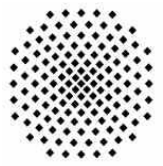

Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D – 70569 Stuttgart

Studienarbeit Nr. 2311

Web-based Application Integration: Advanced Business Process Monitoring in WSO2 Carbon

Jakob Krein

Studiengang:	Informatik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. David Schumm
begonnen am:	25.11.2010
beendet am:	27.05.2011
CR-Klassifikation:	C.2.4, D.2.2, H.4.1, H.5.2, H.5.3

Abstract

Today's business processes tend to get more and more complex and can sometimes have hundreds of activities. Maintaining and monitoring these business processes can get hard for companies' IT experts. Process views support those experts by providing user-defined views of a process. Complexity can be reduced by aggregating activities or leaving them out. It is also possible to highlight parts of a process that are more important than others.

Business Process Illustrator is an open-source, web-based business process monitoring tool, developed at the Institute of Architecture of Application Systems at the University of Stuttgart. This tool uses process views to simplify the monitoring of complex business processes. It has a built-in adapter for the Apache ODE process engine, but it can basically support any type of Workflow Management System.

WSO2 Carbon is a modular open-source middleware platform which hosts a rich set of middleware components encompassing capabilities such as business process management and business activity monitoring. It is extensible and allows the installation of custom components through a technology called OSGi. WSO2 Business Process Server is a WSO2 Carbon-based product that executes business processes defined using the WS-BPEL standard. The WSO2 Process Server is powered by Apache ODE and therefore predestinated for the Business Process Illustrator to be integrated with WSO2 Carbon.

The goal of this student thesis is to provide an open-source integration of the Business Process Illustrator with the WSO2 Carbon platform as an OSGi bundle and to establish a connection to the Business Process Server component.

Table of Contents

Table of Figures

List of Tables

Table of Listings

List of Abbreviations

BPEL	Business Process Execution Language
BPI	Business Process Illustrator
BPS	Business Process Server
IAAS	Institute of Architecture of Application Systems
JSF	JavaServer Faces
JSP	JavaServer Pages
JVM	Java Virtual Machine
OSGi	Open Services Gateway initiative
POJO	Plain Old Java Object
SVG	Scalable Vector Graphics
WfMS	Workflow Management System

1 Introduction

1.1 Task of the Student Thesis

The task of this student thesis is to integrate the open-source, web-based Business Process Illustrator with the WSO2 Carbon framework. This includes providing the Business Process Illustrator as an OSGi bundle and establishing a connection to the Business Process Server component. At the beginning of this work, it was considered to Implement Web service interfaces for integration with the Carbon platform. A more thorough analysis has shown that a successful integration is not depending on those Web service interfaces. They are considered as a “nice-to-have” feature that can be implemented in a future release.

1.2 Structure of the Document

This document is divided into 7 chapters.

Chapter 1 gives a short overview of the task and the structure of this student thesis.

Chapter 2 covers the basics and gives an introduction on the used technologies.

Chapter 3 describes the Business Process Illustrator (BPI) and gives a deeper insight in the underlying architecture. The chapter also focuses on the use of adapters and gives a first overview on the challenges that one has to consider before integrating the BPI with other applications.

Chapter 4 focuses on the WSO2 Carbon framework and the WSO2 Business Process Server. It describes, how components can be developed to extend the framework and what the requirements are, that those components have to meet.

Chapter 5 shows the architecture of the resulting OSGi-based BPI application as a high level view. The resulting solution, as well as the steps that have to be taken is described in this chapter.

Chapter 6 highlights the implementation aspects. This is a low-level perspective on the application.

Chapter 7 finally gives a short outlook and points out some possible improvements.

2 Technologies

This chapter covers the basics and gives an introduction of the used technologies. This includes a short introduction to workflow management, the *Business Process Execution Language (BPEL)* and *Process Views*. The Business Process Illustrator uses a technology called *Scalable Vector Graphics (SVG)* to generate the graph of a monitored process or instance. Another important technology is the *Open Services Gateway initiative (OSGi)* that is used by the WSO2 Carbon platform. The final section of this chapter introduces *Maven* – a tool that can be used to build a Java project and allows a project to be packaged as an OSGi bundle.

2.1 Workflow Management

The following is a collection of definitions of terms and terminologies that are used in context with workflow, and is based on the terminology and glossary defined by the Workflow Management Coalition (WfMC) [1].

2.1.1 Workflow

Workflow is the structured controlling of processes or the automation of a business process, in whole or part [2]. It provides the possibility to conduct business processes by software where multiple collaborators are involved in a certain order. In the context of this student thesis, workflow refers to the technical implementation of a business process.

2.1.2 Workflow Management System (WfMS)

A Workflow Management System (WfMS) is a system to define, create and manage the execution of workflows by using software that runs on one or more workflow engines. The software interprets the process definition and interacts with workflow participants. Where needed, it also invokes applications and IT tools. It also provides functions to administrate and monitor the overall system as well as individual process instances.

2.1.3 Process Model

To allow automated manipulation of a business process by the workflow management system, the business process is represented in a form called a process model. It consists of one or more linked activities and their connections. The process model also holds some attributes that define when to start or terminate a process, as well as some information about the activities, e.g. who is participating, what data is needed and which application has to be invoked.

2.1.4 Process Instance

A process instance is a representation of a single enactment of a process. It is created and managed by the workflow management system according to its process definition. Each instance has its own data associated and can be independently controlled by a workflow management system. It also has an internal state that indicates its progress towards completion.

2.1.5 Process Engine

The process engine provides the run time execution environment for a process instance. It knows how to interpret the process definition and creates instances. These instances are managed by the process engine (e.g. starting, stopping, suspending and resuming of the instance). The engine navigates between activities and creates the work items to be processed.

2.1.6 Deployment

Deployment means to put a process model into production (i.e. make it ready for execution). The corresponding model data is usually translated into a different format, e.g. the workflow management system does not directly support the metamodel of the imported model. Process instances of a process model can be created once it is deployed. If a new version of an already existing process model is deployed, all existing instances of the old version are run according to the process definition that was valid at the time of their instantiation.

2.1.7 Execution Events and States

The status of a process instance or an activity is maintained by workflow management systems as part of their workflow control data. A process instance follows a series of transitions between the various states during its execution. Table 1 and Table 2 show different states for activities and instances defined by the WfMC Reference Model [1].

Status	Description
Inactive	The activity instance has been created but may not have been activated yet (there is no work item for that activity)
Active	One or more work items have been created and assigned for processing
Suspended	The activity instance is suspended and no further work items are started until it is resumed
Completed	The activity has completed and any post-completion system activities are running (e.g. audit logging)

Table 1 – States of activities [1]

Status	Description
Initiated	The process instance has been created, but may not yet be running
Running	The process instance has started execution and one or more activities may be started
Active	One or more activities are started and activity instances exist
Suspended	The process instance is suspended and no further activities are started until it is resumed
Complete	The process instance has completed and any post-completion system activities are in progress (e.g. audit logging)
Terminated	The execution of the process has been stopped (abnormally) due to an error or user request
Archived	The process instance has been placed in an indefinite archive state but may be retrieved for process resumption (typically only supported for long-lived processes)

Table 2 – States of a process instance [1]

An event is an occurrence of a particular condition (external or internal) that causes the workflow management system to take one or more actions, e.g. the arrival of an email may cause the creation of an instance of a specific process model. An event has a trigger (cause) and an associated action (response). The workflow management system can react directly to events, but they may also be monitored by an application that initiates action (e.g. through API calls).

2.1.8 Business Process Monitoring

A workflow management system provides access to the actual state of each workflow. This is referred to as monitoring. The status is usually visualized so one can quickly identify how far the execution has progressed. Different people benefit from monitoring. Administrators can identify problems during a test phase, for example. Monitoring also allows to detect out-of-line situations and to react accordingly (e.g. staff can be reassigned when work piles up). Another user of monitoring is a client who wants to know the status of their order, for example. The Business Process Illustrator is a business process monitoring tool.

2.1.9 Process Views

Business process monitoring is used by different people, e.g. administrators, staff members and clients. All these people need different views or different amounts of information about the same process model, e.g. the management department needs only a quick overview while the IT department wants all details. Process views are usually

used to reduce complexity of a process model. Another usage of process views is in the integration of companies in insourcing or outsourcing scenarios. Sometimes a company needs to provide parts of its processes (private views) as process views (public views) for a business partner [3]. This ensures that the partner receives only the relevant data. The Business Process Illustrator can apply process views to process models.

2.1.10 Business Process Execution Language (BPEL)

BPEL is used to describe workflows as orchestration of Web Services. It is a recursive aggregation model that means: tasks in BPEL processes are Web Services and the process itself is a Web Service again. It is based on XML and originated from the Web Service Flow Language (WSFL) by IBM [4] and XML Business Process Language (XLANG) by Microsoft [5]. In 2002 the two companies released the first version under the name BPEL4WS [6]. The second version was released in 2007 under the name WS-BPEL [7] by the OASIS consortium that took over standardization. It supports primarily automated business processes but BPEL4People [8] is an extension that allows the integration of people.

BPEL distinguishes between executable and abstract process models. The latter can be seen as process views to hide internal details, for example. A BPEL model consists of different main building blocks like *Partner Links*, *Variables*, *Correlation Sets*, *Handlers* and *Activities* (see [7] for a complete reference).

2.2 Scalable Vector Graphics (SVG)

Scalable Vector Graphics (SVG) is based on XML and serves as a description of 2-dimensional vector-graphics, both static and dynamic [9]. It has been developed by the World Wide Web Consortium (W3C). As SVG images and their behavior are defined in XML files, they can be searched, indexed and scripted. Creation and editing of SVG files can be done with any text editor. Almost every modern browser supports SVG either natively or via additional plug-ins.

2.2.1 Structure

The structure of an SVG document is a tree structure of different elements and their attributes (see Listing 1). It starts with the xml-declaration and the document-type-declaration like it is common with all XML files. The start-tag is the <svg>-tag that holds all sub elements. There are three different types of elements.

- Vector graphics that are constructed out of graphical primitives like rectangles and circles
- Raster graphics like bitmaps
- Text of a specific font type and style

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:ev="http://www.w3.org/2001/xml-events"
    version="1.1" baseProfile="full"
    width="400mm" height="300mm">

<!--Content -->


</svg>
```

Listing 1 – Simple SVG document

The corresponding xml-tags for those elements are <rect /> for rectangular areas, <circle /> and <ellipse /> for round objects, <image /> for bitmap images, <text /> for representations of text and many others that can be found in [9]. The most powerful element in SVG is the <path /> element. It defines a list of arbitrary combinations of lines, ellipses, bézier curves with absolute or relative coordinates. Everything that can be drawn by one of the graphical objects can be drawn by the <path />-element, too. The other graphical objects are just for better usability.

2.2.2 SVG in HTML pages

SVG was developed for the World Wide Web. Most web browsers can display SVG without additional plug-ins. Adobe Systems developed a viewer for SVG that is used in different browsers on different platforms but the project is officially discontinued. Microsoft Internet Explorer supports SVG natively since version 9 (current version as of this writing). Other browsers like Opera, Safari, Google Chrome and Mozilla Firefox support SVG natively to a certain degree.

There are different methods to embed SVG into HTML pages. SVG images can be included directly in XHTML pages using XML namespaces. Listing 2 shows the usage of SVG as inline code in an XHTML document.

```
<svg xmlns="http://www.w3.org/2000/svg" width="20" height="20"
    version="1.1">
   <circle cx="10" cy="10" r="10" />
</svg>
```

Listing 2 – SVG inline code in an XHTML document

There are also three different tags that can be used to embed SVG files into HTML pages. Listing 3 to Listing 5 show the usage of the <embed>-, <object>- and <iframe>-tag.

```
<embed src="circle.svg" width="50" height="20" type="image/svg+xml"
    pluginspage="http://www.adobe.com/svg/viewer/install/" />
```

Listing 3 – Usage of the <embed>-tag

```
<object data="circle.svg" width="50" height="20" type="image/svg+xml"
    codebase="http://www.adobe.com/svg/viewer/install/" />
```

Listing 4 – Usage of the <object>-tag

```
<iframe src="circle.svg" width="50" height="20">
</iframe>
```

Listing 5 – Usage of the <iframe>-tag

The <embed>-tag is supported by all major browsers and is recommended by the Adobe SVG Viewer though it cannot be used in XHTML files. The <object>-tag is an HTML4 standard. It is supported by most new browsers but it does not allow scripting in contrast to the <embed>-tag. The <iframe>-tag works in most browsers.

2.3 Open Services Gateway initiative (OSGi)

OSGi is a dynamic module system for Java [10]. The problem today is that a lot of software development consists of adapting existing programs so that they may run in a different environment. This is because programs often use already developed building blocks that have become a standard. One example is the success of open software. The complexity of libraries makes the integration process very difficult and comes with a lot of problems. This is where OSGi supports the developer. Java provides the technology to run programs on different platforms. OSGi provides the technology to construct applications from reusable and collaborative components. One benefit of the service platform is the possibility to install, update, start, stop and uninstall service applications (*Bundles*) both dynamically and controlled at run time. Those independent and modular bundles can run in parallel inside the same *Java Virtual Machine (JVM)* and they can be managed and updated throughout the whole lifecycle. Dependencies between bundles are automatically resolved and an intelligent version management is available.

The origin of OSGi is in embedded systems and that is why it is often used in automobiles, mobile devices and building automation like assisted living and facility management. A famous example of the usage of OSGi is the *Eclipse IDE*. Eclipse uses the Equinox OSGi framework and since version three of Eclipse, every plug-in is an OSGi bundle. As of this writing the current version of the OSGi specification is 4.3.

2.3.1 Classification

Figure 1 shows a classification of layers of a typical OSGi architecture. In this context “server” usually means an embedded system, not necessarily an Enterprise Server or Desktop Client.

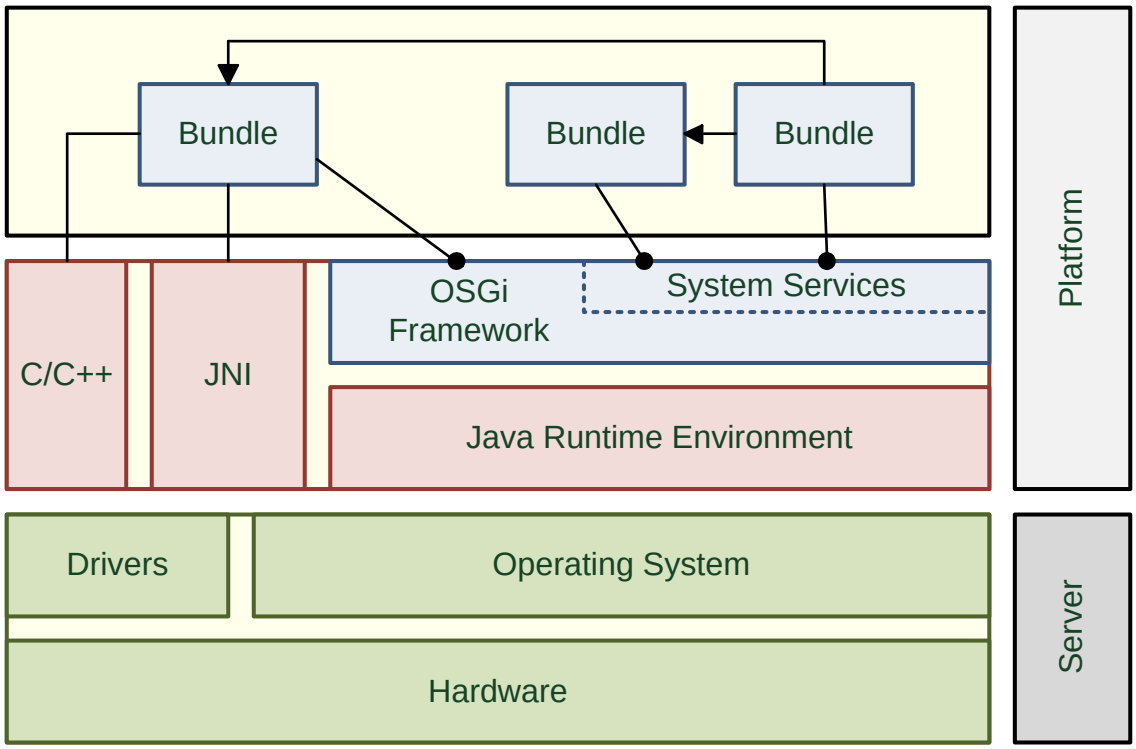

Figure 1 – OSGi Layer (adapted from [11])

2.3.2 Framework

An OSGi framework is an open, modular and scalable service delivery platform on a Java basis. It provides a standardized environment to applications (bundles). It is a component model with a service registry but the term “service” means nothing more than an interface and is not to be mistaken for the term service in a *Service Oriented Architecture (SOA)*, though OSGi can be used as a fundamental component model for a SOA. The OSGi Alliance specifies only the execution environment, the API and the test cases for third party OSGi implementations. A reference implementation of an OSGi framework is provided by the OSGi Alliance but it is not intended for productive use. Figure 2 shows the different layers of the framework.

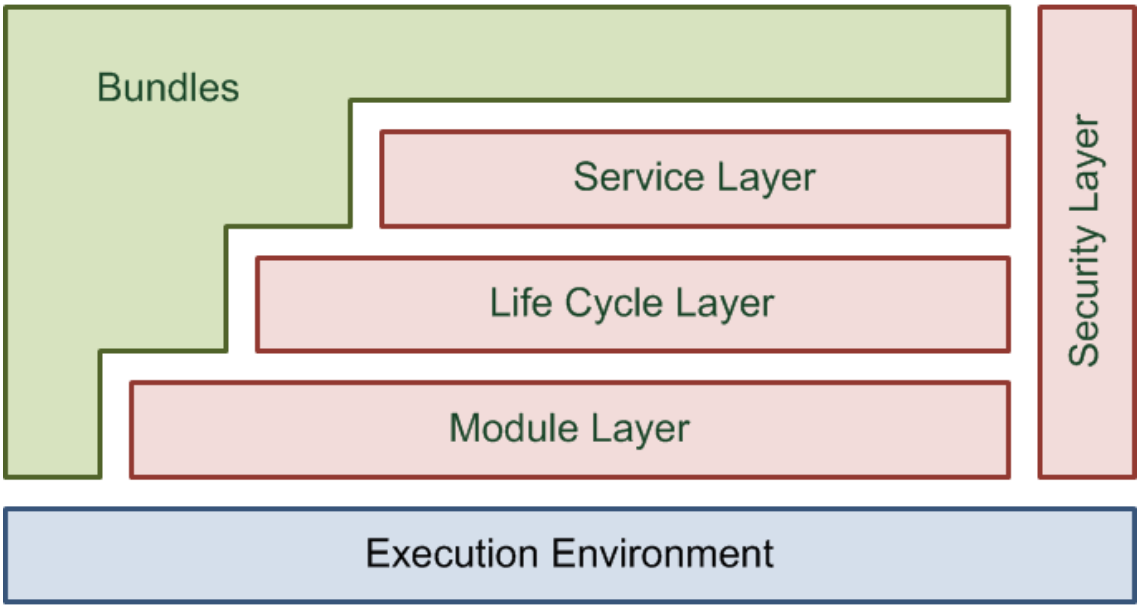

Figure 2 – OSGi Service Gateway Architecture (adapted from [12])

Table 3 describes the different components of an OSGi framework and its respective function.

Component	Function
Execution Environment	The execution environment is specified in a way that it is executable under different Java platforms. Instead of requiring a specific Java version, it just defines the classes, interfaces and methods that have to be available
Modules Layer	The modules layer defines the class loading policies. Java usually has a single classpath that contains all classes and resources. This layer adds private classes for modules and controlled linking between them on top of Java [10]. It defines how bundles can import and export code.
Life-Cycle	Adds the possibility to manage the bundles at runtime by providing an API for installing, updating, starting, stopping and uninstalling bundles. Dependency mechanisms assure the correct operation of the environment.
Services / Service Registry	The service layer adds a service registry so that bundles can be registered in the framework. In contrast to SOA OSGi does not directly address distributed systems.

Security	Provides the possibility to restrict execution rights of bundles. It is based on Java and the Java 2 security model.
Bundles	Defines the smallest unit inside the OSGi framework. It consists of Java classes and the required resources and it holds a MANIFEST.MF file that defines the dependencies with other bundles and the exported and imported packages.

Table 3 – Different Components of an OSGi framework

2.3.3 Bundles

Bundles are JAR files that contain resources and classes. Dependencies between bundles can be managed by the OSGi framework. Classes in a bundle are not visible to other bundles by default. They have to be exported through the manifest file. The manifest is a special file that can contain information about the files packaged in a Java archive [13] and it resides inside the META-INF folder. Each archive can only have one manifest file. Entries in the manifest are in the form of "header: value" pairs. The default manifest of a jar created by the *Java Development Kit (JDK)* version 1.2 is shown in Listing 6.

