractMyPluginCommandHandler . . . 51
A.8 PluginPropertiesContainer . . . 52
A.9 Example of Service Binding . . . 53
A.10 Binding Methods - MyPluginComponent . . . 53
A.11 Binding Methods - AbstractMyPluginCommandHandler . . . 54
A.12 Manifest File - Service Component Declaration . . . 54
A.13 Service Binding - Commands Component(GUI) . . . 54
A.14 Activator - getCommands() (GUI) . . . 55
A.15 Manifest File - Service Component Declaration (GUI) . . . 55
A.16 Using the Commands component (GUI) . . . 55
A.17 IMyPlugin . . . 56
A.18 MyPluginComponent . . . 57
A.19 AbstractMyPluginCommandHandler . . . 58
A.20 MyPluginCommands . . . 59
A.21 MyPluginMyCommand . . . 60
A.22 MyPluginItem . . . 61
A.23 MyPluginItemPayload . . . 62
A.24 PluginPropertiesContainer . . . 63

A.25 MANIFEST File 64
A.26 myplugincomponent.xml 64
A.27 myplugincommands.xml 65
A.28 mypluginmycommandcommand.xml 65

1 Updating the non-OSGi™[1] library (.jar)

1. Right click on the project name (LifeSocial core) -> Export.. -> Jar File (see figure 1.1).
2. Check the options: (see figure 1.2):
   - Export all output for checked projects
   - Export java source files and resources
3. Select only the resources shown in figure 1.2. Choose a directory to save the file and click next.
4. Accept default settings from the next screen (JAR Packaging Options, see figure 1.3)
5. On the JAR Manifest Specification screen, choose "Generate MANIFEST file" and select the main class "main.MainSystem" (see Figure 1.4).
6. Click Finish

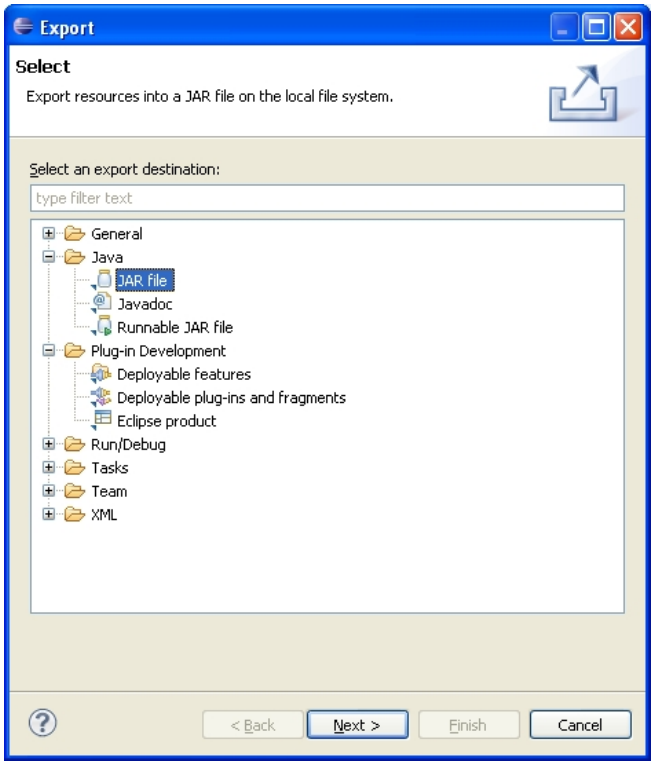


Figure 1.1: Jar Creation Step 1

[1] OSGi is a trademark or a registered trademark of the OSGi Alliance in the United States, other countries, or both

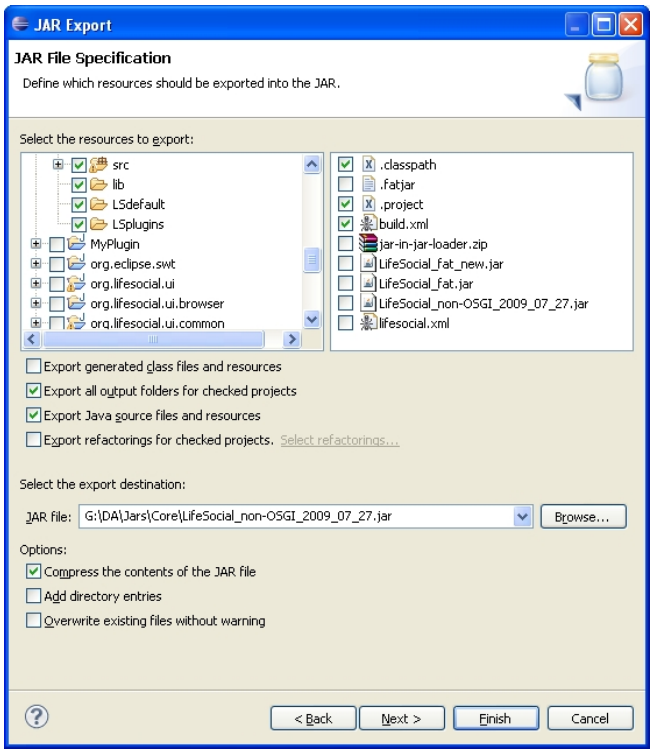


Figure 1.2: Jar Creation Step 2

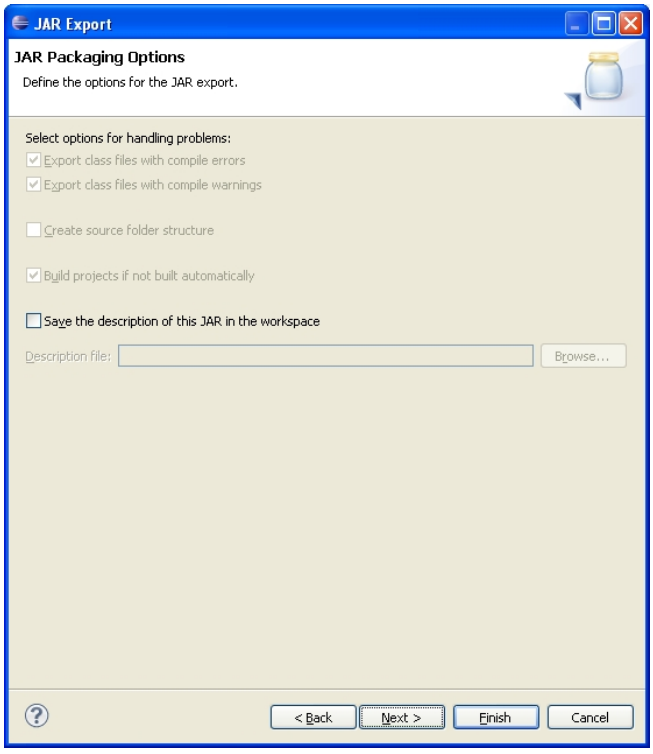


Figure 1.3: Jar Creation Step 3

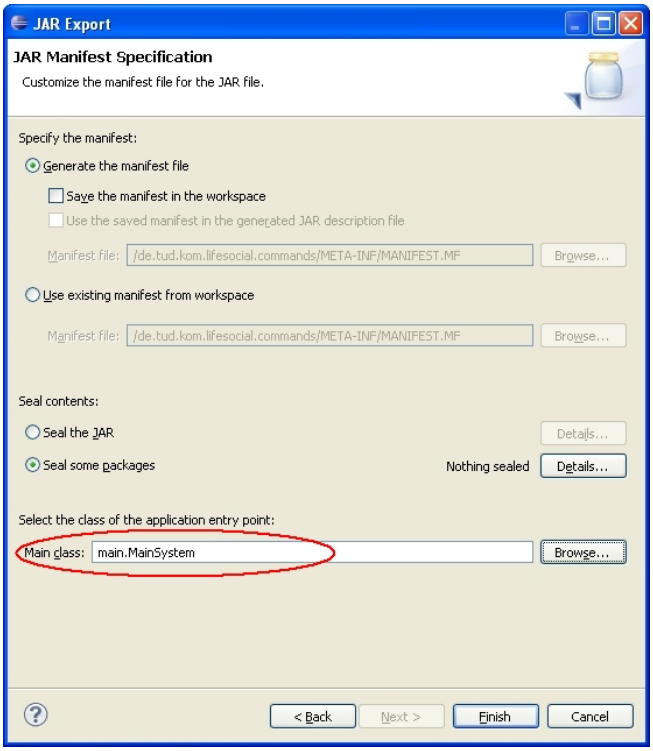


Figure 1.4: Jar Creation Step 4

2 Setting up the OSGi Framework

In this chapter we describe the steps necessary to set up the OSGi framework using Eclipse for the development of LifeSocial plug-ins and the use of the OSGi based LifeSocial core interface.

2.1 Prerequisites

Firstly it is necessary to download Eclipse and Equinox and set them up for the development of Bundles. The reader should download the "Equinox + RCP" package from the eclipse web site [Fou09a]. Next, the equinox platform together with the declarative services and log bundles must also be installed. These can be obtained under [Fou09b], by choosing the version to match your Eclipse distribution (latest is recommended). The following files should be downloaded:

Equinox Bundles		
Name	Filename	Current Version[1]
Equinox SDK	`equinox-SDK-<version>.zip`	3.5
OSGi Services API	`org.eclipse.osgi.services_<version>.jar`	3.2.0.v20090520-1800
Declarative Services	`org.eclipse.equinox.ds_<version>.jar`	1.1.0.v20090601
Log Service	`org.eclipse.equinox.log_<version>.jar`	1.2.0.v20090520-1800
Utils bundle	`org.eclipse.equinox.util_<version>.jar`	1.0.100.v20090520-1800

Table 2.1: Equinox Bundles

Make sure Eclipse is not running and unpack the contents of the Equinox SDK to the eclipse folder and copy the other bundles to the "plugins" directory. Restart eclipse and now the previously added plug-ins should be available.

2.2 Obtaining the core LifeSocial plug-ins

The next step is to obtain the basic core plug-ins necessary for the platform to run. Currently the LifeSocial source code is maintained at KOM[2][Lab09b] Developer's Repositories ([Lab09a]), which can be accessed in Eclipse via svn[3] using an Eclipse plug-in such as Subclipse ([Tig09]). Next, add the LifeSocial repository (`https://dev.kom.e-technik.tu-darmstadt.de/svn/lifesocial`) and checkout the OSGi implementation of LifeSocial found under OSGI\Core. The following projects are the basic for the platform to work:

- `de.tud.kom.lifesocial.commands`
- `de.tud.kom.lifesocial.core`

[1] as of 26.07.2009
[2] Multimedia Communications Lab
[3] Subversion, see `http://en.wikipedia.org/wiki/Subversion_(software)`

- de.tud.kom.lifesocial.emailsender
- de.tud.kom.lifesocial.filelogger
- de.tud.kom.lifesocial.filesystem
- de.tud.kom.lifesocial.localstorage
- de.tud.kom.lifesocial.logger
- de.tud.kom.lifesocial.loginplugin
- de.tud.kom.lifesocial.messagedispatcher
- de.tud.kom.lifesocial.network
- de.tud.kom.lifesocial.overlaynetwork
- de.tud.kom.lifesocial.storage
- de.tud.kom.lifesocial.utilities

The Login Plug-in is needed because to do virtually anything in the platform the user must be logged in. Other mandatory plug-ins of the platform which are used by many other plug-ins are:

- de.tud.kom.lifesocial.friendsplugin
- de.tud.kom.lifesocial.groupsplugin
- de.tud.kom.lifesocial.messsagingplugin
- de.tud.kom.lifesocial.profileplugin

Furthermore, when developing non-GUI related functionalities, it is advised to use the module **de.tud.kom.lifesocial.equinoxconsole**, which is an extension of the standard Equinox command line console developed during the first OSGi implementation, when there was still no GUI, and which also accepts LifeSocial commands besides the regular Equinox commands. This type of interaction is still possible and useful to isolate eventual errors and debugging purposes when implementing non graphical components, since it is then guaranteed it is not a GUI issue and more information about the state of the Bundles can be obtained through the OSGi console.
To activate the extension of the Equinox Console, the component has to be started and its dependencies satisfied. For a list of the currently available commands, refer to [Alv09] or simply type "help" once the equinoxconsole bundle is running.

2.3 Run configuration for the Equinox Console Extension

Figure 2.1 shows a typical Run Configuration for the LifeSocial platform in Eclipse using Equinox. Under "Workspace", select the desired bundles/plug-ins and click on "Add Required Bundles", which will automatically include the required bundles to the launch configuration.

Under "Target Platform" it is important to make sure the OSGi (`org.eclipse.osgi`), Declarative Services (`org.eclipse.equinox.ds`), and Services (`org.eclipse.osgi.services`) bundles are selected. It is also a good idea to check the "Validate Bundles automatically prior to launching" Checkbox and clicking the "Add required Bundles" if any problems are found while validating the configuration. This way all dependencies of the bundles are automatically added to the configuration (as is the case with .util (`org.eclipse.osgi.util`) in this case). Also make sure to add the arguments `-console` and `-consoleLOG` in the "arguments" tab of your launch configuration. This will display the Equinox console and eventual error messages.

When using the log service (any bundle using `org.osgi.service.log`, as the current implementation of the LifeSocial logger), it is also necessary to select this bundle under "Target Platform", as shown in Figure 2.1.

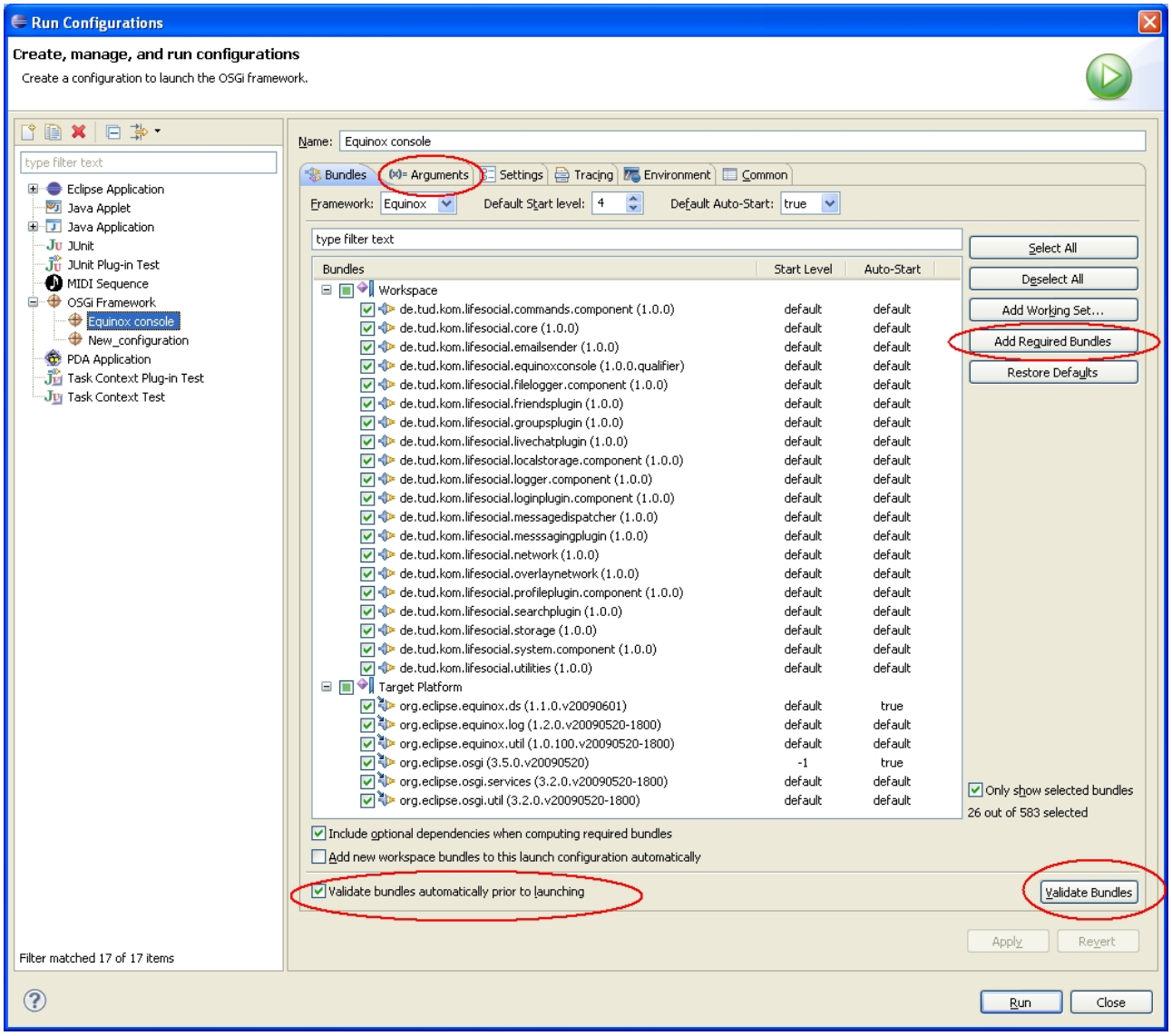


Figure 2.1: Run configuration for the Equinox Console Extension in Eclipse

2.4 Troubleshooting

- Commands are not available in the Equinox Console

  The Bundles:

  - `de.tud.kom.lifesocial.commands` and
  - `de.tud.kom.lifesocial.equinoxconsole`

  must be selected in the Run Configuration (refer to Figure 2.1 for the appropriate Run Configuration.

3 Development of new Plugins in the LifeSocial Platform

In LifeSocial, the GUI[1] is developed as an extension of the Eclipse IDE and uses SWT[2] and JFace[3] for the graphical part and OSGi to control the application's life cycle. The plug-in core implementation is developed using OSGi and Declarative Services (see [Bar07] for a good introduction) for the dynamic management of plug-ins. The integration between the plug-in core implementation and its GUI is performed through the "Commands" Bundle (`de.tud.kom.lifesocial.commands`). In this chapter we explain how to develop new plug-ins for the LifeSocial Platform and how to link the GUI and the core plug-in implementation.

3.1 The core plug-in implementation

In this section we describe the steps involved in creating a basic plug-implementation in LifeSocial. This is the actual implementation of the plug-in functionalities and not the GUI, which will be later described in Section 3.4. Refer to Appendix A for an example of an hypothetical plug-in or checkout the Project "MyPlugin" located under OSGI\Examples\MyPlugin.

3.1.1 Creating the basic structure of the plug-in

The first step is to create a new Plug-in project in Eclipse and (without an Activator, see detailed steps in [Bar07]) and create the basic packages needed for the implementation. These packages are described in Table 3.1.

Basic plug-in packages		
Package	Description	Example
api	contains the interface the Plugin will expose to others	api.ILoginPlugin
component	contains the actual Plugin implementation of the interface defined in the api package. This package should not be visible to other bundles, only the interface (api)	component.LoginComponent (implementation of api.ILoginPlugin)
commands	contains the commands the plug-in will offer to the platform	commands.LoginLoginStatus
data [4]	contains the plug-in specific data types	data.LoginItem
utils	contains plug-in utilities, including the PluginPropertiesContainer (see Section 3.1.6)	utils.PluginPropertiesContainer
docs	contains the plug-in documentation	docs.javadocs

Continued on next page

[1] Graphical User Interface
[2] SWT: The Standard Widget Toolkit, see `http://www.eclipse.org/swt/`
[3] "JFace is a UI toolkit with classes for handling many common UI programming tasks. JFace is window-system-independent in both its API and implementation, and is designed to work with SWT without hiding it", see `http://wiki.eclipse.org/index.php/JFace`
[4] under implementation. all data types are currently available in the core de.tud.kom.lifesocial.core.data package

Table 3.1 – continued from previous page

Basic plug-in packages		
Package	Description	Example

Table 3.1: Plug-in packages

3.1.2 Dependency Declaration

Now the basic dependencies of the plug-in need to be declared in the MANIFEST file. Open the manifest file editor and choose the "Dependencies" tab. Under "Imported packages", add the following packages:

- `de.tud.kom.lifesocial.core.commoninterfaces`
- `de.tud.kom.lifesocial.core.commands`
- `de.tud.kom.lifesocial.utilities`
- `de.tud.kom.lifesocial.core.data.interfaces`
- `de.tud.kom.lifesocial.core.data.impl`

Any other dependencies the plug-in has will need to be added here too. These dependencies vary of course for each plug-in and the list above includes only the basic packages needed for a simple plug-in.

