APPROVAL PAGE FOR GRADUATE THESIS OR PROJECT

SUBMITTED IN PARTIAL FULFILLMENT OF REQUIREMENTS FOR DEGREE OF MASTER OF SCIENCE IN COMPUTER SCIENCE AT CALIFORNIA STATE UNIVERSITY, LOS ANGELES

Martin J. Olsen
Candidate

Computer Science
Field of Concentration

TITLE: AN ECLIPSE-BASED WEB SERVICE COMPOSITION TOOL

APPROVED: Dr. Jiang Guo
Faculty Member
Signature

Dr. Behzad Parviz
Faculty Member
Signature

Dr. Raj S. Pamula
Department Chairperson
Signature

DATE: 03/08/2007
AN ECLIPSE-BASED WEB SERVICE COMPOSITION TOOL

A Thesis

Presented to

The Faculty of the Department of

Computer Science

California State University, Los Angeles

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

By

Martin J. Olsen

March 2007
© 2007

Martin J. Olsen

ALL RIGHTS RESERVED
ACKNOWLEDGMENTS

I would like to thank Dr. Jiang Guo and Dr. Raj Pamula for their direction and assistance. In particular I have to thank Dr. Guo, my thesis advisor, for giving me enough directions to finish my project, as well as giving me enough liberty to get ideas and inspiration by exploring new and interesting topics by myself. Special thanks go to my fellow students and faculty at the Computer Science department for a thriving academic environment and for encouraging technical insight.
ABSTRACT

AN ECLIPSE-BASED WEB SERVICE COMPOSITION TOOL

By

Martin J. Olsen

This thesis describes the development of a graphical plug-in for Eclipse in which the user can create a web service composition. By using web service composition, different web services can be connected through their inputs and outputs. The project includes different technologies such as web service composition, Java, Web services, Eclipse plug-in development and more. Eclipse GEF is used as the editing framework.

Dynamic, stub-less, invocation serves as basis for the connection framework enabling connection to any kind of web services regardless of design and implementation. The user will be required to specify the location of a WSDL file, either manually or through UDDI upon which the system will build a local model of the web service structure.

The plug-in is given the name “Eclipse Web Service Architecture Plug-in”, or just as an abbreviation: “EWAP”. This is one of the first successful implementations of web service composition as a graphical plug-in for Eclipse.
## TABLE OF CONTENTS

ACKNOWLEDGMENTS .......................................................................................................................... iii

ABSTRACT ........................................................................................................................................ iv

LIST OF TABLES ................................................................................................................................... viii

LIST OF FIGURES .................................................................................................................................. ix

CHAPTER

1. Introduction....................................................................................................................................... 1

2. System Architecture....................................................................................................................... 3

2.1. Eclipse SDK ............................................................................................................................. 3

2.2. Web services ............................................................................................................................ 5

2.2.1. The Client ........................................................................................................................... 5

2.2.2. The Server ........................................................................................................................... 6

2.2.3. UDDI ................................................................................................................................. 7

2.2.4. WSDL ............................................................................................................................... 7

2.3. Plug-in Architecture .................................................................................................................. 9

2.4. The Traversal Algorithm ....................................................................................................... 11

3. The Eclipse Graphical Editing Framework ................................................................................ 13

3.1. Overview ............................................................................................................................... 13

3.2. The Model ............................................................................................................................. 15

3.3. Notification Routines .......................................................................................................... 18
3.4. The View........................................................................................................ 19

3.5. Controllers.................................................................................................... 20

4. Design and Implementation ............................................................................ 22

4.1. Building a Plug-in in Eclipse ....................................................................... 23

4.1.1. The Plug-in Development Environment ............................................... 24

4.1.2. The Plug-in Manifest ............................................................................... 24

4.1.3. Importing Existing Plug-in Source Code into Eclipse ......................... 28

4.1.4. Testing the Plug-in ................................................................................ 28

4.1.5. Deploying the Plug-in .......................................................................... 30

4.2. Model Overview .......................................................................................... 30

4.3. Representation Model Overview ............................................................... 33

4.4. Invocation Model Overview ....................................................................... 35

4.5. The View Model ......................................................................................... 38

4.5.1. ModelElement ...................................................................................... 38

4.5.2. Shape .................................................................................................. 39

4.5.3. ShapesDiagram ................................................................................... 41

4.5.4. Input and Output .................................................................................. 41

4.5.5. Web service .......................................................................................... 42

4.6. The Semantic Model ................................................................................ 44

4.6.1. WSWesbservice .................................................................................. 44

4.6.2. WSOOperation .................................................................................. 45
4.6.3. WSArgument .................................................................................... 45
4.3.4. WSResult .......................................................................................... 45
4.6.5. WSType ............................................................................................ 46
4.6.6. The Invocation Data Model .............................................................. 46
4.7. The Controller .......................................................................................... 47
4.8. The Views ................................................................................................ 48
4.9. Commands ............................................................................................... 50
4.10. Adding Tools to the Palette................................................................. 51
4.11. Utility Classes ........................................................................................ 52
5. Future Work .................................................................................................... 54
5.1. Data Mapper............................................................................................. 54
5.2. Invocation Element .................................................................................. 56
5.3. Describing the Flow Composition ........................................................... 58
5.4. Adding UDDI Interactivity ...................................................................... 59
6. Conclusion ...................................................................................................... 60
References ........................................................................................................... 61
LIST OF TABLES

Table 1 - Plug-in Extension Points ................................................................. 4
Table 2 - Packages ......................................................................................... 9
Table 3 - Plug-in Settings .............................................................................. 25
LIST OF FIGURES