```
Manifest-Version: 1.0
```

Listing 6 – Default manifest of a jar created by the JDK 1.2

There is no assumption about what information should be recorded in a manifest. The information recorded depends on the intended use of the JAR file. For example, if an application is bundled as a JAR file, it contains the header *Main-Class: classname* that tells the Java Virtual Machine where the entry point of the application is. The OSGi framework defines additional headers like *Export-Package* and *Bundle-Classpath*, for example. Listing 7 shows an example of a manifest of an OSGi bundle and Table 4 describes the different headers. For a complete list of headers see [12].

```
Manifest-Version: 1.0
Created-By: Apache Maven Bundle Plugin
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.example.helloworld
Bundle-Name: helloworld
Bundle-Version: 1.0.0
Bundle-Activator: org.example.Activator
Bundle-Description: A simple Bundle
Import-Package: org.osgi.framework
Export-Package: org.example.*
```

Listing 7 – Example manifest of an OSGi bundle

Header-Name	Description
Manifest-Version	This is required for all JAR file manifests and must be the first entry.
Created-By	Gives some information about the vendor or tool that generated this manifest.
Bundle-ManifestVersion	Defines that the bundle follows the rules of this specification (1 for Release 3 bundles, 2 for Release 4 and later bundles)
Bundle-SymbolicName	This is a mandatory header that specifies a non-localizable name for this bundle. Together with the Bundle-Version it identifies a unique bundle. It should be based on the reverse domain name convention.
Bundle-Version	Defines the version of this bundle.
Bundle-Activator	Specifies the name of the class used to start and stop the bundle. See next chapter.
Bundle-Description	Defines a description of this bundle.
Import-Package	A coma-separated list of packages this bundle is depending on. The version of an imported package can be committed and packages can be selected as optional, meaning the bundle can resolve without the package being available.
Export-Package	The Export-Package is similar to the Import-Package definition. A coma-separated list of package-names defines the packages that are exported and hence made available for other bundles.

Table 4 – Different manifest headers of an OSGi bundle

2.3.4 Life Cycle

The life cycle layer adds the mechanisms to dynamically install, update, start, stop and uninstall bundles. Figure 3 and Table 5 show the different bundle states and the order of their transitions.

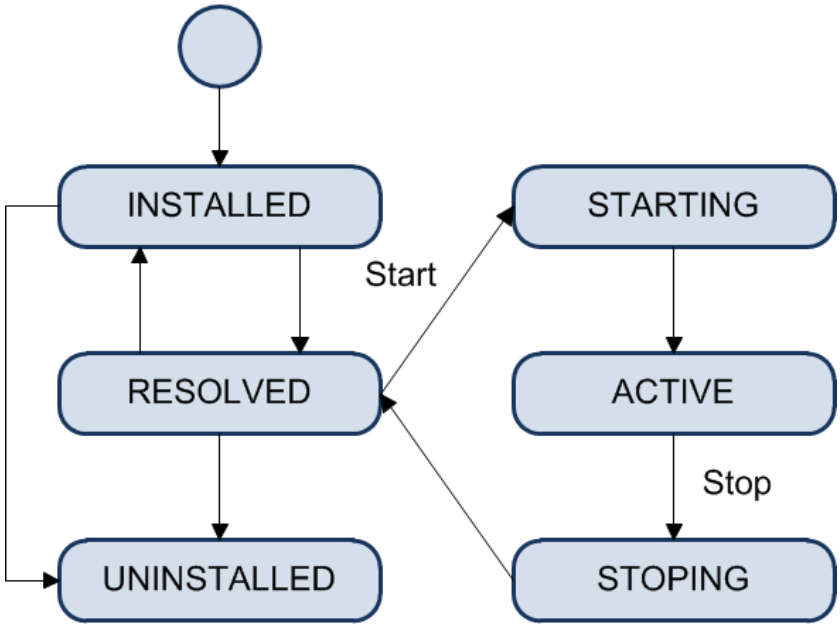

Figure 3 – Bundle life cycle

After installation the bundle tries to resolve all the required packages and enters the state RESOLVED if all packages are found. Once resolved the bundle can be started and is active as soon as its BundleActivator has finished its initialization. When it is active it can be stopped. The bundle enters the RESOLVED status again as soon as the stop method of the BundleActivator has returned. Of course it can be uninstalled when it is in the status INSTALLED or RESOLVED.

Bundle State	Description
INSTALLED	The bundle is successfully installed
RESOLVED	The classes that are needed by the bundle are available. This means that the bundle is either ready to be started or it has stopped.
STARTING	The bundle is starting and the BundleActivator.start() method is called but has not yet returned.
ACTIVE	The bundle is active and running and the BundleActivator.start() method has returned.
STOPPING	The bundle is stopping and the BundleActivator.stop() method is called but has not yet returned.
UNINSTALLED	The bundle has been uninstalled.

Table 5 – Bundle states

As described in the chapter before, the bundle can have a bundle activator. The bundle is activated by calling its bundle activator, if one exists. The BundleActivator-header

must be specified in the manifest by the developer. Listing 8 shows an example of a simple BundleActivator.

```
package org.helloworld;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {
	private BundleContext context;

	public void start(BundleContext context) throws Exception {
		System.out.println("Hello World");
		this.context = context;
	}

	public void stop(BundleContext context) throws Exception {
		System.out.println("Goodbye World");
		this.context = null;
	}
}
```

Listing 8 – BundleActivator sample

The Activator implements the BundleActivator interface that requires an implementation of the two methods *start* and *stop*. In this example the start-method simply prints out a text and saves the current *BundleContext* that was given as a parameter. The BundleContext grants access to other methods so that the bundle can interact with the framework e.g. subscribe to events published by the framework or receive a list of installed bundles. When the stop-method is called it simply prints out a text and sets the context to *null*. The two import declarations are needed for the BundleActivator interface and the BundleContext class.

2.3.5 Services

A service is a *Plain Old Java Object (POJO)* or a *Plain Old Java Interface (POJI)* that is registered in the service registry under an interface name. Registration is not persistent and exists only throughout runtime. Even during runtime services can register and unregister and hence not be available anymore. The API provides three mechanisms called *Service Listener*, *Service Tracker and Declarative Service* to react to those changes. The OSGi framework also provides a lot of standard services like http-service that can be used by other bundles (see [10] for more standard services).

2.4 Maven

Maven[14] is a build-management tool developed by the Apache Software Foundation that is based on Java. Especially Java programs can be created and managed with Maven. It is essentially a project management and comprehension tool [15] and provides a way to help with managing:

- Builds
- Documentation
- Reporting
- Dependencies
- Software Configuration Management (SCM)
- Releases
- Distribution

It is similar to Ant [16] but it is very useful for complex multi-projects, while Ant can be easier used in small projects. Maven tries to follow the idea of *Convention over Configuration*, a software design paradigm that aims to minimize the decisions a developer has to make. This means a developer only has to write code if he deviates from a predetermined path, e.g. the tool expects a certain project folder structure. Deviating from this folder structure leads to additional configuration steps.

2.4.1 Installation

Maven can run on Java 1.4 but the latest stable Java Development Kit (JDK) should be installed[1], which at the time of this writing is Java 6. The current installed version of Java can be determined by running *java –version* on the command line.

```
% java -version
Java version "1.6.0_24"
Java(TM) SE Runtime Environment (build 1.6.0_24-b07)
Java HotSpot(TM)Client VM (build 19.1-b02, mixed mode, sharing)
```

Listing 9 – Determining the current Java version

Maven can be downloaded[2] from the project website. The current version as of this writing is 3.0.3. It can be installed on different operating systems like Mac OS X and Microsoft Windows. Installation steps are similar on all operating systems. They usually only differ in the installation location and environment variables. The following steps describe the procedure on Microsoft Windows (see [17] for instructions on different operating systems like Mac OS X, Linux and FreeBSD). Maven needs a correct setting in two environment variables: *M2_HOME* and *PATH*. Listing 10 shows how to set the environment variables in Microsoft Windows, assuming Maven is unpacked and installed under C:\Program Files\apache-maven-3.0.3.

[1] http://www.oracle.com/technetwork/java/javase/downloads/index.html

[2] http://maven.apache.org/download.html

```
C:\Users\krein > set M2_HOME=c:\Program Files\apache-maven-3.0.3
C:\Users\krein > set PATH=%PATH%;%M2_HOME%\bin
```

Listing 10 – Setting environment variables under Windows

Once the configuration is completed the installation can be tested by running *mvn –v* on the command line (see Listing 11).

```
$ mvn -v
Apache Maven 3.0.2 (r1056850; 2011-01-09 01:58:10+0100)
Java version: 1.6.0_23, vendor: Sun Microsystems Inc.
Java home: C:\Programme\Java\jdk1.6.0_23\jre
Default locale: de_DE, platform encoding: Cp1252
OS name: "windows xp", version: "5.1", arch: "x86", family: "windows"
```

Listing 11 – Testing the Maven installation

If the mvn command is not found then the M2_HOME and PATH variables are not set properly.

2.4.2 Repositories

When Maven runs the first time, it creates a user-specific configuration file and a local repository. A repository holds build artifacts and dependencies of varying types. Maven distinguishes between local and remote repositories. The local repository serves as a cache for remote repositories and contains also the temporary build artifacts that are not yet released. When an artifact is not found in the local repository it is downloaded from a remote repository into the local one. The standard repository that is searched is *http://repo1.maven.org/maven2* but other repositories can be defined, too. The local repository resides in the user's home directory under *~/.m2/repository*.

2.4.3 Project Object Model (POM)

The *Project Object Model (POM)* is the central concept of Maven. This is where the structure of a project is declared, builds are configured and projects are related to one another [17].A Maven project is defined by the presence of a *pom.xml* file. It is an XML representation of a Maven project and can be compared to a *Makefile* or a *build.xml* file in Ant. The majority of the POM deals with descriptions of where the source code is, where the resources are and what the packaging is. A build.xml in Ant will look different because it contains explicit instructions for tasks like compiling a set of Java classes. Although the Maven Ant plug-in allows including procedural customizations, developers usually will not have to deal with such procedural details.

Though Maven's default plug-ins are targeted at building JAR artifacts, the POM is not specific to building Java applications, e.g., it is possible to define a project that contains C# sources and produces a proprietary Microsoft binary [17].

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <!-- The Basics -->
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <packaging>...</packaging>
  <dependencies>...</dependencies>
  <parent>...</parent>
  <dependencyManagement>...</dependencyManagement>
  <modules>...</modules>
  <properties>...</properties>

  <!-- Build Settings -->
  <build>...</build>
  <reporting>...</reporting>

  <!-- More Project Information -->
  <name>...</name>
  <description>...</description>
  <url>...</url>
  <inceptionYear>...</inceptionYear>
  <licenses>...</licenses>
  <organization>...</organization>
  <developers>...</developers>
  <contributors>...</contributors>

  <!-- Environment Settings -->
  <issueManagement>...</issueManagement>
  <ciManagement>...</ciManagement>
  <mailingLists>...</mailingLists>
  <scm>...</scm>
  <prerequisites>...</prerequisites>
  <repositories>...</repositories>
  <pluginRepositories>...</pluginRepositories>
  <distributionManagement>...</distributionManagement>
  <profiles>...</profiles>

</project>
```

Listing 12 – Example pom.xml [14]

Listing 12 shows the elements directly under the POM's project element. As a complete description of all elements would go beyond the scope of this student thesis, the following is just a description of the elements needed to create a simple project, more precisely an OSGi bundle for the WSO2 Carbon Framework.

Note: modelVersion is set to 4.0.0. This is the only version supported since version 2 of Maven.

The minimum fields in a pom.xml that are required by maven are *groupId*, *artifactId* and *version*. The groupId and version elements can be inherited from a parent project and do not need to be specified explicitly. Listing 13 shows a minimal pom.xml.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>example-app</artifactId>
  <version>1.0</version>

</project>
```

Listing 13 – Minimum pom.xml required by Maven

The three fields act like an address and are also referred to as the coordinates of the project.

- **groupId:** This is often unique amongst an organization or a project, e.g. all core Maven artifacts have the groupId org.apache.maven. The use of the dot notation is recommended but not mandatory. Inside a repository the groupId acts like a Java package in an operating system, meaning dots are replaced by directory separators so that the *org.example* group in the example above can be found under the *org/example* folder inside the repository.
- **artifactId:** The artifactId is basically the name of the project. Along with the groupId it identifies a project uniquely. The project in the example above would reside under *org/example/example-app* in the repository.
- **version:** The artifactId and the groupId may identify a project uniquely but a project may evolve and come in different versions, hence each version has to be identifiable. In a repository, the example above could be found under *org/example/example-app/1.0*.
- **packaging:** There is one more label that belongs to the address part, the packaging. This is set to JAR per default so it does not have to be declared explicitly assuming that JAR is the desired packaging. Listing 14 shows how to declare the packaging as a WAR.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...
  <packaging>war</packaging>
  ...
</project>
```

Listing 14 – Specifying the packaging

2.4.4 Dependencies

Maven allows relationships between projects through dependencies, inheritance and aggregation. It can manage both internal and external dependencies. External ones might be a library like *Log4J.* An internal one is illustrated by a web application depending on another application that can contain classes and objects. Listing 15 shows a sample usage of dependencies.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                http://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <dependencies>
  <dependency>
   <groupId>org.example</groupId>
   <artifactId>another-example-app</artifactId>
   <version>2.0</version>
   <type>jar</type>
   <scope>compile</scope>
   <optional>true</optional>
  </dependency>
  ...
 </dependencies>
 ...
</project>
```

Listing 15 – Dependencies in a POM

All dependency elements are grouped under the dependencies element. Maven downloads and links dependencies automatically and also brings in the transitive dependencies, so developers only have to focus on direct dependencies.

- **groupId, artifactId, version:** These elements have already been described in the previous chapter. Projects can only depend on Maven artifacts so that all dependencies are managed by Maven. When a project depends on another project that cannot be made available in a central repository (e.g. because its license does not allow it) then Maven provides a method to install the project into the local repository through a plugin as shown in Listing 16.

```
mvn install:install-file
  -Dfile=example-app-non-maven.jar
  -DgroupId=org.example
  -DartifactId=example-app-non-maven
  -Dversion=1
  -Dpackaging=jar
```

Listing 16 – Installation of non-Maven projects (adapted from [14])

To install the project, the plugin requires the coordinates for the project and automatically generates a POM.

- **type:** This defines the packaging type. Examples are *jar*, *war* and *bundle*. It is possible to define new types through plugins. The type usually represents the file extension.
- **scope:** There are five different scopes available that control which dependencies are available in which classpath and which are included in an application.
    - **compile:** The default scope that is used if none is specified. Dependencies are packaged and will be available in all classpaths.
    - **provided:** This indicates that the projects expects the JDK or a container to provide the dependency at runtime, e.g. a web application could need the Servlet API to compile the project but it would not be included in the WAR. The application server or servlet container would provide it.
    - **runtime:** This indicates that the dependency is not required to compile the project but it is needed at runtime, e.g. a JDBC API JAR is needed at compile time and the JDBC driver implementation only at runtime.
    - **test:** The dependency is not needed for the execution of the application but for compilation and execution phases of the tests.
    - **system:** This is similar to *provided*, except that a developer has to provide the JAR explicitly. The artifact is always available and not looked up in a repository.
- **optional:** Lets other projects know that they do not require this dependency in order to work correctly when they use this project.

Dependencies allow to exclude transitive dependencies, e.g. if a developer wants to replace the transitive dependency with another dependency that provides the same functionality. Listing 17 shows how to use exclusions in dependencies.

