Department of Computer Science

Daniel Jansson

Software Development Kit for a Telecom Services Architecture

D-level Thesis (30 ECTS)

2005:03

Software Development Kit for a Telecom Services Architecture

Daniel Jansson

This thesis is submitted in partial fulfillment of the requirements for the Masters degree in Computer Science. All material in this thesis which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Daniel Jansson

Approved, May 27, 2005

Opponent: Martin Blom

Advisor: Katarina Asplund

Examiner: Donald F. Ross

Abstract

This Master's thesis is the result of a project performed at Incomit AB in Karlstad. The content describes the requirements gathering for, and the development of, a Software Development Kit (SDK) for telecom service creation. The SDK facilitates the development of network plug-ins that enable new protocol specific behaviour in the Incomit Application Hub. As part of the requirement gathering for the SDK, two surveys were performed among the developers at Incomit. The first survey regarded experiences with development environments and the second survey regarded their opinion of development environment features.

The results from the two surveys and requirements from an old requirement specification were used as a base for creating the requirement specification for the SDK.

The SDK consists of interface documentation and a utility that generates compilable Java code in order to make it easier for the developer of the network plug-in to concentrate on protocol specific implementation. The code generation of the SDK has a graphical user interface implemented as a plug-in in the eclipse platform.

The SDK works well in a Windows XP environment. The SDK has very loosely coupled parts which makes it ideal for extension and modification which was part of the requirements.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Development of telecom services is often tied to telecom protocols. Since telecom network operators often refrain from opening their network because of security and billing complications, most of the service development is done within the operator company. It is very expensive and complex for outside developers to create a new service (e.g. a voting service that registers SMS-votes) since they have to work very closely with the network operator. This is not only a disadvantage for external developers but for the network providing company as well. The reason for this is that the operator could benefit from more services as it may attract more customers.

The company Incomit solves the above described problem by providing plug-in-based products with simple northbound interfaces[1] (see Section 2.3). Incomit AB was founded in March 2000. The company has two offices[2], one in Karlstad and one in London. The office in London is mostly sales based where the office in Karlstad is mostly product development based. The company, in total, has about 30 employees.

[1]interfaces provided to the layer above. As an example, the transport layer in the TCP/IP stack has an northbound interface toward the application layer

[2]Incomit AB was acquired by BEA Systems in December 2004

1.1 Thesis Assignment

Incomit has a product called The Application Hub (see Section 2.3). This product acts as a gateway between different networks. For example, one could through the Application Hub access the GSM-network from the Internet. In order to support new functionality the developers at Incomit create new software modules within the Application Hub. It is the creation of these modules that would need some help.

When developers at Incomit design a new plug-in in order to extend the functionality of their Application Hub, they tend to reuse old code and change it in order to create their new plug-in. This method has one major advantage and that advantage is time. The development time is greatly reduced when reusing existing code and replacing only certain parts of it with new code. However, there are also several disadvantages with this approach. First, using code that was specifically designed for one purpose may not be optimal for the new purpose. Second, source code that comes from old code can linger unused in the new code, and errors that could exist in the old code propagates to the new code. One solution to this problem is a Software Development Kit (SDK). The SDK should facilitate development of new telecom services such as network plug-ins. It should do this in such a way that the developer gets the required help to create the service fast and still avoid the problems described above.

The purpose of the project described in this thesis is to design and develop an SDK, which could be used to create network plug-ins. A network plug-in could, for example, be a messaging type plug-in that forwards text over a certain network protocol. An example of what a message plug-in could do is sending an application generated SMS to a GSM mobile phone.

The SDK is supposed to provide a development environment for the developer. This development environment should, through wizards and code manipulating features such as code completion, make the code creation of telecom services faster and easier. The SDK should also facilitate the testing of created network plug-ins by providing easy access to

Incomits own Network Simulator (see Section 2.6).

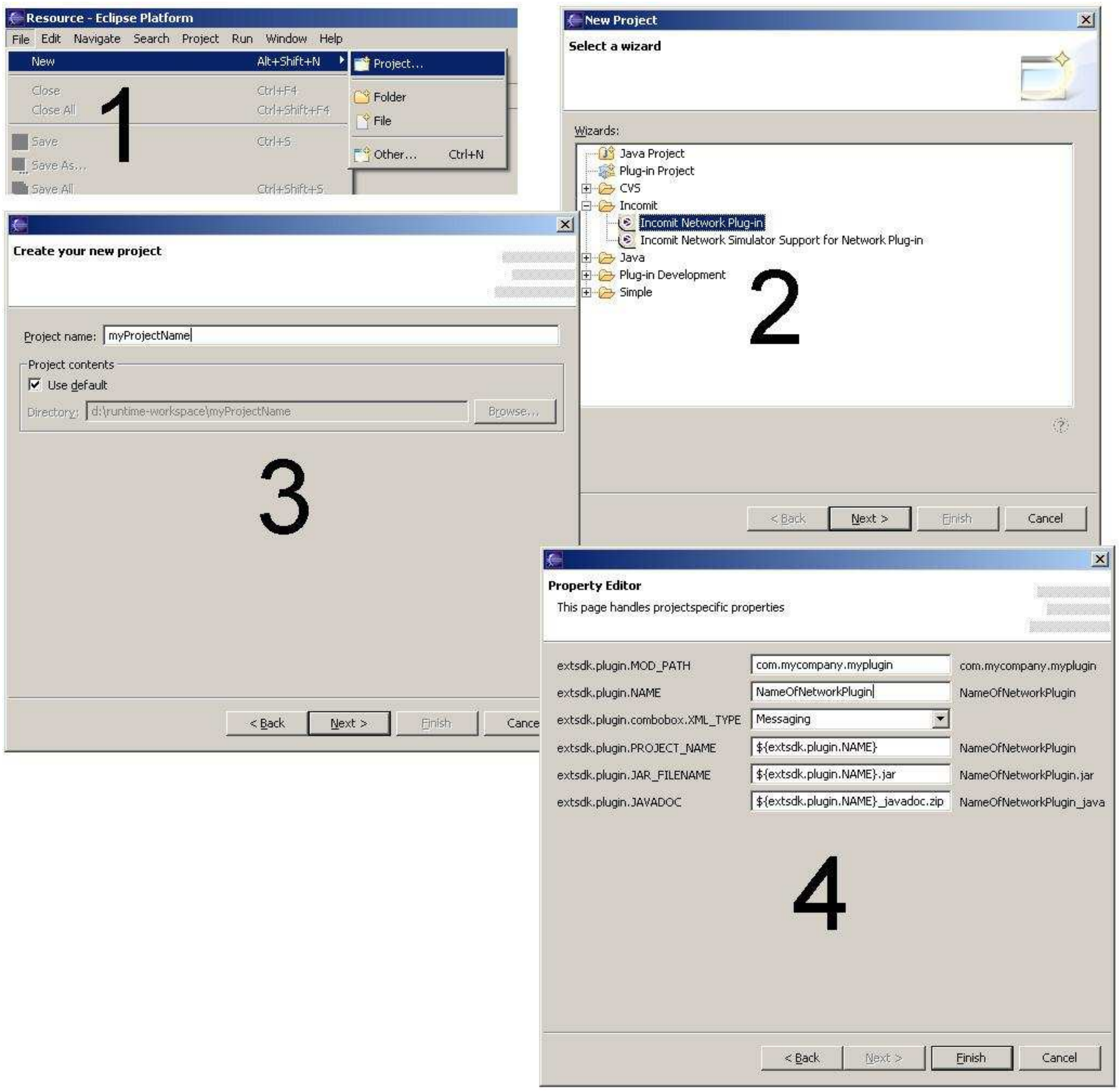
In order to find out what kind of parts that were essential for the SDK and the priorities of these, a quite extensive requirement gathering has been done. Incomit had an old requirement specification which was examined and modified. The modification was done according to two surveys performed among the Incomit developers. The surveys were done as a part of this thesis. Furthermore, in order to find what Graphical User Interface (GUI) the SDK should support, an evaluation was made of the two development environments Incomit suggested, JBuilder and eclipse. The evaluation part was seen as an important part by Incomit, therefore it is quite extensive.

The result of the work for this thesis is an SDK which is called "the Extension SDK" since it extends the existing functionality of the telecom services architecture. The SDK provides code examples which developers can use as a reference when creating their own service. Javadoc for utilities and necessary APIs are included into the SDK, as well as a plug-in to the popular development environment eclipse. This plug-in creates code from code templates, where the developer's own preferences for name and type of the services are taken into consideration and affects the result. The ability to receive help with code creation speeds up development of new telecom network services, reducing the overall implementation time.

In order to clarify how this is all implemented three figures are provided. These figures explain the different steps a developer is following when using the eclipse plug-in to create a new Network plug-in.