Figure 1 - EWAP Plug-in Architecture................................................................. 4
Figure 2 - Client Server Architecture..................................................................... 5
Figure 3 - A Simple Soap Client............................................................................. 6
Figure 4 – Server Setup............................................................................................ 6
Figure 5 - Locating Web Services Using UDDI....................................................... 7
Figure 6 - Example WSDL ....................................................................................... 8
Figure 7 - Plug-in Architecture Overview ............................................................. 10
Figure 8 - Traversing a Web Service Composition ............................................... 12
Figure 9 - Model View Controller ......................................................................... 14
Figure 10 - Two Models ......................................................................................... 16
Figure 11 - One Model Approach........................................................................... 17
Figure 12 - Publisher Listener Design Pattern....................................................... 18
Figure 13 - Notification in GEF ............................................................................. 19
Figure 14 - The EWAP Plug-in .............................................................................. 22
Figure 15 - Creating a New Plug-in Project............................................................ 24
Figure 16 - The Plug-in Overview ......................................................................... 25
Figure 17 - The Plug-in Dependencies ................................................................. 26
Figure 18 - The Plug-in Runtime ......................................................................... 26
Figure 19 - The Plug-in Extensions ...................................................................... 27
Figure 41 - InputFigure Class ................................................................. 49
Figure 42 - WebserviceFigure Class ......................................................... 49
Figure 43 - ConnectionCreateCommand Class ......................................... 50
Figure 44 - The Palette .............................................................................. 51
Figure 45 - ShapesEditorPaletteFactory Class ......................................... 51
Figure 46 - WSDLUtil Class ................................................................. 52
Figure 47 - WSConnection Class ............................................................ 53
Figure 48 - Introducing the Data Mapper ................................................. 55
Figure 49 - Introducing the Ivocation Element ......................................... 57
CHAPTER 1

Introduction

Web services are a growing trend in application development and architecture. Web services are platform-independent software components that can be accessed via distributed environments. More parts of applications tend to be assembled from a set of web services and are no longer developed manually. Composition of web services has a lot of potential in streamlining business-to-business and enterprise application integration.

Web service functionality and semantics are described through additional pieces of information. Either through annotation of what is being done or function annotation of behavior. The web service specification is expressed in WSDL, Web service Definition Language. WSDL specifies the syntax of messages entering or leaving applications.

Currently there are many specification languages to describe the flow between web services. However, the composition of the flow is still manually obtained. That is, it needs to be manually created.

Semantic annotation is a widely discussed topic in the Semantic Web community. Understanding the meaning of messages poses no problem. However, in web service composition, the semantic-web community relies heavily on AI planning, which investigates the problem of synthesizing complex behavior given an initial state and an explicit goal. The implicit goal of a business process will always be to enable the correct
handling and creation of data objects manifested in the persistent documents. As an example, a travel agency reservation process organized the travel, while the creation of the required travel documents is the implicit goal.
CHAPTER 2
System Architecture

EWAP is an Eclipse Plug-in. It builds on and is dependent on other Eclipse plug-ins. In addition to being a plug-in it is also implements a web service client interface using Apache Axis. There are several protocols for web service and web service client communication. One of them is the SOAP protocol which is widely used and supported by EWAP.

2.1. Eclipse SDK

EWAP builds on several other plug-ins. Some plug-ins are standard Eclipse plug-ins. [1] Other plug-ins are 3rd party plug-ins that do not come with the Eclipse distribution.

A standard approach for building modular software systems is to avoid tight coupling between components. The more components are integrated to each other, the harder it becomes to integrate those components into other components. Also replacing a component will be nearly impossible.

Eclipse uses loose coupling as much as possible. This is achieved through the mechanism of extensions and extension points. Extension points can be thought of as electrical outlets; the outlet being the extension point and the plug being the extension.
Extension points are declared through XML files and Java interfaces that extensions must conform to. Plug-ins that wants to connect to an extension point must implement that interface. A key characteristic of any extending plug-in is that they know nothing of which plug-ins they extend. Table 1 shows the different extension points.

![EWAP Plug-in Architecture](image)

**Figure 1 - EWAP Plug-in Architecture**

**Table 1 - Plug-in Extension Points**

<table>
<thead>
<tr>
<th>Plug-in</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse Workspace</td>
<td>Provides interaction with resources such as projects and files.</td>
</tr>
<tr>
<td>Eclipse Workbench</td>
<td>Provides ability to extend Eclipse user interface with menu selections, toolbar buttons, notification and event framework, etc.</td>
</tr>
<tr>
<td>Eclipse GEF</td>
<td>Eclipse Graphical Editing Framework provides the ability to create rich graphical editing environments for applications using the Eclipse Platform.</td>
</tr>
</tbody>
</table>
2.2. Web services

The EWAP plug-in works as a client towards web services. This means that it is in the consumer end of a web service system. On the other end there is the server that the web service is deployed on, often called the provider. The system typically operates on a network such as internet. Overall the system conforms to the client server paradigm (see Figure 2).

![Client Server Architecture](image)

**Figure 2 - Client Server Architecture**

2.2.1. The Client

In order to perform web service invocation a client framework is needed. For this specific purpose Apache Axis has been chosen. Apache Axis is in it’s essence a SOAP engine, or a SOAP implementation. SOAP is an XML-based communication protocol and encoding format for inter-application communication or web services. The Java implementation of the Apache Axis framework was chosen to easily integrate it into the EWAP client application. A simple implementation of the Axis Framework can be seen in Figure 3.
public class TestClient {
    public static void main(String[] args) {
        try {
            String endpoint =
                "http://localhost:8080/axis/services/echo";
            Service service = new Service();
            Call call = (Call) service.createCall();
            call.setTargetEndpointAddress(new java.net.URL(endpoint));
            call.setOperationName(new QName("http://soapinterop.org/", echoString"));
            String ret = (String) call.invoke(
                new Object[] { "Hello!" });
            System.out.println("Sent 'Hello!', got " + ret + "'\n");
        } catch (Exception e) {
            System.err.println(e.toString());
        }
    }
}

Figure 3 - A Simple Soap Client

2.2.2. The Server

To verify the implementation a server environment was set up. The server implementation consisted of Apache Tomcat web server, with Apache Axis deployed web application (see Figure 4). Together this provided the necessary components to serve as a provider of SOAP web services.

![Figure 4 – Server Setup](image)
2.2.3. UDDI

UDDI, or Universal Description, Discovery and Integration, is an XML based protocol. [2] Using UDDI businesses can list themselves on the Internet and discover other services. Typical listings include names, products, locations and web services offered. Using UDDI a web service provider can register the URL of the WSDL file. A client can locate the service and investigate the service closer by looking at the WSDL file. Note that any invocation is separated from the UDDI Registry. See Figure 5 for more details.