```
<dependency>
 <groupId>org.example</groupId>
 <artifactId>example-app</artifactId>
 <version>1.0</version>
 <exclusions>
  <exclusion>
   <groupId>org.example</groupId>
   <artifactId>example-app-dependency</artifactId>
  </exclusion>
</exclusions>
```

Listing 17 – Using exclusions in dependencies

2.4.5 Inheritance

All POMs inherit from a so called “Super POM” that defines standard configuration variables (see [17] for a description of the complete Super POM). It can be compared to the *Object* class in a Java world, where all classes implicitly inherit from this super class. A project can also inherit from a parent POM (Listing 18). This is useful when large systems are built and developers do not want to repeat dependencies. When a

project specifies a parent it inherits all information from the parent. These settings can be overridden and new values can be added. When a POM inherits from a parent, it does not inherit directly from the super POM because the parent, or at least one of its "ancestors", already inherited from the super POM.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <parent>
    <groupId>org.example</groupId>
    <artifactId>parent-app</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>example-app</artifactId>
  ...
</project>
```

Listing 18 – Specifying a parent

As Listing 18 shows the project itself does not specify a groupId and a version. It specifies only the artifactId. The groupId and version are inherited from the parent. The resulting POM when merging the super POM, the parent POM and the current project POM is called the effective POM. Running *mvn help:effective-pom* on the command line inside a project would retrieve the effective POM.

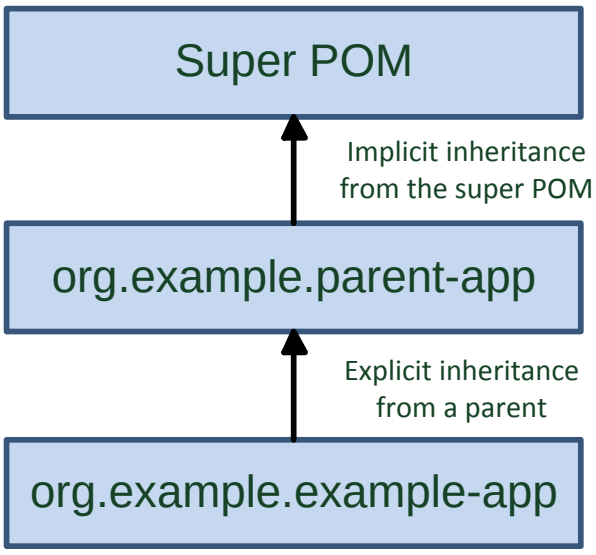

Figure 4 – POM Inheritance (adapted from [17])

Maven assumes that the parent POM is either available in the local repository or it is available in the parent directory of the current project. This default behavior can be overridden via the *relativePath* element if neither of the two is the location of the parent.

2.4.6 Aggregation

A project that contains modules is called a multi-module or aggregator project. The difference between inheriting from a parent and being managed by a multi-module project is in the parent project passing its values to its children, while a multi-module project just manages a group of other projects. Listing 19 shows an example project that uses modules. The projects packaging type has to be set to *pom*.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>example-app</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>

  <modules>
    <module>example-module-one</module>
    <module>example-module-two</module>
  </modules>
</project>
```

Listing 19 – A sample project that contains modules

2.4.7 Build Lifecycle

Another big part of the POM is the build section that defines the directory structure of the project and manages the plugins.

Maven follows the central concept of build lifecycles, which define the process for building and distributing artifacts. The three built-in build lifecycles are *default*, *clean* and *site*. They handle the project's deployment, cleaning and site documentation creation. A build lifecycle is made up by phases or stages. Some of the phases of the default lifecycle are described in Table 6 (see [14] for a complete reference).

Phase	Description
Validate	Check if the project is correct and all necessary information is available.
Compile	Compile the source code of the project.
Test	Test the source code.
Package	Package the code in its distributable format, e.g. a JAR or a WAR.
Verify	Verify that the package is valid and meets defined quality criteria.

Install	Install the package into the local repository so that it can be used by other projects.
Deploy	The final package is copied to a remote repository for sharing.

Table 6 – Some phases of the default lifecycle.

To run the complete lifecycle, one would have to call *mvn deploy* on the command line inside the current project folder. This would execute all the different lifecycle phases in sequential order (plus the ones that are not shown here). If the project should only be packaged and neither installed nor deployed then one would only call *mvn package* on the command line. This would execute all the phases of the lifecycle till it finishes packaging. Later lifecycle phases are not executed.

The clean lifecycle consists of the three phases *pre-clean*, *clean* and *post-clean*. It basically removes all the files created by the previous build. It is possible to run two lifecycles in sequence, e.g. *mvn clean install* first runs the clean lifecycle and then the install lifecycle.

2.4.8 Plug-ins and goals

Plug-ins are libraries that implement specific goals. Goals are tasks that are finer than build phases and comparable to an Ant tasks. They are usually bound to a build phase, but they could be executed by direct invocation, e.g. *mvn archetyp:generate* calls the *generate* goal of the *archetype* plugin. All goals that belong to a certain lifecycle phase are called automatically during the execution of the lifecycle phase. Listing 20 shows an example of how to use plugins.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <build>
  …
  <plugins>
   <plugin>
    <groupId>…</groupId>
    <artifactId>…</artifactId>
    <version>…</version>
    <configuration>…</configuration>
    <executions>…</executions>
   </plugin>
  </plugins>
 </build>

</project>
```

Listing 20 – Example of a plugin

The *configuration* section is specific to each plugin and specifies properties that the plugin expects. As plugins can have many goals, the *execution* section configures each goal of a plugin. There are many Maven plugins (see [18] for a list of available Maven plugins) but the important plugins for the task of this student thesis are the *maven-bundle-plugin*, *carbon-p2-plugin*, *maven-antrun-plugin* and *build-helper-maven-plugin*.

The maven-bundle-plugin allows one to create OSGi bundles. This is especially useful because it helps to create the OSGi specific headers in the manifest. The carbon-p2-plugin is used to create features and repositories for carbon (see chapter 4.4 – Developing components and features). As the name implies the maven-antrun-plugin can run Ant tasks, in this case we need to run the *WSDL2Java* tool to create so called *Stubs* for the provided WSDL files of the Web services of the Apache ODE API. The last plugin is the build-helper-maven-plugin that can add the generated code of the WSDL2Java tool as source directories to the POM. The plugins are described in chapter 6 – Implementation.

3 Business Process Illustrator

Business Process Illustrator (BPI) is a tool that has been developed in the course of a diploma thesis [3] written by Gregor Latuske in 2010 at the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart. BPI is a business process monitoring tool that lists all deployed process models and created process instances. As main feature the BPI provides the ability to apply process views to a monitored process model or instance. Figure 5 shows a screenshot of how the deployed process models and process instances of Apache ODE are listed in the BPI.

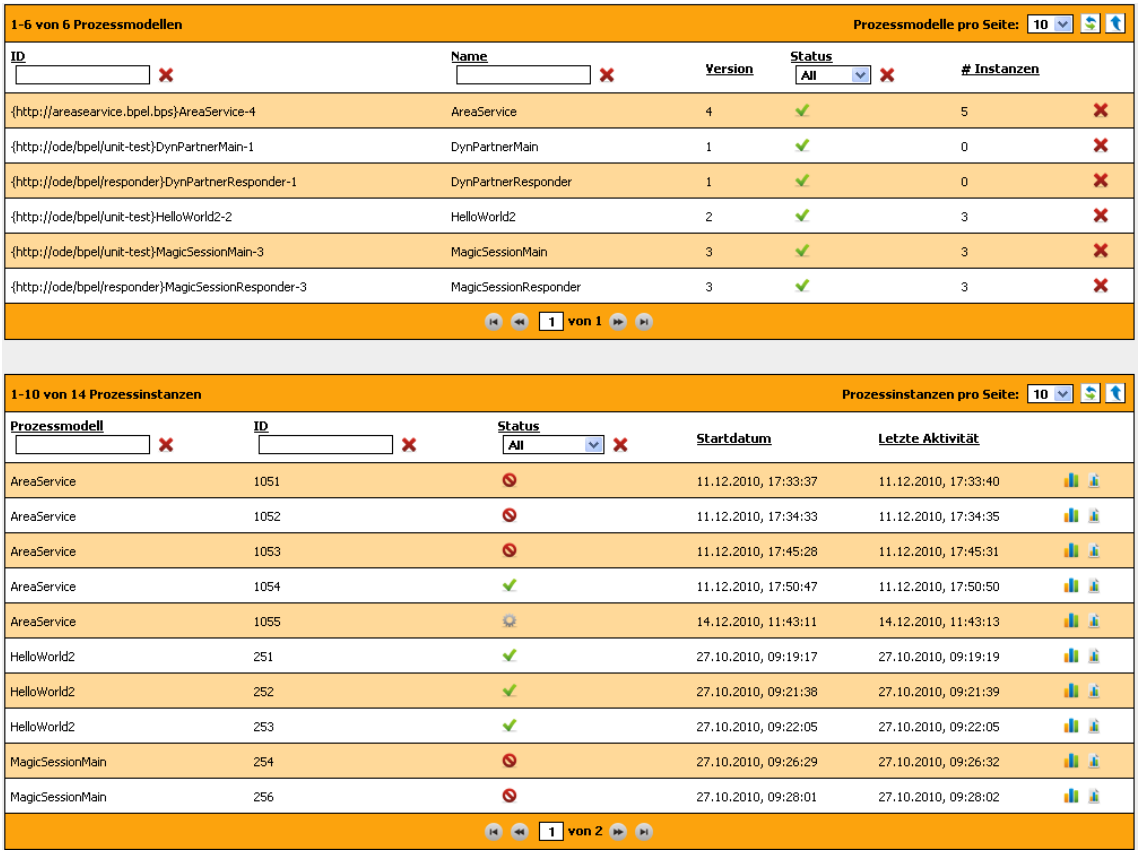

Figure 5 – Screenshot of process models and process instances in BPI

The list at the top shows the deployed process models, the one at the bottom shows running or past process instances. Both tables can be adjusted in a similar way, e.g. the number of models and instances respectively can be set in the top right corner in steps of 10, 20 and 50 rows per table and the tables can be refreshed and minimized here. There are also buttons for browsing through the pages at the bottom of the tables if the number of entries exceeds the allowed entries per page. Entries can be sorted by every column and some columns allow filtering.

The process models list shows the *id*, *name*, *version*, *status* and the *number of instances* for each model. It is possible to filter the ids and names or to just show the entries with *active* or *retired* status.

The process instances list shows the *process model*, *id*, *status*, *start date* and the *date of last activity* for each instance. Entries can be sorted by every column and filtered by process model and id. It is also possible to filter entries according to their status, i.e. *all*, *active*, *suspended*, *completed*, *terminated*, *failed* and *error*.

The real benefit of the BPI is its ability to show process instances as a graph and to apply process views. Therefore, every entry in the process instances list has a button for showing its graph in either the same browser window or in a new browser tab. Figure 6 shows a graphical representation of a process instance and Figure 7 shows what the same graph can look like when process views are applied.

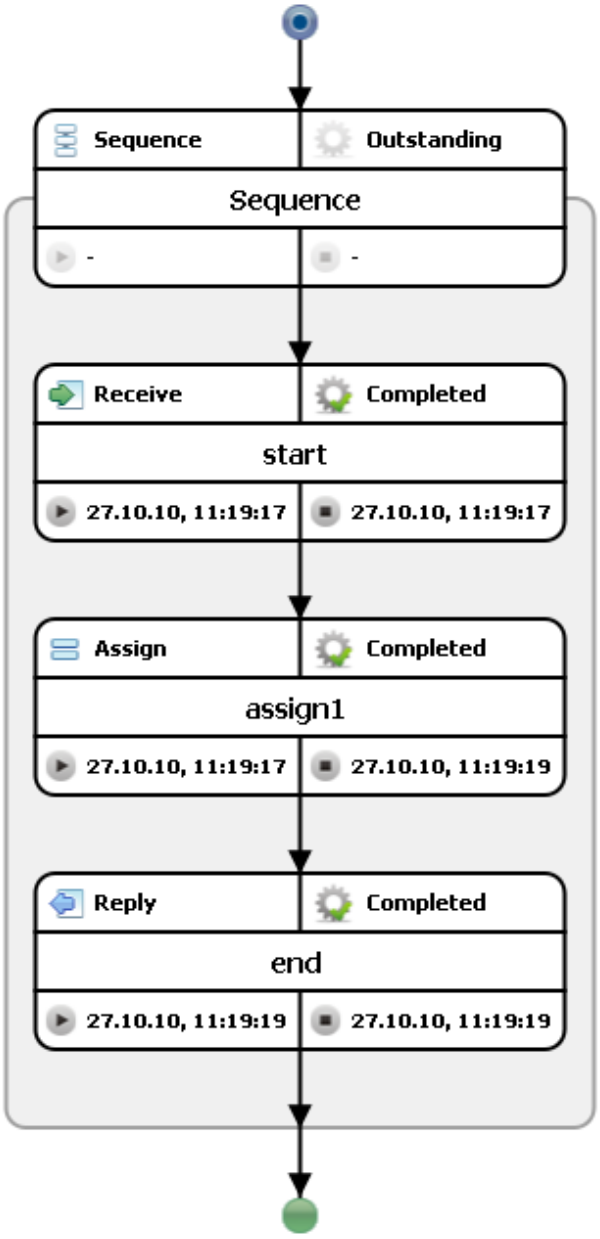

Figure 6 – Graphical representation of a process instance

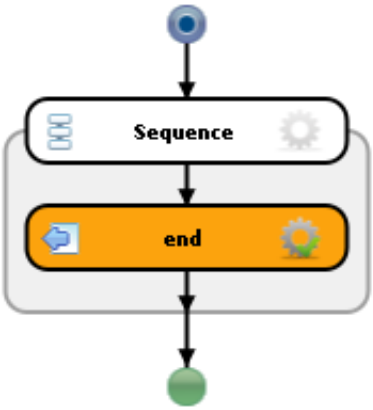

Figure 7 – Graph with applied process view.

As seen in Figure 7, the receive and assign activities are left out, the end activity is highlighted and some of the information like start and end times are hidden which results in a more compact view, compared to the one seen in Figure 6. To allow those adjustments, the BPI supports the user with additional graphical controls which are positioned above the graph. Figure 8 shows those controls with the settings that resulted in the earlier seen collapsed graph.

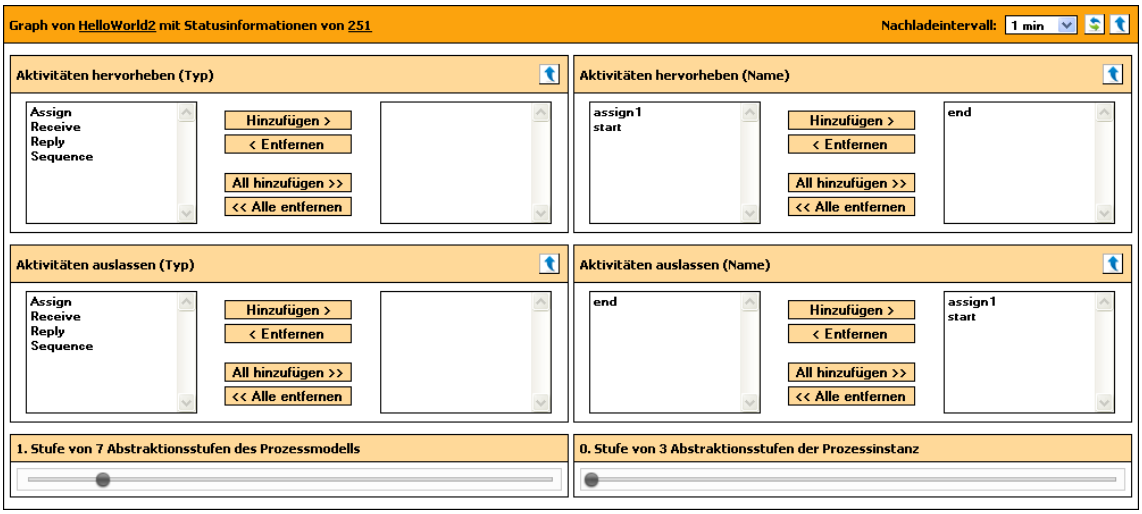

Figure 8 – Graphical controls to adjust and apply process views

The bar on top shows some information about the currently selected process instance. On the top right are again buttons for refreshing and minimizing the graph and a box for selecting the refreshing interval at which the graph is automatically refreshed.

Each of the four boxes in the middle allows the user to highlight, respectively hide certain activities (right boxes) and activity types (left boxes). In this instance, the *end* activity is marked as highlighted while the *assign1* and *start* activities are marked as hidden. To mark activities, the user can select an activity and use the *add* and *remove* buttons. Each box can be minimized or maximized by clicking the button in the top right corner of the box.

The two sliders on the bottom allow the user to automatically reduce the process graph. The number of steps, by which the graph can be reduced, depends on the complexity of the process instance. For more details see [3] and [19].

3.1 Architecture

The BPI is implemented as a 3-tier *Rich Internet Application* consisting of a client, a web server and a workflow management system. The great benefit of this architecture is its ability to be accessed from anywhere. All a client needs is a web browser that supports JavaScript and SVG. There's no need to install any further software. Figure 9 shows the underlying architecture of the BPI.

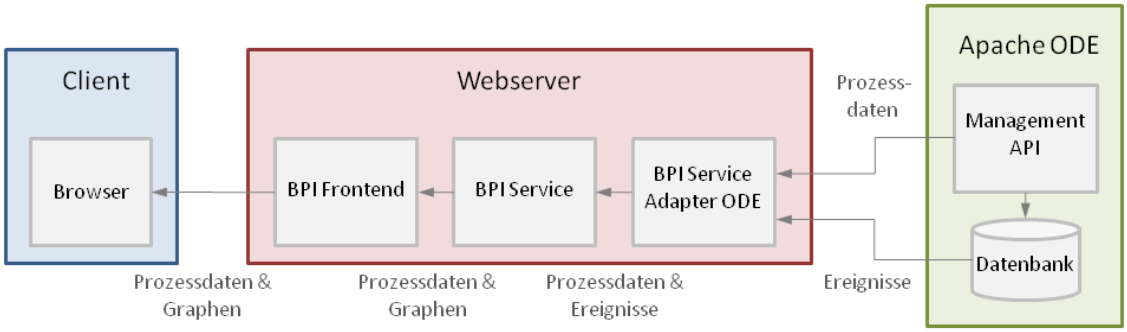

Figure 9 – BPI Architecture [3]

The 2nd tier consists of an application server hosting a Java web application. It consists of three components called BPI Frontend, BPI Service and BPI Service Adapter ODE. The BPI Service is the central component that calls the process data and events from the WfMS through a BPI Service Adapter and generates the graphs. The BPI Frontend presents the aggregated status information and graphs to the user. To retrieve the process data and events from the different WfMS, the BPI Service uses adapters. Each WfMS must have its own adapter. The BPI currently comes with an adapter for Apache ODE.