Accordingly, the interface the plug-in will offer to others must be declared. Select the "Runtime" tab and under "Exported Packages", click on "Add.." and choose the api package, e.g.:

- `de.tud.kom.lifesocial.myplugin.api`

3.1.3 The Plugin Interface

Any new plug-in developed for the LifeSocial Framework must extend the `Plugin` interface from the core LifeSocial Bundle (`de.tud.kom.lifesocial.core.commoninterfaces`). The plug-in Interface (e.g. api.IMyPlugin, see Listing 3.1) will extend the `Plugin` interface on the plug-in specific commands and the actual implementation (e.g. component.MyPlugin, see Listing 3.2) will then implement this newly created interface.

3.1.4 Plugin Data types

The data types the plug-in implements (e.g. data.MyPluginItem, see Listing 3.3) must extend the class `SharedItem` (`de.tud.kom.lifesocial.core.data.impl`) and usually contain a payload (e.g. data.MyPluginItemPayload, see Listing 3.4) which implements the `Payload` (located at `de.tud.kom.lifesocial.core.data.interfaces`) and `Serializable` (`java.io`) interfaces.

```
1  import data.MyPluginItem;
   import de.tud.kom.lifesocial.core.commoninterfaces.Plugin;
3
   /** @author Luciana Alvite
5   * @created 26.07.2009 */

7  public interface IMyPlugin extends Plugin {

9    // this is one of my commands
     public String myCommand1();
11
     // this is another of my commands
13   public int myCommand2(String myParameter, String myParameter2);

15   // returns my item
     public MyPluginItem getMyPluginItem(String myParameter);
17
     /** more commands here **/
19 }
```

Listing 3.1: IMyPlugin

```
1
package component;
3 import api.IMyPlugin;
import data.MyPluginItem;
5
/**
7  * @author Luciana Alvite
 * @created 26.07.2009
9  */
public class MyPluginComponent implements IMyPlugin {
11
  @Override
13  public String myCommand1() {
    // TODO the implementation of the command goes here
15    int i = this.doSomething();
    // do something with the int
17    // returns a String
    return null;
19  }

21  @Override
  public int myCommand2(String myParameter, String myParameter2) {
23    // TODO the implementation of the command goes here
    return 0;
25  }

27  @Override
  public MyPluginItem getMyPluginItem(String myParameter) {
29
    // do something using the overlay network (network, storage) to obtain
31    // my item
    /****** ........ ******/
33
    // TODO should return the obtained item
35    return null;
  }
37
  @Override
39  public String getPluginId() {
    return PluginPropertiesContainer.PluginID;
41  }

43  /*** PRIVATE METHODS ***/
  private int doSomething (){
45  //does something
  }
47 }
```

Listing 3.2: MyPluginComponent

```
package data;

import de.tud.kom.lifesocial.core.data.impl.SharedItem;
import de.tud.kom.lifesocial.core.data.interfaces.Header;
import de.tud.kom.lifesocial.core.data.interfaces.StorageKey;

/**
 * @class MyPluginItem
 * @author Luciana Alvite
 * @created 03.08.2009
 */
public class MyPluginItem extends SharedItem {

  private static final long serialVersionUID = -7908436216462985959L;
  private MyPluginItemPayload payload;

  /**
   * Standard constructor
   */
  public MyPluginItem(StorageKey storageKey, Header header,
      MyPluginItemPayload payload) {
    super(storageKey, header);
    this.payload = payload;
  }

  /**
   * @return the payload
   */
  public MyPluginItemPayload getPayload() {
    return payload;
  }

  /**
   * @param payload
   *            the payload to set
   */
  public void setPayload(MyPluginItemPayload payload) {
    this.payload = payload;
  }

  /**
   * Returns a {@link String} representation of the item.
   */
  @Override
  public String toString() {
    return this.storageKey.toString() + this.header.toString()
        + this.payload.toString();
  }

}
```

Listing 3.3: MyPluginItem

```
package data;

import java.io.Serializable;

import de.tud.kom.lifesocial.core.data.interfaces.Payload;

/**
 * @class MyPluginItemPayload
 * @author Luciana Alvite
 * @created 03.08.2009
 */
public class MyPluginItemPayload implements Payload, Serializable {

  /**
   *
   */
  private static final long serialVersionUID = 126098134721871659 9L;

  private String myVariable;

  /**
   * Returns a {@link String} representation of the object.
   */
  @Override
  public String toString() {
    String result = "";
    result = result.concat("\nMyPlugin item payload:" + "\n");
    result = result.concat("MyPlugin variable: " + this.myVariable + "\n");
    return result;
  }

}
```

Listing 3.4: MyPluginItemPayload

3.1.5 Plugin Commands

In our implementation every plug-in has its own set of commands which are made available to the platform only after the corresponding plug-in is activated, since it does not make sense to have the commands available if the plug-in itself is not available. The class diagram 3.1 illustrates the command architecture for the Profile Plugin.

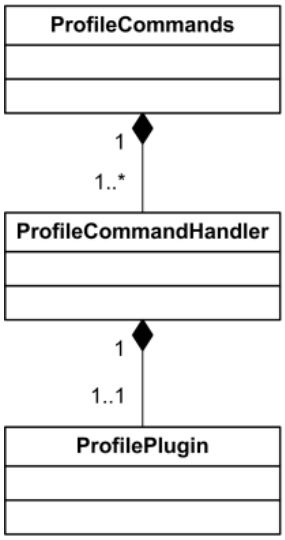


Figure 3.1: Example of the command architecture

Each command (e.g. commands.MyPluginMyCommand, see Listing 3.5) a Plugin offers must be implemented in a component extending the AbstractCommandHandler (`de.tud.kom.lifesocial.core.commands`) interface and all CommandHandlers are aggregated in a component (e.g. MyPluginCommands, see Listing 3.6) that extends the class AbstractCommands (found in the same package). This class act as an aggregator of 1 or more CommandHandlers for a certain plug-in and act as a proxy for command execution. It contains a identifier of the plug-in it is associated with and maps commands to the corresponding CommandHandler. It also offers other functionalities such as help for all plug-in's commands and checking command availability. The CommandHandlers are the ones responsible for actually interacting with the plug-in when executing the command and returning appropriate results back.

The commands are then automatically made available to the platform when the related plug-in is present. Notice that plug-in's commands are ONLY available when the corresponding plug-in is activated, and the departure of a plug-in means the deregistration of its commands from the platform, i.e. commands may become unavailable at times. The platform is designed to return a "Command Not Available" message when the user tries to execute a command which is (temporarily or not) unavailable.

Usually command creation is accomplished by first creating an abstract CommandHandler class (e.g AbstractMyPluginCommandHandler, see Listing 3.7) which extends the AbstractCommandHandler class mentioned above and will be the basis for all other plug-in command handlers. In

```
package commands;

import utils.PluginPropertiesContainer;
import data.MyPluginItem;

/**
 * @author Luciana Alvite
 * @created 26.07.2009
 *
 */
public class MyPluginMyCommand extends AbstractMyPluginCommandHandler {

  public MyPluginMyCommand() {
    // name the command will be known in the platform
    this.command = PluginPropertiesContainer.CommandGetMyPluginItem;
    this.help = "<myParameter1> - Explain how to use the command here";
  }

  @Override
  public Object executeCommand(String arg) {

    // EXAMPLE of parameter validity checking
    String myParameter1 = arg;

    if (!(this.checkValidArg(myParameter1)))
      return PluginPropertiesContainer.ParameterMissingMsg;

    // calls method on component to obtain a MyPluginItem
    MyPluginItem result = this.myPlugin.getMyPluginItem(myParameter1);
    // returns the obtained item
    return result;
  }

  @Override
  public String executeCommandToString(String arg) {
    // EXAMPLE of parameter validity checking
    String myParameter1 = arg;

    if (!(this.checkValidArg(arg)))
      return PluginPropertiesContainer.ParameterMissingMsg;

    // calls method on component to obtain a MyPluginItem
    MyPluginItem result = this.myPlugin.getMyPluginItem(myParameter1);
    // returns the obtained item
    return result.toString();
  }
}
```

Listing 3.5: MyPluginMyCommand

```
1  package commands;

3  import utils.PluginPropertiesContainer;
   import de.tud.kom.lifesocial.core.commands.AbstractCommands;
5
   /**
7   * @author Luciana Alvite
    * @created 26.07.2009
9   */
   public class MyPluginCommands extends AbstractCommands {
11
     public MyPluginCommands() {
13     this.pluginId = PluginPropertiesContainer.PluginID;
     }
15 }
```

Listing 3.6: MyPluginCommands

this abstract class the common variables and methods for all other plug-in commands are defined, which typically means the bind and unbind methods and the myPlugin instance for all commands (more on this in section 3.2).

The addition of the CommandHandlers to the Commands (e.g. MyPluginCommands) component is already implemented in the superclass AbstractCommands, so what the new subclass needs to override is mainly only the PluginID, which should be defined in the properties class PluginPropertiesContainer (see Section 3.1.6).

Obviously all Plugin commands depend on the plug-in and this dependency must be declared on the component .xml file (see Section 3.2) for each command and the corresponding binding methods declared on the (abstract) CommandHandler .

```
1  package commands;

3  import api.IMyPlugin;
   import de.tud.kom.lifesocial.core.commands.AbstractCommandHandler;
5
   /**
7   * @author Luciana Alvite
    * @created 26.07.2009
9   */
   public abstract class AbstractMyPluginCommandHandler extends
11      AbstractCommandHandler {

13    protected IMyPlugin myPlugin = null;

15    /***
       * if all command handlers for the plugin depend on a certain plugin,
17     * declare the binding methods here.
       *
19     * Usually the command handlers depend AT LEAST on the plugin it represents
       * (MyPlugin here)
21     *
       * The commands will then only be available when the corresponding plugin is
23     * available (which makes sense in this case)
       *
25     */

27    /********** BINDING METHODS **************/
      public synchronized void setMyLoginPlugin(IMyPlugin myPlugin) {
29      this.myPlugin = myPlugin;
      }
31
      public synchronized void unsetMyLoginPlugin(IMyPlugin myPlugin) {
33      this.myPlugin = null;
      }
35 }
```

Listing 3.7: AbstractMyPluginCommandHandler

3.1.6 Plugin Properties

Previously all Platform constants were stored in the Class PropertiesContainer and any changes in status messages, plug-in ID's, etc required the modification of this class. The new implementation of this architecture splits the properties in global properties (which should be know to all Plug-ins) and plug-in specific properties such as Plug-in ID and the commands it offers, which is a contract made between the plug-in implementation and the GUI and does not need to be known by the whole platform. This was implemented by creating the new PluginPropertiesContainer class in the utils package of every plug-in, which is a class that extends the main PropertiesContainer class (found in the `de.tud.kom.lifesocial.utilities` Bundle) and adds its own constants to it. An example of a PluginPropertiesContainer can be seen in Listing 3.9.

When new plug-in commands are added, the name convention shown in Listing 3.8 should be adopted:

```
1 String <PluginID><.><Command>
```

Listing 3.8: Command Naming Convention

The pluginID can be added by concatenating PluginPropertiesContainer.PluginID, as seen in Listing 3.9.

In the future this could be implemented in a property file and localization of return messages should also be considered, but this is still under development.

3.2 Extending the Plugin on DS

In order to be able to interact with the platform, the plug-in must comply with the adopted Declarative Services architecture. In this Section we elaborate on how to extend the core plug-in implementation so it can interact with the Platform and its plug-ins. So far, we have explained how to create a plug-in, including its data types and commands, but its function are still not available to the Platform, since the OSGi Framework is not aware of any services the Bundle is offering and therefore can't interact with it. In order to make its services available to the LifeSocial Platform, a plug-in must expose them by creating so called "Service Components" (see [All07]) for all components which will offer some service to the platform, be it commands or the actual plug-in interface. The steps necessary to extend the core plug-in implementation on DS are:

1. Create the component definitions (.xml Files)
2. Declare the exposed services
3. Declare the components dependencies
4. Add the bind and unbind methods for the dependencies to the Component class
5. Inform the MANIFEST file about the newly created service component
6. Add the service component definition to the build path

Following we explain the steps in details:

```
/**
 *
 * @class PluginPropertiesContainer
 * @author Luciana Alvite
 * @created 30.07.2009
 */
package utils;
import de.tud.kom.lifesocial.utilities.PropertiesContainer;

/**
 * @author Luciana Alvite
 * @created 30.07.2009
 */
public class PluginPropertiesContainer extends PropertiesContainer {

  /**
   * GENERAL FIELDS, WILL BE AVAILABLE TO ALL PLUGINS!!!
   *
   * @see de.tud.kom.lifesocial.utilities.PropertiesContainer)
   **/

  /******** PLUGIN SPECIFIC FIELDS ********/
  /*** plugin ID ***/
  public static final String PluginID = "MyPlugin";

  /*** Plug-in Commands ***/
  public static final String CommandMyCommand1 = PluginPropertiesContainer.PluginID
      .concat(".").concat("myCommand1");
  public static final String CommandMyCommand2 = PluginPropertiesContainer.PluginID
      .concat(".").concat("myCommand2");
  public static final String CommandGetMyPluginItem = PluginPropertiesContainer.
     PluginID
      .concat(".").concat("getMyPluginItem");

  /*** Return Status ***/
  public static final Integer myCommand1Success = 1;
  public static final Integer myCommand1Failure = 2;
  public static final Integer myCommand2Success = 1;
  public static final Integer myCommand2Failure = 2;
  public static final Integer getMyPluginItemSuccess = 1;
  public static final Integer getMyPluginItemFailure = 2;

  /*** Return Status Messages ***/
  public static final String getMyPluginItemFailureMsg = " Error obtaining
     MyPluginItem";
}
```

Listing 3.9: PluginPropertiesContainer

1. Create the component definitions (.xml Files)

   For each component which will offer some service to the platform a Service Component must be created. In general the following components must be created:

   - The plug-in component (1, e.g. MyPluginComponent)
   - The command aggregator (1, e.g. MyPluginCommands)
   - One for each command offered by the plug-in (1-n, e.g. MyPluginMyCommand)

   The first step is to create a folder named `OSGI-INF` in the project's main folder (see Figure 3.2). Then for each service component create an .xml by right clicking on the `OSGI-INF` folder, choose: New -> Other -> Plug-in Development -> Component Definition and choosing a suitable name for the component. The naming convention for the components is:

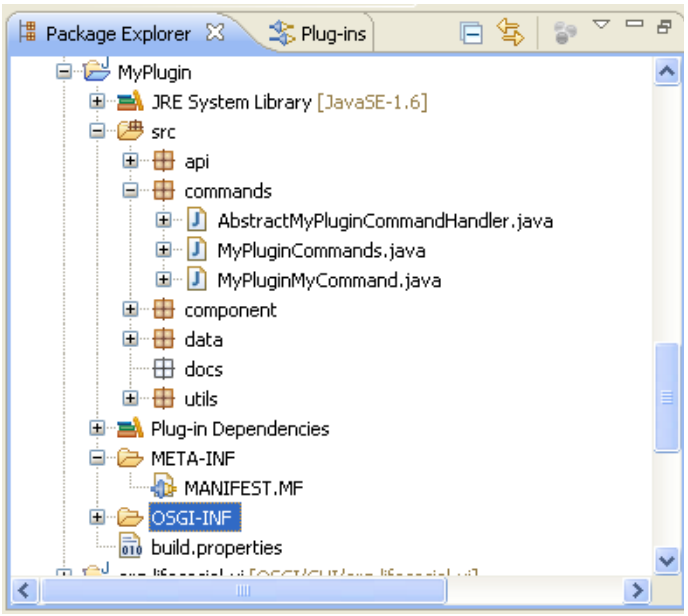


Figure 3.2: OSGI-INF Folder

   - `<pluginname>component` for the plug-in component (e.g. myplugincomponent.xml)
   - `<pluginname>commands` for the command aggregator (e.g. mypluginCommands.xml)
   - `<pluginname><command_name>command` for the commands offered by the plug-in (e.g. mypluginmycommandcommand.xml)

   The name of the component should be the same as the .xml file for clarity.

   Next, choose the class of the component (this would be the class that implements the component, MyPluginComponent in this case, see Figure 3.3). Click on finish. Do this for all components (3 in our example, see Figures 3.4 and 3.5). In the `OSGI-INF` folder there should now be 3 components, as can be seen in Figure 3.6.

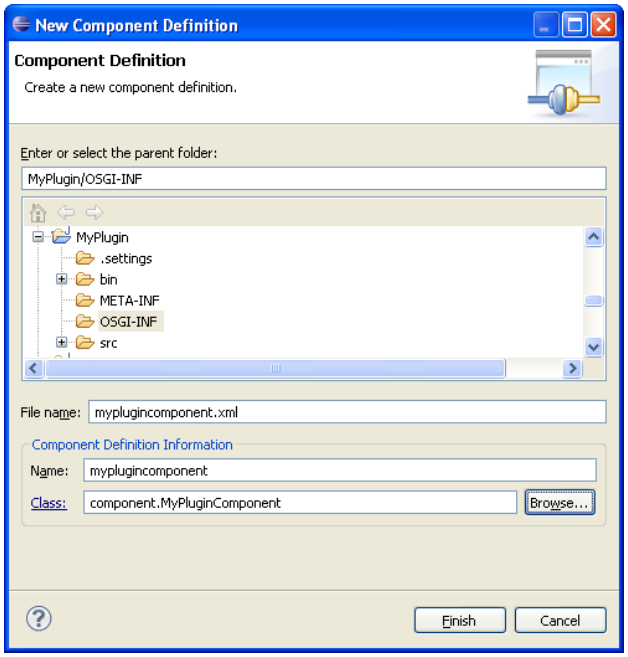


Figure 3.3: Component Definition - The main plug-in component

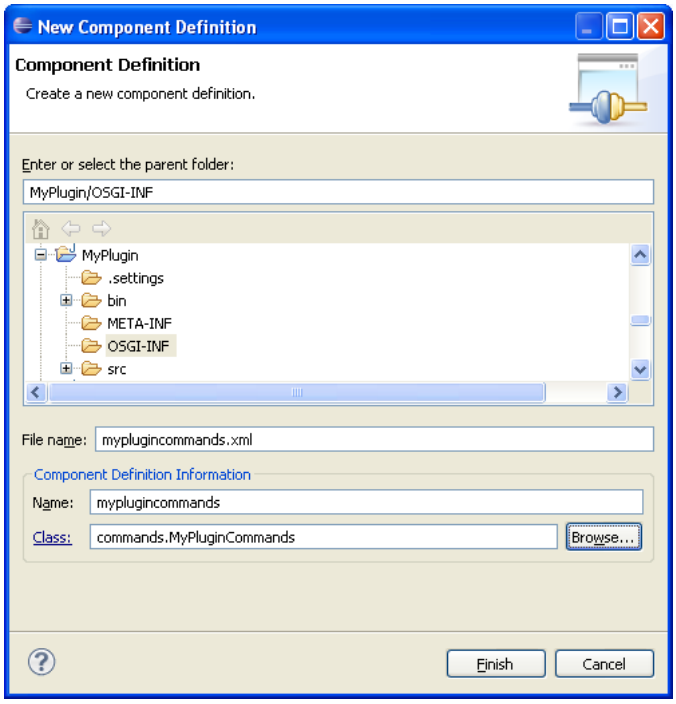


Figure 3.4: Component Definition - Commands

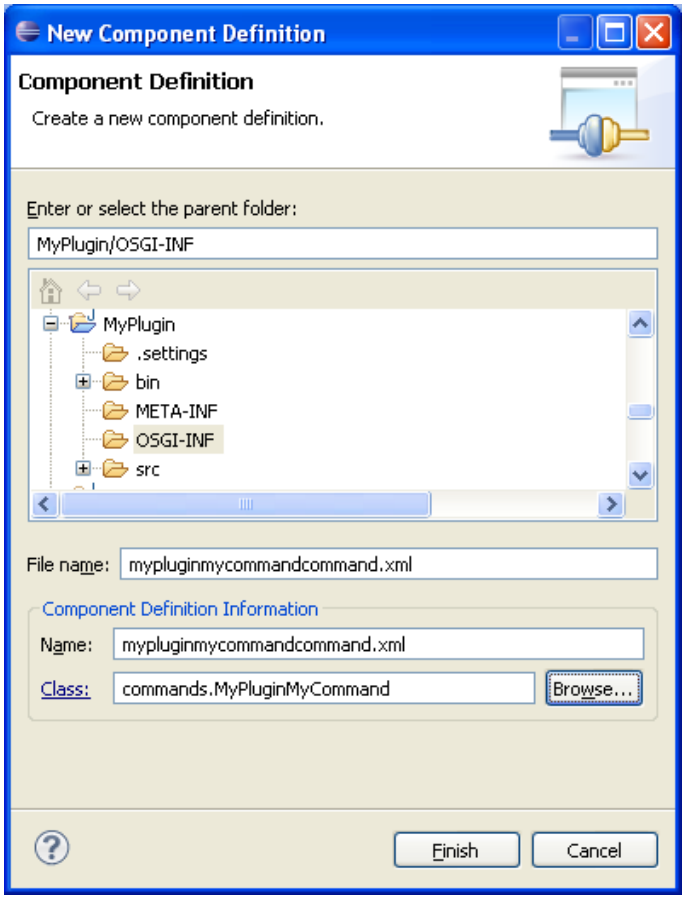


Figure 3.5: Component Definition - My Command

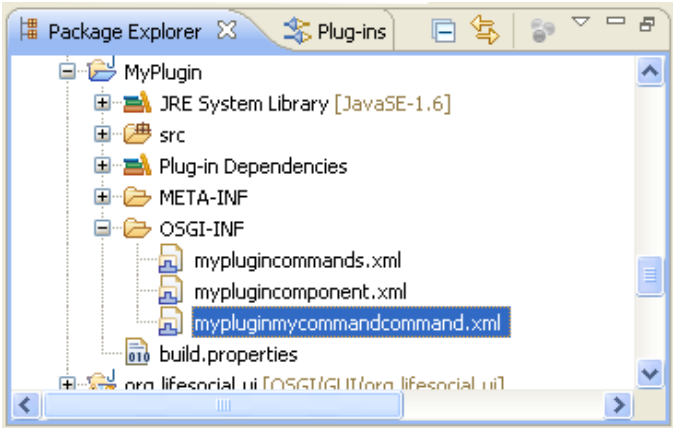


Figure 3.6: OSGI-INF folder - all components

2. Declare the exposed services

Now for each component, open the component editor file (right click on the .xml file), choose the "Services" tab and under "Provided Services" click on "Add...". Choose the interface the component will be exposing to the platform (api.IMyPlugin for the MyPlugin Component, see Figure 3.7). Notice the plug-in component will be exposing the interface and not the actual plug-in implementation in this case! Do the same for the remaining components (the exposed services will be `de.tud.kom.lifesocial.core.commands.ICommands` for the commands aggregator and commands.AbstractMyPluginCommandHandler for all commands, see Figures 3.8 and 3.9).