![Figure 5 - Locating Web Services Using UDDI](image)

2.2.4. WSDL

WSDL, or Web service Description Language, is an XML based language that provides a model for describing web services. The WSDL description tells a client how to connect to a web service. WSDL is often used in combination with SOAP to provide a full featured deployment environment being able to provide and describe a service. An example WSDL file can be seen in Figure 6.
<?xml version="1.0"?>
<definitions name="StockQuote"
    targetNamespace="http://example.com/stockquote.wsdl"
    xmlns:tns="http://example.com/stockquote.wsdl"
    xmlns:xsd1="http://example.com/stockquote.xsd"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
    <types>
        <schema targetNamespace="http://example.com/stockquote.xsd"
            xmlns="http://www.w3.org/2000/10/XMLSchema">
            <element name="TradePriceRequest">
                <complexType>
                    <all>
                        <element name="tickerSymbol" type="string"/>
                    </all>
                </complexType>
            </element>
            <element name="TradePrice">
                <complexType>
                    <all>
                        <element name="price" type="float"/>
                    </all>
                </complexType>
            </element>
        </schema>
    </types>
    <message name="GetLastTradePriceInput">
        <part name="body" element="xsd1:TradePriceRequest"/>
    </message>
    <message name="GetLastTradePriceOutput">
        <part name="body" element="xsd1:TradePrice"/>
    </message>
    <portType name="StockQuotePortType">
        <operation name="GetLastTradePrice">
            <input message="tns:GetLastTradePriceInput"/>
            <output message="tns:GetLastTradePriceOutput"/>
        </operation>
    </portType>
    <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
        <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetLastTradePrice">
            <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>
    <service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
            <soap:address location="http://example.com/stockquote"/>
        </port>
    </service>
</definitions>

Figure 6 - Example WSDL
2.3. Plug-in Architecture

The EWAP plug-in consists of 7 parts or packages (see Table 2) and a total of 48 classes. A conceptual overview can be seen in Figure 7. For more information about class implementation see chapter on implementation.

Table 2 - Packages

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.eclipse.gef.ewap</td>
<td>Plug-in entry classes, extension interface implementations and actions.</td>
</tr>
<tr>
<td>Org.eclipse.gef.ewap.model</td>
<td>Model (View)</td>
</tr>
<tr>
<td>Org.eclipse.gef.ewap.wsdata</td>
<td>Model (Semantic)</td>
</tr>
<tr>
<td>Org.eclipse.gef.ewap.model.commands</td>
<td>Model creation, deletion etc.</td>
</tr>
<tr>
<td>Org.eclipse.gef.ewap.parts</td>
<td>Views and Controllers.</td>
</tr>
<tr>
<td>Org.eclipse.gef.ewap.wsconn</td>
<td>Web service connection and invocation utilities.</td>
</tr>
<tr>
<td>Org.eclipse.gef.ewap.wsutil</td>
<td>Utility classes used to parse WSDL into representation model, and mapping of XML data objects.</td>
</tr>
</tbody>
</table>
Ewap controls the MVC modules through the GEF framework.

Figure 7 - Plug-in Architecture Overview
As can be seen in the figure Ewap is an extension to Eclipse GEF. The entry point for the plug-in is partly defined in the ewap package and in the plug-in manifest. The Model view controller (MVC) module consists of three packages:

- ewap.parts – containing view and view controllers.
- ewap model – containing the view model.
- ewap.mode.commands – semantic rules that are enforced upon deletion and removal of model elements.

All the semantic data that is relevant to a web service is stored in the ewap.wsdata package. Ewap.wsconn and ewap.wsutil packages contain a collection of utilities used to interact with web service.

2.4. The Traversal Algorithm

After a composition has been modeled using the graphical editor it’s time to run the web service composition. This is done by invoking each service starting with the root elements of the composition going on to its children. The root element can be either an Input or WS object. A successful traversal can be seen in Figure 8.
The traversal algorithm works as follows:

1. Find the root element of the composition. This can be either an Input or WS object.
2. Get the output of the element.
3. Use the output to set the input of the next element in the queue.
4. If the child element is a WS object invoke the web service by calling the invoke() method in the Web service object. This will return either true or false depending of the outcome.
5. If the outcome of the invocation is true set the output of the web service (Go to step 2). If the outcome is false this is an indication that the invocation has failed.
CHAPTER 3
The Eclipse Graphical Editing Framework

In order to understand the nature of the plug-in, the Eclipse Graphical Editing Framework must be understood. This section describes the initial steps involved in creating an Eclipse-based application using the Graphical Editing Framework (GEF). GEF has been used to build a variety of applications for Eclipse, including state diagrams, activity diagrams, class diagrams, GUI builders for AWT, Swing and SWT, and process flow editors. Eclipse and GEF are both open source technologies. [3]

3.1. Overview

The initial assumption when using GEF is that a model should be displayed and edited graphically. To achieve this kind of functionality, GEF provides viewers, EditPartViewer. GEF viewers are much like JFace viewers, and are adapters on a SWT Control. However, GEF viewers distinguish themselves in that they are Model-View-Controller based. (See Figure 9)
The different views and the model are connected via controllers, EditParts. The EditParts are responsible for mapping the model to a view, and making changes to the model. In addition to this the EditParts will observe the model for any changes and update the view to reflect this.
3.2 The Model

The model represents everything, and it should be persisted and restored upon all changes. When editing the graphical view, the model is the only thing that should endure. All else will eventually be garbage collected and recreated when needed.

When interacting with an EditPart, the model will be manipulated through an extra abstraction layer called commands. Commands are created to encapsulate changes. By doing it this way it’s easier to implement undo and redo features as well as validation of any user actions. The Commands work as a part of the model, in the way that they edit the model. Commands should only know about the model.

The simplest GEF application is an editor for drawing figures and diagrams which have no other properties than location, color etc. These properties are strictly tied to the visual representation of a model, and have no other purposes.