The 3rd tier is the WfMS. Every WfMS, that has a Management API or a database that is accessible from the outside, should be able to be integrated with the BPI through an adapter that is written for that specific WfMS. The following chapters take a closer look at the three components, how they are implemented and what technologies they use.

3.2 Implementation

The different parts of the source code[3] are split across different eclipse projects. Table 7 shows the different projects and their role.

[3] Source code is available under http://sourceforge.net/projects/bpi/

Project Name	Description
Frontend	Dynamic Web Project. Packaged as a Web Archive (WAR). Generates dynamic HTML pages. Imports the other projects.
Libraries	A library-only project. Organizes the vast amount of libraries into folders (e.g. the Apache ODE libraries).
Model	Holds the classes for the underlying generic data model.
Service	Provides an interface for the frontend component. Allows changing the used adapter without the frontend knowing and without code changes.
ServiceAdapterODE	An adapter for Apache ODE.
ServiceAdapterTest	A test adapter mainly for developing purposes. No WfMS is needed.
Tools	Different tools to test the application and to deploy data models.

Table 7 – Different projects of the BPI source code

3.2.1 BPI Frontend

The previous chapters already explained how the GUI works. The basic idea and the graphical representation of the BPI are based on SUN's BPEL2SVG Generator [20]. It holds a table for process models and instances and a graphical representation for a selected process model as well.

One goal of the development of the BPI was a userfriendly Graphical User Interface in terms of design and installation. The BPI runs inside a web browser so the user has no need to install any software. The only software needed is a browser of course that has JavaScript enabled, but most Operating Systems provide a browser. The generated HTML sites contain parts with SVG code. Most browsers support SVG, either natively or via plugins.

In order to dynamically generate the HTML pages, the frontend is mainly built with *JavaServer Faces (JSF)* and *Facelets* with *AJAX* and some Managed Beans. Facelets provide a more "pluggable" ViewHandler framework for JSF that is more designer friendly. The Facelets used are usual XHTML files which contain no Java code anymore compared to *JavaServer Pages (JSP)*. The benefit of JSF is the possibility to reuse components through *composite components*. This is very useful, as the frontend has a lot of graphical components that have similar functionalities and hence similar designs. For example, every component e.g., the process model table or the process instance table, has the possiblity to be minimized. With JSF those parts have to be

programmed only once and can then be reused in every component that needs minimizing or maximizing functionality.

This is one of the first requirements to the environment the BPI will be running inside – in this case, the WSO2 Carbon Framework. If the environment does not support JSFs, one option is to take the main funcionality of the BPI (the listing of process models and instances and the graphical representation of a process model) an pack them into a JSP file. Almost every component that is developed for the WSO2 Carbon Framework consists of JSP files. But as for now, there is no component developed by the WSO2 Carbon team that already uses Facelets.

The frontend uses JavaSript to automate the refreshing of the website and to display the sliders. Sliders are not supported for HTML versions below 5 so the frontend uses input fields as sliders and changes the presentation and behaviour through JavaScript. The website needs to be refreshed automatically because a polling mechanism is used for reading events from the WfMS. To prevent a complete reloading of the site each time the user minimizes or maximizes one of the tables or boxes the BPI uses a technology called *Asynchronous JavaScript and XML (AJAX).* AJAX can replace or reload parts of a site without completely reloading the site so the user gets the feeling that he is working with a desktop application rather than a website.

The SVG code is not generated by JavaServer Faces but it is dynamically created by a Servlet and embedded inside the HTML code using the <object> tag, which allows to embed basically any content inside a HTML file.

```
<object data="rect.svg" width="300" height="100" type="image/svg+xml"
codebase="http://www.adobe.com/svg/viewer/install/" />
```

Listing 21 – Example usage of the <object>-tag

Data that needs to be available for a whole session or for a request is stored in two Managed Beans. *MainBean* is a so called *SessionBean* that is available throughout the whole session and holds information about the current process models and instances and the current settings for the main window. *SVGBean* is a *RequestBean* and is connected to a request when the user wants to monitor a process instance in a separate window e.g. to monitor different instances at the same time. It holds information about the graph and the settings of the current window.

The Java classes of the frontend, e.g. the ManagedBeans, are organized in the *net.latuske.bpi.frontend* package and subpackages. All the web-related files are stored under the *WebContent* folder. There is also a subfolder called *resources* that holds part of the XHTML files and all the Stylsheets, Scripts and Icons that are used for the design of the webpages.

The frontend calls all the information and data it needs from an interface called *BPIService*. The next chapter explains how the BPIService works and why it is needed.

3.2.2 BPI Service

The BPIService is the central part of the application. It calls the process models and process instances from the WfMS and generates the SVG graph. It provides a clear interface that can be used by the frontend. An instance of the BPIService is created through the use of the *BPIServiceFactory*. The benefit of the use of a factory pattern [21] is the possibility to choose the actual implementation at runtime. The BPI Service also contains the definition of the BPI Service Adapter Interface.

The concrete implementation of the BPI Service Adapter is detected at runtime through the use of a properties file. It contains the name of the classes that implement the different interfaces. Thus the used WfMS can be changed by modifying the properties file, even at runtime. The loading of the classes is done through the class.forName() method as shown in Listing 22.

```
public ProcessModelService<?> createService() throws BPIException {
    try {
        String cName = PropertiesUtil.getInstance().getProperty(SERVICE_NAME);
        return (ProcessModelService<?>) Class.forName(cName).newInstance();
    } catch (Throwable t) {
        throw new BPIException(t);
    }
}
```

Listing 22 – Dynamic loading of the BPI Service Adapter [3]

The first thing the BPI Service does is calling the process models from the WfMS through the BPI Service Adapter. At this point caching the process models can be very effective. Process models are not changed very often, so caching lifetime can be set very high. Afterwards the process instances are loaded, either all instances or just the ones belonging to a certain process model. This depends on the WfMS and the corresponding adapter. Once the process models and instances are loaded, a component called BPEL Parser parses the process models and creates a *DOM (Document Object Model)*. It is a structure that represents XML documents as trees that can be traversed recursively. The parser just loads activities and links at the moment. The process model class gets a reference to the root activity.

Before the SVG graph is generated, the events of a selected process instance are called from the WfMS. Some workflow management systems like the Apache ODE do not provide those events via API's. The service adapter for ODE that comes with the BPI operates directly on the database. This can be problematic if the WfMS forbids direct access to the database, e.g. the database that comes with ODE is not accessible and is replaced by a MySQL database.

The WSO2 Business Process Server is based on ODE so this is another requirement for the integration process. As later chapters will show, the Business Process Server provides a different API than ODE where events are available through API calls rather than calling them from the database.

3.2.3 BPI Service Adapter (ODE)

A BPI Service Adapter has to implement the three interfaces *ProcessModelService*, *ProcessInstanceService* and *EventService*. Apache ODE provides a management API that the ProcessModelService and ProcessInstanceService use to get the process models and instances. Listing 23 shows an example of an API call that returns a list of all process instances. The actual addresses of the Web Services are also stored in a properties file and retrieved at runtime.

```
ServiceClientUtil client = new ServiceClientUtil();
OMElement root = client.buildMessage(
    "listAllProcesses", new String[] {}, new String[] {});
OMElement result = client.send(
    msg, "http://localhost:8080/ode/processes/ProcessManagement");
```

Listing 23 – Apache ODE Management API [3]

The EventService accesses the database directly because there is no such functionality implemented in the API. The service establishes connectivity through the use of JDBC, a database interface for relational databases that is part of the Java Standard Edition. The name of the JDBC driver and the access data of the database are both stored in a properties file, so that any database that has a JDBC driver can be used. To change the database, the JAR of the JDBC driver has to be included in the *classpath* of the application and the properties file must be adapted accordingly.

After connecting to the database all events belonging to a particular instance are loaded from the ODE_EVENT table. Every entry has a *BLOB (Binary Large Object)* that is de-serialized into a Java object. This object is an instance of *ActivityEvent* defined by Apache ODE and can be used to create an *ActivityExecEvent* of the underlying data model.

This shows another requirement for the integration process. Though the Business Process Server is based on Apache ODE, it uses a different API. From this follows that a new BPI Service Adapter has to be written for the Business Process Server because the provided adapter for ODE does not work.

4 WSO2 Carbon

WSO2 [22] is a company that was founded in 2005 by Sanjiva Weerawarana and Paul Fremantle, pioneers in XML and Web services technologies and standards as well as open source. They offer a complete SOA platform that is completely free and open source. The business model of the company is based on providing training, consultancy and support for the software.

In the beginning the company had developed different products like an *Enterprise Service Bus (ESB)*, *Web Services Application Server (WSAS)* and *Business Process Server* that all had some common code base and were used together by customers. When the code base changed, it had to be copied from the code base to all products, so the company had the idea to develop a component-based SOA platform. They called it the Carbon framework.

4.1 Architecture

The Carbon framework is a modular middleware based on Equinox and OSGi. Everything in the Carbon framework is closely based on Apache technology, e.g. Apache Axis2, Apache ODE, Apache Tomcat and many others. It is completely build out of OSGi components. Developers like to call it “Eclipse for Servers”, referring to the possibility to reconfigure the framework, e.g. to install or uninstall features. The Carbon framework is split into a core section and a features section as shown in Figure 10.

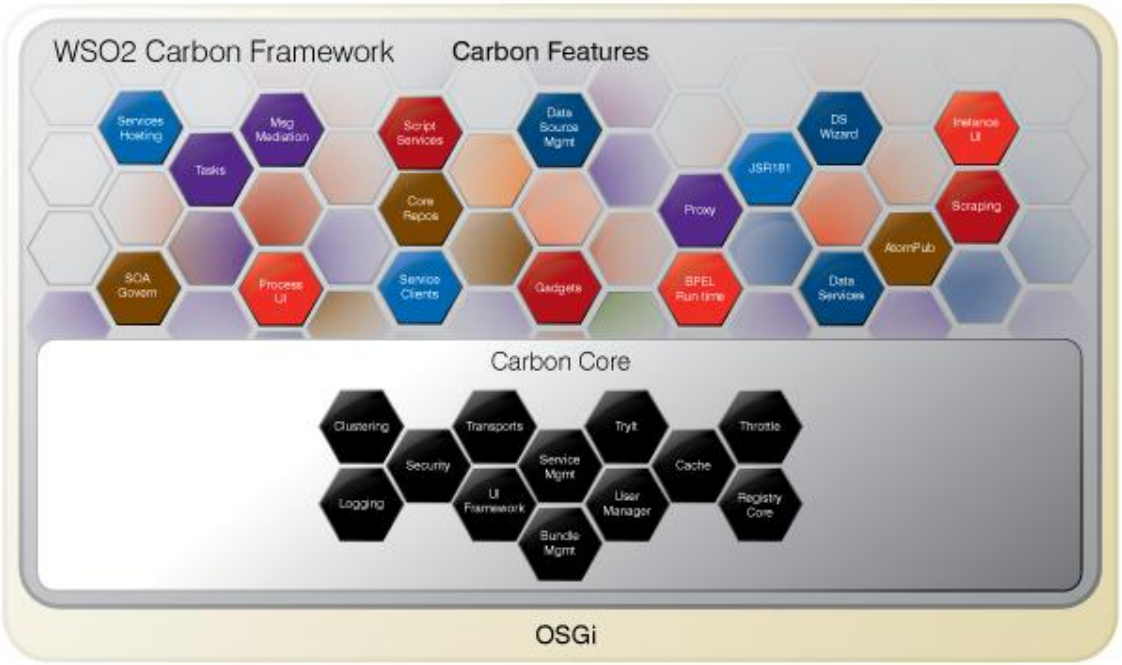

Figure 10 – WSO2 Carbon Framework Architecture [23]

The core consists of a set of common middleware components that are useful in any enterprise project, while additional components can be added to solve a specific enterprise scenario. It provides WSO2 middleware products with a consistent set of enterprise-class management, security, clustering, logging, statistics, tracing, throttling, caching, and other capabilities as well as a management UI framework [23]. The central component is a solid and high performance SOA and Web services engine. When components are added, associated management components are automatically added to the UI. The framework supports a front-end and back-end separation so that everything can be controlled through a remote WSO Carbon UI or through a Web services interface.

When setting up an environment that is based on WSO2 Carbon, one could build the Carbon core, pick the components that are needed and integrate them into the Carbon build, having a customized product. Another way would be to choose one of the Carbon based products that reflect common middleware product categories like an ESB, WSAS or a Mashup Server and extend it with additional required functionality (illustrated by Figure 11).

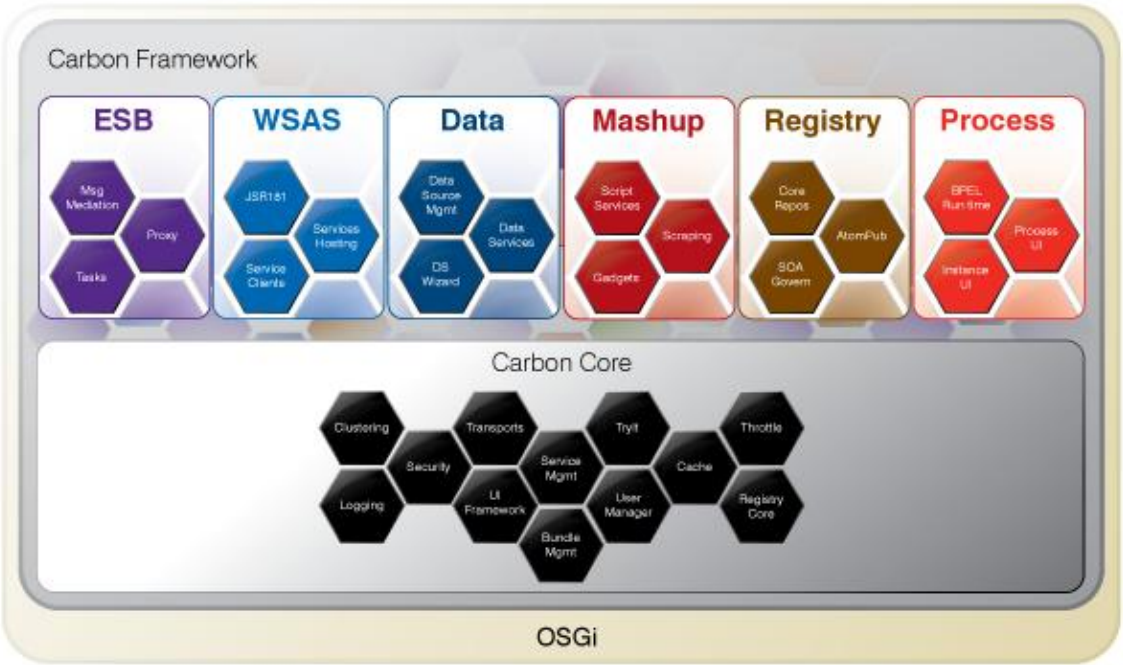

Figure 11 – Carbon-based products [23]

4.2 Management Console

One of the rich features of the Carbon framework is the management UI framework. Once WSO2 Carbon is installed and started, the management console is accessible from the URL https://localhost:9443/carbon. Pointing to this URL in a web browser will open a sign-in page. The default username is *admin* and the password is also *admin*. After login the management console presents a components menu on the left side where installed components can plug in their UI. Figure 12 shows the menu of the

Business Process Server. The Business Process Server has a separate menu section called *Business Processes* that allows managing processes and instances as well as deploying new processes (see chapter 4.3.1 – Managing).

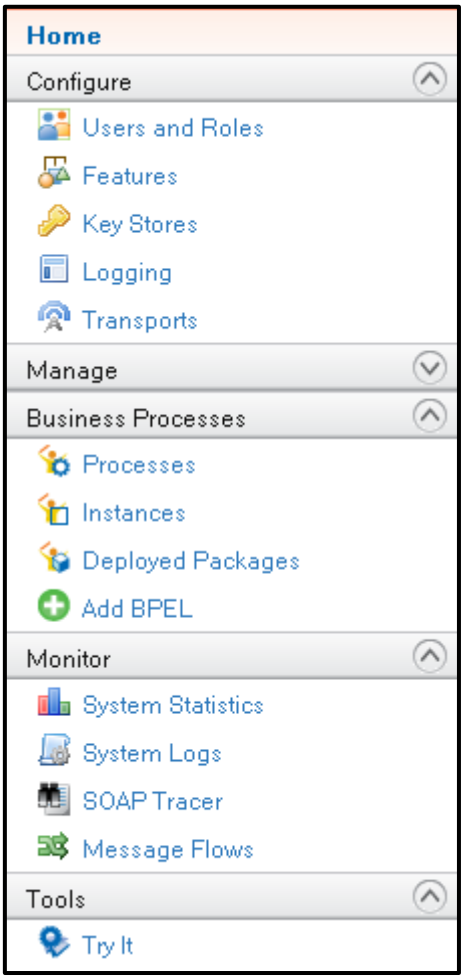

Figure 12 – Management Console Components Menu

Two important configuration options that are available in all Carbon products are the *Users and Roles* and the *Features* menu.

4.2.1 User and role management

The Users and Roles menu allows administrators to manage users and roles. The *user store* in Carbon can operate in two modes, “Read/Write” mode and “Read only” mode. In read/write mode one can add, modify and remove user accounts, reset passwords, manage roles of users and bulk import users from other user stores, while in read only mode it is only possible to view user accounts. Figure 13 and Figure 14 show how to manage users and roles. Users can also be imported from other user stores, e.g. from a relational database or a LDAP server. Carbon allows importing of comma separated lists (*.csv* files) or Excel sheets (*.xls* files). There are many visual LDAP tools available that support exporting to the .csv format.