Figure 1.1 explains the creation of the project in the eclipse environment. First, the `File->New->Project...` is clicked. Second, the type of service which is to be created is chosen. Third, the project name is entered. Fourth, the service-specific information is entered, such as name, module name and what the `.jar` file will be called after compilation.

Figure 1.2 shows how the working project looks like after the creation of the project, which envolves generating code depending on the decisions of the developer. This is where



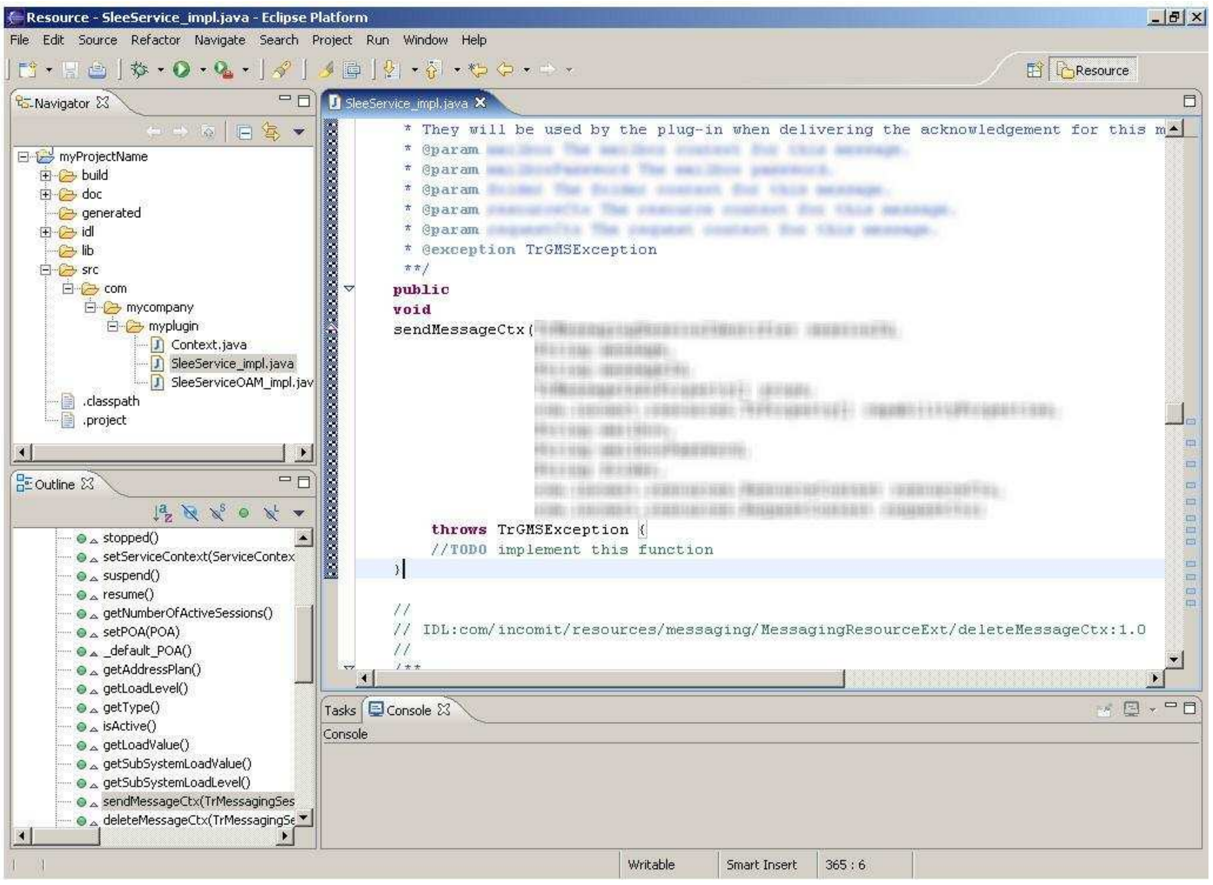
Project name: myProjectName

[✓] Use default

Directory: d:\runtime-workspace\myProjectName

extsdk.plugin.MOD_PATH	com.mycompany.myplugin	com.mycompany.myplugin
extsdk.plugin.NAME	NameOfNetworkPlugin	NameOfNetworkPlugin
extsdk.plugin.combobox.XML_TYPE	Messaging	
extsdk.plugin.PROJECT_NAME	${extsdk.plugin.NAME}	NameOfNetworkPlugin
extsdk.plugin.JAR_FILENAME	${extsdk.plugin.NAME}.jar	NameOfNetworkPlugin.jar
extsdk.plugin.JAVADOC	${extsdk.plugin.NAME}_javadoc.zip	NameOfNetworkPlugin_java

Figure 1.1: Creating a Network Plug-in

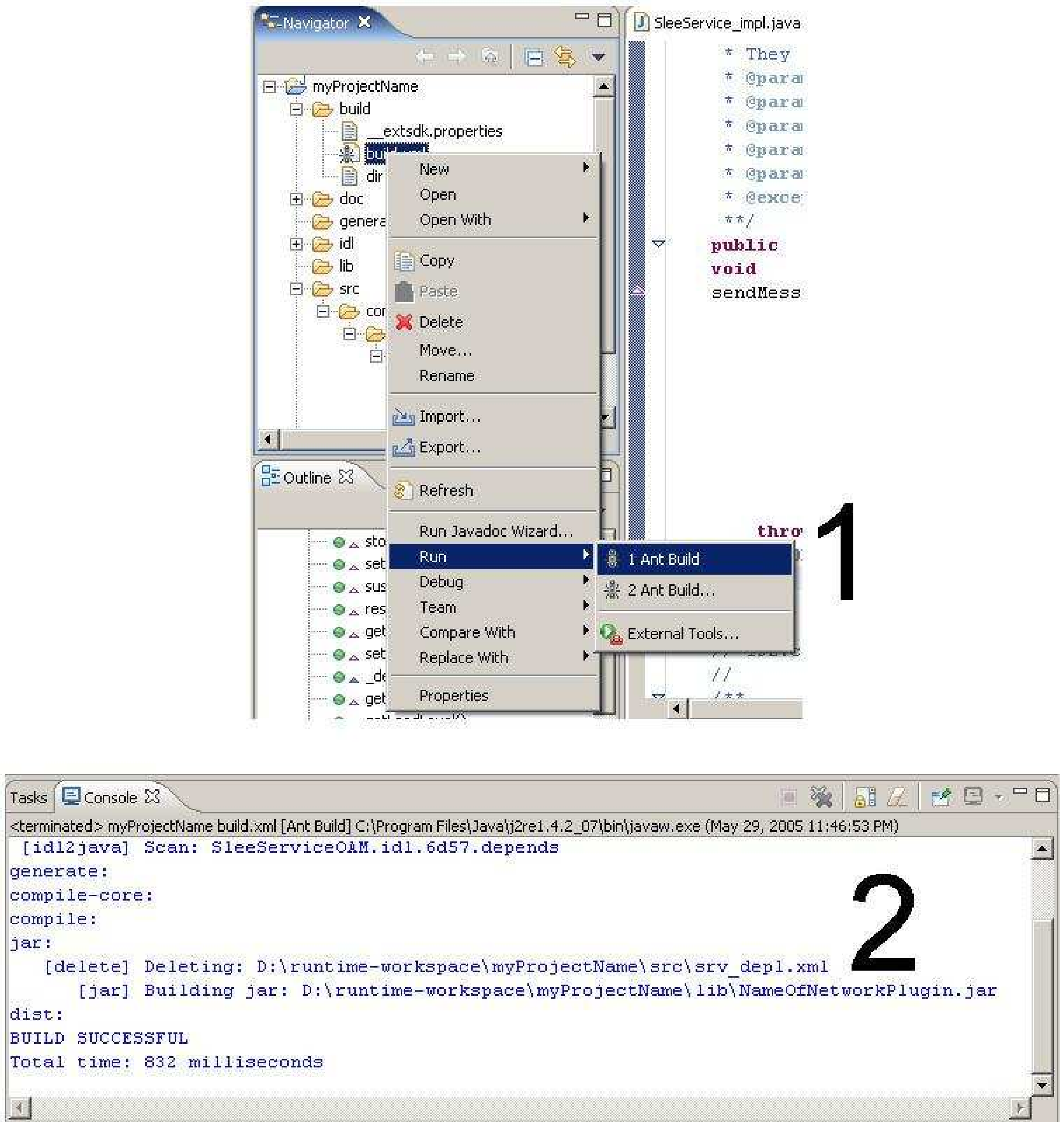


```
 * They will be used by the plug-in when delivering the acknowledgement for this m
 * @param [redacted]
 * @param [redacted]
 * @param [redacted]
 * @param [redacted]
 * @param [redacted]
 * @exception TrGMSException
 **/
public
void
sendMessageCtx([redacted]

   throws TrGMSException {
   //TODO implement this function
}

//
// IDL:com/incomit/resources/messaging/MessagingResourceExt/deleteMessageCtx:1.0
//
/**
```

Figure 1.2: Development Environment



```
<terminated> myProjectName build.xml [Ant Build] C:\Program Files\Java\j2re1.4.2_07\bin\javaw.exe (May 29, 2005 11:46:53 PM)
 [idl2java] Scan: SleeServiceOAM.idl.6d57.depends
generate:
compile-core:
compile:
jar:
   [delete] Deleting: D:\runtime-workspace\myProjectName\src\srv_depl.xml
      [jar] Building jar: D:\runtime-workspace\myProjectName\lib\NameOfNetworkPlugin.jar
dist:
BUILD SUCCESSFUL
Total time: 832 milliseconds
```

Figure 1.3: Compilation

the code has been generated and the developer can add his or her own functionality.