However, in other cases, the editor is not only used to represent visual diagrams, but more complex model. The model still needs to keep track of the location and color, but in addition the model needs to keep track of other data. For example in the case of a class diagram, the model should also keep track of its variables and its methods.

It makes a lot of sense to avoid pollution a model with attributes that don’t make sense. For example, mixing the visual attributes of a model with its inner attributes may not be appropriate. Using a “business” model to represent the important semantic details, and a “view” model to represent the view of the business model is a useful implementation. (See Figure 10)
A business/view split model is not a critical requirement for GEF. In fact, the model can be split into multiple resources, containing different information about the same visual representation. A one model approach is also perfectly legal (see Figure 11). The term model is often just used about the application model. Objects on screen may correspond to several model objects. In GEF this mapping is easily obtained using the EditParts.
Depending on the complexity of the model, different design decisions can be made. As a rule a complex view model and a simple business model can simply be put into one model. A simple view model and a complex business model can also be put in the same model. However, a complex view model and a complex business model should always be separated.
3.3. Notification Routines

The view should always be a result of notification from the model. Therefore the model should acquire some sort of notification mechanism that can be mapped to appropriate update points in the application.

![Publisher Listener Design Pattern](image)

Figure 12 - Publisher Listener Design Pattern

Usually a distributed or a centralized object notification strategy will do. Using a domain notifier will know about all changes in the model and broadcast these to the domain listeners. This conforms to the publisher listener design pattern (see Figure 12).

In GEF the model is the publisher, GEF handles the notifying, and the EditParts act as listeners, eventually performing changes to the view (see Figure 13). By employing this notification model a domain listener can be added to each viewer. Thus, when receiving a change, the affected EditPart will be looked up, re-dispatching the change appropriately.
The alternative is a distributed notification strategy, in which each EditPart will add its own listener to all model objects affecting it.

![Diagram of notification in GEF](image)

**Figure 13 - Notification in GEF**

3.4. The View

The view is a visual representation of the model. The view is represented using the Eclipse Draw2D plug-in that is part of GEF. Some models require more complex figures than others, while some figures can be used directly to display a model. For example a Label figure can be used to display a String. More complex figures can be
composed by combining different figures, layout managers and borders. In other cases
none of the predefined figures will work and it might be better writing figure
implementations that paint in a specific way.

In most cases it is not necessary to write custom figures. A combination of
provided layout managers can usually do the work. It is also smart to keep the controllers
and the views separated. A general approach is to subclass Figure and hide the details of
its structure. Now the EditPart could use this, or even better, write a minimalized API that
can be used. This is often referred to as the Separation of Concern pattern and is a way to
prevent coupling and bugs.

The figure should not have references to the model or its EditPart. However,
sometimes the EditPart can be added as a listener to the figure, through the Listener
interface. A content pane can be used to contain other graphical elements, when you
need decorations around the outside of the container. For example a box, with a top
portion that is a label, and the bottom portion is a label. This can be done by composing
multiple figures together. First a title box, while another figure is designed as the content
pane.

3.5. Controllers

Between the model and the view are the controllers. In GEF they are called
EditParts. The EditParts are in many ways what glues it all together or makes it into an
intricate framework. The GEF provides several classes for the implementation of the
controllers. However, they are abstract and require an active implementation. In many ways using sub-classing is a very flexible and straightforward way of doing this.

There are different base implementations provided for sub-classing. AbstractTreeEditPart should be used for EditParts that appear in the tree viewer. AbstractGraphicalEditPart and AbstractConnectionEditPart can be used for graphical viewers.

All viewers are configured with a factory for creating EditParts. This is the creation point of the EditParts. When setting the viewer’s content, it should be done by providing the model object that represents the input for that viewer. In most cases the input will typically be the top-most model object. From this model object all other model objects can be traversed. From here on each EditPart will populate and manage its own children. When new model objects are added by the user, the EditPart responsible for those objects will respond by constructing corresponding EditParts. After each EditPart is created and added to its parent, the same thing will happen to view objects.

The EditParts will be garbage collected as soon as the user deletes a model object. However, if the user undoes this delete, a new EditPart will be constructed to represent the restored object. This is the reason why no long-term information should be kept in the EditParts.
The implementation of the EWAP plug-in has required use of different technologies. First of all the plug-in is built on Eclipse and inherits a lot of its qualities from its SDK. In addition to the Eclipse SDK itself, GEF was required to implement the graphical features of the plug-in (see Figure 14). Features such as creating toolbars, creating shapes and figures and being able to connect them are provided through GEF. Just as important is the notification system which lets data flow seamlessly between model and figures upon changes. However, the EWAP plug-in is after all a web service composition plug-in and therefore a lot of the implementation also has to do with technicalities regarding web services. The following sections will go through the implementation details one by one, and briefly explain each of them.

Figure 14 - The EWAP Plug-in
4.1. Building a Plug-in in Eclipse

The first challenge any plug-in developer faces is the challenge of where to start and what to look for. Luckily, when developing a plug-in for Eclipse there are a lot of resources and books that can offer a lot of help. First of all, it’s necessary to understand that except from a small run-time kernel, everything in Eclipse is a plug-in, and built like a plug-in. The most crucial plug-ins are the Eclipse Workbench and the Eclipse Workspace. These are essential and basically provide an extension point to be used by other plug-ins. All plug-ins require an extension point to plug into.

In the case of the Eclipse Workbench, it provides the ability to extend the Eclipse user interface with menu selections, toolbar buttons, notification and event framework, and the ability to create new views. On the other hand the Eclipse Workspace lets you interact with resources such as projects and files. Needless to say these plug-ins provide the basic features of all other plug-ins. However, these are just a few of numerous extension points that are available in the Eclipse SDK.

It’s also important to understand that creating a plug-in is just like creating any Java project. It requires java programming and understanding. This section is not a tutorial but more a first look at the plug-in development cycle.
4.1.1. The Plug-in Development Environment

The most common approach when creating an Eclipse plug-in is to use the Eclipse Plug-in Development Environment (PDE). The PDE provides wizards to help the developer when creating plug-ins. A plug-in project can be started just like a regular java project (see Figure 15).