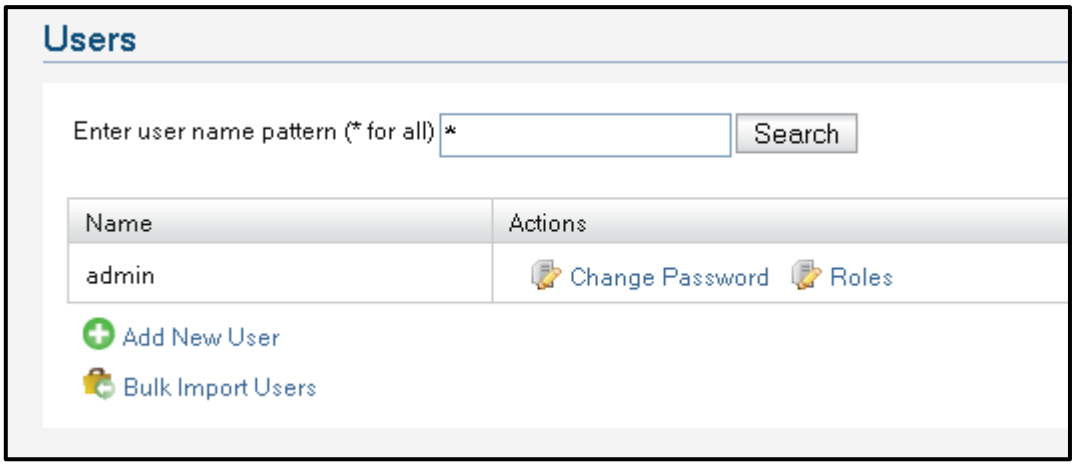

Figure 13 – Managing users in WSO2 Carbon

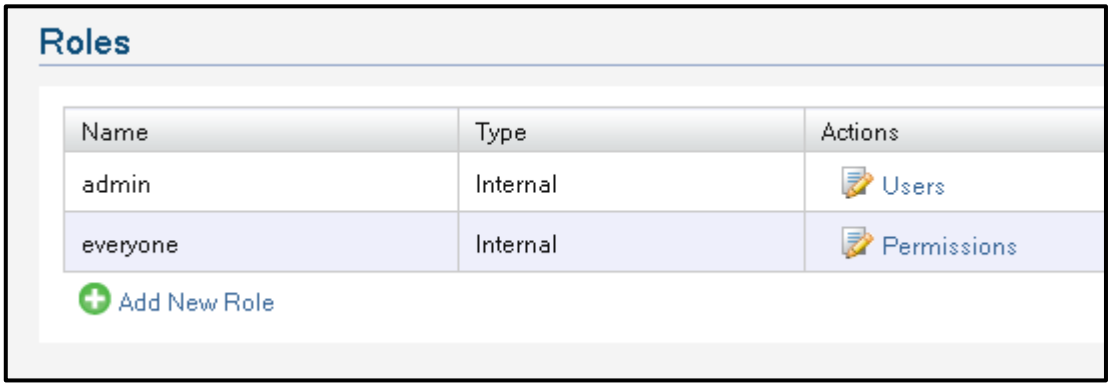

Figure 14 – Managing roles in WSO2 Carbon

4.2.2 Feature management

Another feature that comes with WSO2 Carbon is the feature manager facility that helps developers to customize Carbon. The possibility to install, update and uninstall features is called provisioning. This can be done manually by dropping bundles and configuration files into certain folders of the framework, but this method is not recommended because it is error-prone. Components are depending on each other and finding the exact set of components is difficult. To handle these issues the developers integrated Equinox P2 with Carbon [23].

Equinox P2 [24] is a framework for provisioning Eclipse-based applications, but it can basically be used as a provisioning platform for any OSGi-based application. The integration of P2 into Carbon enables users to extend the framework by installing various features. The feature manager provides a GUI to perform these actions. Figure 15 shows part of the feature management console.

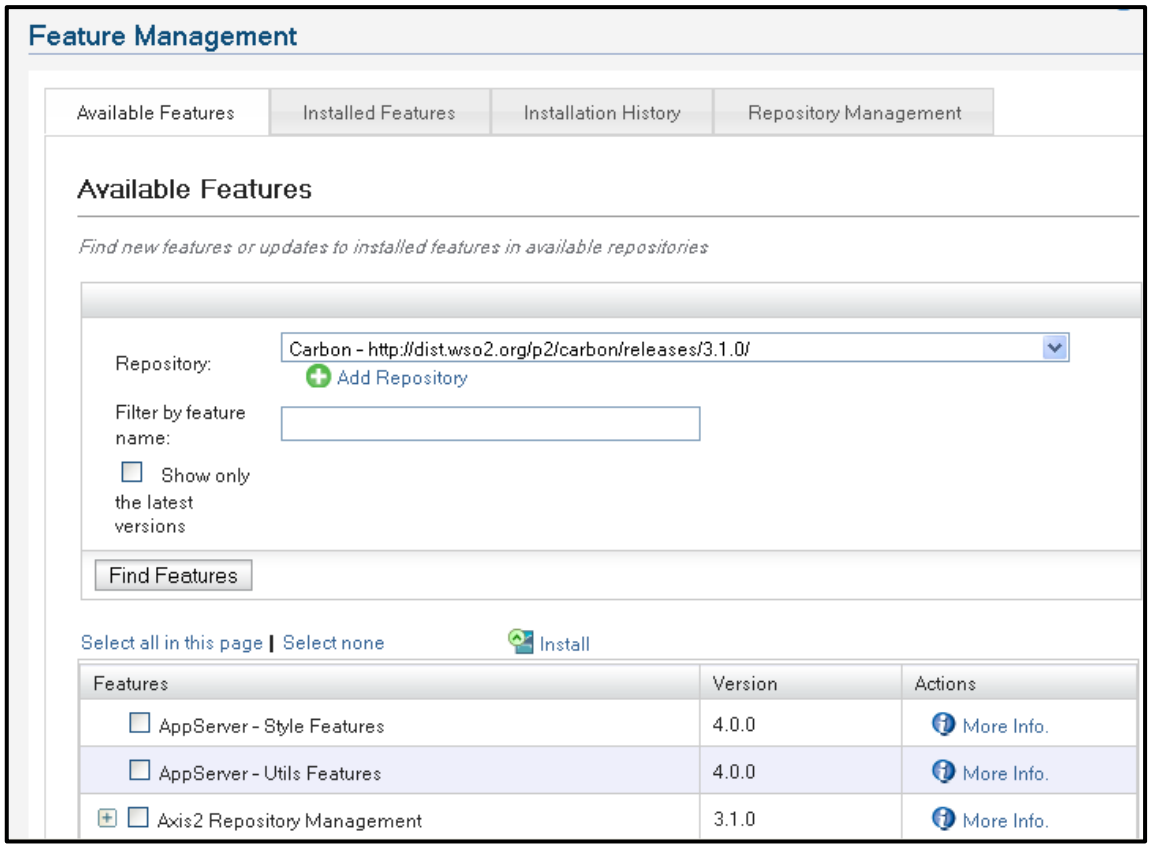

Figure 15 – Managing features in WSO2 Carbon

Carbon differentiates between repositories, features and components. A repository consists of one or more features and a feature can consist of one or more components (see chapter 4.4 – Developing components).

Before installing a new feature one has to specify a repository that contains the feature. For example, the standard repository of all the features of the Carbon 3.1.0 release is http://dist.wso2.org/p2/carbon/releases/3.1.0/. Clicking the “Find Features” button will list all the available features of a repository. The required features can then be selected and installed.

Afterwards, Carbon shows the install details with the installation size. As some features may depend on others, Equinox P2 would automatically complain that an installation could not be performed, if a dependency is missing. Finally, the user has to accept the license agreements and installation will finish after a restart of the server.

The feature will be added to the installed features list in the *Installed Features* section as illustrated by Figure 16. Of course each feature can also be uninstalled, but only if there is no other feature depending on it. Not only can a feature be uninstalled but the front-end/back-end separation (see chapter 4.4 – Developing components) allows one to install or uninstall just the front-end or back-end feature. The list allows filtering of components so that the front-end or back-end components are listed only.

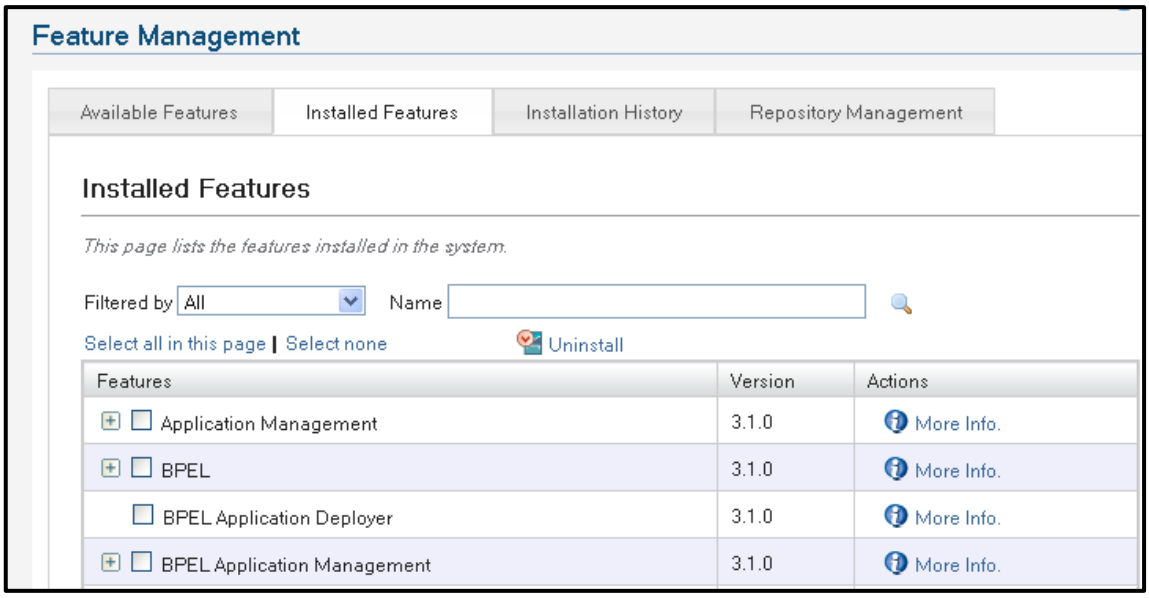

Figure 16 – Managing installed features in WSO2 Carbon

The last two sections of the feature management console are the *Installation History* and the *Repository Management*. The installation history shows the previous states of the framework and provides the possibility to revert the current configuration to a previous configuration. When a configuration is selected for an undo, the changes that the configuration will do are shown, e.g. the installed and uninstalled features.

Note: Equinox P2 does not delete OSGi bundles when they are uninstalled, so the bundles will not have to be downloaded when they are selected for installation again.

The repository management section lists the available repositories and allows one to add, remove, modify and disable repositories. One can either provide a URL to the repository or the path of the directory if the repository was downloaded to the local file system.

4.3 WSO2 Business Process Server

The WSO2 Business Process Server (BPS) is one of the Carbon-based products. It is open source and can execute business processes written following the WS-BPEL standard. Powered by Apache ODE the WSO2 BPS provides a web-based graphical console to deploy, manage and monitor business processes and process instances [25].

Apache ODE is one of the most popular BPEL engines, especially because it is open source. Running Apache ODE as a standalone server would make it impossible to use some functionalities and features provided by WSO2 Carbon, so they build their own Business Process Server by creating an Apache ODE integration layer. The integration process was straightforward because the Web services based transport in ODE is implemented on top of Apache Axis2, as is the Carbon core [26].

Through the component-based architecture of the carbon framework, BPEL support can be added to any Carbon-based product, e.g. one can add BPEL support to WSO2 ESB. Figure 17 shows the Business Process Server in context with other products and technologies.

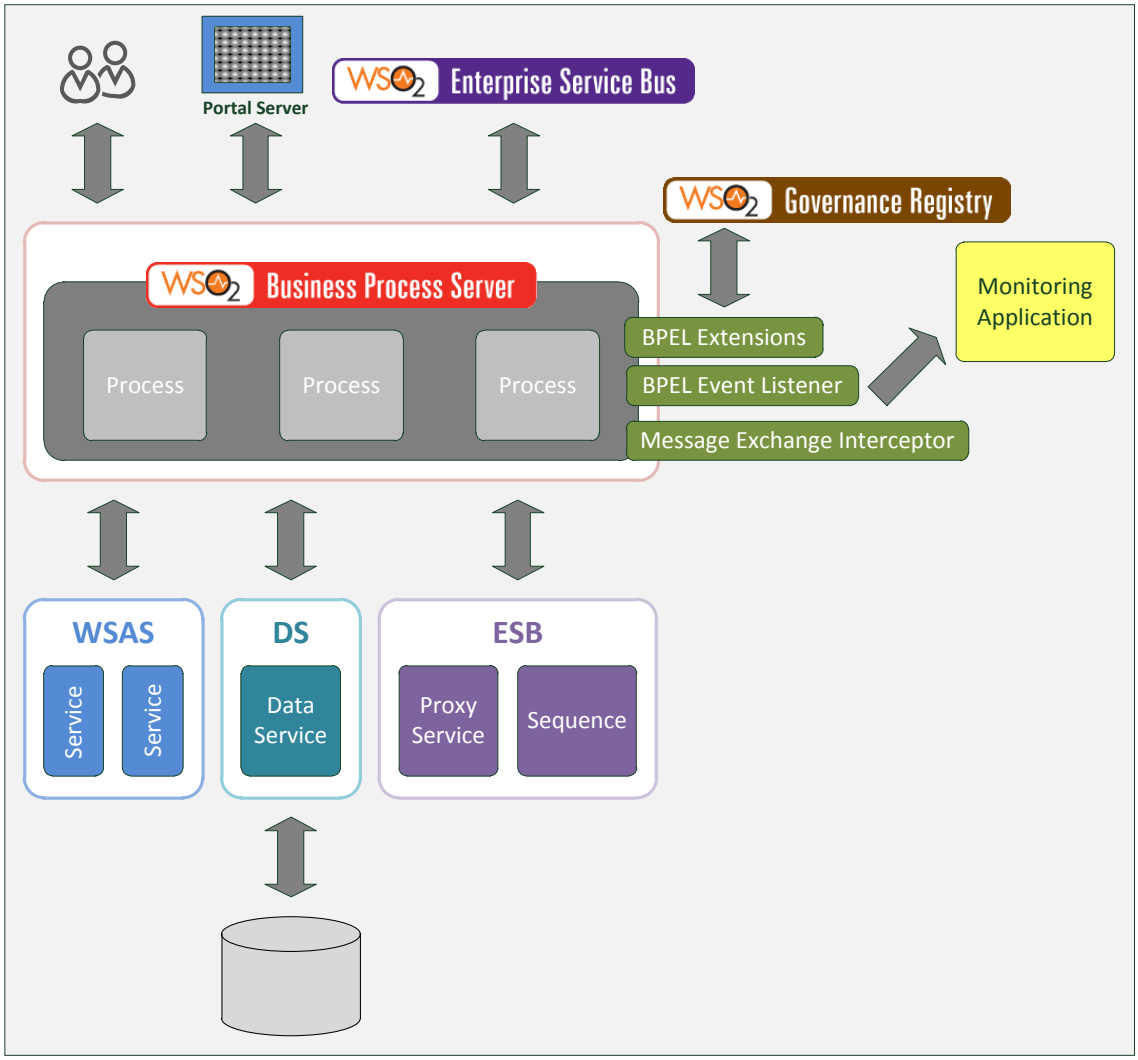

Figure 17 – WSO2 Business Process Server [27]

The latest addition to the Business Process Server is the multi-tenant capability. In early 2010 the developer team added Human Task support with moderate success because of a lack of tooling and support from the underlying platform. Current development is focusing on a multi-tenanted Human Task implementation with Eclipse-based tooling support [26].

4.3.1 Managing the BPS

The BPS already comes with a UI component to manage process models and instances and to install new BPEL processes. All these features are available in the *Business Processes* category of the components menu on the left of the management console. When selecting the *Processes* menu item, the management console shows a list of the deployed process models with some additional information like deployment date. The Instances menu item shows a list of process instances, also with some additional information like id, status and start date. Both lists are comparable to the lists provided by the Business Process Illustrator. Figure 18 and Figure 19 show the lists of processes and instances.

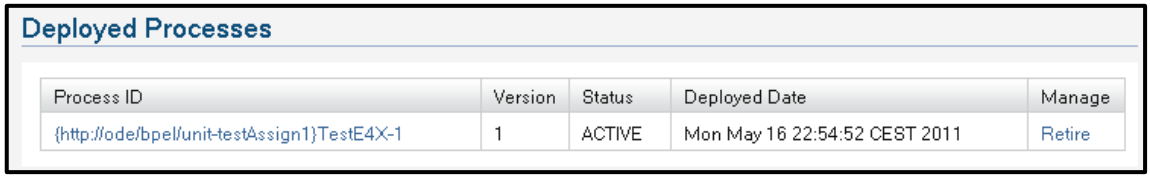

Figure 18 – Managing BPS processes

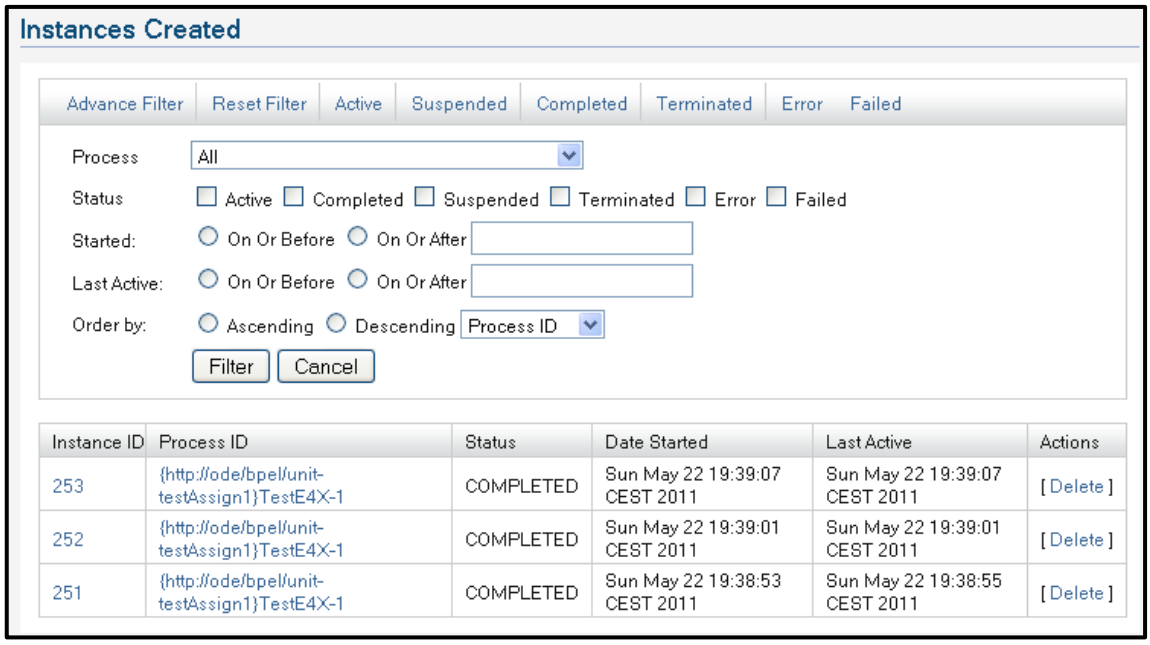

Figure 19 – Managing BPS instances

The managing component also provides more specifics to each process instance when pointing to the link in the *Process ID* column of either of the two lists in Figure 18 or Figure 19. The *Process Information* page shows a lot more information of a process model, e.g. a summary of the instances of the current selected process model. A chart on the right visualizes the number of instances in their different states, i.e. *Active*, *Completed*, *Error*, *Failed*, *Suspended* and *Terminated*. The page also shows the process definition and a graphical representation of the model (Figure 20 and Figure 21).