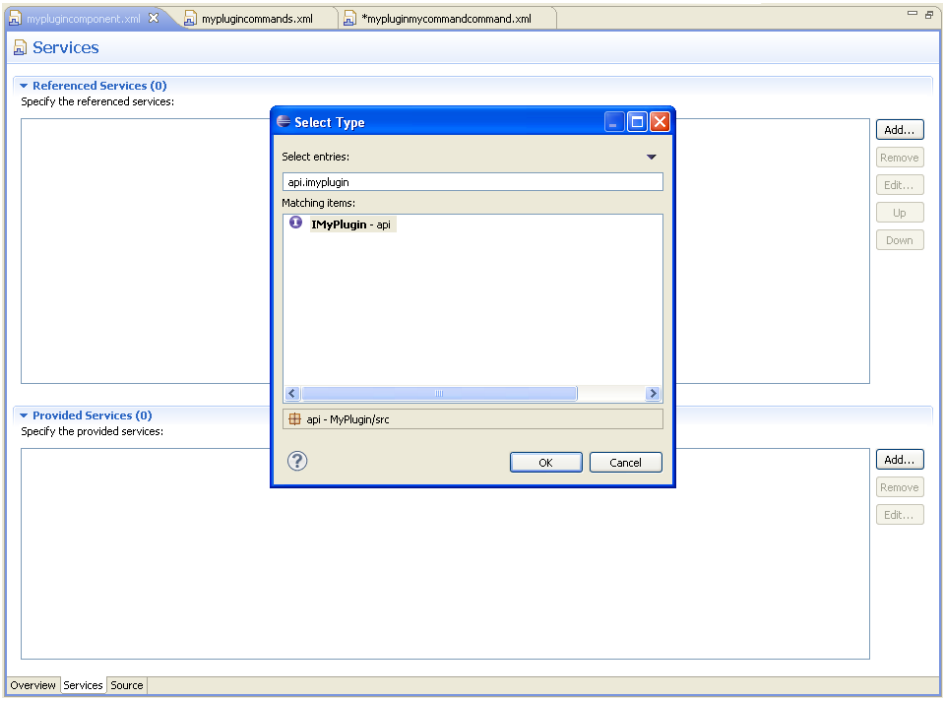


Figure 3.7: Component Definition - Exposing the plug-in interface

Finally, if our plug-in wants to offers services not only to its own bundle but also to other modules/plug-ins in the framework, the plug-in interface also needs to be exported. This can be done through the MANIFEST editor, Tab "Runtime", under "Exported Packages". In general, the only package that needs to be exported is the one containing the plug-in interface (the api package). Figure 3.10 shows the exported packages for our example:

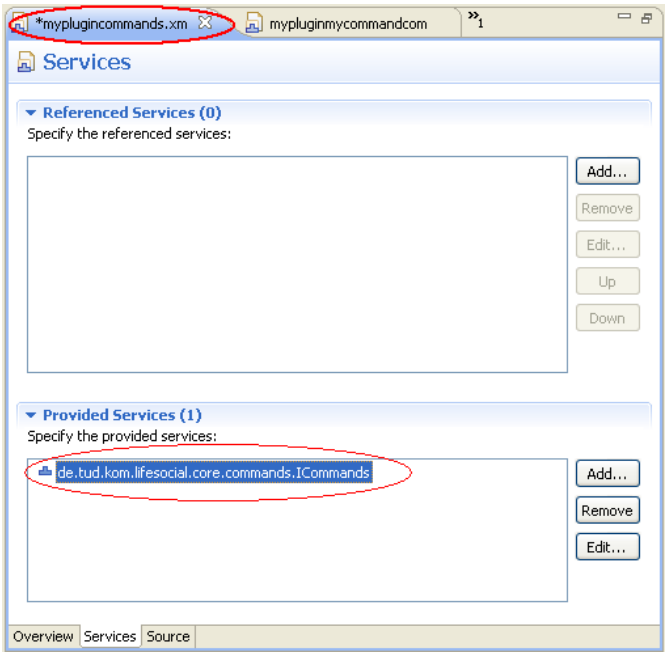


Figure 3.8: Component Definition - Exposing the command aggregator

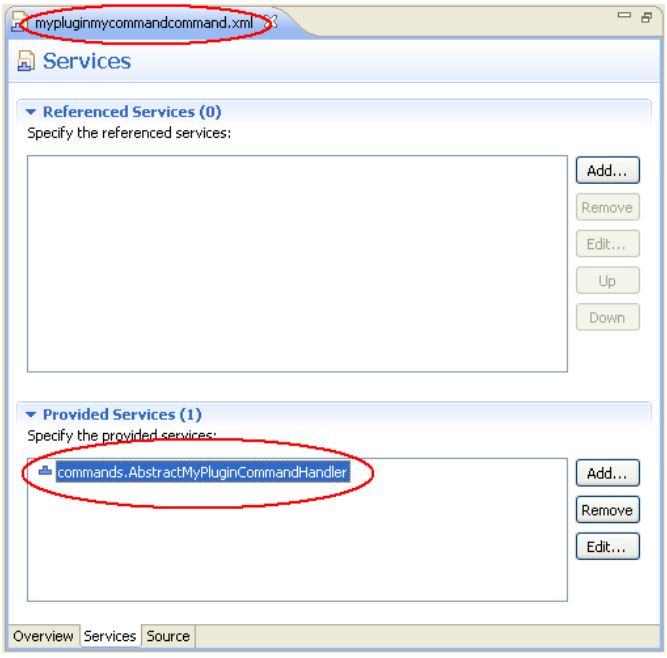


Figure 3.9: Component Definition - Exposing a command

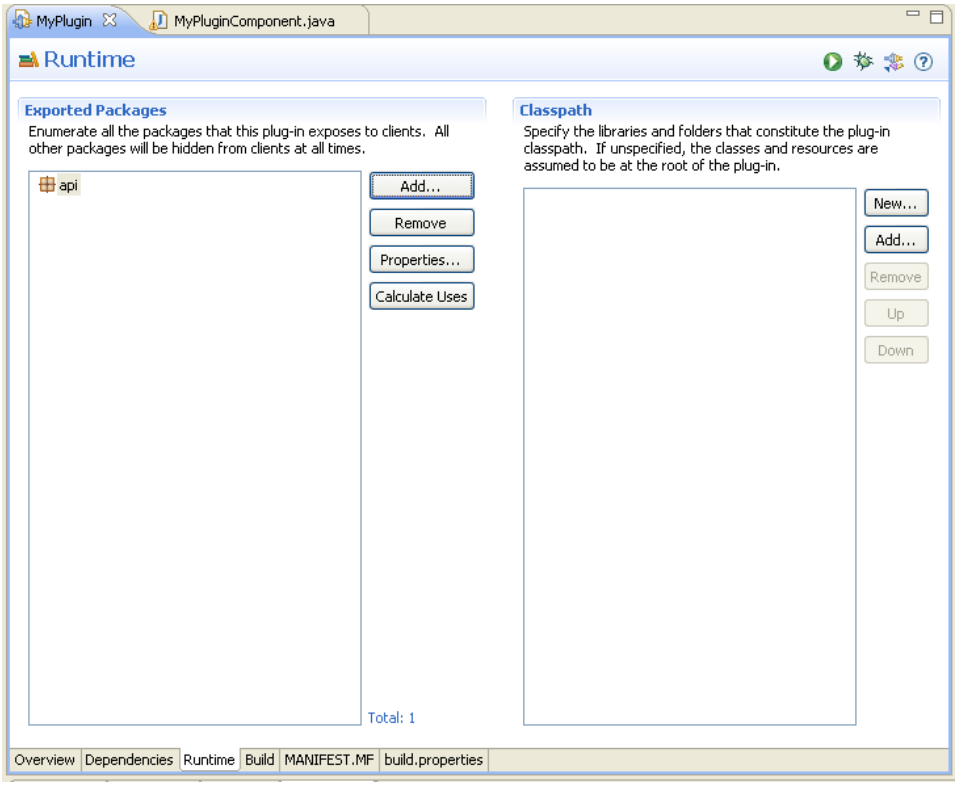


Figure 3.10: Manifest File - Exporting a Package

3. Declare the components dependencies

   Now we need to declare the dependencies from the components we have just created. As seen in Section 3.1.5, the command aggregator will depend on 1 or more CommandHandlers and the CommandHandler will depend (at least) on the plug-in they represent. We need to declare this in the component definition file. Back to the component editor, select the "Services" tab again and under "Referenced Services" click on "Add...". The dependencies of our example plug-in components will be:

   - 0 or more plug-in/modules for the plug-in component (the modules could be the network, storage, message dispatcher or other plug-ins like the login plug-in). In our example, the plug-in component depends on the MessageDispatcher only.
   - 1 or more AbstractMyPluginCommandHandler for the command aggregator (it needs at least one command)
   - (At least) 1 and only one MyPlugin for all commands. Depending on the function the command executes, it might have dependencies on other plug-ins/module as well (i.e. the Login plug-in to check if the user is logged in or not, etc). In our case, all commands also depend on 1 and only one Login plug-in.

   Figure 3.11 shows the dependencies for the MyPluginComponent (the IMessageDispatcher Interface, located at `de.tud.kom.lifesocial.messagedispatcher.api`). Make sure to choose the following options (see Figure 3.12) for the service (by clicking on "Edit..." after the dependency is added):

   - Name: MessageDispatcher
   - Cardinality: 1..1 (one and only one reference)
   - Policy: dynamic (will be dynamically informed of changes in the service)
   - Bind: setMessageDispatcher (this is the name of the bind method)
   - Unbind: unsetMessageDispatcher (accordingly, this is the name of the unbind method)

   Figures 3.13 and 3.14 show the dependencies for MyPluginCommands and MyPluginMyCommand respectively. Notice the command handlers depend on both the IMyPlugin and the ILoginPlugin interfaces. The options for these dependencies are:

   - Name: CommandHandlers
   - Cardinality: 1..n (one or more commands)
   - Policy: dynamic
   - Bind: addCommandHandler (bind method)
   - Unbind: removeCommandHandler (unbind method)

   - Name: MyPlugin

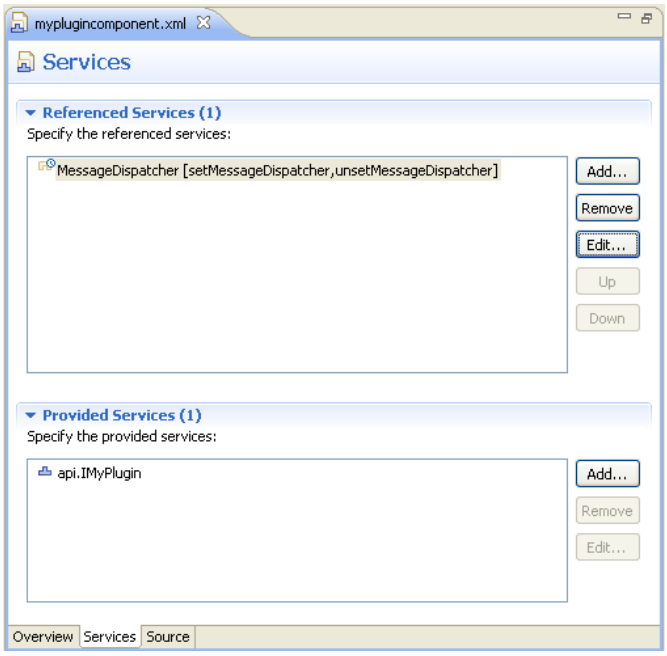


Figure 3.11: Component Definition - Declaring a Dependency

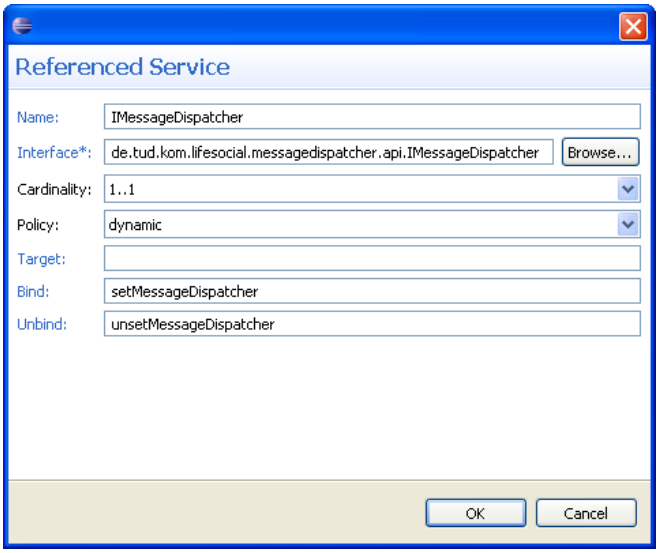


Figure 3.12: Component Definition - Dependency Options

- Cardinality: 1..1
- Policy: dynamic
- Bind: setMyPlugin (bind method)
- Unbind: unsetMyPlugin (unbind method)

- Name: LoginPlugin
- Cardinality: 1..1
- Policy: dynamic
- Bind: setLoginPlugin (bind method)
- Unbind: unsetLoginPlugin (unbind method)

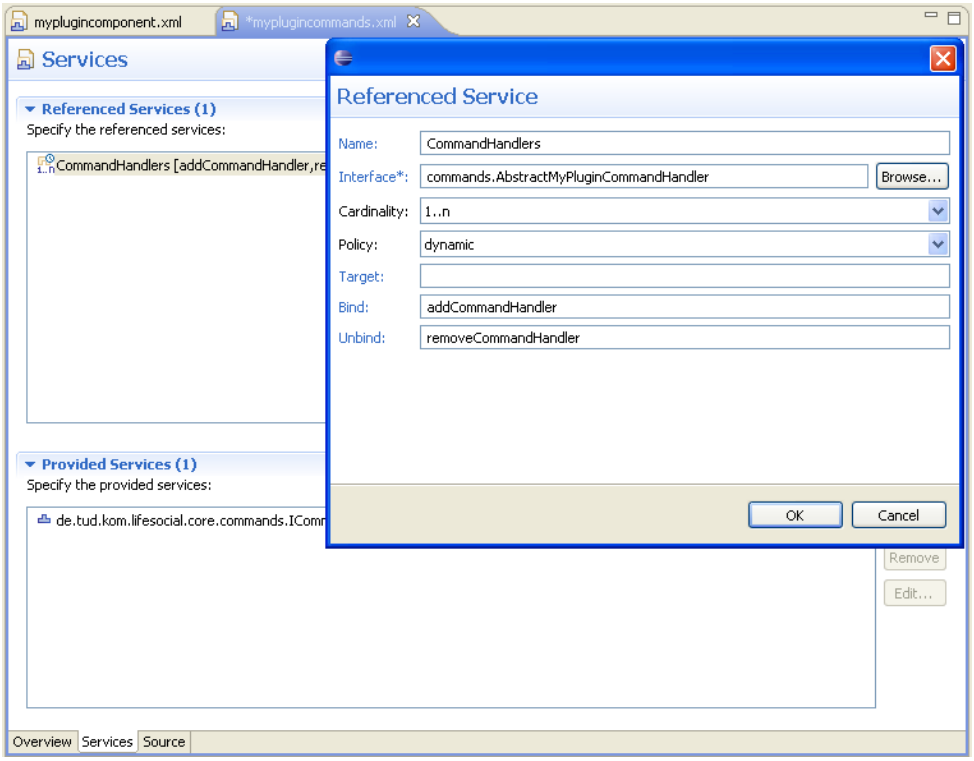


Figure 3.13: Component Definition - Command Aggregator Dependencies

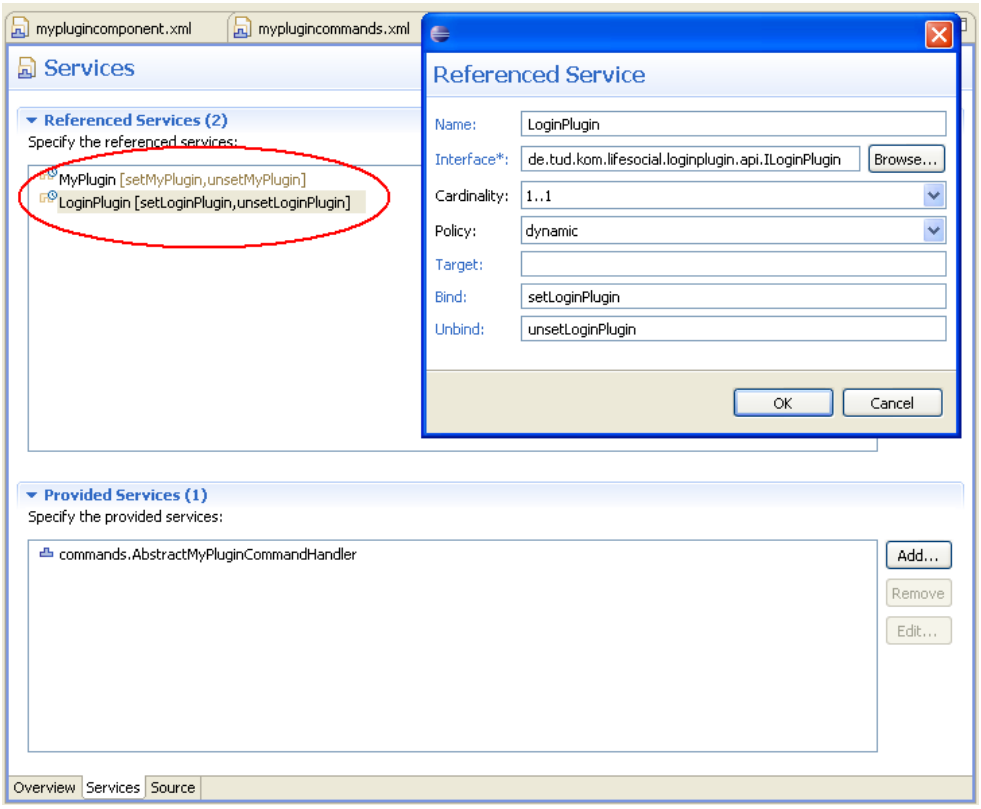


Figure 3.14: Component Definition - Command Handlers Dependencies

4. Add the bind and unbind methods for the dependencies to the Component class

   In order to be informed when its dependencies are available in the framework, a component must implement bind and unbind methods which will be called back when the dependencies are registered in the framework. The component must have a pair of bind/unbind methods for each of its dependencies. The naming convention for these methods is:

   - `add<dependency_name>` and
   - `remove<dependency_name>`, when the dependency is multiple (0/1..n). OR

   - `set<dependency_name>` and
   - `unset<dependency_name>`, when the dependency is unitary (1..1)

   An example of a pair of binding methods is shown in Listing 3.10. Note that a instance variable of the same type should be declared to hold the reference to the service.