![Creating a New Plug-in Project](image)

Figure 15 - Creating a New Plug-in Project

4.1.2. The Plug-in Manifest

After a project has been created the plug-in manifest should be investigated. When double-clicking on the plugin.xml file in the package explorer a number of different tabs are available to explore and specify different parts of the plug-in (see Table
3. These settings can also be edited directly in the files MANIFEST.MF, plugin.xml, and build.properties.

Table 3 - Plug-in Settings

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview (see Figure 16)</td>
<td>Plug-in id, version, name, provider and activator settings.</td>
</tr>
<tr>
<td>Dependencies (see Figure 17)</td>
<td>Other plug-ins the plug-in dependent on.</td>
</tr>
<tr>
<td>Runtime (see Figure 18)</td>
<td>Runtime required packages and jar files.</td>
</tr>
<tr>
<td>Extensions (see Figure 19)</td>
<td>Plug-in extension settings.</td>
</tr>
<tr>
<td>Build (see Figure 20)</td>
<td>Resources to include in build.</td>
</tr>
</tbody>
</table>

Figure 16 - The Plug-in Overview
Figure 17 - The Plug-in Dependencies

Figure 18 - The Plug-in Runtime
4.1.3. Importing Existing Plug-in Source Code into Eclipse

In many cases you find a plug-in that has some of the features you would like to use. Sometimes the plug-ins will also contain the source code. In these cases the source code for the plug-in can be loaded into Eclipse using the Import Plug-in and Fragments feature (see Figure 21). In addition to being a great time saving tool it’s also one of the best ways to learn how to create plug-ins by looking at other project source codes.

![Import Plug-in and Fragments](image)

Figure 21 - Import Plug-in and Fragments

4.1.4. Testing the Plug-in

The testing cycle for an Eclipse plug-in is in many ways a tedious procedure. First of all it requires a new Eclipse instance to be started. The instance will use its own workspace called Runtime Eclipse Application (see Figure 22). It’s always faster to test
components isolated from the Eclipse Runtime environment, then later implement them into the project and test the plug-in using the Runtime instance.

![Image of Eclipse Application Launching](image-url)

Figure 22 - Launching an Eclipse Application
4.1.5. Deploying the Plug-in

As a last instance in the plug-in development cycle is the actual build process. This is done through the Export Plug-in and Fragments feature (see Figure 23).

![Figure 23 - Export Plug-in and Fragments](image)

4.2. Model Overview

The implementation of the model was split into two parts: a “view model” and a “business model”. The view model contains application specific data such as the location, color and shape of the figures. In addition to this all incoming and outgoing connections are also referenced here.
The deeper, semantic, web service specific data is separated from the view model. Its job is to have all information about the structure of a web service. This is obtained from the WSDL, and then parsed into this model. In addition it contains information about a specific invocation on this web service. The invocation model will contain more specific data. In order to do this it inherits the structure from the structure model and adds the data. In other words it serves as a wrapper for the structure model.

Figure 24 shows how the model is connected to the figures through the EditParts. The main model elements are the Input, WebService and Output. They each have controllers: InputEditPart, WebServiceEditPart and OutputEditPart. The controllers control the views, InputFigure, WebServiceFigure, and OutputFigure. When any changes are made to the model a notification, firePropertyChange(…), is sent through the event framework. Each EditPart is a listening subscriber to any changes made to the model, and will pick up the property change notifications. The editparts can then perform the corresponding change to the figures. Note that the event notification framework starts with the model then goes through the controllers to the view. The controllers do not listen to changes to the view, only to the model.
Figure 24 - Model Overview
The Input and Output model elements do not have any underlying data except the view data. In addition to having the standard view data, they have a string that represents either the input or the output of a webservice.

All the base model elements, Input, Output, and WebService, have two methods; setInput(…), and getOutput(). The setInput(…) method is called when data should be set for that specific model element. For example when the user defines the input of an Input element the setInput(…) method is used. The traversal algorithm will use the getOutput() method to fetch this data and call the setInput(…) method of the WebService model element with this data. The data flows in a similar fashion through all the queued model elements. However, before the traversal algorithm can continue to the next WebService element an invocation is needed.

4.3. Representation Model Overview

In order to build the representation model the corresponding WSDL must be parsed. A utility class, WSDLUtil, was implemented for this specific purpose. It contains a method parseWSDL(String URL) which returns the WSWebservice reference model. Note that the reference model does not contain the invocation data. (See Figure 25)
Figure 25 – The Web Service Representation Model

To build the representation model WSDL4J was used. WSDL4J, or Web Services Description Language for Java Toolkit, allows the creation, representation, and
manipulation of WSDL documents. The reference implementation for JSR110 'JWSDL' (jcp.org). The schema support in WSDL4J is not complete. Retrieving information such as simple data types works. Retrieving information about complex data types does not work. In order to completely parse the WSDL XML Schema a different more complete schema parser, such as Apache WSIF, or Web Services Invocation Framework, could be used. However, WSIF has not been implemented on this project. Developers working on similar project are recommended to take a look at the WSIF application framework. At the current time it does a better job parsing WSDL schemas.

Note that neither Apache WSIF nor WSDL4J has a completely implemented schema parser. Both frameworks use JAX-RPC, or Java API for XML-based RPC, to parse the WSDL XML file. In cases where neither frameworks can do the job, JAX-RPC can be used directly to write more specific parser. Obviously this will be a more time consuming task, and more suitable for a tighter scoped project.

4.4. Invocation Model Overview

When the model has been set up and the corresponding reference model has been created through WSDL4J, the composition can be traversed. During a traversal each Web Service will be invoked with the specific operation and arguments the user has specified.

Before invocation a WSInvocation object will be created based on what operation is selected. The WSInvocation object directly depends on the WSOperation object, rather
the WSWebservice object. The WSWebservice object will always be the same for all
invocations on the same reference model, but the WSOperation may be different
depending on the selected operation.