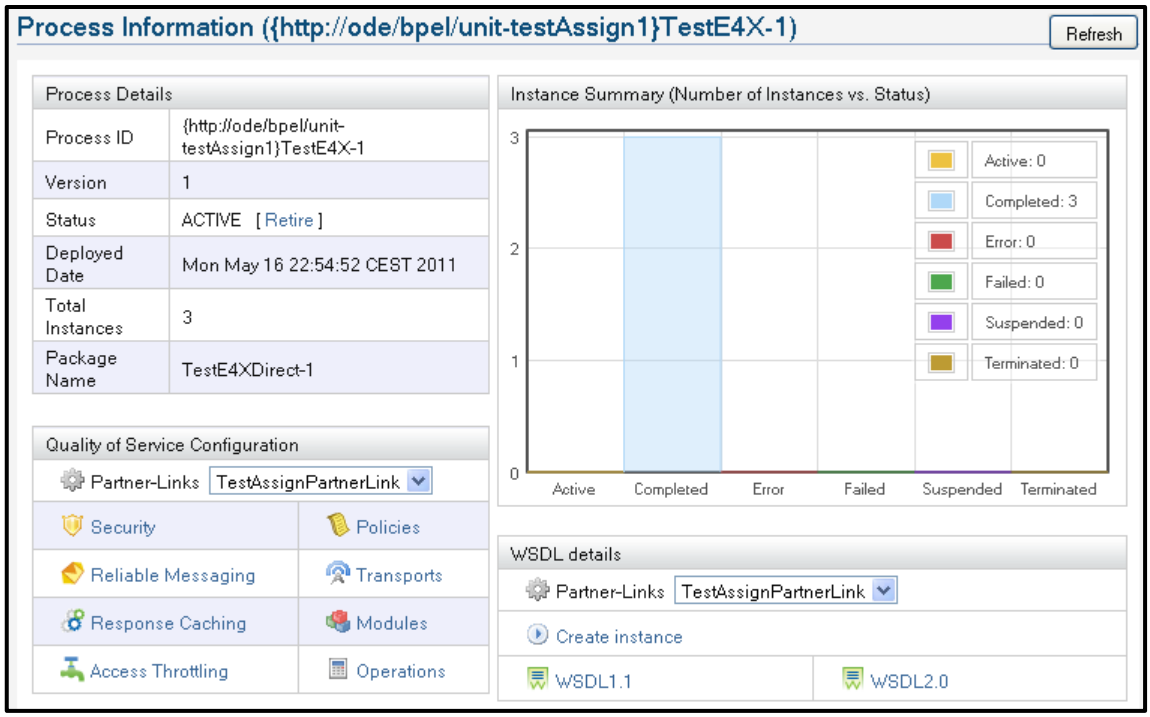

Figure 20 - Process Information

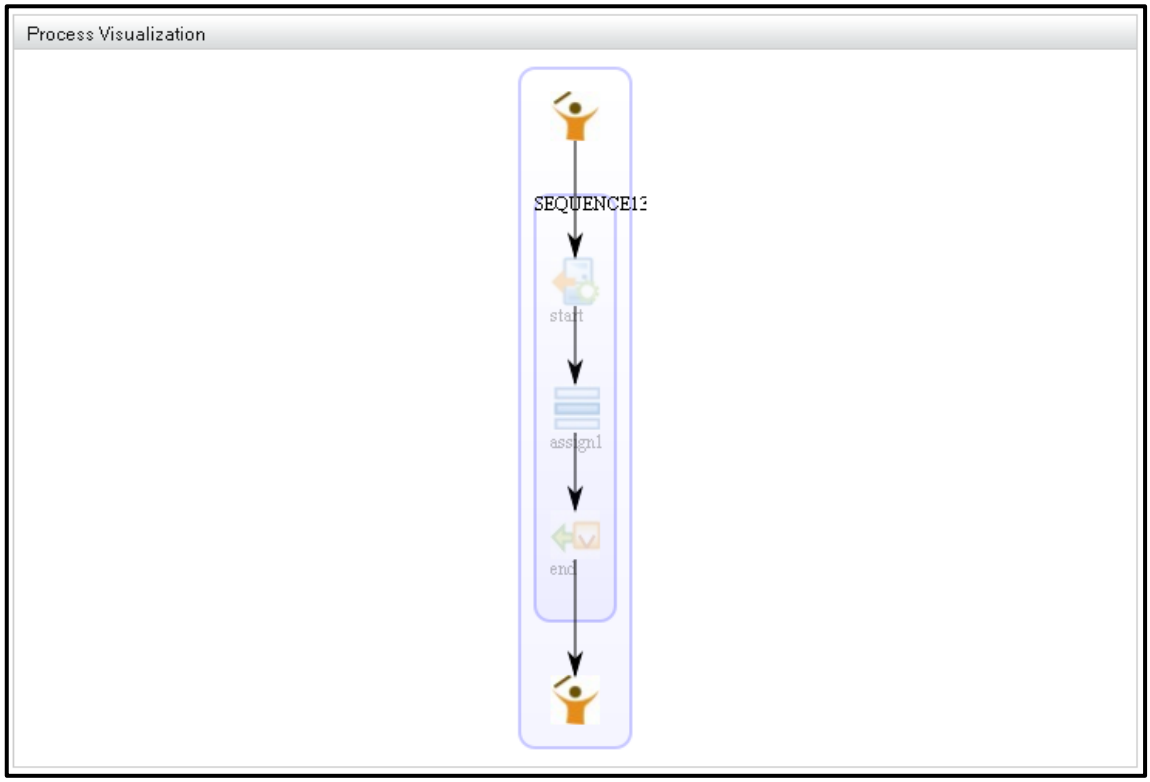

Figure 21 - Process visualization

There is also the possibility to create instances of a process model by calling the *TryIt* service. The TryIt feature is one of the core components of Carbon and enables one to test Web services by generating a webpage with the required input fields.

4.3.2 Monitoring the BPS

The Business Process Illustrator is a monitoring tool and the core of this student thesis is to connect the Business Process Illustrator to the Business Process Server. This requires understanding the provided mechanisms to monitor the Business Process Server.

The Apache ODE Adapter of the Business Process Illustrator uses two Web services to get the process models and instances. To get the activity events, it accesses the underlying database directly, because this functionality is not yet provided by the Web services. Apache ODE does not run as a standalone application inside the framework. It had to be adapted to be able to run inside Carbon. This also means that the provided API of the Apache ODE engine does not work anymore.

The Business Process Server provides a different API to read the process models and instances. It even provides the possibility to get the activity events through the API, instead of accessing the database directly. All these functions are available through different Web services called *process management* and *instance management*. The usage of these services is explained in the implementation chapter.

4.4 Developing components and features

The previous chapters already explained how to manage features in WSO2 Carbon. This chapter focuses on component and feature development [28].

The concept of features in Carbon is very similar to the concept used in the Eclipse IDE. In Eclipse, a feature is a grouping of logically related plug-ins (bundles). They can be installed using the *Update Manager*. In Carbon, a feature can be thought of as an installable form of one or more logically related carbon components. They can be installed using the *Feature Manger*.

Features are units that can be installed in any Carbon-based product and shared with others by packaging them as a repository (a repository can be compared to an update site in Eclipse). Features allow one to specify the requirements of the Carbon component, e.g. dependencies to other features and bundles.

A component is a set of OSGi bundles living in the Carbon framework. The best-practices when developing components suggest a separation of front-end and back-end component as illustrated by Figure 22.

The development process is divided into three steps.

1. Develop the front-end and back-end components
2. Develop the corresponding features, i.e. front-end (UI) feature and back-end (Server) feature and a composite or aggregated feature that contains both features.
3. Install the feature into Carbon by developing a feature repository and installing it using the feature manager.

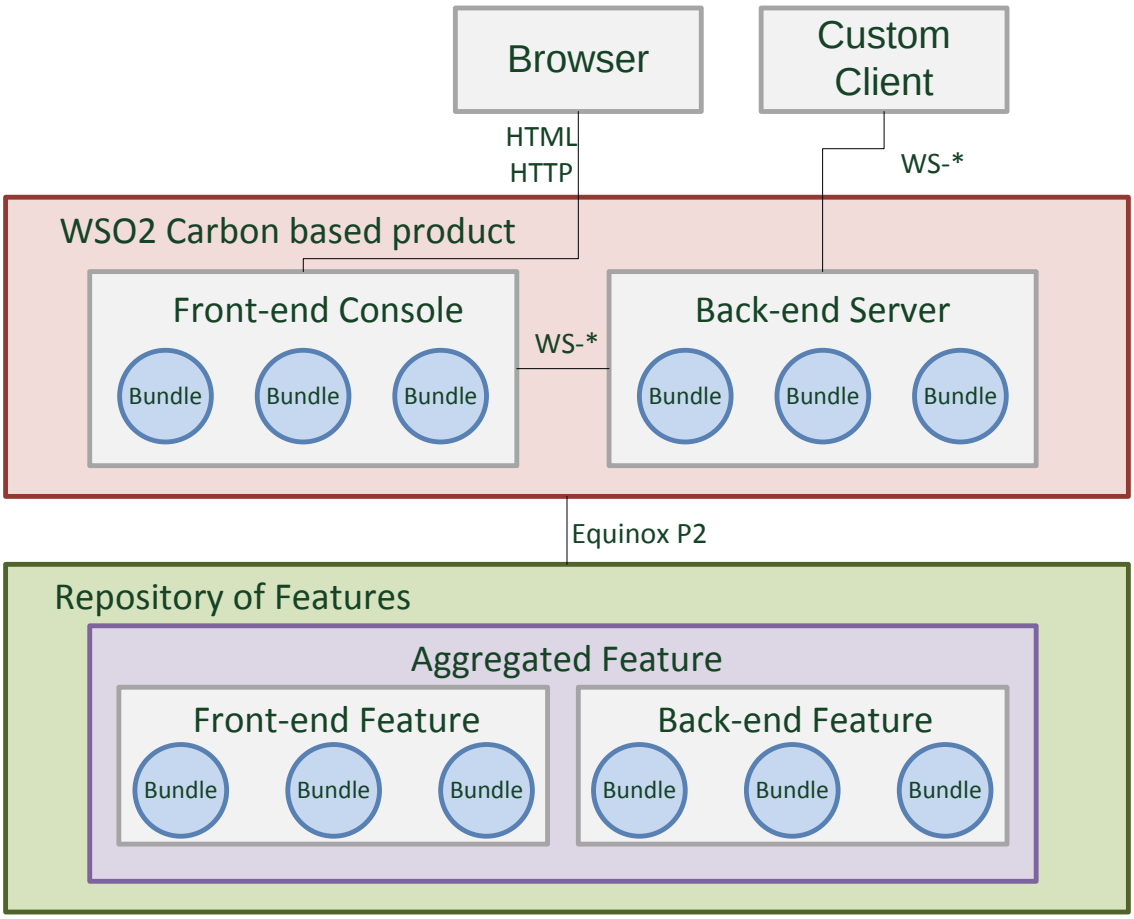

Figure 22 – Big Picture of Components and Features (adapted from [28])

Note: This is just a general guideline and depends on the Carbon component one develops. It is possible to create components that just have a front-end or back-end component.

The figure illustrates the difference between front-end and back-end component. A client uses a browser to connect to the front-end component. The front-end component translates client requests into Web service calls to the back-end component. Of course a client could also call the Web services of the back-end component directly through a custom client.

The bottom of the figure shows the aggregated feature installed in a repository. A feature can also be thought of as an installable form of a component, while the component is the functionality that is developed.

Maven provides the plugins to create the components, features and repositories. The maven bundle plugin is used to create the components while the maven p2 feature plugin is used to create the features and the repository. The implementation chapter explains the usage of these plugins.

5 Integration Architecture

The previous chapters have covered the different technologies and tools that are used for the integration process. The analysis of the Business Process Illustrator and the Business Process Server has already indicated some problems for the integration process. This chapter explains the problems and the resulting decisions that had to be made, and led to the integration architecture described in this chapter.

5.1 Frontend – JavaServer Faces and Facelets

One of the first problems when writing some test examples was the missing support of JavaServer Faces and Facelets in an OSGi environment. At the beginning of this writing, there was no working OSGi bundle of the Apache MyFaces implementation of JavaServer Faces. The complexity of the MyFaces implementation even caused Apache developers to question, if MyFaces is OSGi-fyable at all. The problem is that class loading in an OSGi environment is handled differently than in a standard Java environment. Though there are some examples, where developers were able to get JavaServer Faces running in an OSGi environment to a certain degree, they all had some code changes of the MyFaces implementation involving.

As these changes were questionable, the decision was to create a prototype that does not contain any JavaServer Faces, but was made out of JavaServer Pages. Almost every UI component inside the Carbon framework is made out of JavaServer Pages and Servlets. This decision is also based on the fact that the core functionality of the Business Process Illustrator – the possibility to apply process views – is not affected by these changes. The graphical representation of a process is generated by Servlets and hence will also work inside the Carbon framework. The changes mainly affect the generation of the process model and instance lists. At the same time, the goal is to keep the current graphical design of the GUI of the Business Process Illustrator.

5.2 BPI Service Adapter (BPS)

The Business Process Adapter is based on Apache ODE, so the first impression was that the current service adapter of Apache ODE could possibly work for the Business Process Illustrator. A more thorough analysis showed that Apache ODE hat to be adapted in order to be integrated with the Carbon framework. These changes also included discarding the old API of Apache ODE and creating a new one. This means that a new adapter has to be written for the new API.

The API of the Business Process Server provides additional information about the process instances that the API of the Apache ODE did not provide. This means that activity events can be retrieved directly through the provided Web services of the API,

instead of a directly accessing the underlying database, hence making the adapter simpler than the one provided for Apache ODE.

5.3 Code Changes

The Business Process Server has a modular architecture. Modular meaning, the frontend can be exchanged and still use the functionality provided by the BPI Service, or the used WfMS can be changed by using a different BPI service adapter.

During the development of the BPI, this was considered very important, as the tool should be able to be used with other WfMS. The only developed and provided adapters are the ones for Apache ODE and a test adapter for developing purposes. Creating another adapter for the Business Process Server revealed some restrictions in the code that made it difficult to impossible to create an adapter for the Business Process Server. This concerns the Business Process Server, but other workflow management systems could work perfectly with the current version of the BPI.

The current version of the BPI expects the process definition of a deployed process to be available via a URL. The BPEL file has to be parsed, in order to create an internal representation of a process model. When processes are deployed in Apache ODE, the BPEL file is accessible via a URL, and this assumption was made on other WfMS as well. The Business Process Server – and possibly many others – does not provide this URL. Of course, the process definition is available as a string-representation through calls to the API, but not the BPEL file itself via a URL. It became clear that the BPI cannot depend on the WfMS to provide the process definition as a BPEL file via a URL, though the process definition still must be available in some way, that the parser can parse it.

Another problem is the loading of the properties file through a helper class in the current version. The location of the file differs, depending on the current environment the BPI runs in. Different environments are already covered by the code but the OSGi environment was not included. This is probably a similar problem to the class loader problem with the Apache MyFaces implementation. The new solution adds OSGi environment support to the code but other environments may not be covered by this approach. The changes in the BPI code only extend the functionality without affecting the BPI in its currently working environments. They can be thought of as bug fixes for a version update.

5.4 Resulting Architecture

The resulting integration architecture is shown in Figure 23. Newly developed components and code adjustments are indicated by a yellow painting. The image shows the old architecture above the new architecture to compare the changes.

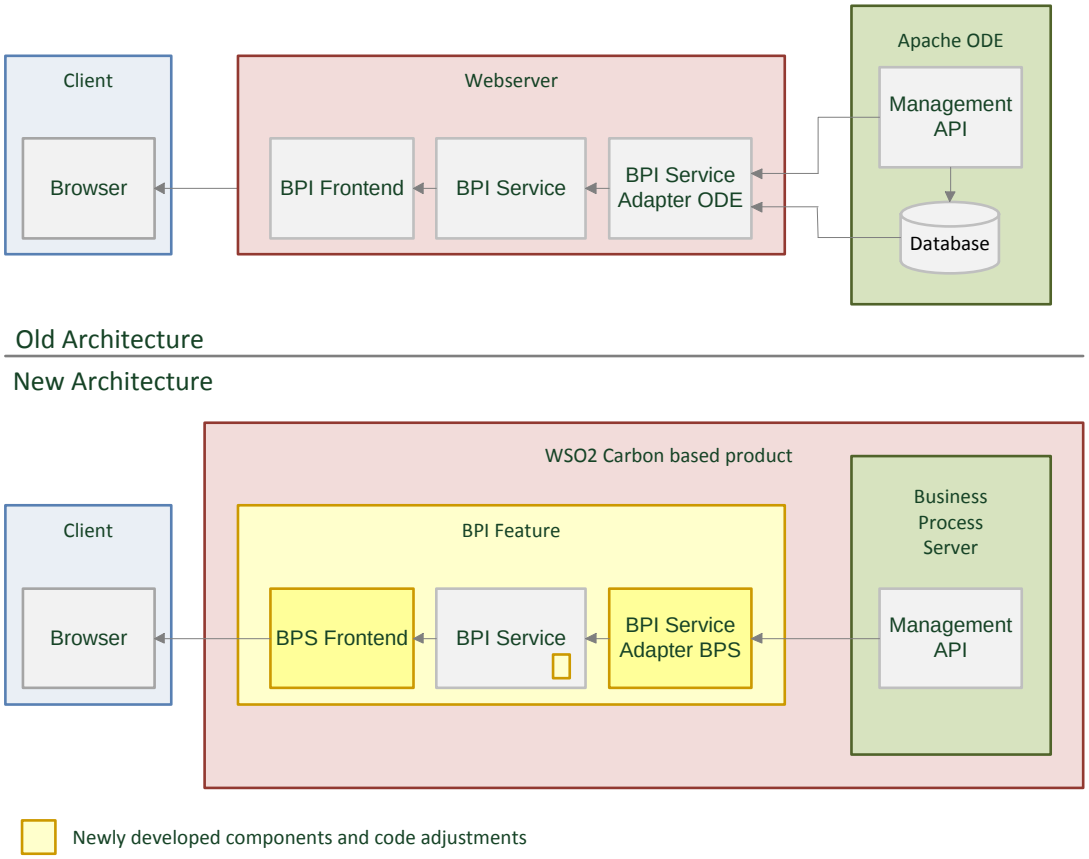

Figure 23 – Integration Architecture

The *BPS Frontend* (compared to BPI Frontend) is the new frontend that is developed because the old one with the Facelets will not work. The same applies for the *BPI Service Adapter BPS* (compared to BPI Service Adapter ODE) because the one for Apache ODE does not work anymore. In the bottom right corner of the *BPI Service* is a small yellow rectangle that indicates that there are some code changes in this component. This is only a very small part of the BPI Service and will also be contained in the general BPI release. The *BPI Feature* illustrates that all the functionality has to be packaged as an OSGi bundle, more precisely a Carbon component (and a Carbon feature) that can be installed into the WSO2 Carbon framework.