Finally, Figure 1.3 shows how the compilation is done in the project. First, the build-file is right-clicked and run. Second, the result from the compilation is shown in the eclipse console.

Note that the above scenario is how the Extension SDK would work in collaboration with eclipse. Using the Extension SDK with scripts instead of the Graphical Interface provided with eclipse would look different, but the result would be the same.

1.2 Reading Guide

To fully understand the contents of this document basic knowledge in computer science is required. Previous experience with SDKs, telecom architectures and object oriented approaches makes the reading easier, but is not required. The thesis is structured as follows:

Chapter 2. Background. This chapter explain knowledge areas needed to comprehend the contents of this thesis.

Chapter 3. Methods and Tools. This chapter describes some of the most important methods and tools used.

Chapter 4. Requirement Gathering. The requirement gathering is discussed in this chapter.

Chapter 5. Requirement Specification. The requirement specification is discussed in this chapter.

Chapter 6. Design and implementation. In this chapter the design and the implementation of the SDK are described.

Chapter 7. Results and Evaluation. This chapter describes the results and evaluation of the thesis.

Chapter 2

Background

2.1 Introduction

This chapter contains information about concepts and technologies which are necessary to comprehend in order to understand the content of this thesis. First, a historic overview of the evolving telecom architecture is given in Section 2.2. Second and third, telecommunication and company specific technologies, such as the Incomit Service Logic Execution Environment and The Incomit Application Hub are described. Fourth, the concept Software Development Kit is described. Finally, the Incomit Network Simulator is briefly discussed.

2.2 History of Open Network APIs

Extending the functionality of a network can from the network provider's point of view be done in two ways; Either by adding the functionality oneself, or by allowing outside developers to do it. The first solution is the easiest if new features will be implemented fairly infrequently. However, with a steadily increasing mobile market the strategy of allowing other developers into the network and extending it can be seen as the winning one. It should be noted that allowing people into the network makes certain restrictions necessary so that third party developers are unable to abuse the network.

The Intelligent Network concept (IN) [29, 11] was the first step in order to standardize an interface for outside developers so that it would not look different from network provider to network provider. The concept separates the service-specific software, which allows external applications to be deployed without interfering with the underlying network, from the service switching points.

Although IN solved the problem with deployment of applications it did less to the standardization part of the interface. It could be described as a first step toward a more open service architecture but it did not reach all the way. The third party developers still needed telecommunication expertise. What this meant for the telecommunication industry was that the development of services was not as fast as it could have been.

Two major standardization groups, JAIN [19] and Parlay [10], had proved that standardized and open network APIs decreased development time. They abstracted the telecommunication service creation from the telecommunication details. JAIN and Parlay did this with concern for the third party developers and the fact that Internet had become increasingly entangled with mobile telecommunication networks.

The Third Generation Partnership Project (3GPP) [2] and ETSI [1] were heading the same way defining the Open Service Access Standard (OSA) [26]. However, instead of being network agnostic like Parlay tried to be, they focused on the 3rd generation network. OSA consists of open network APIs that enables third party developers to develop and deploy their services into a telecommunication network without reducing security. Part of these APIs were taken from Parlay. Since the three organizations had common goals they decided to create a single development community by creating a Joint API group. This allow the organizations to synchronize their APIs. For further information about the history of the APIs of ETSI, 3GPP and Parlay see Chapter 2 and 3 in "Opening the networks with PARLAY/OSA APIs: standards and aspects behind the APIs" [3].

Incomit has built products on the open standards described above. In the next section the Incomit Application Hub will be discussed.

2.3 Incomit Application Hub

Incomit has two main products that aims to tie the data communication world to the telecommunication world. These are the Application Hub[13] and the OSA Gateway[14]. In this thesis the focus is on the Incomit Application Hub and the OSA Gateway will not be further described.

The Application Hub is intended to act as a gateway between the traditional telecom network and the IP network. It consists of both hardware and software components, both from Incomit and from third party companies. Incomit is able to include third party software and hardware into their Application Hub because of their decision to base their interfaces on open standards and open architecture (see Section 2.2). The application Hub is based on the Incomit SLEE described in the next section.

The Application Hub has several northbound APIs based on Java, CORBA or Web Services standards. The choice of API depends mostly on the preference of the developers and the demands of the application. Since security is often a concern of the operator, the Application Hub uses a central point of authentication for both third party applications and internal applications alike. One central point means only one point to protect.

Policy-based access control means that operators can dynamically configure their Service Level Agreement (SLA) [30] data according to contracts and security specifications. The fact that the policies can be changed dynamically means that operators can change rules without affecting the rest of the system.

Using policies in the Application Hub together with the subscriber profiles [1] allows access control to the services. For example, the Incomit Application Hub can check which

[1]data of people subscribing to services running on the Application Hub

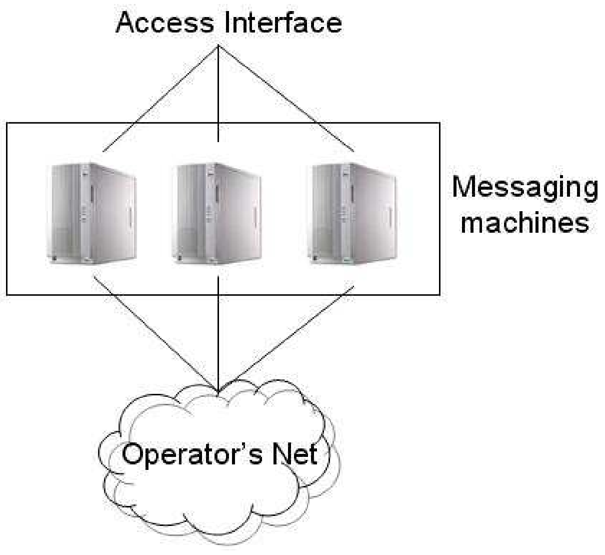


Figure 2.1: Horizontal Scalability

services the users subscribe to, which payment method to use, account status and more.

There is support for a wide range of standard network protocols which enables fast creation of applications that target the mobile networks directly through IP-based service nodes or via OSA/Parlay gateways (such as the Incomit OSA Gateway). The Application Hub also has built-in routing of services requests to other networks nodes, depending on the destination address.

The Application Hub is built with a modular architecture in mind, which makes it easier for service application developers to form their own unique service application from their needs and requirements.

How the system scales is very important in large commercial systems and Incomit's Application Hub is very scalable. It is scalable in both horizontal and vertical aspects. Horizontal scaling means that a specific part of the Application Hub can be run on several computers so that the traffic load is balanced. Thus, the Application Hub becomes capable of higher throughput. An example could be that a computer which runs a whole messaging

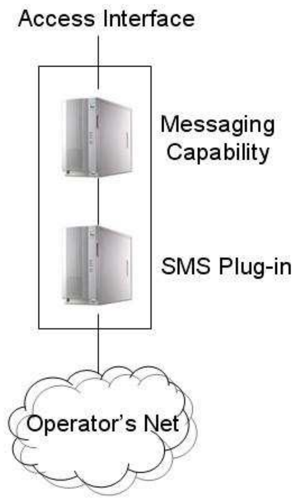


Figure 2.2: Vertical Scalability

pipe[2] is scaled over a number of computers. These computers run the same software, but they have a common access layer which selects which computer gets to handle the event from above (see Figure 2.1). Vertical scalability is when a software module is divided into pieces which are distributed over a number of computers. The messaging pipe could be divided so that an SMS plug-in resides on one computer and another part, for example the messaging service capability (which handles charging database access and routing of messages), resides on a different computer (see Figure 2.2).

The functionality of the Application Hub is created through the fact that it utilizes the Incomit SLEE technology which is described in the next section.

2.4 Incomit Service Logic Execution Environment

A Service Logic Execution Environment (SLEE, pronounced `[sli:]`) [15] is a container for applications and is a high throughput environment with low latency. The SLEE provides functionality for applications inside the SLEE for communicating with each other.

[2]all the way from the access interface on the top to the network specific plug-in at the bottom

Applications in the SLEE are referred to as services.