The WSArgumentData object will be created right away based on what the user
has set to be the input value, or argument, for the operation. Note that the
WSArgumentData is really just a wrapper around the WSArgument class, adding the
corresponding data. Both the WSArgument and WSReturn have references to WSType.
WSType is an enumerative data representation that is used to determine the XML-type of
the data represented in the Web Service. The WSReturnData is not initialized until the
invocation is being performed.

A utility class, WSConnection, was created in order to perform the actual
invocation of the web service. The WSConnection class uses Apache Axis to perform the
actual connection and invocation of the web service. Apache Axis takes care of all the
back and forth SOAP transactions involved. (See Figure 26)

Based on the result of the invocation the invoke() method will set the
WSReturnData of the WSIInvocation representation model. A firePropertyChange(…) notification will be sent through the notification framework notifying the controllers of
the outcome. The invoke() method in the WebService class will return either true or false
depending on the outcome of the invocation. This, in turn, will determine if the next
element in the queue of the composition should be traversed or not.
Figure 26 - The Invocation Model

Note that a web service can not be invoked unless the WSWebservice representation model has been initialized. Also, when the model is saved, then restored, a
re-initialization of the model will done. This is a design choice, simply to verify the existence of a web service.

4.5. The View Model

A good place to start implementing a GEF project is the view model. The view model has the following responsibilities:

- Storing all data that may be edited or viewed by the user.
- Model persistency.
- Provide a notification strategy by letting others listen to changes in model.

4.5.1. ModelElement

The base class of all model elements is the ModelElement class. It adds three features that are common to all model elements; namely persistency, property change, and property source support. Persistency is added by implementing the java.io.Serializable interface. Model changes are propagated through a property event system. Editparts can subscribe and unsubscribe as receivers of property change notifications. A property change is posted by calling the firePropertyChange method. IPropertySource interface is implemented to integrate with the Properties View of the Eclipse Workbench. An outline of the implementation can be seen in Figure 27.
public abstract class ModelElement implements IPropertySource, Serializable {

    private transient PropertyChangeSupport pcsDelegate =
    new PropertyChangeSupport(this);

    public synchronized void
        addPropertyChangeListener(PropertyChangeListener l) {
        if (l == null) {
            throw new IllegalArgumentException();
        }
        pcsDelegate.addPropertyChangeListener(l);
    }

    protected void firePropertyChange(String property,
        Object oldValue,
        Object newValue) {
        if (pcsDelegate.hasListeners(property)) {
            pcsDelegate.firePropertyChange(property, oldValue, newValue);
        }
    }

    private void readObject(ObjectInputStream in) throws IOException,
    ClassNotFoundException {
        in.defaultReadObject();
        pcsDelegate = new PropertyChangeSupport(this);
    }

    public synchronized void
        removePropertyChangeListener(PropertyChangeListener l) {
        if (l != null) {
            pcsDelegate.removePropertyChangeListener(l);
        }
    }

    ...
}

Figure 27 - ModelElement Class

4.5.2. Shape

The Web service, Input, and Output objects have certain common elements that
have been factored into the Shape class. The features that have been factored into Shape
is the ability to having Connection elements ending and originating in it. In addition, information such as location and size are implemented.

```java
public abstract class Shape extends ModelElement {
    private static final long serialVersionUID = 1;
    public static final String SIZE_PROP = "Shape.Size";
    public static final String LOCATION_PROP = "Shape.Location";
    public static final String SOURCE_CONNECTIONS_PROP = "Shape.SourceConn";
    public static final String TARGET_CONNECTIONS_PROP = "Shape.TargetConn";
    public List sourceConnections = new ArrayList();
    public List targetConnections = new ArrayList();
    protected static Image createImage(String name) {
        ...
    }
    abstract void addConnection(Connection conn);
    public abstract Image getIcon();
    public abstract Point getLocation();
    public abstract IPropertyDescriptor[] getPropertyDescriptors();
    public abstract IPropertyDescriptor[] getBasicDescriptors();
    public Object getPropertyValue(Object propertyId) {
        ...
    }
    public abstract Dimension getSize();
    public abstract List getSourceConnections();
    public abstract List getTargetConnections();
    abstract void removeConnection(Connection conn);
    public abstract void setLocation(Point newLocation);
    public String getOutput() {
        ...
    }
    public void set PropertyValue(Object propertyId, Object value) {
        ...
    }
    public abstract void setSize(Dimension newSize);
}
```

Figure 28 - Shape Class
4.5.3. ShapesDiagram

ShapesDiagram is a top-level implementation of the Shapes interface. In addition to the basic ModelElement features it adds a container that contains other shapes. Its main responsibility is to maintain information about collections of shapes and notify listeners about collection changes.

```java
public class ShapesDiagram extends ModelElement {
    ...
    private Collection shapes = new Vector();

    public boolean addChild(Shape s) {
        if (s != null && shapes.add(s)) {
            firePropertyChange(CHILD_ADDED_PROP, null, s);
            return true;
        }
        return false;
    }

    public List getChildren() {
        return new Vector(shapes);
    }

    public boolean removeChild(Shape s) {
        if (s != null && shapes.remove(s)) {
            firePropertyChange(CHILD_REMOVED_PROP, null, s);
            return true;
        }
        return false;
    }
}
```

Figure 29 - ShapesDiagram Class

4.5.4. Input and Output

The Input and Output model elements are the base for the Input and Output shapes respectively. They do not contain a lot of information except values that represents Input and Output. The Input and Output implementations are similar.
public class Input extends Shape {

    private static final Image RECTANGLE_ICON =
            createImage("icons/rectangle16.gif");
    private static final long serialVersionUID = 1;
    private String inputInfo = "";
    public static final String INPUT_SETINFO =
            "Input.Info";

    ...

    public void setInput(String input) {
        this.inputInfo = input;
        updateUI();
    }

    public String getInput() {
        return inputInfo;
    }

    public void updateUI() {
        firePropertyChange(this.INPUT_SETINFO, null,
                "Value: " +
                this.inputInfo);
    }

    public String getOutput() {
        firePropertyChange(this.INPUT_SETINFO, null,
                "Value: " +
                this.inputInfo);
        return this.getInput();
    }
}