   Only adding the methods is not sufficient though, since the components are not known in the build path of the plug-in. We first have to import the relevant packages from the modules our plug-in depends on in the MANIFEST file. In our example plug-in, these packages would

```
2   //the local variable for the service reference
    protected IMyPlugin myPlugin = null;
4
    /********** BINDING METHODS **************/
6   public synchronized void setMyPlugin(IMyPlugin myPlugin) {
      this.myPlugin = myPlugin;
8   }

10  public synchronized void unsetMyPlugin(IMyPlugin myPlugin) {
      this.myPlugin = null;
12  }
```

Listing 3.10: Example of Service Binding

be:

- `de.tud.kom.lifesocial.core.commands` (for the Commands aggregator)
- `api.IMyPlugin` (for the Command Handlers)
- `de.tud.kom.lifesocial.loginplugin.api` (also for the Command Handlers)
- `de.tud.kom.lifesocial.messagedispatcher.api` (for the plug-in component)

Open the manifest file editor, select the "Dependencies" Tab and under "Imported packages" add the packages mentioned above. Note that the first import was already added earlier in the development and the second one is local to the Bundle, therefore already known. Therefore, only the 2 last packages needed to be imported. Figure 3.15 shows the result after all imports needed for our plug-in have been added.

Now we are able to add the binding methods to the components. In general, the binding methods for the plug-in dependency (IMyPlugin in our example) of the command handlers are added to the abstract superclass of all command handlers, AbstractMyPluginCommandHandler, since ALL commands will depend on this service. In our case all commands will also depend on the Login plug-in and therefore we also added its binding methods to the abstract superclass. When only some commands need to access functions from other plug-ins, it makes sense to add the bind methods to the specific handler and not to the superclass.

The plug-in component will need binding methods for the Message Dispatcher and the Command aggregator would need binding methods for the command handlers, but its superclass AbstractCommands (see `de.tud.kom.lifesocial.core.commands`) already have them, so nothing needs to be added to MyPluginCommands.

Listings 3.11 and 3.12 shows the binding methods for MyPluginComponent and AbstractMyPluginCommandHandler respectively

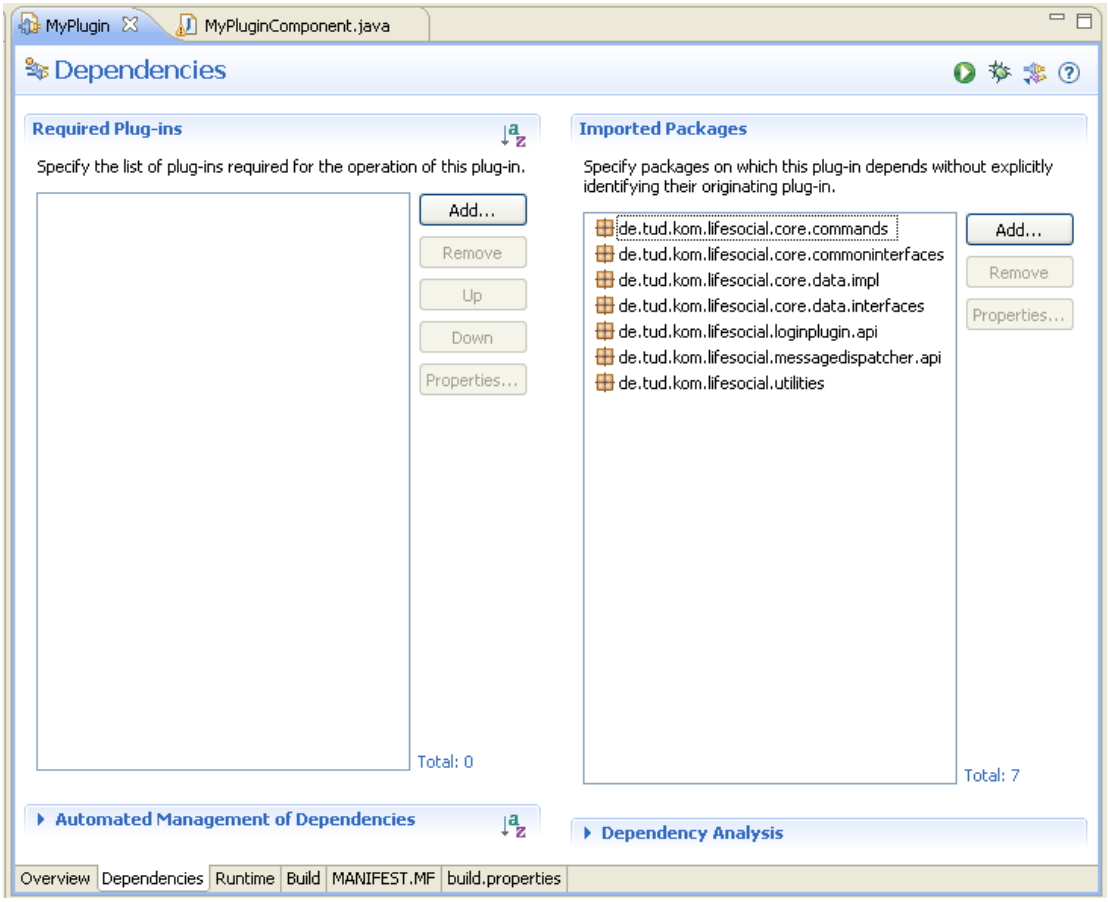


Figure 3.15: Manifest File - Imported Packages

```
2 /**
  * @author Luciana Alvite
4 * @created 26.07.2009
  */
6 public class MyPluginComponent implements IMyPlugin {

8   //the local variable that will hold the service reference
    private IMessageDispatcher messageDispatcher = null;
10
    /******* BINDING METHODS *******/
12  public synchronized void setMessageDispatcher(IMessageDispatcher md) {
      this.messageDispatcher = md;
14  }

16  public synchronized void unsetMessageDispatcher(IMessageDispatcher md) {
      this.messageDispatcher = null;
18  }

20 }
```

Listing 3.11: Binding Methods - MyPluginComponent

```
/**
 * @author Luciana Alvite
 * @created 26.07.2009
 */
public abstract class AbstractMyPluginCommandHandler extends
    AbstractCommandHandler {

  //the local variables that will hold the service references
  protected IMyPlugin myPlugin = null;
  protected ILoginPlugin login = null;


  /********** BINDING METHODS **************/
  public synchronized void setMyPlugin(IMyPlugin myPlugin) {
    this.myPlugin = myPlugin;
  }

  public synchronized void unsetMyPlugin(IMyPlugin myPlugin) {
    this.myPlugin = null;
  }

  public synchronized void setLoginPlugin(ILoginPlugin login) {
    this.login = login; // don't forget to declare a local variable for the
    // plugin!
  }

  public synchronized void unsetLoginPlugin(ILoginPlugin login) {
    this.login = null;
  }
}
```

Listing 3.12: Binding Methods - AbstractMyPluginCommandHandler

5. Inform the MANIFEST file about the newly created service component

   The last step is to add the service components to the manifest file. Open the manifest file for the project and click on the MANIFEST.MF tab. If the Service component was created using the steps mentioned in step 1, there should be now a new entry in the MANIFEST file:

```
Service-Component: OSGI-INF/myplugincomponent.xml,
 OSGI-INF/mypluginmycommandcommand.xml
```

Listing 3.13: Manifest File - Service Component Declaration

   If the .xml file was created manually or copied from another project, make sure to insert this entry into the MANIFEST file. This makes sure the newly created components are known to the Framework.

6. Add the service component definition to the build path

   Add the `OSGI-INF` to the Build Configuration in the MANIFEST file (Manifest editor -> Build Tab -> Binary Build, check the `OSGI-INF` folder).

We have now created the service components for the plug-in, declared its dependencies and made services available to the platform. This is all a plug-in wanting to interact with the platform needs to start interacting with the core OSGi based LifeSocial platform. When using the equinox console extension module, its plug-ins should now be available. For more information about how to integrate the GUI and the core plug-in implementation, refer to section 3.4.

3.3 Interacting with the platform - The Commands Bundle

A lot of work is being done in developing GUIs for the LifeSocial network and currently there is an unified interface to interact with the core modules of the platform. Using the "Commands" interface, the developer simply needs to call[5]:

```
execute(<PluginID><.><Command_Name> <Parameter1> <Par2> <ParN>)
```

The Commands interface will then be responsible for checking if the plug-in responsible for the Command is currently loaded and executing the command on the plug-in. In case the plug-in is not loaded, it returns an error message informing the user. The "Commands" interface is also implemented using Declarative Services and in order to integrate it with the GUI some modifications to the GUI code need to be done. This will be the subject of Section 3.4.

[5] for a list of the commands currently available in the LifeSocial platform, refer to [Alv09]

3.4 Linking the GUI

As seen in Section 3.3, the OSGi version of the core LifeSocial platform relies on Declarative Services to deal with dynamic plug-in management. The commands interface is also itself implemented using DS and therefore to access it GUI developers need to adapt the code to comply with the new interface. The steps necessary to link the GUI and the core DS based platform are:

1. Create a component definition for the GUI (.xml File)
2. Declare the components dependencies ("Commands")
3. Add the bind and unbind methods to the Activator class of the GUI (`add` and `removeCommands()`)
4. Add a static `getCommands()` methods to the Activator
5. Inform the MANIFEST file about the newly created service component
6. Adapt the relevant classes to use the new commands interface (typically the handlers, threads or any other class that needs to use plug-in functions)
7. Modify the Run Configurations for the whole platform (see section 3.5)

Following we explain the steps in details:

1. Create a component definition for the GUI
   - Create a folder named `OSGI-INF` in the project's main folder
   - Right click on the `OSGI-INF` folder, choose: New -> Other -> Plug-in Development -> Component Definition (see Figure 3.16), choose a suitable name for the .xml File. Suggestion: `<plugin_name>GUIcomponent`. The name of the component should be the same as the .xml file for clarity.
   - Choose the class of the component (this would usually be the Activator class of the GUI, see Figure 3.17). Click on finish. This should open the newly created component editor ("Overview").

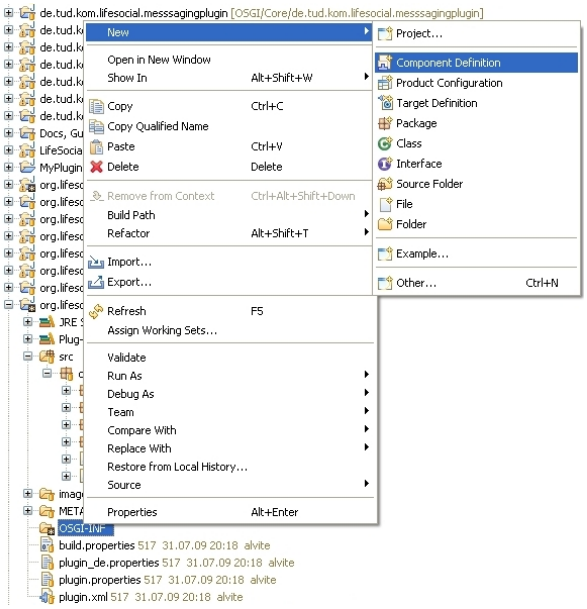


Figure 3.16: Creating a Service Component (GUI)

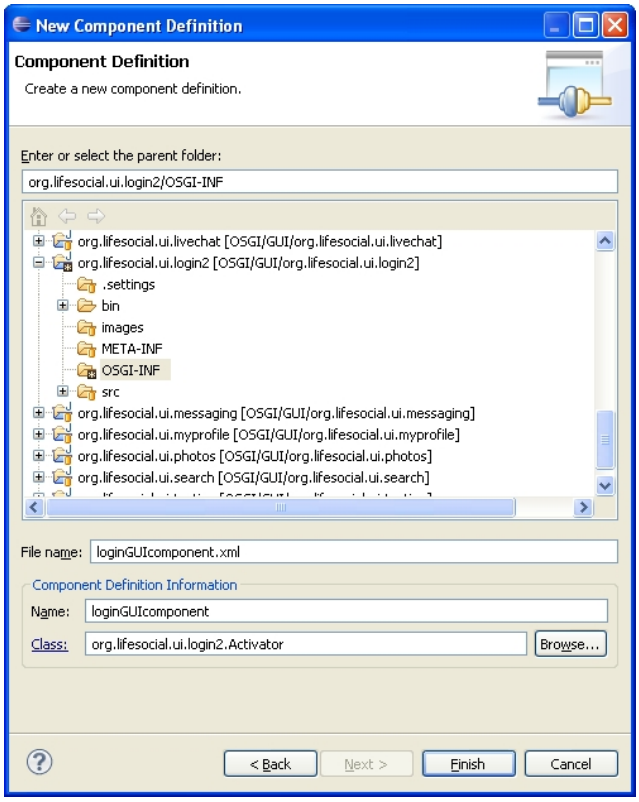


Figure 3.17: Creating a Service Component - Component Definition (GUI)

2. Declare the components dependencies

   Select the "Services" tab and under "Referenced Serviced", click on "Add..". Choose the ICommandsComponent interface (located at `de.tud.lifesocial.api.commands`) (see Figure 3.18) and click on ok. Back to the "Services" tab, choose the newly added dependency and click on "Edit.." and make sure the following options and selected (see Figure 3.19):

   - Name: COMMANDS
   - Cardinality: 1..1 (one and only one reference)
   - Policy: dynamic (will be dynamically informed of changes in the service)
   - Bind: addCommands (this is the name of the bind method)
   - Unbind: removeCommands (accordingly, this is the name of the unbind method)

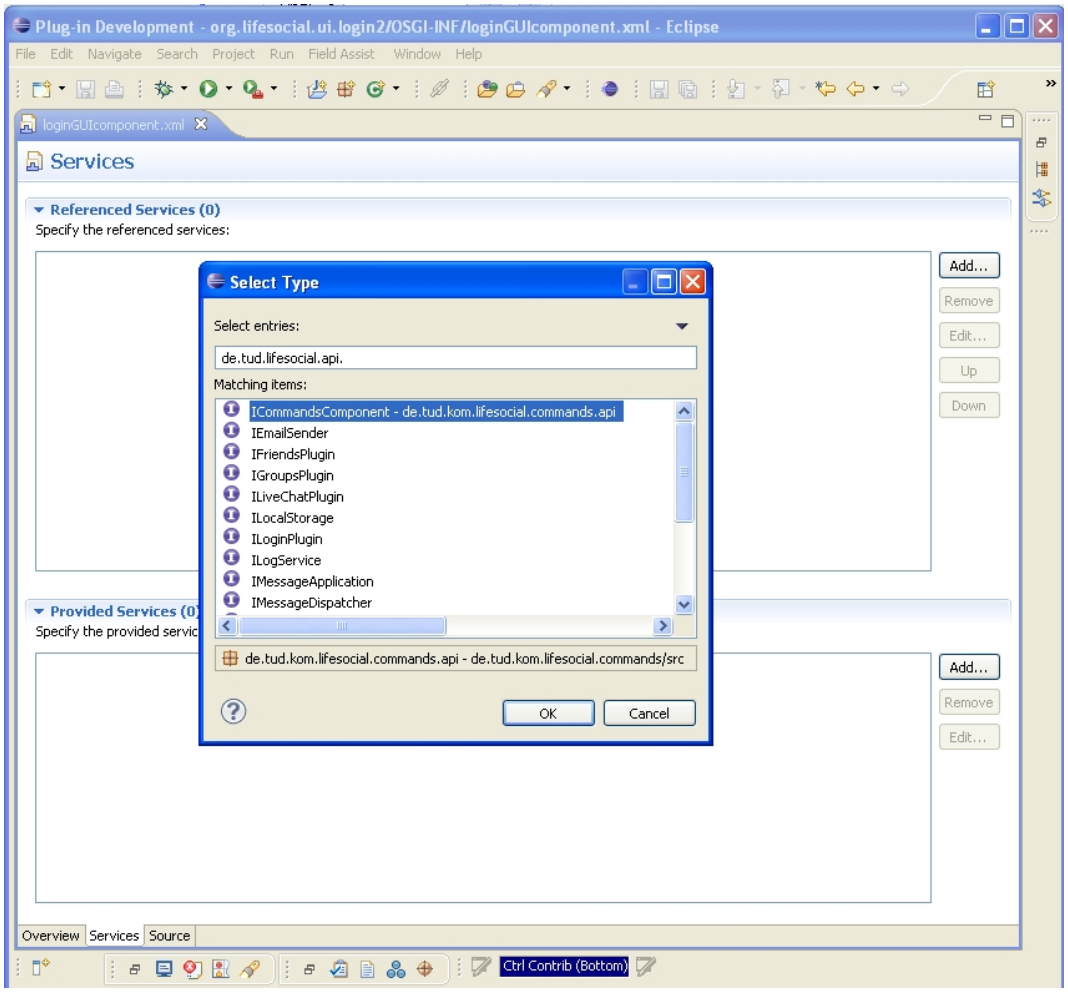


Figure 3.18: Service component - Adding a dependency (GUI)

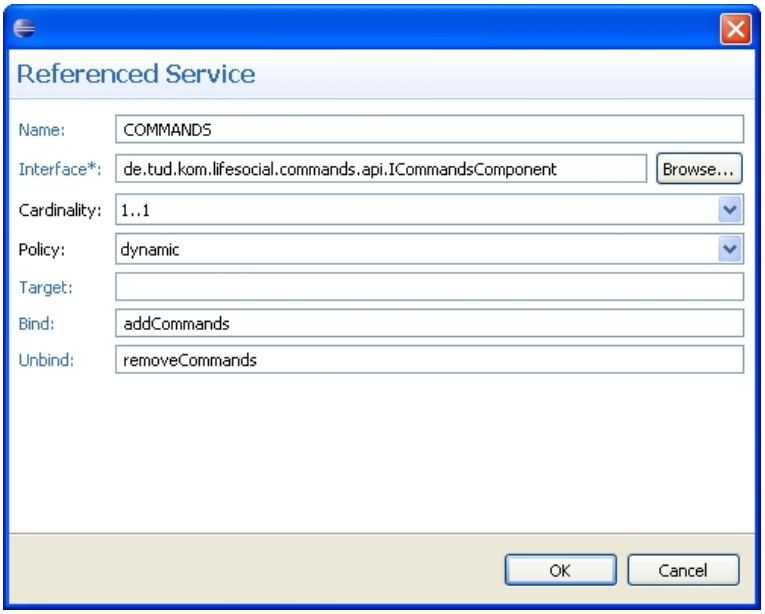


Figure 3.19: Service component - Editing a dependency (GUI)

3. Add the bind and unbind methods to the Activator class of the GUI

   We now need to actually implement the bind and unbind methods on the desired class (e.g. Activator). Add the following at the end of the GUI's Activator class:

```
1
  /********** BINDING METHODS **************/
3
    public synchronized void addCommands(ICommandsComponent command) {
5     this.commands = command;
    }
7
    public synchronized void removeCommands(ICommandsComponent command) {
9     this.commands = null;
    }
```

Listing 3.14: Service Binding - Commands Component(GUI)

   and of course add a private static variable named "commands".