The Incomit SLEE runs as a Java application. The communication between the services in the Incomit SLEE is handled via a standardised architechture called the Common Object Request Broker Architecture (CORBA) [27]. CORBA is language independent and provides network distribution of objects which enables remote management of services in the SLEE. Below are the three interfaces described which has a connection to SLEE services.

ServiceDeployable. This interface has to be implemented in order for the SLEE service to be deployed and recognized by the SLEE.

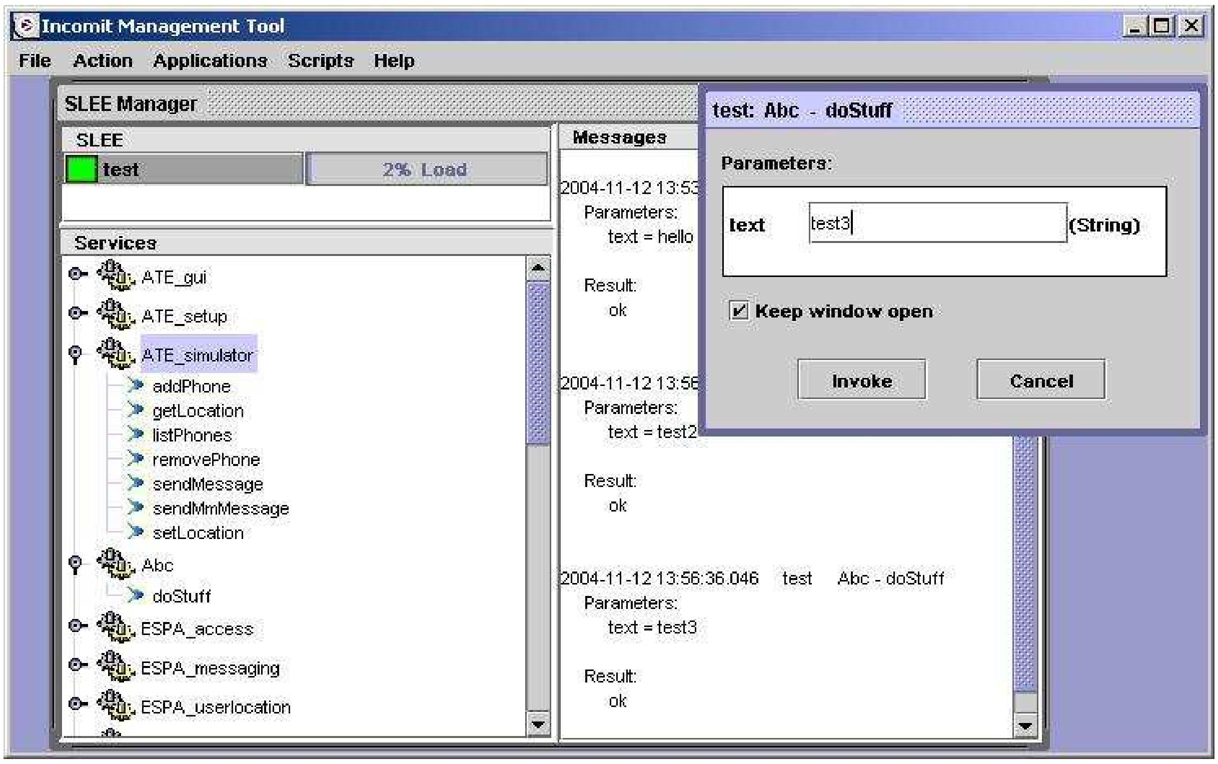
ServiceAccessible. If the service should be reachable by other services the ServiceAccessible interface has to be implemented. This interface enables external access to the SLEE service.

ServiceManageable. In order for the SLEE service to be managed by the SLEE manager this interface has to be implemented.

The SLEE has a number of SLEE utilities which can provide help with functionalities such as database access, timer functions, alarm generation and more. The SLEE utilities are also SLEE services.

The SLEE manager application exists within the Incomit Management Tool and is used to monitor and manage SLEE services (see Figure 2.3). On the lower left side the services deployed in the SLEE are listed. When clicked on, they display their accessible methods. The methods can be run through the manager by double clicking on them, inserting the necessary parameters into an appearing dialog, and then pressing the "invoke"-button. To the right, in the message field, responses and alarms are printed.

There are various designs and implementation of SLEEs. Other than Incomit's SLEE (the one described above), there is also the JAIN[19] SLEE architecture which has a technical approach which will not be described in this thesis.



text test3 (String)

```
2004-11-12 13:53
  Parameters:
    text = hello

  Result:
    ok

2004-11-12 13:56
  Parameters:
    text = test2

  Result:
    ok

2004-11-12 13:56:36.046     test     Abc - doStuff
  Parameters:
    text = test3

  Result:
    ok
```

Figure 2.3: Incomit Management Tool

2.5 Software Development Kit

Creating software code that is intended to be accessed by other developers produces a number of problems. One of the problems is how to easily educate developers in how to use the interface of that software code. This is commonly done by providing a Software Development Kit (SDK).

An SDK is a programming toolkit which can vary in form and therefore the definition also varies. It could be everything from a simple Application Programming Interface (API) to an extensive library of helpful documentation, code examples, tutorials and software wizards. This thesis will use the latter definition of an SDK, thinking of it more as giving a person not accustomed with a code library a helping hand in order to use its functionality.

In short an SDK is a number of parts that will help a developer understand how to use a piece of software. It is a kit with more or less advanced utilities and documentation. There are a number of SDKs out on the market; every big software product usually has one. As an example, in order to develop code in the programming language Java, it is recommended to use the Java SDK. The Java SDK consists of the Java API documentation, utilities for compilating and debugging created software as well as run-time files so the developer can execute the programs which he or she created. The approach with these parts, the API documentation, utilities and run-time files are quite common when describing an SDK.

2.6 Incomit Network Simulator

The Incomit Network Simulator is an application that simulates a real telecom network. Such an application is extremely valuable when testing a network plug-in as the testing can be done on a single workstation.

The network simulator is based on random behaviour. For example, in a call related environment, the application could simulate a normal call, a busy call, no answer or an error, depending on random values. The behaviour of the network simulator is configurable;

this means that the probability of, for example, a normal call could be altered so that it occurs more often.

The simulator provides a plug-in interface for each supported services capabilities (such as Call Control). In order to develop a network simulator plug-in the plug-in must conform to that interface.

The Network Simulator is used when testing new functionality included in the Application Hub. The SDK described in this thesis provides templates for Network Simulator interfaces, so that testing new services will be easier.

2.7 Summary

In this chapter the historical background of open network APIs has been described. Following the historic view, the Incomit products were explained, which are based on these open standards. The project described in this thesis is the creation of an SDK, therefore a definition of what an SDK is was described in this chapter. Finally, the Incomit Network Simulator was briefly described since the SDK will facilitate creation of plug-ins to the Network Simulator in order to simplify testing new parts. The next chapter consists of a description of the methods and tools used to create the SDK.

Chapter 3

Methods and Tools

3.1 Introduction

This chapter describes the different methods and tools which were used during the project. First, the development method used is explained. Second, the building tool Ant is described. Third, some important features with the programming language Java is listed and explained, since this is the language used throughout the project. Fourth, a description of the extensive markup language is made, and finally, other tools which have been used are listed.

3.2 Development Method

No known development method has been followed during the development of the SDK and it was never intended to use a method similar to eXtreme Programming (XP) [17]. Nevertheless, the way that refactoring was made, and the short deliveries with extensive testing was similar to that of XP. However, there are some major differences between the method the author of this thesis used and that of XP. First of all, XP recommends pair programming, which was impossible since there was only one person developing the

Extension SDK. Second, instead of tasks there were requirements which could have been divided into tasks, but this was never done. Instead, the requirements were implemented as they were, but divided into smaller parts by the developer. This method worked okay since there was only one person developing. However, it would have been chaos if more than one person had been involved. Another big difference is that XP programming more or less demands that the customer works physically close to the developers. In this case, the developer worked in the environment of the customer, but the developer was also one of the people making the requirements for the Extension SDK and therefore the customer *was* one of the developers in a sense.

3.3 Ant

Ant[7] is a java-based build tool which is used to compile Java code, comparable to the build tool "make". It is designed to be cross-platform, easy to use, and very flexible so that it can be used efficiently in a large as well as a small project.

It uses eXtensible Markup Language (XML) [28] as a file format, which enables very simple syntax, especially for those who have used XML before but also for beginners who manage to overcome the threshold one experiences when first encountering XML. The core concepts of Ant are simplicity, understandability and extensibility, which all basically express that it should be easy to comprehend and use.