The blue parts of the image show the client side, the red parts illustrate the server side and the green parts show the workflow management systems. Note that the WfMS in the new architecture – the Business Process Server – is not a standalone application like Apache ODE in the old one, but it is contained as a feature in a WSO2 Carbon-based product.

Note: Business Process Server is a Carbon-based product, so when downloading the Business Process Server, one actually gets the Carbon framework plus the features that make up the Business Process Server. That is why the Business Process Server is illustrated as a feature inside the Carbon framework in the image above.

There is also no database shown in the Business Process Server as all the information is directly available through the API.

6 Implementation

The previous chapter explained the architecture of the integration process. This chapter focuses on the implementation aspects, by describing which Java classes are created and what their functionality is, as well as the JavaServer Pages and the OSGi bundles that are created.

6.1 Frontend

Usually, developers would consider moving from a JavaServer Pages project to a JavaServer Faces project, but in the integration process of this student thesis, the opposite is the case. The project is a JSF project and must be ported to a JSP project, or at least some basic functionality must be available in the OSGi version. If the graphical design of the current project is desired then there is a simple and obvious solution for small projects like the Business Process Illustrator: let the project with the JavaServer Faces run on a server and let it generate the main page (by issuing a client request). Looking at the code of the generated page, one can easily see that the page contains almost only HTML code. This is just a current static “image” of the dynamic page on the server, e.g. the process model list in the Business Process Illustrator shows the first 10 models when the page is initially opened (illustrated by Listing 24).

```
<tbody>
    <!-- First Entry -->
    <tr class="hover light-orange">
        <td>1</td>
        <td>Process Model 1</td>
        <td>1.0</td>
        <td><img
                src="./resources/icons/status/processModel/active.png"
                alt="Active" title="Active" />
        </td>
        ...
    </tr>
    <!-- Second Entry -->
    <tr class="hover light-orange">
        <td>2</td>
        <td>Process Model 2</td>
        <td>1.0</td>
        <td><img
                src="./resources/icons/status/processModel/active.png"
                alt="Active" title="Active" />
        </td>
        ...
    </tr>
    <!-- 8 more -->
</tbody>
```

Listing 24 – Static "image" of a dynamic page

This code can be transformed into a dynamic version. As the listing showed, every entry has the same structure. The idea is to take one entry, enclose it with a for loop that loops over all process models, and exchange all the static values with the ones retrieved dynamically by calling the WfMS. Listing 25 shows a dynamic version of the former static one.

```
int i = 0;
for (ProcessModel processModel : processModels) {
    i++;
    %>
     <tbody>
        <tr class="<%=(i%2==0?"hover white":"hover light-orange")%>">
            <td><%=processModel.getPid()%></td>
            <td><%=processModel.getName()%></td>
            <td><%=processModel.getVersion()%></td>
            <td><img
                    src="./resources/icons/status/processModel/active.png"
                    alt="Active" title="Active" />
            </td>
            ...
        </tr>
    </tbody>
    <%
}
```

Listing 25 – Dynamic version of the code

The code iterates over all process models and the *id*, *name*, *version* and all the other information of a process model are dynamically placed in the code. This approach is applied for the instance list and the settings for the graph as well.

The generation of the SVG graph was originally done through the *SVGServlet*. The same applies for the new OSGi version of the frontend, but the servlet has to be adapted, because the original servlet uses the managed beans that are connected to the facelets. Adapting the code is simple. The process id and the instance id of the current selected instance are added as parameters to the request and the servlet generates the SVG code according to these parameters (see Listing 26).

```
String pid = req.getParameter("pid");
String iid = req.getParameter("iid");

try {
     BPIService service = new BPIServiceFactory().createService();

     if(!(iid == null || iid.isEmpty() || iid.equals("null"))) {
          ProcessModel pModel = service.getProcessModel(pid);
          ProcessInstance pInstance = pModel.getProcessInstance(iid);
          return service.getSVG(pInstance, new Settings());
     }
} catch(Exception e) {
     e.printStackTrace();
}
```

Listing 26 – Code changes in the SVGServlet

The servlet is included in the JSP page using the object tag as it was already used in the facelets.

Before the service component can call the process models and process instances form the workflow management system (using the provided web services), it needs to be logged in. The user is usually already logged in to the management console because most features require administrative roles to work. The same applies for the developed feature in this student thesis. The feature will not be listed in the components menu (see chapter 4.2 – Management Console) unless the user is logged in. When the user logs in, a cookie is created and this cookie has to be passed on to all the calls that require administrative rights. The cookie is stored in a class called *BPIUtil*, so it is available to the later described service adapter. Listing 27 shows the initialization of the JSP page. Once the cookie is saved, the service can call the process models.

```
String iid = request.getParameter("iid");
String pid = request.getParameter("pid");

String cookie =
      (String)session.getAttribute(ServerConstants.ADMIN_SERVICE_COOKIE);

BPIUtil.setLogged(true);
BPIUtil.setCookie(cookie);

try {
      BPIService service = new BPIServiceFactory().createService();
      Collection<ProcessModel> processModels = service.getProcessModels();
      ...
}
```

Listing 27 – Initialization of the JSP page

The resulting JSP page is called *index.jsp* and is automatically opened when the feature is selected from the component menu.

6.2 Service Adapter (BPS)

An adapter for a workflow management system has to implement three different interfaces (see chapter 3.2.3 – BPI Service Adapter (ODE)). The three Java classes that implement the interfaces are called *ProcessModelServiceImpl.java*, *ProcessInstanceServiceImpl.java* and *EventServiceImpl.java*. They all use another Java class called *BPIUtil.java* that handles login and authentication.

6.2.1 BPIUtil

WSO2 Carbon provides a web service called *AuthenticationAdmin* that can be used to login into Carbon. The BPIUtil.java class uses this web service to handle cases where no login was made before. A login object that holds the *username*, *password* and *remote address* is required by the web service. When a login attempt was successful, the web service creates a cookie that is stored in the BPIUtil class and used for later calls

to other web services, namely the *ProcessManagement* service and the *ProcessInstance* service. The code for the web service call is placed inside the *authenticate()* method of the BPIUtil class as shown in Listing 28.

```
public static boolean authenticate() throws AuthenticationExceptionException,
RemoteException {
	String serviceURL = backendServerURL +
		AUTHENTICATION_ADMIN_SERVICE;
	AuthenticationAdminStub stub =
		new AuthenticationAdminStub(null, serviceURL);

	/* Create a loginRequest */
	Login loginRequest = new Login();
	loginRequest.setUsername(USERNAME);
	loginRequest.setPassword(PASSWORD);
	loginRequest.setRemoteAddress(ADDRESS);

	Options option = stub._getServiceClient().getOptions();
	option.setManageSession(true);

	/* Call Web-Service */
	LoginResponse loginResponse = stub.login(loginRequest);

	isLogged = loginResponse.get_return();
	if (isLogged) {
		/* Save cookie */
		cookie =
		    (String) stub._getServiceClient().getServiceContext().getProperty(
			HTTPConstants.COOKIE_STRING);
	}

	return isLogged;
}
```

Listing 28 – Authentication method of the BPIUtil class

The AuthenticationAdminStub in the code listing is created by the maven-antrun-plugin (described in chapter 6.3 – UI Component). Once an instance of the stub is created, a login request is send to the service. If the login attempt was successful, the cookie is saved. Finally the status of the login attempt is returned.

6.2.2 ProcessModelServiceImpl

The ProcessModelServiceImpl class has almost nothing in common with the implementation provided for the Apache ODE. To call the process models from the workflow management system, the ProcessModelServiceImpl for the WSO2 BPS uses a stub that is also generated by the maven plugin from the provided WSDL files of the management API – in this case the *process_mgt.wsdl*. There are two functions of the web service that are of interest: *getPaginatedProcessList()* and *getProcessInfo()*. The getPaginatedProcessList method returns a list of all the processes, each of them containing limited process information, e.g. the version of a process model. To get all the information needed, one has to call the getProcessInfo() method that returns additional

information not available in the limited information type, e.g. the process definition. Listing 29 illustrates the usage of the two functions.

```
try {
      processList = getPaginatedProcessList("name}}* namespace=*",
            "deployed name", 0);
} catch (RemoteException e) {
      e.printStackTrace();
}

LimitedProcessInfoType[] processes = processList.getProcessInfo();
for (LimitedProcessInfoType process : processes) {

      ProcessInfoType processInfo = null;
      try{
            processInfo = getProcessInfo(process.getPid());

            ProcessModel processModel = new ProcessModel(
                  process.getPid(),
                  processInfo.getDefinitionInfo().getProcessName().toString(),
                  process.getVersion(),
                  mapToStatus(process.getStatus().getValue()),
                  processInfo.getDefinitionInfo().getDefinition().
                        getExtraElement().toString());

            processModels.add(processModel);
      } catch(Exception e) {
            e.printStackTrace();
      }
}
```

Listing 29 – Calling the process models from the WfMS

After retrieving the process list from the WfMS, each process model is saved into the internal process list of the BPI. This is done by looping over all process models and calling additional information for each process model from the WfMS. Before these calls can be made, the user must be authenticated, of course.

6.2.3 ProcessInstanceServiceImpl

The ProcessInstanceServiceImpl works almost exactly as the ProcessModelServiceImpl, except the WSDL file is called *instance_mgt.wsdl* and the methods are called *getPaginatedInstanceList()* and *getInstanceInfo()*. One thing to note here is that the getPaginatedInstanceList already provides almost all the information needed. The only missing information is the date of last error. This means, that there has to be a web service call for every process instance to get this little piece of information. Considering the fact that there are usually way more process instances than models in a production environment, this could be a noticeable overhead. The BPI would definitely be one of the beneficiaries of a modified version of the getPaginatedInstanceList method that provides this information in the first place.

6.2.4 EventServiceImpl

While the EventServiceImpl of the Apache ODE had to access the database directly, the EventServiceImpl for the WSO2 BPS adapter can use the provided API. The EventServiceImpl also uses the InstanceManagement web service that provides a function called *getInstanceInfoWithEvents()*. This method retrieves all activities and events of a provided process instance. Once all the activities are retrieved, a loop iterates over the activity list and creates an internal object of the activity in the BPI as illustrated by Listing 30.

```
InstanceInfoWithEventsType instance = getStub().getInstanceInfoWithEvents(
      Long.valueOf(processInstance.getIid()));
ActivityInfoWithEventsType[] activities = instance.getRootScope().
      getActivitiesWithEvents().getActivityInfoWithEvents();

for(ActivityInfoWithEventsType activity : activities) {
      EventInfo[] eventInfos = activity.getActivityEventsList().getEventInfo();
      for(EventInfo info : eventInfos) {

      ActivityStatusType status = null;

      if(info.getName().toLowerCase().contains("activityenabledevent")){
            status = ActivityStatusType.ENABLED;
      } else if(info.getName().toLowerCase().contains("activityexecstartevent")){
            status = ActivityStatusType.STARTED;
      } else if(info.getName().toLowerCase().contains("activityexecendevent")){
            status = ActivityStatusType.COMPLETED;
      } else if(info.getName().toLowerCase().contains("activityfailureevent")){
            status = ActivityStatusType.FAILURE;
      }

      /*
      * Create an ActivityExecEvent and add it to the list of activities if the event
      * was an ActivityEvent (status != null)
      */
      if(status != null) {
            ActivityExecEvent instanceEvent = new ActivityExecEvent(
                  activity.getActivityInfo().getName(),
                  mapToStatus(status),
                  eventInfo.getTimestamp(),
                  processInstance);
            activityEvents.add(instanceEvent);
      }
      }
}
```

Listing 30 – Calling events from the WfMS

The listing also shows the status mapping of external WSO2 BPS events to internal BPI events.

Note: the two sets of activity states do not match perfectly, e.g. both BPI and WSO2 BPS know the activity states ENABLED, STARTED, COMPLETED and FAILURE, but

the BPI knows the two additional states RECOVERY and SKIPPED, while the WSO2 BPS knows the state DEAD, which has no representation in the BPI.

6.3 UI Component

The whole project has to be packaged as an OSGi bundle (Carbon component). This is achieved through the use of Maven (see chapter 2.4 – Maven) and the provided plugins to package everything as a bundle.

6.3.1 Project Setup

The first thing to do is to set up a new Maven project by running the following command on the console.

```
mvn archetype:generate
-DgroupId=org.wso2.carbon.bpi
-DartifactId=org.wso2.carbon.bpi.ui
-DarchetypeArtifactId=maven-archetype-quickstart
```

Listing 31 – Creating a new Maven project

This will create a new project through the *archetype* plugin and the *generate* goal. The *groupId* of the plugin will be *org.wso2.carbon.bpi* and the *artifactId* will be *org.wso2.carbon.bpi.ui*. The *archetypeArtifactId* tells the plugin which project template to use, in this case a quick start project that already follows Maven's suggested folder structure. There are two files created automatically, called *App.java* and *AppTest.java.* They can be deleted, because they are not of any use for our task.

As development and coding is done in the Eclipse IDE, the project should be converted to an Eclipse project by running the command in Listing 32. This will create the Eclipse-specific project files.

```
mvn eclipse:eclipse
```

Listing 32 – Converting the project to an Eclipse project.

Note: Eclipse needs a plugin to be able to run Maven projects from the IDE. One of the plug-ins providing Maven integration for Eclipse is M2Eclipse [29].

Setting up the project also includes creating a resources folder that contains the web contents (i.e. the JSP files, images and style sheets), the WSDLs of the API, the properties file and a Carbon specific component.xml inside the META-INF folder. The packaging of the project has to be set to bundle.

6.3.2 Component Configuration

The Carbon framework provides a way to configure the behavior of a component inside the components menu through the *component.xml* file (illustrated by Listing 33)

```
<component xmlns="http://products.wso2.org/carbon">
    <menus>
        <menu>
            <id>bpi_menu</id>
            <i18n-key>bpi.menu</i18n-key>
            <i18n-bundle>
                org.wso2.carbon.bpi.ui.i18n.Resources
            </i18n-bundle>
            <parent-menu>bpel_menu</parent-menu>
            <link>../bpi/index.jsp</link>
            <region>region2</region>
            <order>50</order>
            <style-class>manage</style-class>
            <icon>../bpi/resources/icons/bpi.png</icon>
            <require-permission>
                /permission/protected/manage
            </require-permission>
        </menu>
    </menus>

    <servlets>
        <servlet id="SVG Servlet">
            <servlet-name>SVG Servlet</servlet-name>
            <url-pattern>/carbon/bpi/svg</url-pattern>
            <display-name>SVG Servlet</display-name>
            <servlet-class>
                net.latuske.bpi.frontend.SVGServlet
            </servlet-class>
        </servlet>
    </servlets>
</component>
```

Listing 33 – component.xml

The component will be added to the components menu when a menu tag is provided. A menu item needs different settings like an id, and icon or the link that is followed when the component was selected. There is also the possibility to specify the parent menu (here the *bpel_menu*). Another important setting is the require-permission setting. This specifies that only users with the given permission are allowed to access the component.

The BPI is a web application as is usually packaged as a WAR file with a WEB-INF folder that contains a web.xml. The web.xml can be used to specify the used servlets in an application. As OSGi bundles are packaged as JAR files, they usually do not know the web.xml file. Carbon allows specifying the servlets in the component.xml. Here, the only servlet needed is the SVG servlet.

6.3.3 Parent

The parent POM of the project is set to org.wso2.carbon.carbon-parent-3.0.0. This bundle is not included in the standard Maven repository so an additional repository has to be specified as illustrated by Listing 34. The repository is the one provided by WSO2.

```
...
<parent>
    <groupId>org.wso2.carbon</groupId>
    <artifactId>carbon-parent</artifactId>
    <version>3.0.0</version>
</parent>
...
<repositories>
    <repository>
        <id>wso2.org</id>
        <name>Mirror of http://dist.wso2.org/maven2/</name>
        <url>http://dist.wso2.org/maven2/</url>
        <layout>default</layout>
    </repository>
</repositories>
...
```

Listing 34 – Specifying a parent and a repository

6.3.4 Dependencies

The project basically only depends on two bundles. The *org.wso2.carbon.ui* bundle is needed because the bundle is a Carbon UI component. To be able to call the web services, the project also depends on the *org.apache.axis2.wso2.axis2* bundle. Listing 35 illustrates the use of dependencies in the project's POM.

```
...
</dependencies>
    <dependency>
        <groupId>org.wso2.carbon</groupId>
        <artifactId>org.wso2.carbon.ui</artifactId>
        <version>3.0.0</version>
    </dependency>

    <dependency>
        <groupId>org.apache.axis2.wso2</groupId>
        <artifactId>axis2</artifactId>
        <version>1.6.0-wso2v1</version>
    </dependency>
</dependencies>
...
```

Listing 35 – Dependencies of the UI component

6.3.5 Stubs Generation

Before calling the web services, the stubs that actually produce the SOAP messages have to be created. The two plugins that support the creation of stubs are the *maven-antrun-plugin* and the *build-helper-maven-plugin*. The usage of the maven-antrun-plugin is showed in Listing 36.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-antrun-plugin</artifactId>
    <version>1.1</version>
    <executions>
        <execution>
            <id>source-code-generation</id>
            <phase>process-resources</phase>
            <goals>
                <goal>run</goal>
            </goals>
            <configuration>
                <tasks>
                    <!-- Tasks -->
                </tasks>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Listing 36 – Maven-antrun-plugin

The maven-antrun-plugin can be used to run basically any type of Ant task. In this example, the plugin is used to call the WSDL2Java tool that creates the stubs.