4. Add a static `getCommands()` methods to the Activator

   Also add the following to the Activator class, so interested parts can have access to the commands component:

```
2 public static ICommandsComponent getCommands() {
    return commands;
4   }
```

Listing 3.15: Activator - getCommands() (GUI)

5. Inform the MANIFEST file about the newly created service component

   If the Service component was created using the steps mentioned in step 1, there should be now a new entry in the MANIFEST file:

```
Service-Component: OSGI-INF/loginGUIcomponent.xml
```

Listing 3.16: Manifest File - Service Component Declaration (GUI)

   If the .xml file was created manually or copied from another project, make sure to insert this entry into the MANIFEST file. This makes sure the newly created components are known to the Framework.

6. Adapt the relevant classes to use the new commands interface (typically the handlers, threads or any other class that needs to use plug-in functions)

   Now we have linked the commands component and the Activator of the GUI. In order to use this component the different classes in the GUI Project must first get a reference for the component by calling:

```
1
    ICommandsComponent commands = Activator.getCommands();
3
    if (commands == null) {
5     //handle the absence of commands, e.g. display an error message
      return;
7   }

9   //my command calls go here, e.g:
    Object result = commands.execute(PluginPropertiesContainer.
        CommandGetMyPluginItem);
11
    if(result instanceof String) {
13    //display error message
      return;
15  }
    if (result instanceof MyPluginItem) {
17    //correct type received, process
    }
```

Listing 3.17: Using the Commands component (GUI)

Notice the "execute" method returns a generic "Object". The developer should make sure to check the received result is of the correct type in order to avoid Runtime errors (e.g. use instanceof). ALL plug-ins should check if returned Object is an instance of String, since some error messages are returned as String Object.

Additionally, as seen in section 3.1.6, each Plug-in (both the actual implementation and the GUI) has now its own "PluginPropertiesContainer", located in the "util" folder, where plug-in specific final variables can be declared. This way, each plug-in can agree on a contract for these fields and the core module does not need to know about it. There are although some basic messages which should be present in ALL "PluginPropertiesContainers", therefore they all extend the core PropertiesContainer, located in the utilities bundle, which has to be added as a dependency too (import the utilities package). See `org.lifesocial.ui.util.PluginPropertiesContainer` for more information.

There are also two new Events called:

- `EVENT_COMMANDS_UNAVAILABLE` and
- `EVENT_COMMANDS_AVAILABLE`

which inform listeners (channel: `CHANNEL_COMMANDS`) the commands component is not present anymore. This should be used to enable/disable buttons and/or display a pop-up window giving the user some feedback.

7. Modify the Run Configurations for the whole platform

   This is explained in more detail in Section 3.5

Now we have created a component and declared a dependency on the Commands service. This is all a GUI wanting to interact needs to start interacting with the core OSGi based LifeSocial platform. It is important to point out that now the launch configuration for the platform will need to be modified to allow the Declarative Services bundle and other relevant Bundles to be automatically started. This is explained in more details in Section 3.5.

3.5 Run configurations for the GUI

Since the core plug-ins rely on Declarative Service to run, it is important to make sure the DS and other relevant bundles are started before the GUI is displayed to the user. Differently from an "OSGi Framework" run configuration, the "Eclipse Application" configuration does not automatically start bundles, therefore it is fundamental that some of the default configurations are changed to assure the core bundles are started before the GUI is presented. Figure 3.20 shows the options that should be selected for the plug-ins in an Eclipse Application launch configuration.

It is important to make sure the core plug-ins are started before the GUI is presented to the user. Also, under "Target Platform" it is important to make sure ds, log and util are automatically started with default level 2, as seen in Figure 3.21.

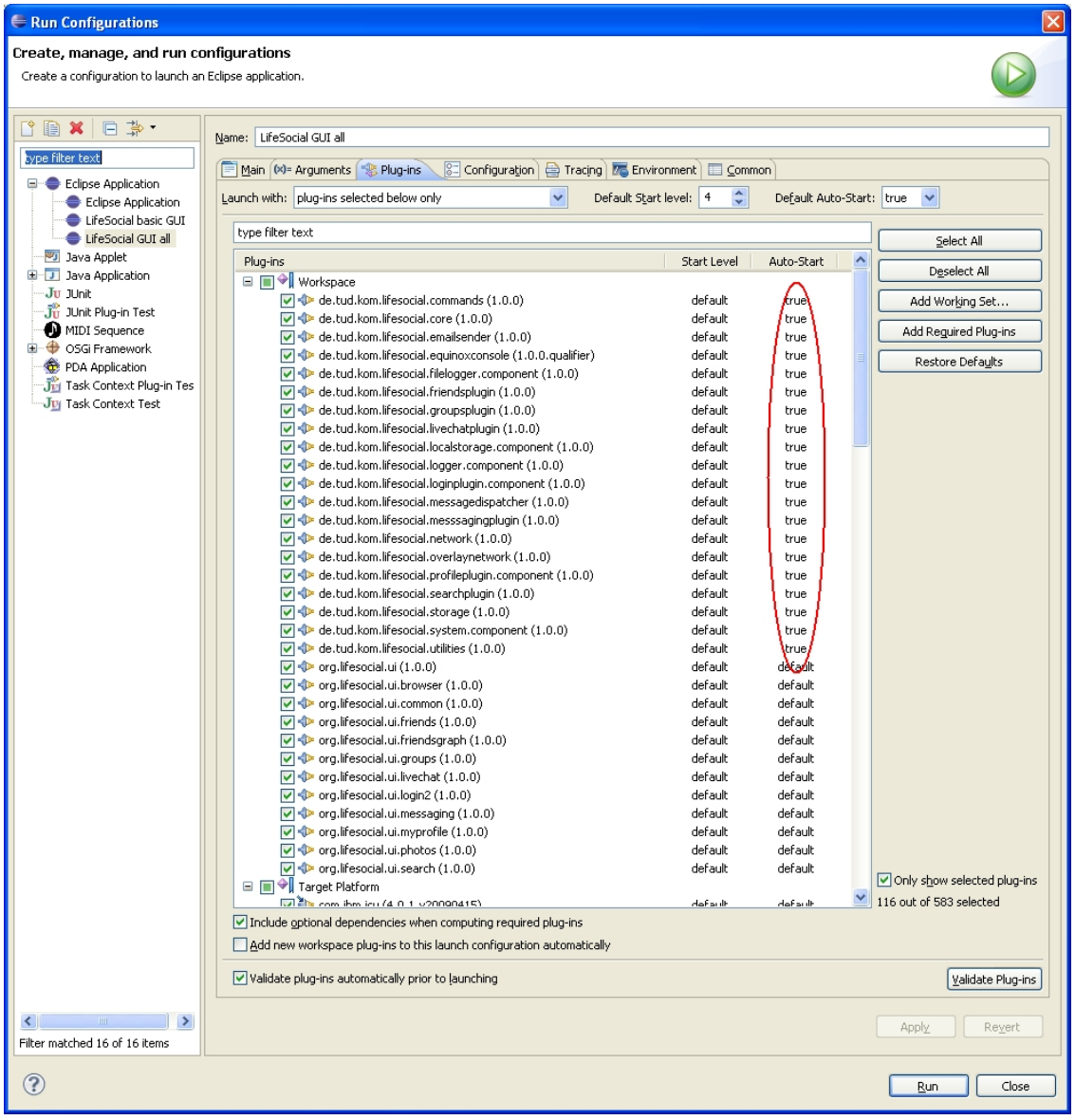


Figure 3.20: Eclipse Application Run Configuration for LifeSocial (GUI)

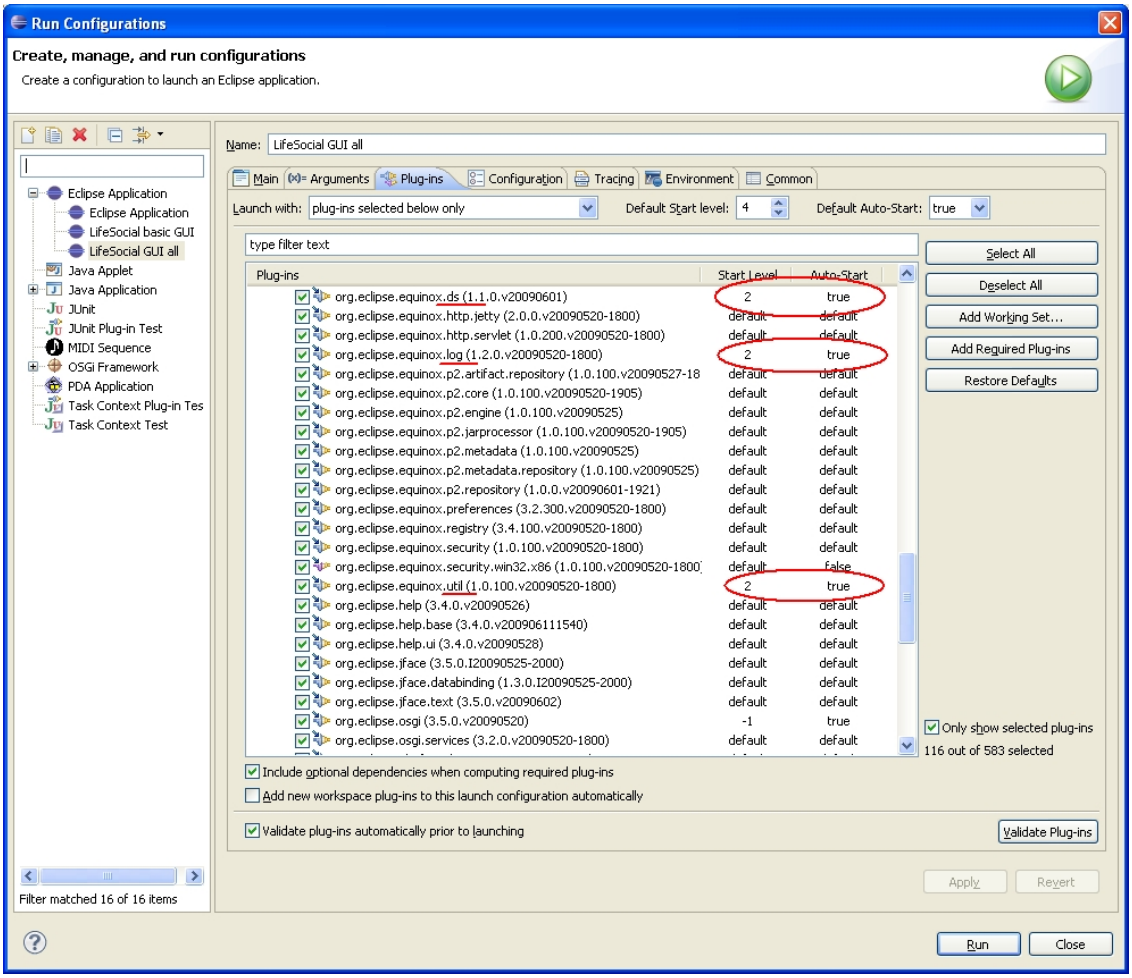


Figure 3.21: Eclipse Application Run Configuration for LifeSocial (GUI)

Additionally, as noted in section 2.3, it is useful to add the `-console` and `-consoleLOG` options to the arguments of the launch configuration (Tab "Arguments" -> "Program arguments").

A MyPlugin Example

This appendix contains an example of a simple plug-in implementation for the LifeSocial Platform. It is also available in the KOM Developer's Repositories [Lab09a] under `OSGI_Examples_MyPlugin`.

```
package api;

import data.MyPluginItem;
import de.tud.kom.lifesocial.core.commoninterfaces.Plugin;

/** @author Luciana Alvite
 * @created 26.07.2009
 */

public interface IMyPlugin extends Plugin {

  // this is one of my commands
  public String myCommand1();

  // this is another of my commands
  public int myCommand2(String myParameter, String myParameter2);

  // returns my item
  public MyPluginItem getMyPluginItem(String myParameter);

  /** more commands here **/
}
```

Listing A.1: IMyPlugin

```
2 package component;
  import api.IMyPlugin;
4 import data.MyPluginItem;

6 /**
   * @author Luciana Alvite
8  * @created 26.07.2009
   */
10 public class MyPluginComponent implements IMyPlugin {

12   @Override
     public String myCommand1() {
14     // TODO the implementation of the command goes here
       int i = this.doSomething();
16     // do something with the int
       // returns a String
18     return null;
     }
20
     @Override
22   public int myCommand2(String myParameter, String myParameter2) {
       // TODO the implementation of the command goes here
24     return 0;
     }
26
     @Override
28   public MyPluginItem getMyPluginItem(String myParameter) {

30     // do something using the overlay network (network, storage) to obtain
       // my item
32     /****** ........ ******/

34     // TODO should return the obtained item
       return null;
36   }

38   @Override
     public String getPluginId() {
40     return PluginPropertiesContainer.PluginID;
     }
42   /*** PRIVATE METHODS ***/
     private int doSomething (){
44   //does something
     }
46 }
```

Listing A.2: MyPluginComponent

```
package data;

import de.tud.kom.lifesocial.core.data.impl.SharedItem;
import de.tud.kom.lifesocial.core.data.interfaces.Header;
import de.tud.kom.lifesocial.core.data.interfaces.StorageKey;

/**
 * @class MyPluginItem
 * @author Luciana Alvite
 * @created 03.08.2009
 */
public class MyPluginItem extends SharedItem {

  private static final long serialVersionUID = -7908436216462985959L;
  private MyPluginItemPayload payload;

  /**
   * Standard constructor
   */
  public MyPluginItem(StorageKey storageKey, Header header,
      MyPluginItemPayload payload) {
    super(storageKey, header);
    this.payload = payload;
  }

  /**
   * @return the payload
   */
  public MyPluginItemPayload getPayload() {
    return payload;
  }

  /**
   * @param payload
   *            the payload to set
   */
  public void setPayload(MyPluginItemPayload payload) {
    this.payload = payload;
  }

  /**
   * Returns a {@link String} representation of the item.
   */
  @Override
  public String toString() {
    return this.storageKey.toString() + this.header.toString()
        + this.payload.toString();
  }

}
```

Listing A.3: MyPluginItem

```
package data;

import java.io.Serializable;

import de.tud.kom.lifesocial.core.data.interfaces.Payload;

/**
 * @class MyPluginItemPayload
 * @author Luciana Alvite
 * @created 03.08.2009
 */
public class MyPluginItemPayload implements Payload, Serializable {

  private static final long serialVersionUID = 126098134721871659 9L;

  private String myVariable;

  /**
   * Returns a {@link String} representation of the object.
   */
  @Override
  public String toString() {
    String result = "";
    result = result.concat("\nMyPlugin item payload:" + "\n");
    result = result.concat("MyPlugin variable: " + this.myVariable + "\n");
    return result;
  }
}
```

Listing A.4: MyPluginItemPayload

```
package commands;

import utils.PluginPropertiesContainer;
import data.MyPluginItem;

/**
 * @author Luciana Alvite
 * @created 26.07.2009
 *
 */
public class MyPluginMyCommand extends AbstractMyPluginCommandHandler {

  /**
   *
   */
  public MyPluginMyCommand() {
    // name the command will be known in the platform
    this.command = PluginPropertiesContainer.CommandGetMyPluginItem;
    this.help = "<myParameter1> - Explain how to use the command here";
  }

  @Override
  public Object executeCommand(String arg) {

    // EXAMPLE of parameter validity checking
    String myParameter1 = arg;

    if (!(this.checkValidArg(myParameter1)))
      return PluginPropertiesContainer.ParameterMissingMsg;

    // calls method on component to obtain a MyPluginItem
    MyPluginItem result = this.myPlugin.getMyPluginItem(myParameter1);
    // returns the obtained item
    return result;
  }

  @Override
  public String executeCommandToString(String arg) {
    // EXAMPLE of parameter validity checking
    String myParameter1 = arg;

    if (!(this.checkValidArg(arg)))
      return PluginPropertiesContainer.ParameterMissingMsg;

    // calls method on component to obtain a MyPluginItem
    MyPluginItem result = this.myPlugin.getMyPluginItem(myParameter1);
    // returns the obtained item
    return result.toString();
  }
}
```

Listing A.5: MyPluginMyCommand

```
package commands;

import utils.PluginPropertiesContainer;
import de.tud.kom.lifesocial.core.commands.AbstractCommands;

/**
 * @author Luciana Alvite
 * @created 26.07.2009
 */
public class MyPluginCommands extends AbstractCommands {

  public MyPluginCommands() {
    this.pluginId = PluginPropertiesContainer.PluginID;
  }
}
```

Listing A.6: MyPluginCommands

```
1  package commands;
2
3  import api.IMyPlugin;
   import de.tud.kom.lifesocial.core.commands.AbstractCommandHandler;
5
   /**
7   * @author Luciana Alvite
    * @created 26.07.2009
9   */
   public abstract class AbstractMyPluginCommandHandler extends
11      AbstractCommandHandler {
12
13    protected IMyPlugin myPlugin = null;
14
15    /***
16     * if all command handlers for the plugin depend on a certain plugin,
17     * declare the binding methods here.
18     *
19     * Usually the command handlers depend AT LEAST on the plugin it represents
20     * (MyPlugin here)
21     *
22     * The commands will then only be available when the corresponding plugin is
23     * available (which makes sense in this case)
24     *
25     */
26
27    /*********** BINDING METHODS **************/
28    public synchronized void setMyLoginPlugin(IMyPlugin myPlugin) {
29      this.myPlugin = myPlugin;
30    }
31
32    public synchronized void unsetMyLoginPlugin(IMyPlugin myPlugin) {
33      this.myPlugin = null;
34    }
35 }
```

Listing A.7: AbstractMyPluginCommandHandler

```
/**
 *
 * @class PluginPropertiesContainer
 * @author Luciana Alvite
 * @created 30.07.2009
 */
package utils;

import de.tud.kom.lifesocial.utilities.PropertiesContainer;

/**
 * @author Luciana Alvite
 * @created 30.07.2009
 */
public class PluginPropertiesContainer extends PropertiesContainer {

  /**
   * GENERAL FIELDS, WILL BE AVAILABLE TO ALL PLUGINS!!!
   *
   * @see de.tud.kom.lifesocial.utilities.PropertiesContainer)
   **/

  /******** PLUGIN SPECIFIC FIELDS ********/
  /*** plugin ID ***/
  public static final String PluginID = "MyPlugin";

  /*** Plug-in Commands ***/
  public static final String CommandMyCommand1 = PluginPropertiesContainer.PluginID
      .concat(".").concat("myCommand1");
  public static final String CommandMyCommand2 = PluginPropertiesContainer.PluginID
      .concat(".").concat("myCommand2");
  public static final String CommandGetMyPluginItem = PluginPropertiesContainer.
     PluginID
      .concat(".").concat("getMyPluginItem");

  /*** Return Status ***/
  public static final Integer myCommand1Success = 1;
  public static final Integer myCommand1Failure = 2;
  public static final Integer myCommand2Success = 1;
  public static final Integer myCommand2Failure = 2;
  public static final Integer getMyPluginItemSuccess = 1;
  public static final Integer getMyPluginItemFailure = 2;

  /*** Return Status Messages ***/
  public static final String getMyPluginItemFailureMsg = " Error obtaining
     MyPluginItem";

}
```

Listing A.8: PluginPropertiesContainer

```
1
   //the local variable for the service reference
3  protected IMyPlugin myPlugin = null;

5  /********** BINDING METHODS **************/
   public synchronized void setMyPlugin(IMyPlugin myPlugin) {
7    this.myPlugin = myPlugin;
   }
9
   public synchronized void unsetMyPlugin(IMyPlugin myPlugin) {
11   this.myPlugin = null;
   }
```

Listing A.9: Example of Service Binding

```
2 /**
   * @author Luciana Alvite
4  * @created 26.07.2009
   */
6 public class MyPluginComponent implements IMyPlugin {

8  //the local variable that will hold the service reference
   private IMessageDispatcher messageDispatcher = null;
10
   /******* BINDING METHODS *******/
12 public synchronized void setMessageDispatcher(IMessageDispatcher md) {
     this.messageDispatcher = md;
14 }

16 public synchronized void unsetMessageDispatcher(IMessageDispatcher md) {
     this.messageDispatcher = null;
18 }
}
```