An example is provided below in order to understand Ant in more detail. To further understand how Ant works compared to "make", we will compare how the building tools handle a building process. The make-example below require some understanding of makefiles and java.

```
All:          project.jar


project.jar:  Main.class XmlStuff.class
```

```
              jar -cvf $@ $<


%.class:      %.java
              javac $<
```

If the directory where the execution of the makefile is made has the files "Main.java" and "XmlStuff.java" then the make-example above will process the following:

```
Javac Main.java XmlStuff.java
Jar -cvf project.jar Main.class XmlStuff.class
```

Make is built on dependencies and targets. "All", "project.jar" and "%.class" are targets. `Project.jar`, located after the target "all", depends on `Main.class` and `XmlStuff.class` which in turn depends on its source code image `Main.java` and `XmlStuff.java` (shown in the example as the wildcards). The row underneath each target specifies the rule which creates the target. `$@` corresponds to the current target and `$<` corresponds to all dependencies.

```
<?xml version="1.0" ?>
<project name="makefile" default="all">
    <target name="all">
        <javac srcdir="." />
        <jar destfile="project.jar" includes="*.class"/>
    </target>
</project>
```

In the Ant-example above the `<javac>` command automatically selects all Java-files in the source directory which it compiles. The result of the two approaches are the same. In ant you define a project name which is not done with make. The default target is "all" which is defined to first compile all source-files and then create an `.jar`-file which consists of all `.class` files.

3.4 Java

This section will explain certain java-specific mechanisms from the perspective of a C++ programmer. Since the practical work in this thesis was done mostly using the Java programming language, an explanation feels justified in order to understand all design and implementation aspects. However, this section only briefly touches a number of features of the Java programming language and should not be seen as a crash course in Java.

Java uses only one file as the main code container, the `.java` file. This file contains one or more classes. A compiler who compiles a `.java` file creates a `.class` file which is executable if there is access to a Java Runtime library. Hence, a `.class` file cannot be executed on its own and demands external help. This approach enables Java to be platform independent even after compilation.

Java does not allow multiple inheritances; instead a class can inherit multiple interfaces, but only one class. In Java there are Interfaces and Classes. An interface cannot be instantiated and does not contain method definitions, only declarations.

The drawback with using Java has been the speed. Since Java is interpreting to some degree, Java is not as fast as native code. However, with the use of Just-In-Time (JIT) compilers and the new HotSpot [23] technology Java code can have near native code performance.

In Java, everything must consist within a class. There is no such thing as global functions or global data in Java. Therefore, to create similar functionality to that of globals, the developer would have to create static methods and static data within a class.

The preprocessor does not exist in Java. Therefore, Java developers use the keyword `import` in order to use classes from other libraries.

Pointers, in the sense from C and C++, do not exist in Java. When an object is created with the keyword `new`, a reference is returned.

In Java there is something called a Garbage Collector (GC). The GC's main task is to destroy objects which no longer have a reference to them. This makes memory leaks much harder to create in Java and makes a lot of programming problems, seen in for example C and C++, disappear.

For more differences and more detailed description about Java features from a C++ view see Homepage Comparing C++ and Java [5].

Further information on Java can be found on Sun's homepage `http://java.sun.com/` where they provide online books which are readable free of charge.

3.5 Extensive Markup Language

In order to explain what the eXtensible Markup Language (XML) [28] is, we should first look at its ancestor the Standard Generalized Markup Language (SGML) [16]. SGML is an international standard describing how data can be structured in a document, where SGML documents consist of text and graphics. SGML enables indentification and naming through its structure on parts of the information in the document. This format enables these parts to be used to create a range of products such as indexing, CD-ROM distribution and translation into foreign languages.

XML is created as a leaner type of SGML. A valid XML document is also a valid SGML document. XML uses only the most common SGML features and is thus easier to work with. XML has become mainly used for distribution of structured information over the web.

Another similar Markup Language is the Hyper Text Markup Language (HTML), which is used to display homepages on the internet. There is a difference, however. HTML has fixed tags that web browsers understand, whereas XML and SGML are "meta" languages where tags are not predefined but specificed by the person creating the XML or SGML file.

The following example is a simple example of an XML file:

```
<?xml version='1.0'?>
```

```
<GARAGE>
  <CAR>
    Ford
  </CAR>
  <CAR>
    Volvo
  </CAR>
</GARAGE>
```

This example illustrates a garage with two cars in it, a Ford and a Volvo. The first row identifies the XML-version of the file. The tag "GARAGE" defines the outer parameter of the file. Inside "GARAGE" are two "CARS".

3.6 Other Tools

For the creation of the diagrams under the requirement chapter (Chapter 4) Microsoft Excel was used. The class diagrams in this thesis were created using Rational Rose Enterprise edition. Some of the pictures under the design chapter were created in Microsoft PowerPoint.

Chapter 4

Requirement Gathering

4.1 Introduction

This chapter contains the different parts that will be considered and judged in order to create the requirement specification for the SDK. First, the requirement specification that already existed before the work for this thesis was begun is considered and evaulated. Second, a review is made of the two development environments JBuilder and eclipse. Third, a survey regarding the thoughts and populartity of different development environments is discussed. Fourth, another survey is presented, discussing the features in development environments. Finally, a conclusion of this chapter is made.

4.2 Old Requirement Specification

Incomit already had a Requirement Specification (RS) for the product that they wanted done. They had named it "Extension SDK" and it was an SDK for Extending the functionality of their Application Hub. It was meant for the Extension SDK to facilitate the creation of services [1] in the Application Hub. The old RS was qritten quite some time

[1]for more information regarding services see Section 2.4

before the work for this thesis had even started. The naming conventions in the old RS was a bit different from what they used at the point in time when this thesis was conducted. That RS was designed for the use in their other product, called the OSA Gateway, so there was a need to rewrite this requirement specification to fit the current needs. The old RS is added to this thesis in the Appendix A.1. It is not documented, hence it should be treated as an external document. The old RS reflects what was availible at the beginning of this thesis.

Some core requirements in the old requirement specification were:

- Code Templates
- Code Examples and Class Libraries
- Facilitating Development of Network Plug-ins
- Standalone
- Documentation
- Installation Wizard
- Other IDE support

For more detailed description of the items above please see Appendix A.1.

4.3 Evaluation of JBuilder and eclipse

In this section a comparison will be made between JBuilder and eclipse. First, the two different development environments are described including their background. Second, the comparison will be made regarding several aspects, including some of the features described in the survey in Section 4.5. This is done in order to determine which development environment to support for the Extension SDK. Research regarding the development environments was something which was regarded as important by Incomit.

4.3.1 JBuilder

Jbuilder is an Development Environment developed by Borland, which comes with many helping utilities out of the box. These helping utilities speeds up the development of Enterprise JavaBeans[22], Web Services[18], XML[28], database applications and more.

Borland started in 1983 to create software to simplify and speed up development of new software. They launched one of the first development environments with "Turbo Pascal", making them one of the leaders in software development tools. Since then, they have refined and tuned their software in order to keep in phase with what the software developers needed next.

Borland's solutions and services are used by organizations all over the world, ranging from public sector and healthcare to telecommunications and financial service firms.

4.3.2 Eclipse

Eclipse is a platform operating under open source that targets tool developers. Being royalty free makes eclipse a attractive choice when deciding which development environment a company should use for their tool extension.

Eclipse uses a plug-in based framework that facilitates creation and integration. It also utilizes software tools, saving both time and money. The tool producer can concentrate on the core functionality of the new software tool since the help from the eclipse platform is pretty extensive. The Eclipse Platform is written in the Java language and comes with extensive plug-in construction toolkits and examples. It has already been deployed on a range of development workstations including Linux, HP-UX, AIX, Solaris, QNX, OSx and Windows based systems. A full description of the Eclipse community and white papers documenting the design and use of the Eclipse Platform are available at `http://www.eclipse.org`.

Borland, IBM, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft and Webgain formed the first eclipse.org Board of Stewards in November 2001. Since then, a number of companies has joined the managing board, companies and groups such as Fujitsu, Hitachi, HP, OMG, Oracle and Intel. Needless to say, many large companies have an eye on eclipse, which predicts a promising future for the platform.

4.3.3 Review Process

In order to compare the two development environments a design for a simple application was made. The application reads from an XML-file and prints a string representation of the file, suitable for debugging. It uses the XML-parser from JDOM[12], which is provided as an external jar-file. The purpose of this application was to compare a number of features the different development environments use. The application was designed to rely on third party software, so that is becomes clear how easy it is to integrate applications with third party software in the development environment. First, every item which is looked at when reviewing each development environment is listed. After that, a walkthrough of how JBuilder 2005 Enterprise (version 11.0.236.0) handled the creation of the application is described. Next, eclipse platform's (version 3.0.0) creation of the application is reviewed. The reviews are made totally separate, and comparisons will be made after both reviews. Finally, a conclusion of the evaluation between JBuilder and eclipse is made.