The *execution* section specifies the goal that should be executed (here the *run* goal). The *phase* specifies in which phase of the build lifecycle the goal should be run, in this case the *process-resources* phase. In the task section are three tasks contained – one for each of the three used web services (*AuthenticationAdmin*, *ProcessManagement* and *InstanceManagement*). Listing 37 shows an example of the generation of the AuthenticationAdmin stub. The same applies for the generation of the other web services.

```
<java classname="org.apache.axis2.wsdl.WSDL2Java" fork="true">
    <arg
        line="-uri src/main/resources/AuthenticationAdmin.wsdl -u -g
        -sn AuthenticationAdmin
        -o target/generated-code -p org.wso2.carbon.bpi.adapter
        -ns2p
            http://authentication.services.core.carbon.wso2.org/xsd
                =org.wso2.carbon.bpi.adapter.types,
            http://authentication.services.core.carbon.wso2.org
                =org.wso2.carbon.bpi.adapter.types" />
    <classpath refid="maven.dependency.classpath" />
    <classpath refid="maven.compile.classpath" />
    <classpath refid="maven.runtime.classpath" />
</java>
```

Listing 37 – Generating a stub

The classname specifies that the WSDL2Java tool should be run and the arg line specifies the arguments for the tool. The argument line contains references to the source WSDL file, the output folder of the task, the package name of the stub and the mappings of namespaces to packages.

The generated files will be available in the *target/generated-code* folder. The build-helper-maven-plugin allows one to add more source directories to the POM. In this case, the sources of the generated stubs should be added to the POM, as illustrated by Listing 38.

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>build-helper-maven-plugin</artifactId>
    <executions>
        <execution>
            <id>add-source</id>
            <phase>generate-sources</phase>
            <goals>
                <goal>add-source</goal>
            </goals>
            <configuration>
                <sources>
                    <source>target/generated-code/src</source>
                </sources>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Listing 38 – Build-helper-maven-plugin

Once the stubs are generated, the plugins will not have to be called again and can be commented. This can only be applied, if the project is run without the *mvn clean* command, because this will delete all the generated stubs.

6.3.6 Bundle Creation

Another very important plugin is the *maven-bundle-plugin*, which packages everything as an OSGi bundle. The plugin automatically sets all the required headers in the manifest according to the configuration of the plugin. Hence the developer does not have to deal directly with the creation of the manifest. Listing 39 shows the usage of the plug-in. The version of the plugin that is used is 1.4.0, although there are newer versions available already. These versions caused different problems, and developers at WSO2 use version 1.4.0 as well.

The instructions section contains the settings that are used for the settings in the manifest. *Bundle-SymbolicName* receives the same value as the project's artifactId. These project properties can be referenced using the $-Symbol, e.g. *${project.artifactId}* resolves to the artifactId of the project. Export-Package and Import-Package specify the packages that are exported and imported by the bundle, e.g. the *net.latuske.bpi.** and

the *org.wso2.carbon.bpi.** package are exported. The *DynamicImport-Package* header takes care that not-specified import-packages are resolved at runtime. *Carbon-component* is a Carbon specific header that tells the framework what kind of component the bundle is. In this case, the component is a UIBundle.

```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <version>1.4.0</version>
    <extensions>true</extensions>
    <configuration>
        <instructions>
            <Bundle-SymbolicName>
                ${project.artifactId}
            </Bundle-SymbolicName>
            <Export-Package>
                net.latuske.bpi.*,
                org.wso2.carbon.bpi.*
            </Export-Package>
            <Import-Package>
                !javax.xml.namespace,
                javax.xml.namespace;version="0.0.0",
                *;resolution:=optional,
            </Import-Package>
            <DynamicImport-Package>*</DynamicImport-Package>
            <Carbon-Component>UIBundle</Carbon-Component>
        </instructions>
    </configuration>
</plugin>
```

Listing 39 – Maven-bundle-plugin

Building this project with Maven would create a file called *org.wso2.carbon.bpi.ui-1.0-SNAPSHOT.jar*. This file can be copied to a folder called *dropins* inside the repository folder of the Carbon application and will be available as a component at the next restart of the server. It is recommended though to create a feature first and install the component through the feature manager, as explained in the next chapter.

Note: The component developed here is a front-end-only component without a back-end component.

6.4 Features

The next step of the process is the creation of the features so that an installable form of the project is available. The plugin which is used for the creation of the features and the repository is the *carbon-p2-plugin*. The projects only consist of a POM and a properties file that contains license information. They are based on a sample application [28] where only the *artifactIds* and the *groupIds* have to be adapted. Listing 40 shows the POM of the UI feature.

```
<plugin>
    <groupId>org.wso2.maven</groupId>
    <artifactId>carbon-p2-plugin</artifactId>
    <version>1.1</version>
    <executions>
        <execution>
            <id>p2-feature-generation</id>
            <phase>package</phase>
            <goals>
                <goal>p2-feature-gen</goal>
            </goals>
            <configuration>
                <id>org.wso2.carbon.bpi.ui</id>
                <propertiesFile>../feature.properties</propertiesFile>
                <adviceFile>
                    <properties>
                        <propertyDef>
                            org.wso2.carbon.p2.category.type:console
                        </propertyDef>
                        <propertyDef>
                            org.eclipse.equinox.p2.type.group:false
                        </propertyDef>
                    </properties>
                </adviceFile>
                <bundles>
                    <bundleDef>
                        org.wso2.carbon.bpi:org.wso2.carbon.bpi.ui
                    </bundleDef>
                </bundles>
                <importFeatures>
                    <importFeatureDef>
                        org.wso2.carbon.core.ui:3.0.0
                    </importFeatureDef>
                </importFeatures>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Listing 40 – UI feature POM

The *carbon-p2-plugin* uses the *p2-feature-gen* goal to generate the feature. The *propertiesFile* tag allows one to specify a properties file that contains the license information. *Bundles* holds the UI components that are included in the feature and *importFeatures* specifies the dependencies of the feature. In the properties section, the *org.wso2.carbon.p2.category.type:console* definition tells the plugin that this feature is a UI component (console) while a back-end component would have an *org.wso2.carbon.p2.category.type:server* definition.

Usually, a Carbon component would consist of a front-end and back-end component and hence a front-end feature and a back-end feature. These two features are then packaged in an aggregated feature. As there is no back-end component in this case, there is also no back-end feature. But the aggregated feature still has to be created. It will then only contain the front-end feature.

The plugin and the goal for the aggregated feature creation are the same as for the UI feature. Listing 41 shows the configuration section of the plugin. The includedFeatures section allows one to specify the features that should be included – in this case the UI feature is the only one.

```
<configuration>
    <id>org.wso2.carbon.bpi</id>
    <propertiesFile>../feature.properties</propertiesFile>
    <importFeatures>
        <importFeatureDef>org.wso2.carbon.core:3.0.0</importFeatureDef>
    </importFeatures>
    <includedFeatures>
        <includedFeatureDef>
            org.wso2.carbon.bpi:org.wso2.carbon.bpi.ui.feature
        </includedFeatureDef>
    </includedFeatures>
</configuration>
```

Listing 41 – Part of the aggregated feature POM

6.5 Repository

The POM of the repository uses the same plugin as the features (carbon-p2-plugin) but with the *p2-repo-gen* goal that creates a repository. Listing 42 shows the *execution* part of the plugin. The important section is the *featureArtifactDef* section that allows one to include the features.

```
<execution>
    <id>2-p2-repo-generation</id>
    <phase>package</phase>
    <goals>
        <goal>p2-repo-gen</goal>
    </goals>
    <configuration>
        <p2AgentLocation>${basedir}/target/p2-agent</p2AgentLocation>
        <metadataRepository>file:${basedir}/target/p2-repo</metadataRepository>
        <artifactRepository>file:${basedir}/target/p2-repo</artifactRepository>
        <publishArtifacts>true</publishArtifacts>
        <publishArtifactRepository>true</publishArtifactRepository>
        <featureArtifacts>
            <featureArtifactDef>
                org.wso2.carbon.bpi:org.wso2.carbon.bpi.ui.feature:1.0.0-SNAPSHOT
            </featureArtifactDef>
        </featureArtifacts>
    </configuration>
</execution>
```

Listing 42 – Part of the Repository POM

7 Discussion and Outlook

OSGi has without any questions many advantages, but the integration process has also shown that developing components for an OSGi environment can be difficult. Developers need a good understanding of the OSGi framework to be able to integrate their projects. The Business Process Illustrator is a relatively small project, yet some of the parts in the code could not be ported into the OSGi environment, at least not in reasonable time and effort. This concerns basically the frontend of the BPI which is made out of JavaServer Faces and Facelets. The problems with the JavaServer Faces – and the used Apache MyFaces implementation – affect all available OSGi frameworks, not only the Equinox framework, which is the foundation of the WSO2 Carbon framework. The currently available solutions which bypass this problem usually involve code changes of the MyFaces implementation. Although first OSGi bundles are released by the development team, there is still a lack of well-documented examples.

The missing support for JavaServer Faces does not affect the main feature of the BPI, which is the possibility to apply process views to monitored process instances and models. The graphs are generated by servlets and work without problems in an OSGi environment. The JavaServer Faces were replaced by JavaServer Pages without losing much functionality of the BPI.

The connection to the Business Process Server was not very difficult, as the management API of the Business Process Server supports all the functionality, required by the BPI. The first assumption that the events will have to be retrieved directly from the underlying database of the workflow management system was not confirmed.

Connecting the Business Process Server to another workflow management system has also revealed some minor bugs in the BPI code. In addition, the code was partly built on assumptions about other workflow management systems that did not hold, e.g. the BPI expected the process definition (the BPEL file) to be available via a URL – as it is the case with the Apache ODE. The WSO2 BPS – and probably many other WfMS – does not provide the BPEL file via a URL. The code has been adapted to support WfMS that do not have this feature. Still, a workflow management system that does not provide a URL must provide the process definition in some other way, e.g. providing a method which returns the content of the BPEL file as a string. The BPI supports now both ways.

An improvement of the current solution could be the separation of the component into a front-end and back-end component. WSO2 provides the framework for this separation and the BPI has a modular architecture that allows a separation of frontend and service as well. Providing the BPIService as a back-end component and hence as a web service would definitely be a valuable improvement.

Figure 24 shows a screenshot of the final application inside the WSO2 Carbon framework.

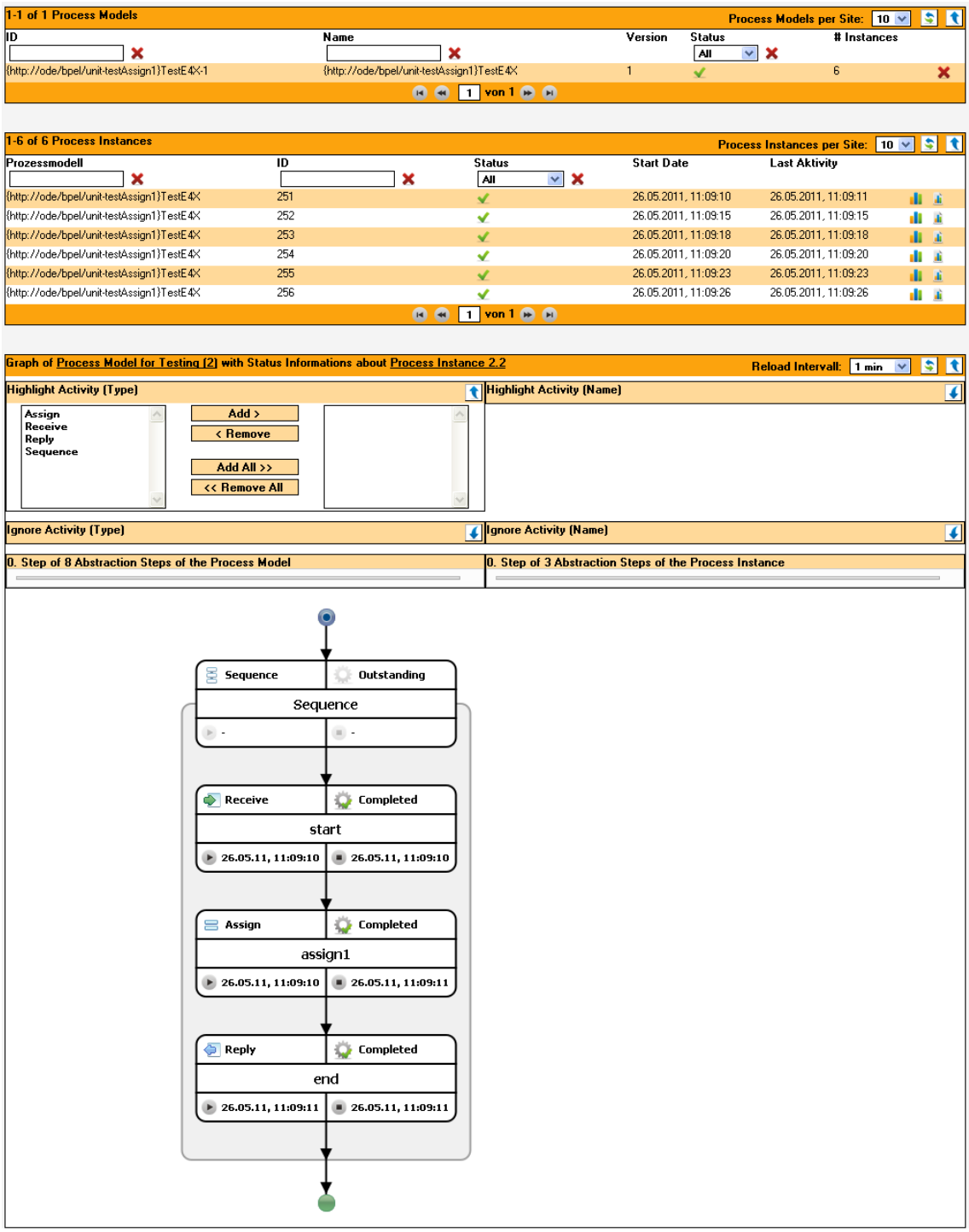

Figure 24 – Screenshot of the application inside the WSO2 Carbon framework

At the time of printing of this document, there is still a minor problem with the layout that will be fixed in the final release.

8 References

1. **Workflow Management Coalition.** *Terminology & Glossary.* 1999.

2. **ITWissen.** Workflow::ITWissen.info. [Online] http://www.itwissen.info/definition/lexikon/Workflow-workflow.html.

3. **Gregor Latuske.** *Sichten auf Geschäftsprozesse als Werkzeug zur Darstellung laufender Prozessinstanzen.* Stuttgart : Institut für Architektur von Anwendungssystemen, 2010.

4. **IBM.** Web Services Flow Language Version 1.0 (WSFL 1.0). [Online] 2001. http://xml.coverpages.org/wsfl.html.

5. **Microsoft.** XML Business Process Language (XLANG). [Online] 2001. http://xml.coverpages.org/xlang.html.

6. **Microsoft, IBM.** Business Process Execution Language for Web Services Version 1.1. [Online] 2003. http://public.dhe.ibm.com/software/dw/specs/ws-bpel/ws-bpel.pdf.

7. **OASIS.** Web Services Business Process Execution Language Version 2.0. [Online] April 11, 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

8. —. OASIS WS-BPEL Extension for People. [Online] 2007. http://www.oasis-open.org/committees/bpel4people/.

9. **W3C.** Scalable Vector Graphics (SVG) 1.1 (Second Edition). [Online] June 22, 2011. http://www.w3.org/TR/SVG11/intro.html.

10. **OSGi Alliance.** OSGi Technology. [Online] http://www.osgi.org/About/Technology.

11. **YarentY.** OSGi. [Online] 2009. http://www.yarenty.com/tools/java/40-java/47-osgi.html.

12. **OSGi Alliance.** OSGi Service Platform Core Specification, Release 4, Version 4.3. [Online] April 2011. http://www.osgi.org/download/r4v43/r4.core.pdf.

13. **Sun Microsystems (SUN).** Understanding the Manifest. [Online] 2010. http://java.sun.com/developer/Books/javaprogramming/JAR/basics/manifest.html.

14. **Apache Software Foundation.** Apache Maven Project. [Online] http://maven.apache.org.

15. —. Maven Getting Started Guide. [Online] 2011. http://maven.apache.org/guides/getting-started/index.html.

16. —. Apache Ant. [Online] http://ant.apache.org/.

17. **Sonatype.** *Maven: The Complete Reference.* 2010.

18. **Apache Software Foundation.** Available Maven Plugins. [Online] http://maven.apache.org/plugins/.

19. **David Schumm, Gregor Latuske and Frank Leymann.** *A Prototype for View-based Monitoring of BPEL Processes.* Universität Stuttgart : Technical Report 2011/04, 2011.

20. **Sun Microsystems (SUN).** BPEL2SVG Generator. [Online] 2008. http://blogs.sun.com/toxophily/entry/open_esb_tip_generating_svg.

21. **Gamma et al.** *Design Patterns.* s.l. : Addison-Wesley Professional, 1994.

22. **WSO2.** WSO2 Oxygen Tank. [Online] http://wso2.org/.

23. —. WSO2 Carbon Framework. [Online] 2010. http://wso2.org/project/carbon/3.1.0/docs/index.html.

24. **Eclipse.** Equinox P2. [Online] http://www.eclipse.org/equinox/p2/.

25. **WSO2.** WSO2 Business Process Server. [Online] 2010. http://wso2.org/project/bps/2.0.2/docs/.

26. **Milinda Pathirage.** WSO2 Business Process Server. [Online] 2011. http://blog.mpathirage.com/2011/02/06/wso2-business-process-server/.

27. **WSO2.** Business processes, simplified, enabled. [Online] http://wso2.com/wp-content/datasheets/BPS.pdf.

28. **Sameera Jayasoma.** Creating your own Carbon component. [Online] 2010. http://wso2.org/premium/webinars/creating-your-own-wso2-carbon-components.

29. **Sonatype.** M2Eclipse. [Online] http://m2eclipse.sonatype.org/.

30. **Holger Funke.** Das OSGi Framework. [Online] April 2009. http://it-republik.de/jaxenter/artikel/Das-OSGi-Framework-2221.html/.

31. **Frank Leymann.** *Managing Business Processes via Workflow Technology.* Rome : s.n., 2001.

32. **Weigle Wilczek.** OSGi Service Platform. [Online] http://www.weiglewilczek.com/fileadmin/Publications/Poster_OSGi_web.pdf.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Studienarbeit selbständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht.

Ort, Datum

Unterschrift