Figure 30 - Input Class

4.5.5. Web service

The Web service model element contains the data of the view/visual
representation of the object. In addition to this it serves as a wrapper for the underlying
business model mainly containing semantic details by storing the WSWebserive and
WSInvokation data. Important methods are setWSDLURL(….) and invoke() which takes care of all the web service invocation later done in the application. Notice that most of the responsibilities of this class are actually done through the WSConnection and WSDLUtil classes that will be discussed later.

```java
public class Web service extends Shape implements Serializable {
    ...
    /* State variables */
    private String selectedMethod = "";
    private String serviceName = "WS";
    private String wsdlUrl = "";
    private WSWebservice wweb = null;
    private boolean error = false;
    private WSInvokation wsinvokation;
    
    private String output = "";
    ...
    public void setWSDLURL(String newURL) {
        ...
    }
    public boolean invoke() {
        ...
    }
    public String getOutput() {
        ...
    }
    public void setOutput(String output) {
        ...
    }
}

Figure 31 - Web service Class
4.6. The Semantic Model

The semantic model, or the business model, contains all the actual data representing the web services. In many ways, it’s simpler than the view model since it’s not coupled with the GEF framework. The model is contained within the org.eclipse.gef.ewap.wsdata package. It’s important to understand that the business model is split into two parts. One part represents the structure of the web service, whilst the other contains all the data representing an invocation of the web service. The separation is obvious; a web service can have more than one single invocation and therefore should not be coupled with the structure of the web service. Note that all the class names start with WS. This is simply to differentiate them from the View Model.

4.6.1. WSWebservice

This class represents the entry point for a web services internal structure. The WSWebservice contains a list with all the operations in the web service.

```java
public class WSWebservice implements Serializable {
    private String targetEndPointAddress;
    private String serviceName;
    private LinkedList<WSOperation> operationList;
    ...
    //getters and setters
}
```

Figure 32 - WSWebservice Class
4.6.2. WSOperation

The WSOperation class represents the structure of a web service operation. This class also contains a list over all the arguments required and the result/return of the operation.

```java
public class WSOperation implements Serializable {
    private String operationName;
    private String nameSpaceURI;
    private LinkedList<WSArgument> argumentList;
    private WSResult result;
    ...
    //getters and setters
}
```

Figure 33 - WSOperation Class

4.6.3. WSArgument

The WSArgument class represents the argument/parameter of an operation. It also contains the specific type of the argument.

```java
public class WSArgument implements Serializable {
    private WSType type;
    private String argumentName;
    ...
    //getters and setters
}
```

Figure 34 - WSArgument Class

4.3.4. WSRResult

WSResult presents the result/return of a web service operation.
public class WSResult implements Serializable {
    private WSType type;
    ...
}

Figure 35 - WSResult Class

4.6.5. WSType

WSType is an enumerative representation of different data types available through different web services.

public enum WSType implements Serializable {
    BASE64BINARY,
    BOOLEAN,
    BYTE,
    DATETIME,
    DECIMAL,
    DOUBLE,
    FLOAT,
    HEXBINARY,
    INT,
    INTEGER,
    LONG,
    QNAME,
    SHORT,
    STRING
}

Figure 36 - WSType Enumeration

4.6.6. The Invocation Data Model

The invocation data model is coupled to the web service structure model. It contains a reference to the operation that is being invoked, the result/return data of that operation and the argument data associated to the operation.
public class WSInvokation implements Serializable {
    private LinkedList<WSArgumentData> argumentDataList;
    private WSResultData resultData;
    private WSOperation operation;
    ...
    //getters and setters
}

Figure 37 – WSInvokation Class

public class WSArgumentData implements Serializable {
    private WSArgument argument;
    private Object value;
    //getters and setters
}

Figure 38 – WSArgumentData Class

public class WSResultData implements Serializable {
    private WSResult result;
    private Object value;
    //getters and setters
}

Figure 39 – WSArgumentData Class

4.7. The Controller

The controller, or EditPart, is responsible for understanding the model, listening to events, and update the views. To have these features the EditPart must implement PropertyChangeListener interface that is registered with the model. When the EditPart is
deactivated it is removed from the list of listeners. The controllers are in the org.eclipse.gef.ewap.parts package.

```java
public abstract class EditPart extends AbstractGraphicalEditPart
    implements PropertyChangeListener {

    public void activate() {
        if (!isActive()) {
            super.activate();
            ((PropertyAwareModel)
                this.getModel()).addPropertyChangeListener(this);
        }
    }

    public void deactivate() {
        if (isActive()) {
            ((PropertyAwareModel)
                this.getModel()).removePropertyChangeListener(this);
            super.deactivate();
        }
    }

    public void propertyChage(PropertyChangeEvent evt) {
        String prop = evt.getPropertyName();
        ...
    }
}
```

4.8. The Views

The views, or Figures as they are called in GEF, are extensions of predefined Figures from the Draw2D framework.

- org.eclipse.draw2d.RectangleFigure – This Figure is extended in order to create the InputFigure and OutputFigure.
- org.eclipse.draw2d.Ellipse – This Figure is extended in order to create the Web serviceFigure.
Note that the fillShape(...) method is overridden. The implementations of OutputFigure and Web serviceFigure are similar.

```java
public class InputFigure extends RectangleFigure {
    private TextFlow textFlow;
    private int topLineHeight = 17;
    private String text = "";
    public InputFigure() {
        ...
    }
    public void setText(String text) {
        ...
    }
    protected void fillShape(Graphics graphics) {
        ...
    }
}

Figure 41 - InputFigure Class
```

```java
public class Web serviceFigure extends Ellipse {
    public void setText(String text) {
        ...
    }
    protected void fillShape(Graphics graphics) {
        ...
    }
    public void redColor() {
        ...
        this.repaint();
    }
    public void greenColor() {
        ...
        this.repaint();
    }
    public void orangeColor() {
        ...
        this.repaint();
    }
}

Figure 42 - WebserviceFigure Class
```
4.9. Commands

Commands are implemented to encapsulate model changes, therefore providing support for undoable changes. Also, commands are responsible for maintaining the “rules” that models have to follow. For example when creating a connection there are certain rules:

- Input can not have a connection targeted at it.
- A Web service can only have one incoming connection.

These rules are enforced through the ConnectionCreateCommand class in the org.eclipse.gef.ewap.model.commands package.