It will be observed how the development environment handles the following features:

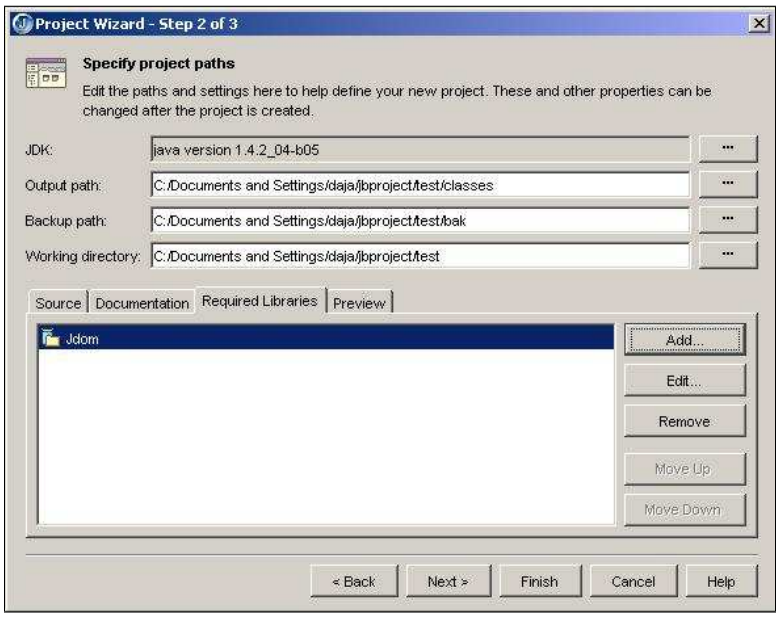
Creation of the Java-class. How does the development provide help when creating a Java-class?

Errors. How does the development environment convey errors?

Code completion. How does the development environment handle code completion [2]?

Execution. How does the development environment handle execution of applications?

[2]code completion is explained further in Section 4.3.4 and 4.3.5


Project Wizard - Step 2 of 3

Specify project paths

Edit the paths and settings here to help define your new project. These and other properties can be changed after the project is created.

JDK: java version 1.4.2_04-b05

Output path: C:/Documents and Settings/daja/jbproject/test/classes

Backup path: C:/Documents and Settings/daja/jbproject/test/bak

Working directory: C:/Documents and Settings/daja/jbproject/test

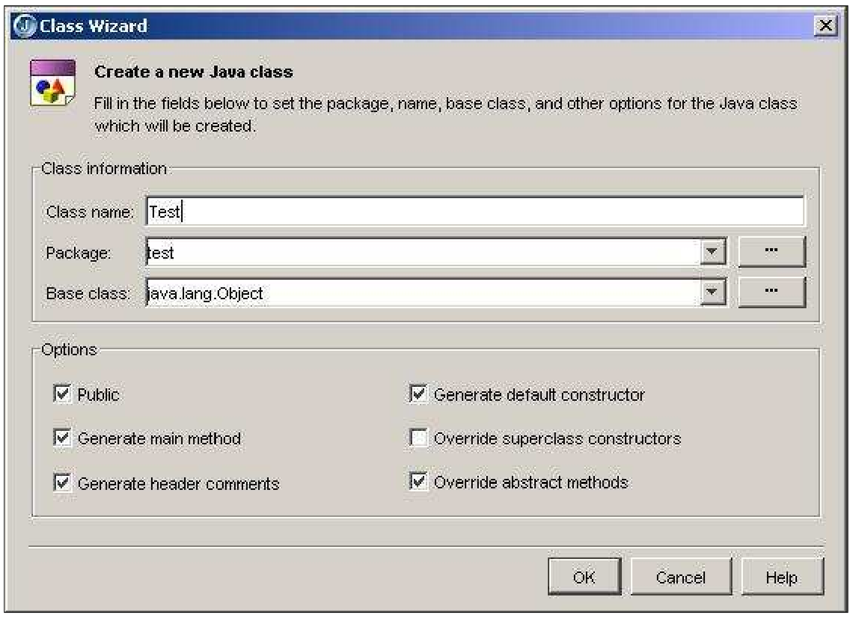
Source | Documentation | Required Libraries | Preview

Jdom

Figure 4.1: JBuilder - Project Wizard

4.3.4 Running a test in JBuilder

Creating a class in JBuilder requires that that class has a project. Therefore, when trying to create a class without having a project, the project wizard is opened so that the creation of a project can be done first. The project wizard's first page is where the name, location and template usage of the project is entered. The second page handles project paths such as location of the Java Development Kit (JDK), the output directory, the backup directory and the working directory. In this page, the required libraries (see Figure 4.1) can be entered. This is where the location of the `jdom.jar` is entered. The third page handles general project settings. When the project wizard has completed, the class wizard pops up. This wizard has only one page where name of the class, the package it should belong to, and base class is specified. The page also contains certain class specific options such as if the class should be accessible from outside the package and if a main method should be created. A main method is preferred in this case since only a few lines of code is necessary rather than some advanced object oriented structure. This is displayed in Figure 4.2.



Class Wizard

Create a new Java class

Fill in the fields below to set the package, name, base class, and other options for the Java class which will be created.

Class information

Class name:	Test
Package:	test
Base class:	java.lang.Object

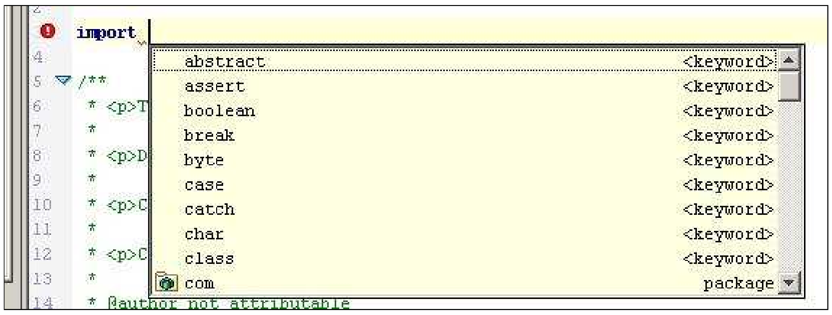
Options

- ✓ Public
- ✓ Generate main method
- ✓ Generate header comments
- ✓ Generate default constructor
- ☐ Override superclass constructors
- ✓ Override abstract methods

Figure 4.2: JBuilder - Class Wizard

When the class wizard is completed, JBuilder creates mandatory code for the class, including javadoc as well as the main method that was chosen during the class wizard. JBuilder added a row in the generated main-method that instansiated the class. This feature would be useful in most cases, but in this cases it is not - therefore it is removed.

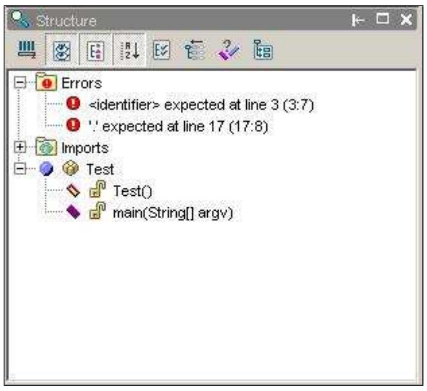
Next comes the importing of the JDOM-specific classes that is needed to retrieve the XML-file. Starting to add the import-line, in order to read the XML from files, streams



```
import
/**
 * <p>T
 *
 * <p>D
 *
 * <p>C
 *
 * <p>C
 *
 * @author not attributable
```

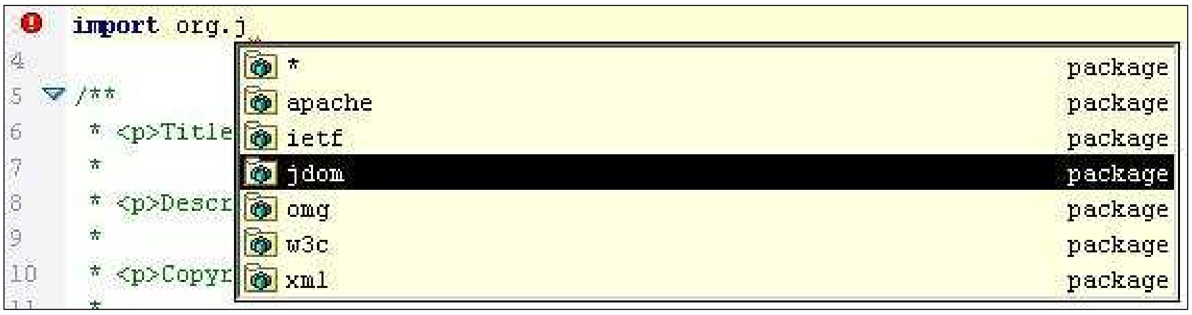
```
abstract        <keyword>
assert          <keyword>
boolean         <keyword>
break           <keyword>
byte            <keyword>
case            <keyword>
catch           <keyword>
char            <keyword>
class           <keyword>
com             package
```

Figure 4.3: JBuilder - Error and Code Completion



```
Structure
Errors
    <identifier> expected at line 3 (3:7)
    ';' expected at line 17 (17:8)
Imports
Test
    Test()
    main(String[] argv)
```