Listing A.10: Binding Methods - MyPluginComponent

```
/**
 * @author Luciana Alvite
 * @created 26.07.2009
 */
public abstract class AbstractMyPluginCommandHandler extends
    AbstractCommandHandler {

  //the local variables that will hold the service references
  protected IMyPlugin myPlugin = null;
  protected ILoginPlugin login = null;


  /********** BINDING METHODS **************/
  public synchronized void setMyPlugin(IMyPlugin myPlugin) {
    this.myPlugin = myPlugin;
  }

  public synchronized void unsetMyPlugin(IMyPlugin myPlugin) {
    this.myPlugin = null;
  }

  public synchronized void setLoginPlugin(ILoginPlugin login) {
    this.login = login; // don't forget to declare a local variable for the
    // plugin!
  }

  public synchronized void unsetLoginPlugin(ILoginPlugin login) {
    this.login = null;
  }
}
```

Listing A.11: Binding Methods - AbstractMyPluginCommandHandler

```
Service-Component: OSGI-INF/myplugincomponent.xml,
 OSGI-INF/myplugincommands.xml,
 OSGI-INF/mypluginmycommandcommand.xml
```

Listing A.12: Manifest File - Service Component Declaration

```

/********** BINDING METHODS **************/

  public synchronized void addCommands(ICommandsComponent command) {
    this.commands = command;
  }

  public synchronized void removeCommands(ICommandsComponent command) {
    this.commands = null;
  }
```

Listing A.13: Service Binding - Commands Component(GUI)

```
2 public static ICommandsComponent getCommands() {
      return commands;
4   }
```

Listing A.14: Activator - getCommands() (GUI)

```
Service-Component: OSGI-INF/loginGUIcomponent.xml
```

Listing A.15: Manifest File - Service Component Declaration (GUI)

```
1
    ICommandsComponent commands = Activator.getCommands();
3
    if (commands == null) {
5     //handle the absence of commands, e.g. display an error message
      return;
7   }

9   //my command calls go here, e.g:
    Object result = commands.execute(PluginPropertiesContainer.
        CommandGetMyPluginItem);
11
    if(result instanceof String) {
13    //display error message
      return;
15  }
    if (result instanceof MyPluginItem) {
17    //correct type received, process
    }
```

Listing A.16: Using the Commands component (GUI)

```
1  /**
   * @project MyPlugin
3  * @package api
   * @class IMyPlugin
5  * @author Luciana Alvite
   * @created 26.07.2009
7  */
   package api;
9
   import data.MyPluginItem;
11 import de.tud.kom.lifesocial.core.commoninterfaces.Plugin;

13 /**
   * @author Luciana Alvite
15 * @created 26.07.2009
   */
17 public interface IMyPlugin extends Plugin {

19   // this is one of my commands
     public String myCommand1();
21
     // this is another of my commands
23   public int myCommand2(String myParameter, String myParameter2);

25   // returns my item
     public MyPluginItem getMyPluginItem(String myParameter);
27
     /** more commands here **/
29 }
```

Listing A.17: IMyPlugin

```
/**
 * @author Luciana Alvite
 * @created 26.07.2009
 */
public class MyPluginComponent implements IMyPlugin {

  // the local variable that will hold the service reference
  private IMessageDispatcher messageDispatcher = null;

  /* @see api.IMyPlugin#myCommand1()    */
  public String myCommand1() {
    // TODO the implementation of the command goes here
    int i = this.doSomething();
    return null;
  }

  /* @see api.IMyPlugin#myCommand2(java.lang.String, java.lang.String)    */
  public int myCommand2(String myParameter, String myParameter2) {
    // TODO the implementation of the command goes here
    return 0;
  }

  /* @see api.IMyPlugin#getMyPluginItem(java.lang.String)  */
  public MyPluginItem getMyPluginItem(String myParameter) {
    // do something using the overlay network (network, storage) to obtain my item
    return null;
  }

  /* @see de.tud.kom.lifesocial.core.commoninterfaces.Plugin#getPluginId()    */
  public String getPluginId() {
    return PluginPropertiesContainer.PluginID;
  }

  /*** PRIVATE METHODS ***/
  private int doSomething() {
    // does something, returns an integer
    return 0;
  }

  /** ******** BINDING METHODS ***********
   * If Plugin depends on any other component, declare the bind/unbind methods
   * here. Here is an example where this plugin depends on the Message Dispatcher
      */

  public synchronized void setMessageDispatcher(IMessageDispatcher md) {
    this.messageDispatcher = md;
  }

  public synchronized void unsetMessageDispatcher(IMessageDispatcher md) {
    this.messageDispatcher = null;
  }
```

Listing A.18: MyPluginComponent

```
2 package commands;

4 import api.IMyPlugin;
  import de.tud.kom.lifesocial.core.commands.AbstractCommandHandler;
6 import de.tud.kom.lifesocial.loginplugin.api.ILoginPlugin;

8 /**
   * @author Luciana Alvite
10  * @created 26.07.2009
   */
12 public abstract class AbstractMyPluginCommandHandler extends
       AbstractCommandHandler {
14
    // the local variables that will hold the service references
16  protected IMyPlugin myPlugin = null;
    protected ILoginPlugin login = null;
18
    /***
20   * If all command handlers for the plugin depend on a certain plugin,
     * declare the binding methods here. Usually the command handlers depend AT LEAST
         on the plugin it
22   * represents (MyPlugin here)
     * The commands will then only be available when the corresponding plugin is
24   * available (which makes sense in this case)
     */
26
    /********** BINDING METHODS **************/
28  public synchronized void setMyPlugin(IMyPlugin myPlugin) {
      this.myPlugin = myPlugin;
30  }

32  public synchronized void unsetMyPlugin(IMyPlugin myPlugin) {
      this.myPlugin = null;
34  }

36  public synchronized void setLoginPlugin(ILoginPlugin login) {
      // don't forget to declare a variable for the service reference!
38    this.login = login;
    }
40
    public synchronized void unsetLoginPlugin(ILoginPlugin login) {
42    this.login = null;
    }
44 }
```

Listing A.19: AbstractMyPluginCommandHandler

```
/**
 * @project MyPlugin
 * @package commands
 * @class MyPluginCommands
 *
 * @author Luciana Alvite
 * @created 26.07.2009
 */
package commands;

import utils.PluginPropertiesContainer;
import de.tud.kom.lifesocial.core.commands.AbstractCommands;

/**
 * @author Luciana Alvite
 * @created 26.07.2009
 */
public class MyPluginCommands extends AbstractCommands {

  /**
   *
   */
  public MyPluginCommands() {
    this.pluginId = PluginPropertiesContainer.PluginID;
  }
}
```

Listing A.20: MyPluginCommands

```
package commands;

import utils.PluginPropertiesContainer;
import data.MyPluginItem;

/**
 * @author Luciana Alvite
 * @created 26.07.2009
 *
 */
public class MyPluginMyCommand extends AbstractMyPluginCommandHandler {

  public MyPluginMyCommand() {
    // name the command will be known in the platform
    this.command = PluginPropertiesContainer.CommandGetMyPluginItem;
    this.help = "<myParameter1> - Explain how to use the command here";
  }

  @Override
  public Object executeCommand(String arg) {

    // EXAMPLE of parameter validity checking
    String myParameter1 = arg;

    if (!(this.checkValidArg(myParameter1)))
      return PluginPropertiesContainer.ParameterMissingMsg;

    // calls method on component to obtain a MyPluginItem
    MyPluginItem result = this.myPlugin.getMyPluginItem(myParameter1);
    // returns the obtained item
    return result;
  }

  @Override
  public String executeCommandToString(String arg) {
    // EXAMPLE of parameter validity checking
    String myParameter1 = arg;

    if (!(this.checkValidArg(arg)))
      return PluginPropertiesContainer.ParameterMissingMsg;

    // calls method on component to obtain a MyPluginItem
    MyPluginItem result = this.myPlugin.getMyPluginItem(myParameter1);
    // returns the obtained item
    return result.toString();
  }
}
```

Listing A.21: MyPluginMyCommand

```
1
package data;
3
import de.tud.kom.lifesocial.core.data.impl.SharedItem;
5 import de.tud.kom.lifesocial.core.data.interfaces.Header;
import de.tud.kom.lifesocial.core.data.interfaces.StorageKey;
7
/**
9  * @class MyPluginItem
 * @author Luciana Alvite
11 * @created 03.08.2009
 */
13 public class MyPluginItem extends SharedItem {

15   private static final long serialVersionUID = -7908436216462985959L;
   private MyPluginItemPayload payload;
17
   /**
19  * Standard constructor
    */
21  public MyPluginItem(StorageKey storageKey, Header header,
        MyPluginItemPayload payload) {
23    super(storageKey, header);
      this.payload = payload;
25  }

27  /**
    * @return the payload
29  */
   public MyPluginItemPayload getPayload() {
31    return payload;
   }
33
   /**
35  * @param payload
    *            the payload to set
37  */
   public void setPayload(MyPluginItemPayload payload) {
39    this.payload = payload;
   }
41
   /**
43  * Returns a {@link String} representation of the item.
    */
45  @Override
   public String toString() {
47    return this.storageKey.toString() + this.header.toString()
          + this.payload.toString();
49  }
}
```

Listing A.22: MyPluginItem

```
/**
 * @project MyPlugin
 * @package data
 * @class MyPluginItemPayload
 * @author Luciana Alvite
 * @created 03.08.2009
 */
package data;

import java.io.Serializable;

import de.tud.kom.lifesocial.core.data.interfaces.Payload;

/**
 * @class MyPluginItemPayload
 * @author Luciana Alvite
 * @created 03.08.2009
 */
public class MyPluginItemPayload implements Payload, Serializable {

  /**
   *
   */
  private static final long serialVersionUID = 126098134721871659 9L;

  private String myVariable;

  /**
   * Returns a {@link String} representation of the object.
   */
  @Override
  public String toString() {
    String result = "";
    result = result.concat("\nMyPlugin item payload:" + "\n");
    result = result.concat("MyPlugin variable: " + this.myVariable + "\n");
    return result;
  }
}
```

Listing A.23: MyPluginItemPayload

```
/**
 *
 * @class PluginPropertiesContainer
 * @author Luciana Alvite
 * @created 30.07.2009
 */
package utils;

import de.tud.kom.lifesocial.utilities.PropertiesContainer;

/**
 * @author Luciana Alvite
 * @created 30.07.2009
 */
public class PluginPropertiesContainer extends PropertiesContainer {

  /**
   * GENERAL FIELDS, WILL BE AVAILABLE TO ALL PLUGINS!!!
   *
   * @see de.tud.kom.lifesocial.utilities.PropertiesContainer)
   **/

  /******** PLUGIN SPECIFIC FIELDS ********/
  /*** plugin ID ***/
  public static final String PluginID = "MyPlugin";

  /*** Plug-in Commands ***/
  public static final String CommandMyCommand1 = PluginPropertiesContainer.PluginID
      .concat(".").concat("myCommand1");
  public static final String CommandMyCommand2 = PluginPropertiesContainer.PluginID
      .concat(".").concat("myCommand2");
  public static final String CommandGetMyPluginItem = PluginPropertiesContainer.
      PluginID
      .concat(".").concat("getMyPluginItem");

  /*** Return Status ***/
  public static final Integer myCommand1Success = 1;
  public static final Integer myCommand1Failure = 2;
  public static final Integer myCommand2Success = 1;
  public static final Integer myCommand2Failure = 2;
  public static final Integer getMyPluginItemSuccess = 1;
  public static final Integer getMyPluginItemFailure = 2;

  /*** Return Status Messages ***/
  public static final String getMyPluginItemFailureMsg = " Error obtaining
      MyPluginItem";
}
```

Listing A.24: PluginPropertiesContainer

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: MyPlugin
Bundle-SymbolicName: MyPlugin
Bundle-Version: 1.0.0
Bundle-Vendor: Luciana Alvite
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: de.tud.kom.lifesocial.core.commands,
 de.tud.kom.lifesocial.core.commoninterfaces,
 de.tud.kom.lifesocial.core.data.impl,
 de.tud.kom.lifesocial.core.data.interfaces,
 de.tud.kom.lifesocial.loginplugin.api,
 de.tud.kom.lifesocial.messagedispatcher.api,
 de.tud.kom.lifesocial.utilities
Service-Component: OSGI-INF/myplugincomponent.xml,
 OSGI-INF/myplugincommands.xml,
 OSGI-INF/mypluginmycommandcommand.xml
Export-Package: api
```

Listing A.25: MANIFEST File

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="
    myplugincomponent">
   <implementation class="component.MyPluginComponent"/>
   <service>
      <provide interface="api.IMyPlugin"/>
   </service>
   <reference bind="setMessageDispatcher"
     cardinality="1..1"
     interface="de.tud.kom.lifesocial.messagedispatcher.api.IMessageDispatcher"
     name="MessageDispatcher"
     policy="dynamic"
     unbind="unsetMessageDispatcher"/>
</scr:component>
```

Listing A.26: myplugincomponent.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="
    myplugincommands">
   <implementation class="commands.MyPluginCommands"/>
   <service>
      <provide interface="de.tud.kom.lifesocial.core.commands.ICommands"/>
   </service>
   <reference bind="addCommandHandler"
      cardinality="1..n"
      interface="commands.AbstractMyPluginCommandHandler"
      name="CommandHandlers"
      policy="dynamic"
      unbind="removeCommandHandler"/>
</scr:component>
```

Listing A.27: myplugincommands.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="
    mypluginmycommandcommand.xml">
   <implementation class="commands.MyPluginMyCommand"/>
   <service>
      <provide interface="commands.AbstractMyPluginCommandHandler"/>
   </service>
   <reference bind="setMyPlugin"
     cardinality="1..1"
     interface="api.IMyPlugin"
     name="MyPlugin"
     policy="dynamic"
     unbind="unsetMyPlugin"/>
   <reference bind="setLoginPlugin"
     cardinality="1..1"
     interface="de.tud.kom.lifesocial.loginplugin.api.ILoginPlugin"
     name="ILoginPlugin"
     policy="dynamic"
     unbind="unsetLoginPlugin"/>
</scr:component>
```