```java
public class ConnectionCreateCommand extends Command {
    public ConnectionCreateCommand(Shape source, int lineStyle) {
        ...
    }
    public boolean canExecute() {
        ...
    }
    public void execute() {
        ...
    }
    public void redo() {
        ...
    }
    public void setTarget(Shape target) {
        ...
    }
    public void undo() {
        ...
    }
}
```

Figure 43 - ConnectionCreateCommand Class
4.10. Adding Tools to the Palette

New tools can be added to the palette (see Figure 44) through the
ShapesEditorPaletteFactory class by adding a new CombinedTemplateCreationEntry
component to the PaletteDrawer components drawer.

```
final class ShapesEditorPaletteFactory {
    private static PaletteContainer createShapesDrawer() {
        ...
    }

    static PaletteRoot createPalette() {
        ...
    }

    private static PaletteContainer createToolsGroup(PaletteRoot palette) {
        ...
    }
}
```

Figure 44 - The Palette

Figure 45 - ShapesEditorPaletteFactory Class
4.11. Utility Classes

A couple of utility classes was implemented to help out with minimizing the complexity of the source code. The WSDLUtil class is used to create a structure model, or WSWebService model, out of a WSDL file residing at some URL. The actual invocation of a web service is done through the WSConnection class.

```java
public final class WSDLUtil {
    public static WSWebService parseWSDL(String url) throws Exception {
        ...
        return web service;
    }

    private static Definition getDefinition(String URL) throws WSDLException {
        ...
    }
}
```

Figure 46 - WSDLUtil Class
public class WSConnection {
    public WSConnection(WSWebservice web service, WSOoperation operation) {
        ...
    }
    public WSConnection(WSWebservice web service, String operationString) {
        ...
    }
    public void invoke(WSInvokation invokation) throws Exception {
        ...
    }
    public WSWebservice getWeb service() {
        ...
    }
    public void setWeb service(WSWebservice web service) {
        ...
    }
    public WSOperation getOperation() {
        ...
    }
    public void setOperation(WSOperation operation) {
        ...
    }
}

Figure 47 - WSConnection Class
CHAPTER 5
Future Work

After finishing the implementation of the plug-in it’s easy to get a feeling of what could have been done different. A large part of the project has been spent on doing research, such as finding suitable API’s, Eclipse GEF and generally about web service composition.

First, it’s obvious that a shift towards a more compliant schema parser would benefit the plug-in. That could be either the WSIF parser or by writing a more specific parser for the application.

5.1. Data Mapper

In addition to the schema parser the next natural step is to include a data mapping element to the model and to the view. With help of a data mapping element, it’s easier to direct the flow between the web services. At the same time only small changes needs to be made to the traversal algorithm, with the introduction of a new element, the data mapper. (See Figure 48)
Figure 48 - Introducing the Data Mapper
5.2. Invocation Element

A more versatile approach would be to implement an invoke element in addition to the data mapper. By doing this, a web service can have several invocations on the same web service element. More complexity is added to the composition, and instead of mapping inputs directly to a web service element they must now be mapped towards the invoke element. Figure 49 shows the same composition, with the additional elements added. With larger scale systems that require more rigorous compositions this seems to be the way to go.
Figure 49 - Introducing the Invocation Element
5.3. Describing the Flow Composition

The interface of a web service is already described through the WSDL file. However, the flow and interaction between the processes in the composition has not yet been discussed as far as how to describe it. BPEL4WS, or Business Process Execution Language for Web Services, is a language specifying business processes and business interaction protocols [4]. The XML-based grammar defines the interaction between processes and its partners during a composition. In addition, states and logic of coordination between the interactions are also described. BPEL is based on the WSDL description model. A BPEL engine has been written for java, BPWS4J.

BPWS4J, or the IBM Business Process Execution Language for Web Services Java, includes a platform that can execute processes written in BPEL4WS. BPWS4J takes a BPEL4WS document, together with the WSDL information [5]. During execution BPWS4J uses Apache Axis to handle the invocation and connection to the web services. The entire process can be made available through a new web service with a soap interface. During runtime a WSDL file describing the interface may also be retrieved.

BPEL4WS provides a more standardized approach to describe compositions. This is good, in the way there are several API’s written specifically to deal with BPEL4WS documents, thus more application support and easier integration.

The EWAP plug-in has a lot of potential as far as describing the composition flow through BPEL4WS, and being able to save them to XML files. This introduces new
possibilities, where developers can design their compositions using the EWAP plug-in, testing it, and then compiling it into a BPEL4WS document. The flow description can then be introduced to other applications and business components that need this specific behavior.

5.4. Adding UDDI Interactivity

Web services are still manually obtained through their WSDL locations. By adding a UDDI search interface the user could connect to a UDDI registry and find suitable services, hence making the application more dynamic. There are two major contributions to the Java community as far as UDDI interaction goes. The first one is JUDDI and is an Apache project. The other one is UDDI4J which is an open source project hosted through Sourceforge.
CHAPTER 6

Conclusion

There are many aspects to the EWAP plug-in. First of all it's a plug-in that runs in Eclipse. Creating a plug-in is a meticulous process. An understanding of the Eclipse SDK is crucial, as well as understanding the Eclipse plug-in framework. The next step would be to start implementing the Eclipse Graphical Editing Framework and creating a graphical editor.

On the other hand the plug-in uses web service composition that requires parsing of a WSDL in order to pull out necessary information to perform the dynamic invocation. After a composition is set up the model can be traversed and each web service can be invoked.

It has been a learning experience, being that this project has required so many different technologies. Puzzling all the different parts together has been a tedious challenge requiring every part of previous knowledge as well as creativity and motivation.

The final conclusion has to be that this is one of the first successful implementations of web service composition specifically developed for the Eclipse platform.
References