Figure 4.4: JBuilder - Structure Window

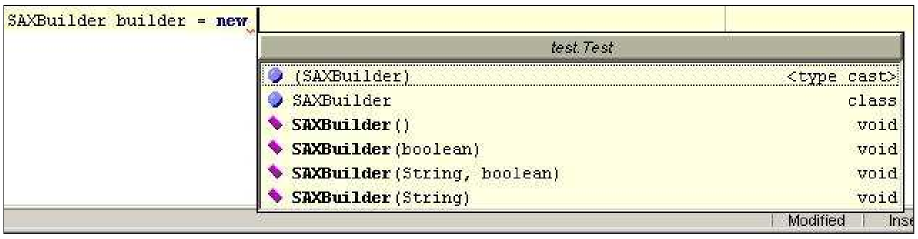


```
import org.j

/**
 * <p>Title
 *
 * <p>Descr
 *
 * <p>Copyr
 *
```

```
*          package
apache     package
ietf       package
jdom       package
omg        package
w3c        package
xml        package
```

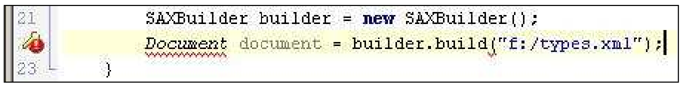
Figure 4.5: JBuilder - Import Code Completion



```
SAXBuilder builder = new
```

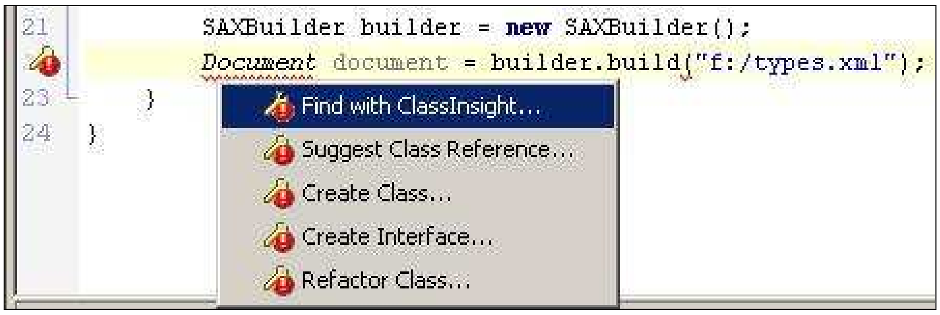
```
test.Test
(SAXBuilder)                  <type cast>
SAXBuilder                    class
SAXBuilder()                  void
SAXBuilder(boolean)           void
SAXBuilder(String, boolean)   void
SAXBuilder(String)            void
```

Figure 4.6: JBuilder - Code Completion



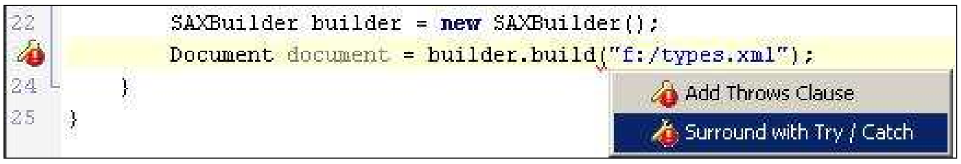
```
        SAXBuilder builder = new SAXBuilder();
        Document document = builder.build("f:/types.xml");
    }
```

Figure 4.7: JBuilder - Fixable Error



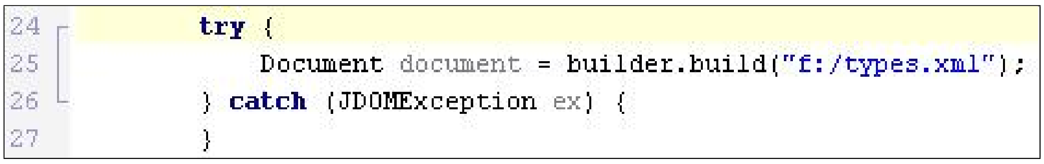
```
        SAXBuilder builder = new SAXBuilder();
        Document document = builder.build("f:/types.xml");
    }
}
```

Figure 4.8: JBuilder - Solution Suggestions



```
        SAXBuilder builder = new SAXBuilder();
        Document document = builder.build("f:/types.xml");
    }
}
```

Figure 4.9: JBuilder - Solution Suggestions 2



```
        try {
            Document document = builder.build("f:/types.xml");
        } catch (JDOMException ex) {
        }
```

Figure 4.10: JBuilder - Added Try and Catch

and more with help from the "SAXBuilder", creates an error which is conveyed in two ways to the developer. The first one is the line (which can be seen as a small blur directly after the word import in Figure 4.3) under recently added text. The second presentation of the error lies in a window to the left and provides more information about the error; what it specifically is and the line number where it is located, as shown in Figure 4.4. As the import code line is fully written the errors disappear. Figure 4.5 shows the code completion JBuilder provides the developer. Starting to add the SAXBuilder into the code reveals another code completion feature from the development environment. Here JBuilder provides a code completion when the class is instantiated (shown in Figure 4.6) which speeded up the coding. JBuilder also conveys which parameters that can be entered into a method in the same way. To retrieve the XML-data, a document to put the data into is needed. Trying to create the document without importing the Document class will cause an error as the compiler is not able to find the Document class. This is a test in order to see if JBuilder can provide help for fixing this error, and if so - how? An icon that looks like a wrench, with a red circle with an exclamation mark in it, appears to the left of the editor window (see Figure 4.7). Had the development environment been unable to help the developer, the icon to the left would have been only a red circle with a white exclamation mark in it. Left clicking on the wrench shows a number of options that will solve the error (shown in Figure 4.8). In this case a new import is preferred and therefore the option "Find with ClassInsight..." is chosen. This option pops up a window with different import-options where ClassInsight has found the class "Document". Choosing an option here and clicking OK will add the import line to the code. When this is done, JBuilder shows the developer that there is still an error with this seemingly error free code line. Apparently, the build method throws an exception which must be caught or thrown. Getting information about this is done by placing the mouse pointer over the red line that underlines a piece of code, or by looking at the window to the right. Luckily, this can be done automatically by JBuilder with a left click yet again on the yellow wrench-icon to

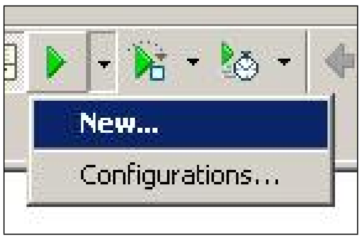
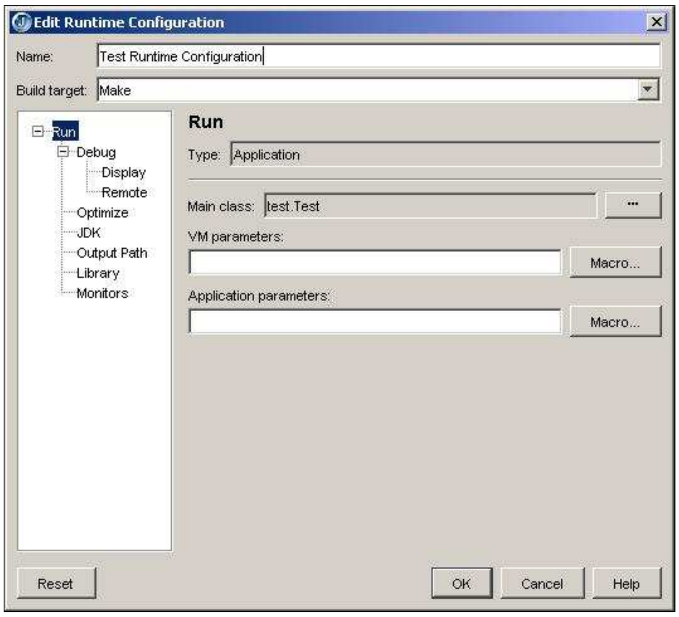


Figure 4.11: JBuilder - Execute

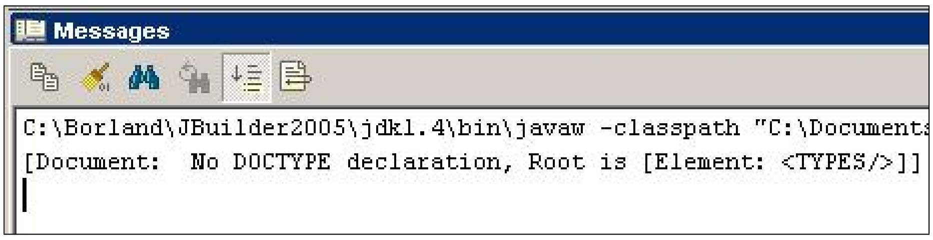


Name: Test Runtime Configuration
Build target: Make
Run
Type: Application
Main class: test.Test
VM parameters:
Application parameters:

Figure 4.12: JBuilder - Config Window

the left of the editor-window (see Figure 4.9). Now there are two different suggestions to solve the error; throwing the exception or enclose it with try and catch brackets. Choosing the second option creates code around the line which the development environment was complaining about (see Figure 4.10). Adding a line, that prints the information about the exception, to the catch bracket provides debugging information. The last code line that is added is the printing of the debug information of the XML-file.