Listing A.28: mypluginmycommandcommand.xml

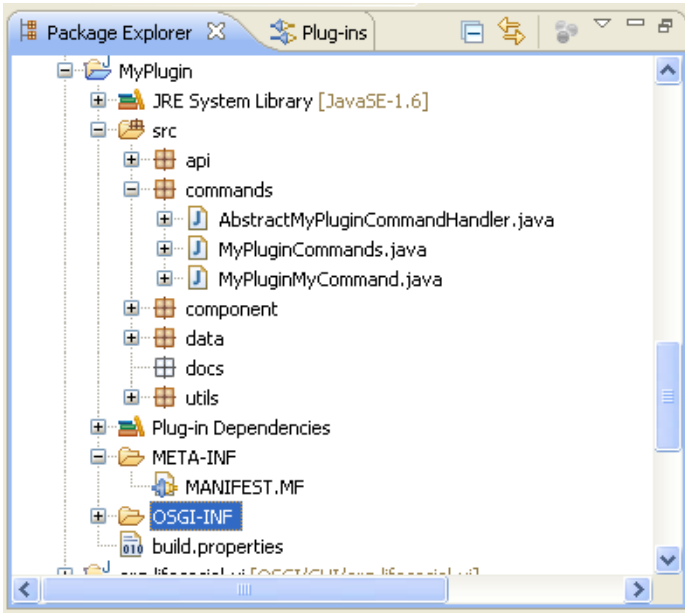


Figure A.1: OSGI-INF Folder

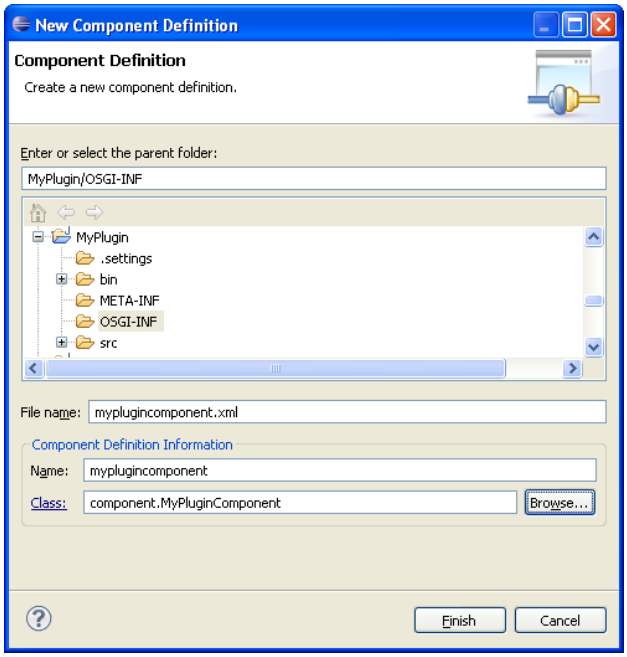


Figure A.2: Component Definition - The main plug-in component

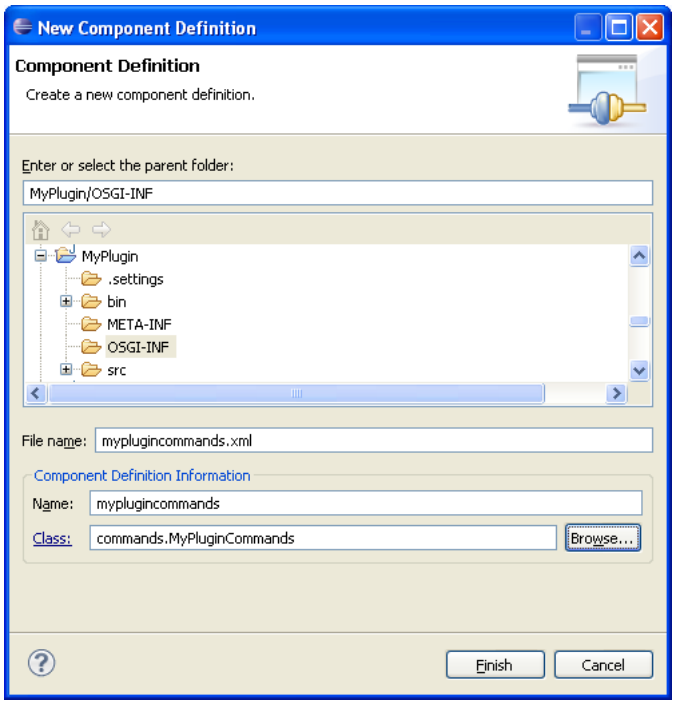


Figure A.3: Component Definition - Commands

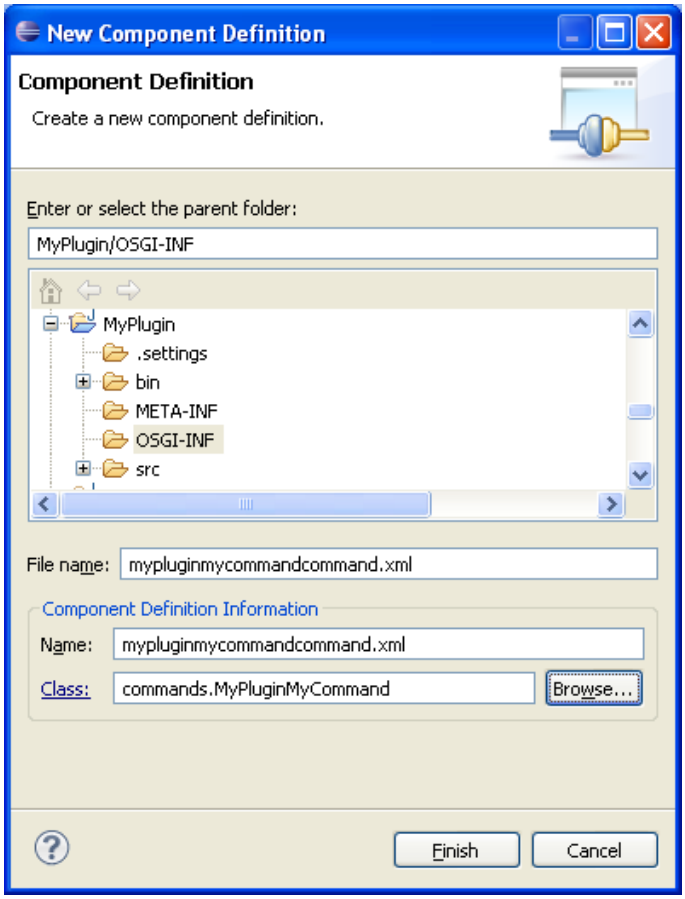


Figure A.4: Component Definition - My Command

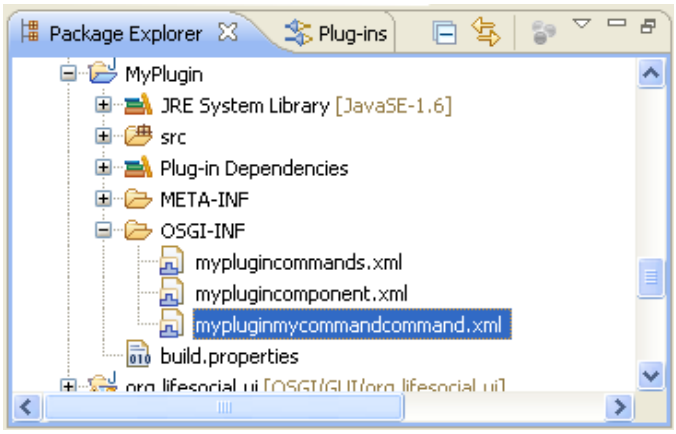


Figure A.5: OSGI-INF folder - all components

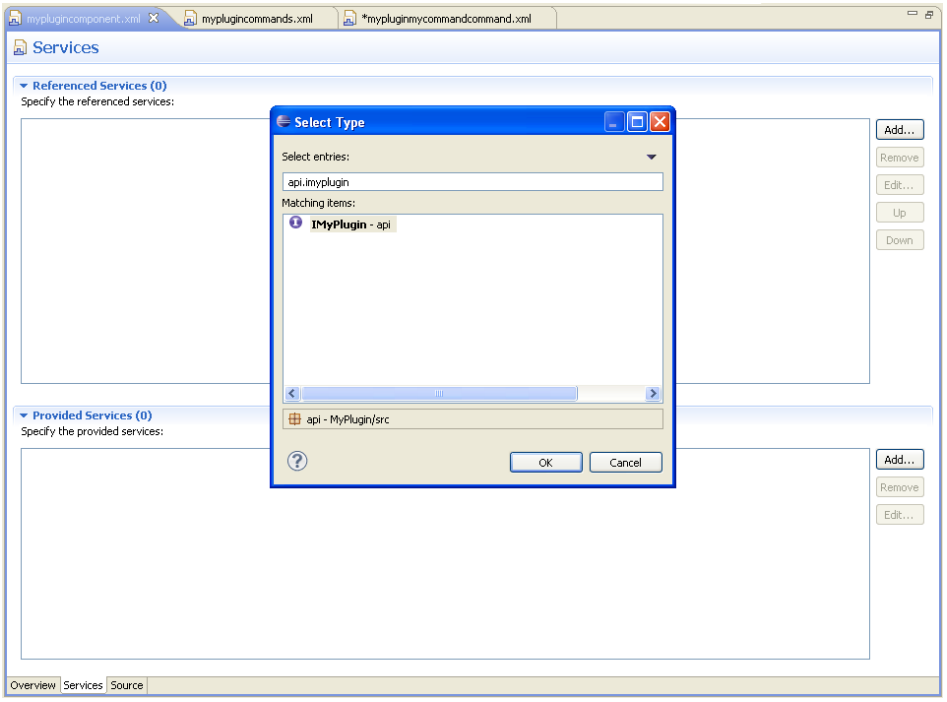


Figure A.6: Component Definition - Exposing the plug-in interface

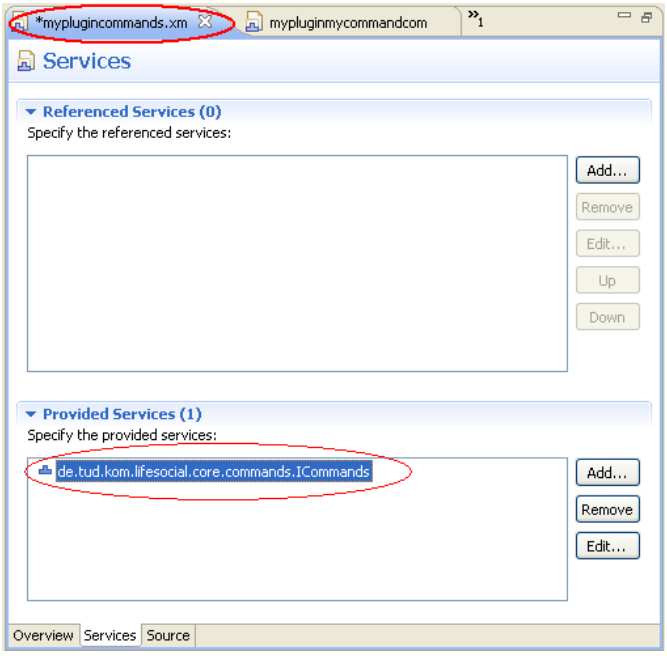


Figure A.7: Component Definition - Exposing the command aggregator

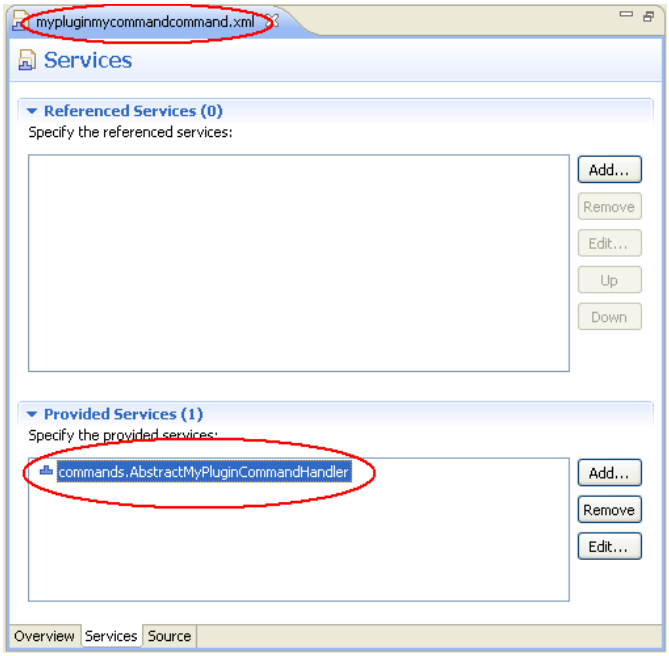


Figure A.8: Component Definition - Exposing a command

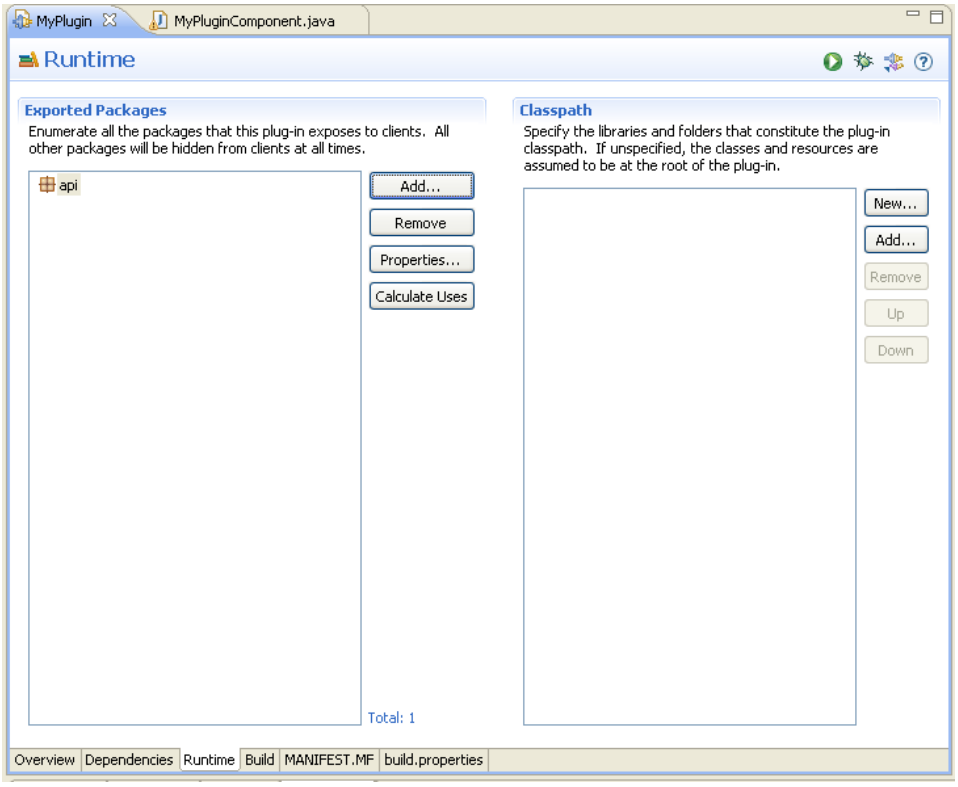


Figure A.9: Manifest File - Exporting a Package

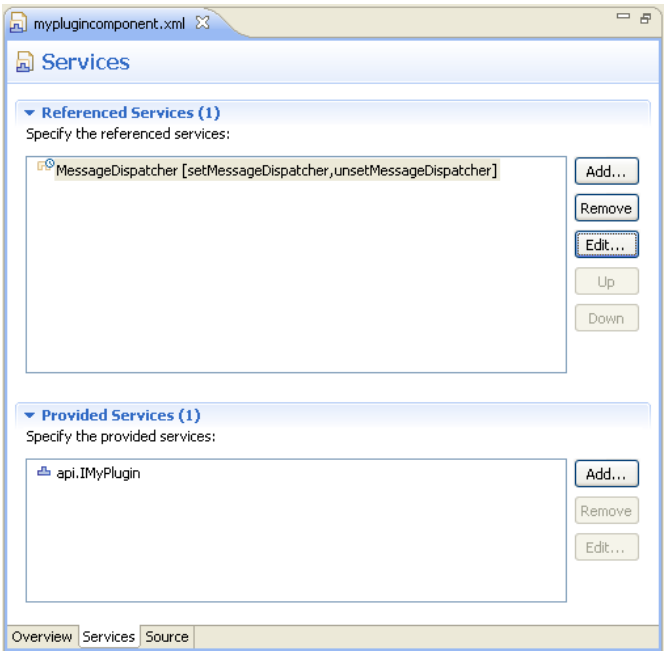


Figure A.10: Component Definition - Declaring a Dependency

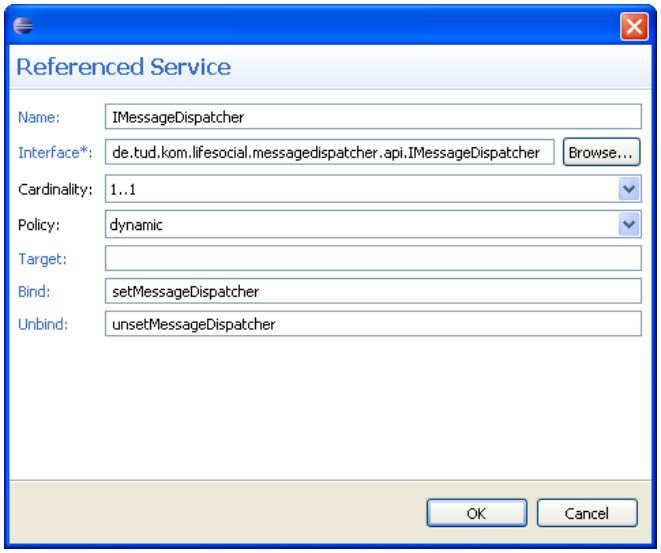


Figure A.11: Component Definition - Dependency Options

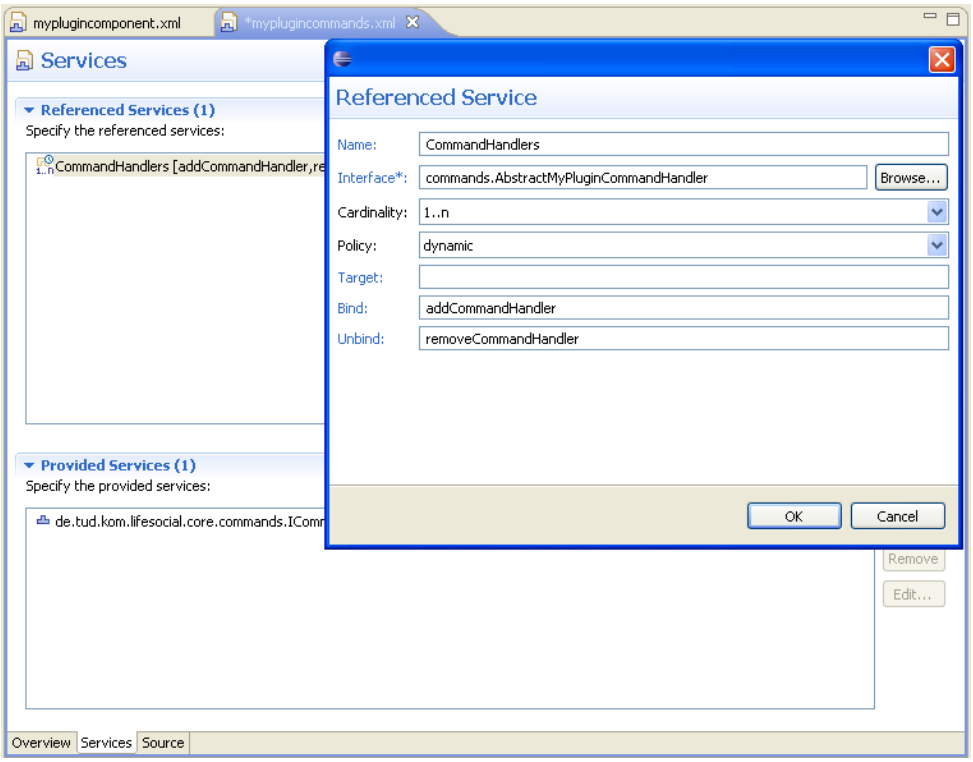


Figure A.12: Component Definition - Command Aggregator Dependencies

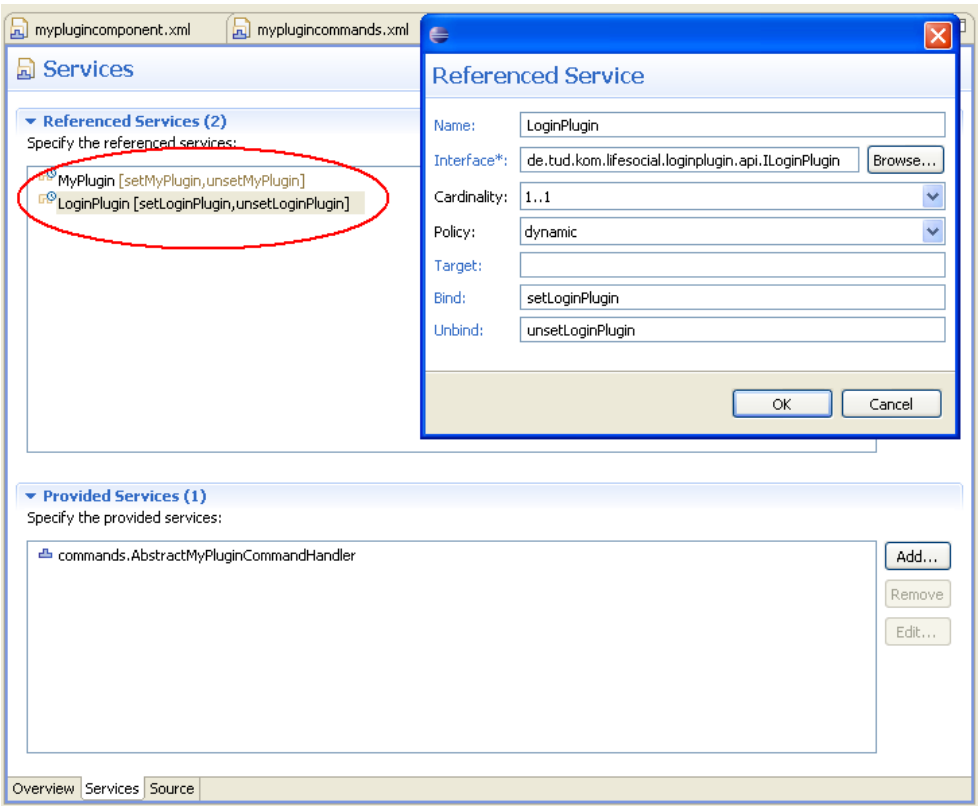


Figure A.13: Component Definition - Command Handlers Dependencies

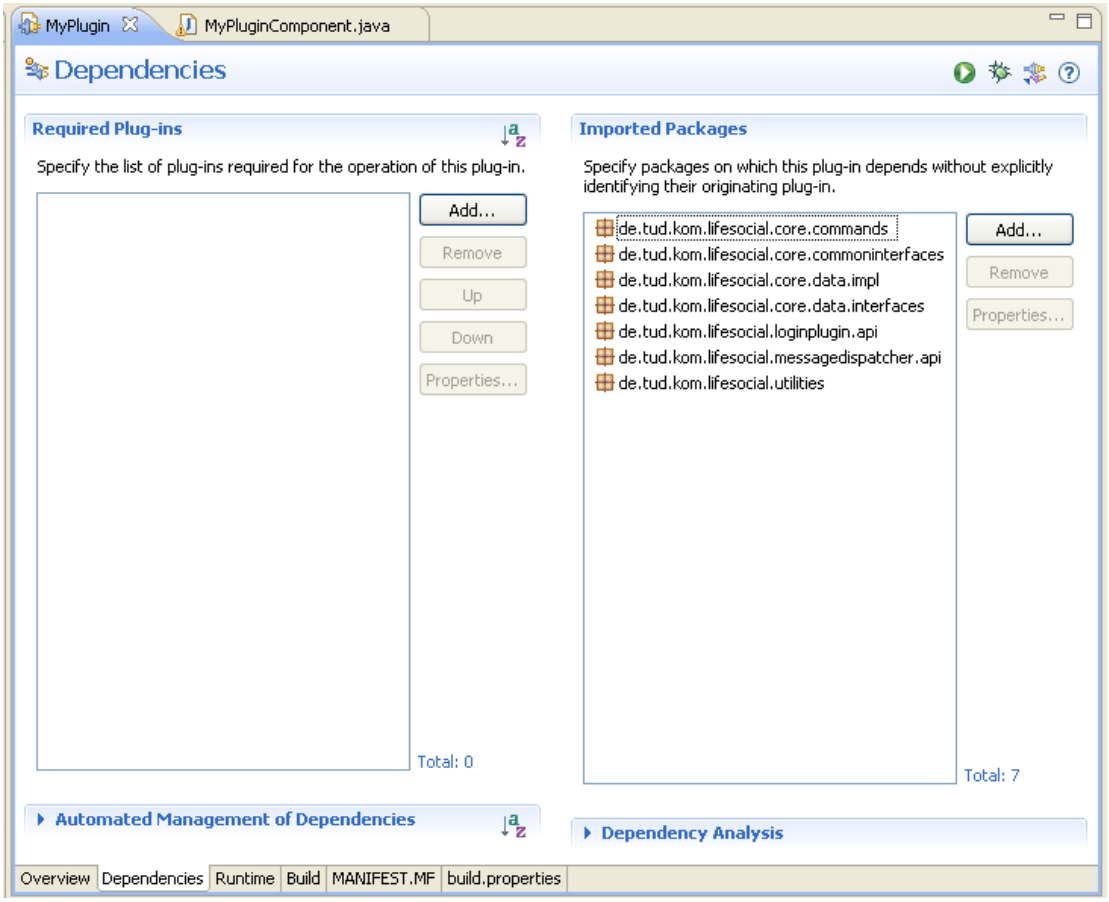


Figure A.14: Manifest File - Imported Packages

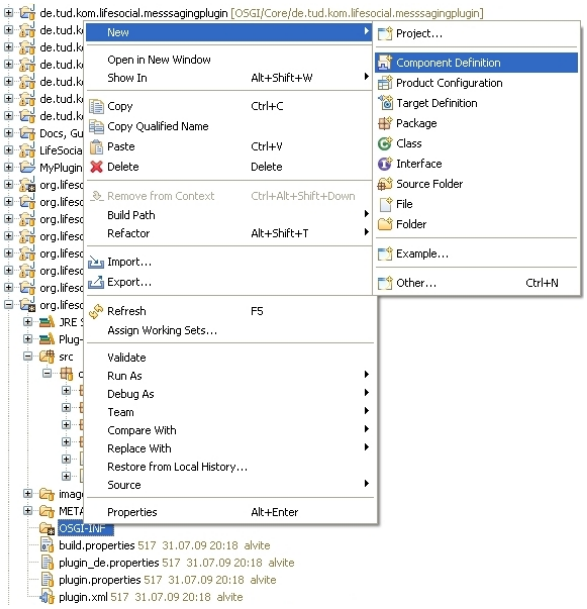


Figure A.15: Creating a Service Component (GUI)

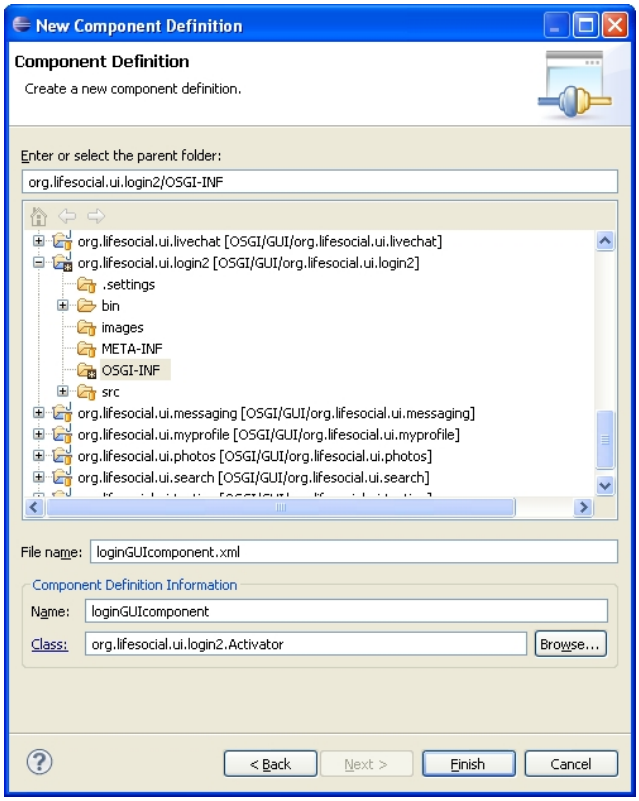


Figure A.16: Creating a Service Component - Component Definition (GUI)

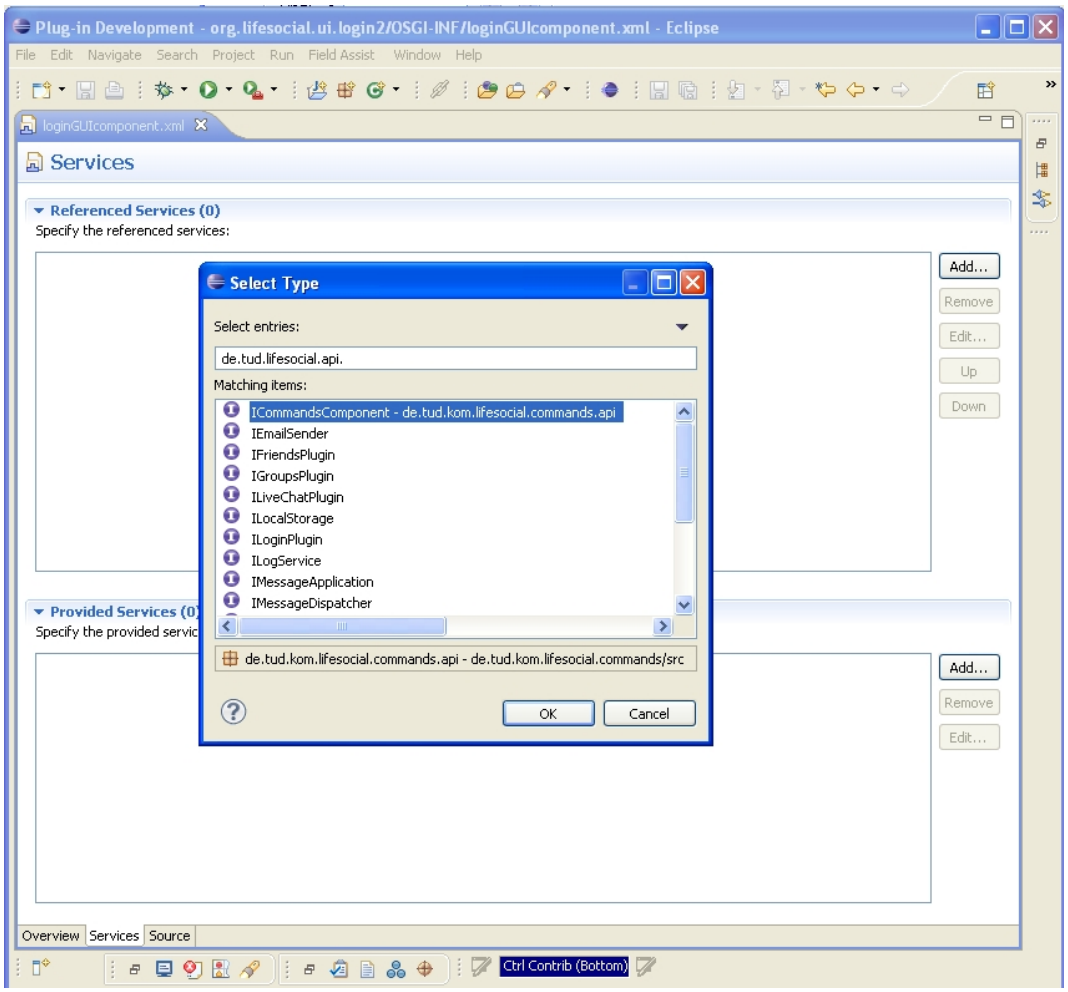


Figure A.17: Service component - Adding a dependency (GUI)

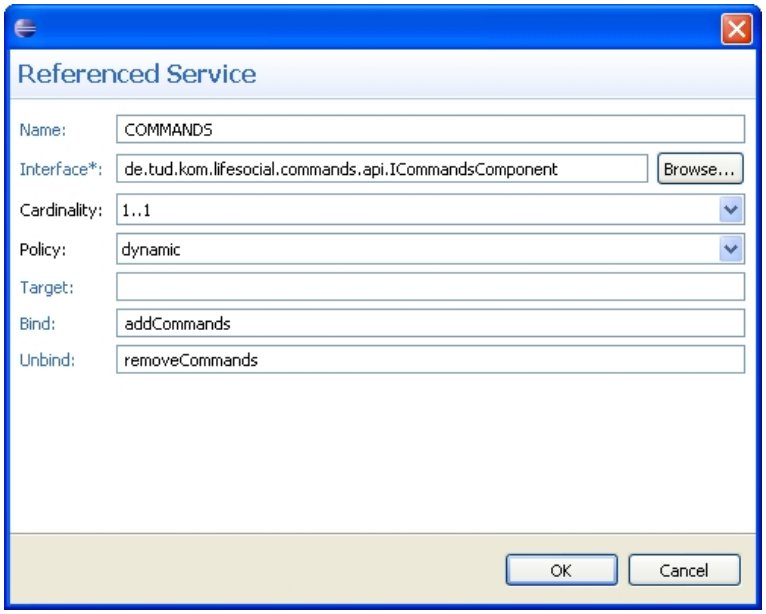


Figure A.18: Service component - Editing a dependency (GUI)

A.3 Run Configurations

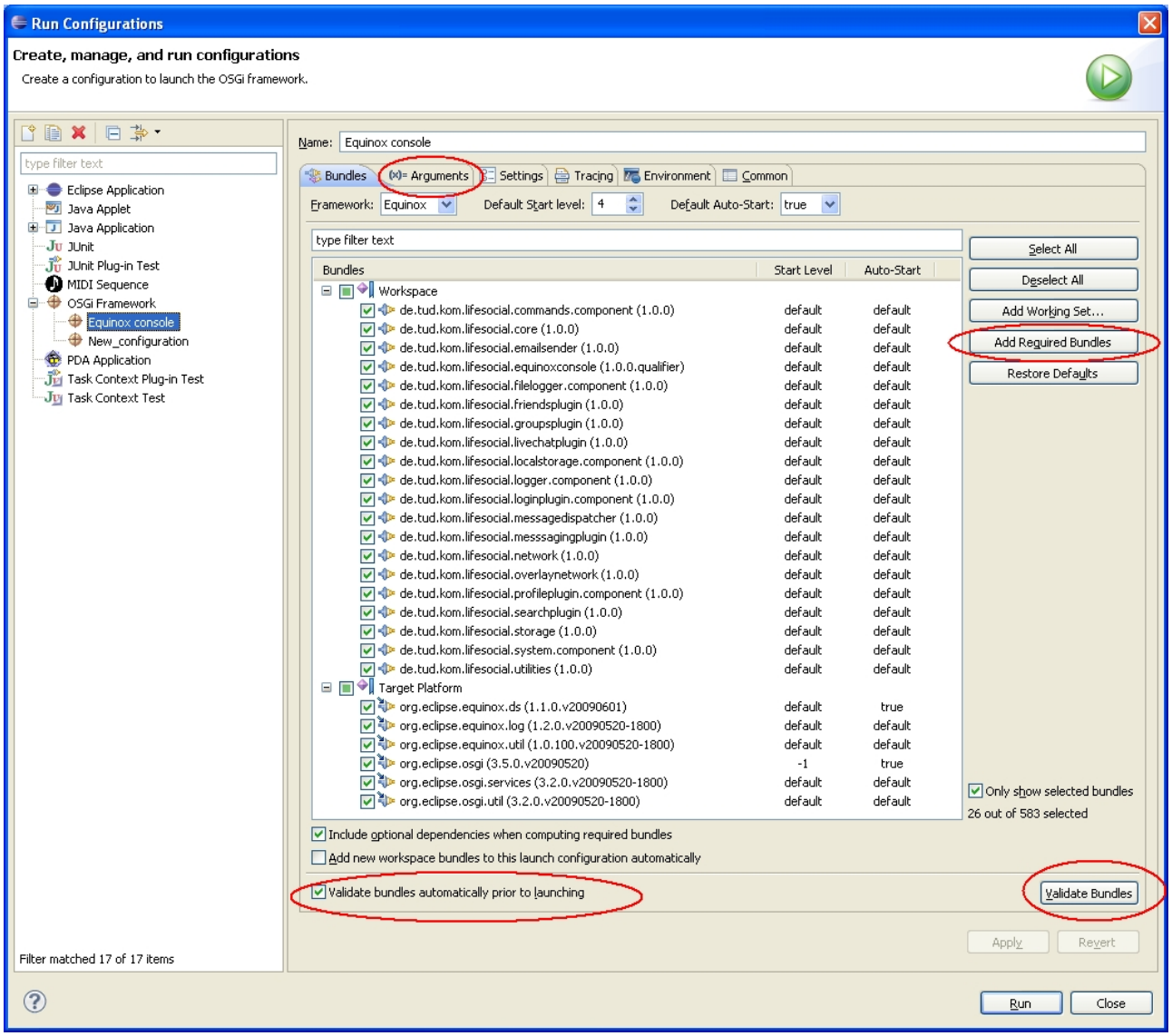


Figure A.19: Run configuration for the Equinox Console Extension in Eclipse

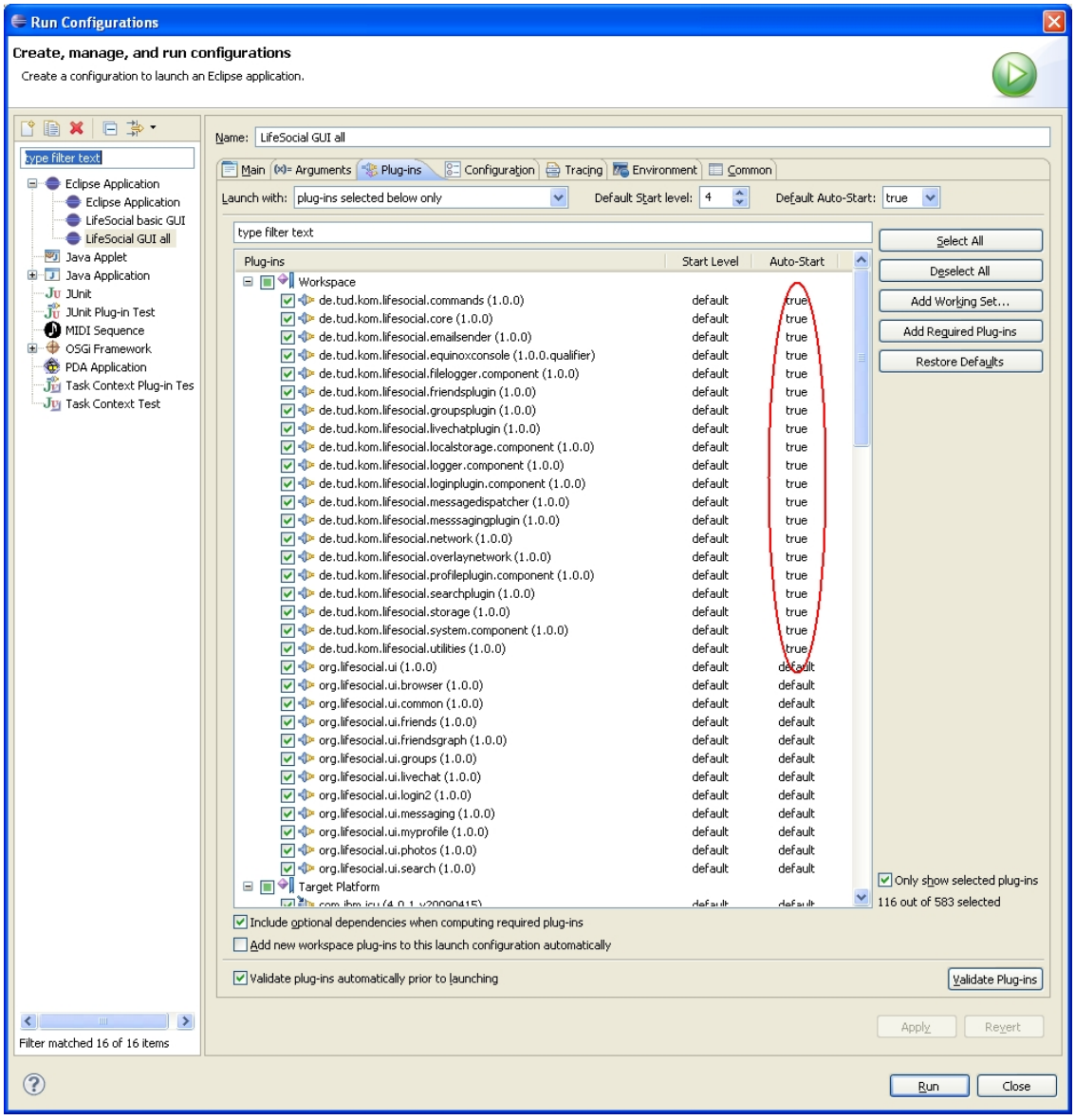


Figure A.20: Eclipse Application Run Configuration for LifeSocial (GUI)

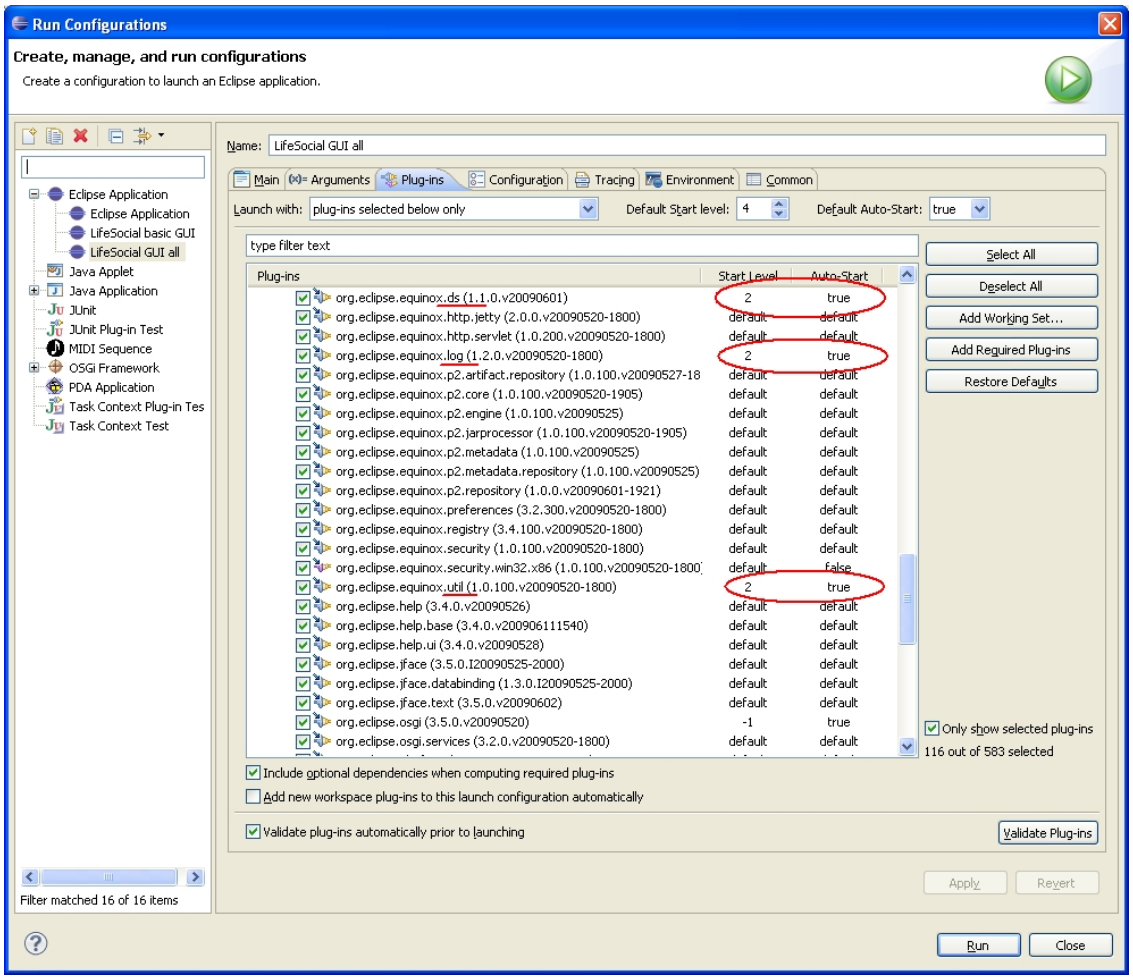


Figure A.21: Eclipse Application Run Configuration for LifeSocial (GUI)

Bibliography

[All07] OSGi Alliance. Osgi service platform release 4 version 4.1 compendium specification, chapter 112, April 2007. Available from: http://www.osgi.org/download/r4v41/r4.cmpn.pdf [cited 2009.08.03]. One citation in page 19.

[Alv09] Luciana Alvite. LifeSocial Commands, July 2009. 2 citations in pages 6 and 34.

[Bar07] Neil Bartlett. Getting started with osgi: Introducing declarative services, June 2007. Available from: http://www.eclipsezone.com/eclipse/forums/t96740.html [cited 2009.07.26]. One citation in page 9.

[Fou09a] Eclipse Foundation. Eclipse for rcp/plug-in developers download, July 2009. Available from: http://eclipse.org/home/categories/index.php?category=equinox [cited 2009.07.26]. One citation in page 5.

[Fou09b] Eclipse Foundation. Equinox download website, July 2009. Available from: http://download.eclipse.org/equinox/ [cited 2009.07.17]. One citation in page 5.

[Lab09a] KOM Multimedia Communications Lab. Kom developer's repositories, July 2009. Available from: https://dev.kom.e-technik.tu-darmstadt.de/ [cited 2009.07.26]. 2 citations in pages 5 and 45.

[Lab09b] KOM Multimedia Communications Lab. Kom's website, July 2009. Available from: http://www.kom.tu-darmstadt.de/en/home/ [cited 2009.07.27]. One citation in page 5.

[Tig09] Tigris.org. Subclipse, July 2009. Available from: http://subclipse.tigris.org/ [cited 2009.07.26]. One citation in page 5.