In order to run the application, the configuration of the project has to be made. This is done through the green "play" button on the toolbar(see Figure 4.11). Since this is the



```
C:\Borland\JBuilder2005\jdk1.4\bin\javaw -classpath "C:\Documents
[Document:  No DOCTYPE declaration, Root is [Element: <TYPES/>]]
```

Figure 4.13: JBuilder - Message Window

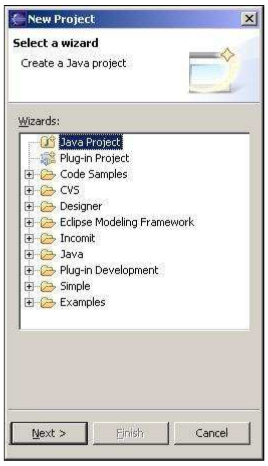
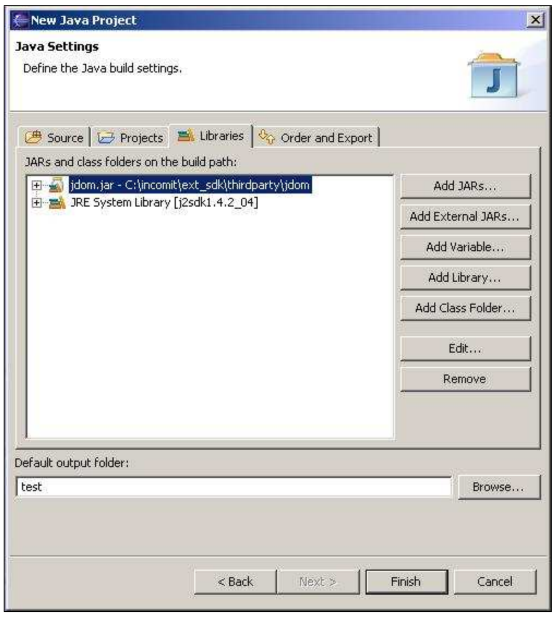


Figure 4.14: Eclipse - New Project

first time that the execution button is pressed for this project, a configuration window (see Figure 4.12) is shown where configuration of how the execution should be made is shown (for instance, what class that should be executed). When all configurations are done and the execution button is pressed, then the system output is presented in the "messages" window at the bottom, as can be seen in Figure 4.13. The output of the application is of less importance for this review.

4.3.5 Running a test in Eclipse

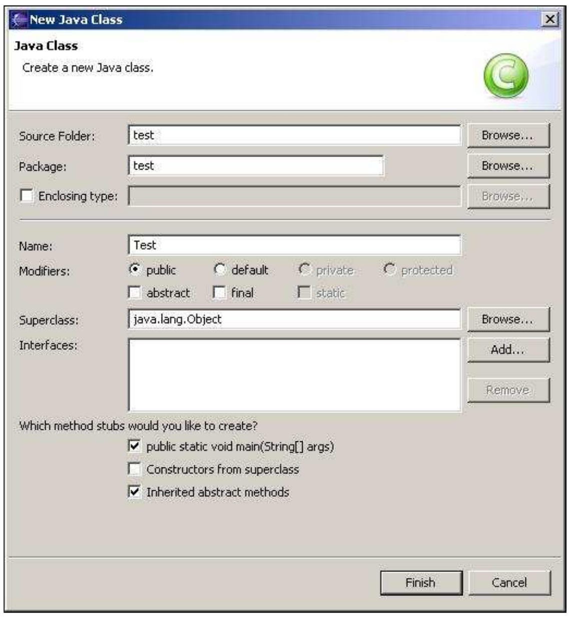
A folder is needed when trying to create a java-class. Eclipse demands that each file is encapsulated in a project and thus a project must be created first and after that a class



JARs and class folders on the build path:
jdom.jar - C:\incomit\ext_sdk\thirdparty\jdom
JRE System Library [j2sdk1.4.2_04]

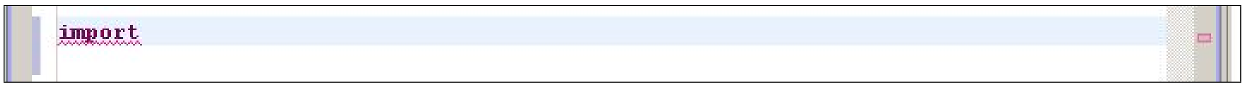
Default output folder: test

Figure 4.15: Eclipse - New Java Project



Source Folder: test
Package: test
☐ Enclosing type:
Name: Test
Modifiers: ● public ○ default ○ private ○ protected
☐ abstract ☐ final ☐ static
Superclass: java.lang.Object
Interfaces:
Which method stubs would you like to create?
☑ public static void main(String[] args)
☐ Constructors from superclass
☑ Inherited abstract methods

Figure 4.16: Eclipse - New Java Class

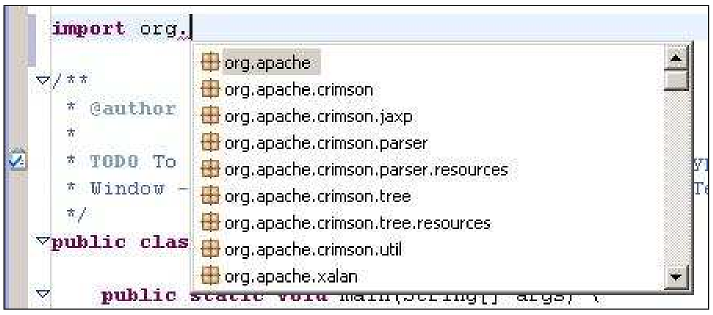


```
import
```

Figure 4.17: Eclipse - Error

can be created. Opening the project creation page shows several different types of projects that can be created. Eclipse is not designed only for Java related creation but is more of a platform with several functionalities built into it. Therefore, creating Java code is only a portion of the eclipse development environment. Selecting the "Java Project" in the project creation window (Figure 4.14) and pressing next show the "New Java Project" page where the name and location of the project is entered. The page shown after the project creation page is about the java settings, where the library tab has functionality for adding the external jar file "`jdom.jar`" which is needed for this application (Figure 4.15). When "Finish" is pressed, eclipse prompts a question if the developer would like to change to the "Java Perspective" (perspectives in eclipse means initial set and layout of views in the desktop development environment). Answering "Yes" to this popup changes the GUI of the development environment. Now the developer can either chose a new class from the File->New->Class or right-click on the "test" folder and choose New->Class. This action will launch the "New Java Class"-wizard which only consists of one page (Figure 4.16). This page enables the developer to enter name, package and - in this case a quite useful feature - if any method stubs should be generated and put into the class. Checking the option that it would be preferred to have the main-class included and then pressing "Finish" generates the java-file.

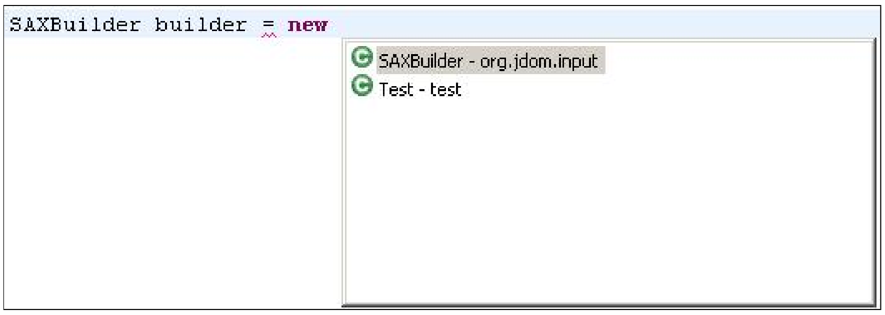
Now when the java-file is created, the external classes have to be imported. Having written an incomplete line renders an error which is marked as a red line under the erroneous code and as a small red box displayed next to the scrollbar to the right (Figure 4.17). Moving the mouse pointer over the red line shows the error message and a possible solution. Continuing to write on the import line shows how eclipse handles the code completion (see



```
import org.
/**
 * @author
 *
 * TODO To
 * Window -
 */
public clas
    public static void main(String[] args) {
```

```
org.apache
org.apache.crimson
org.apache.crimson.jaxp
org.apache.crimson.parser
org.apache.crimson.parser.resources
org.apache.crimson.tree
org.apache.crimson.tree.resources
org.apache.crimson.util
org.apache.xalan
```

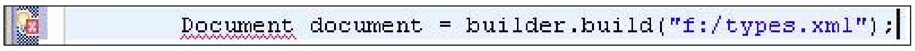
Figure 4.18: Eclipse - Import Code Completion



```
SAXBuilder builder = new
```

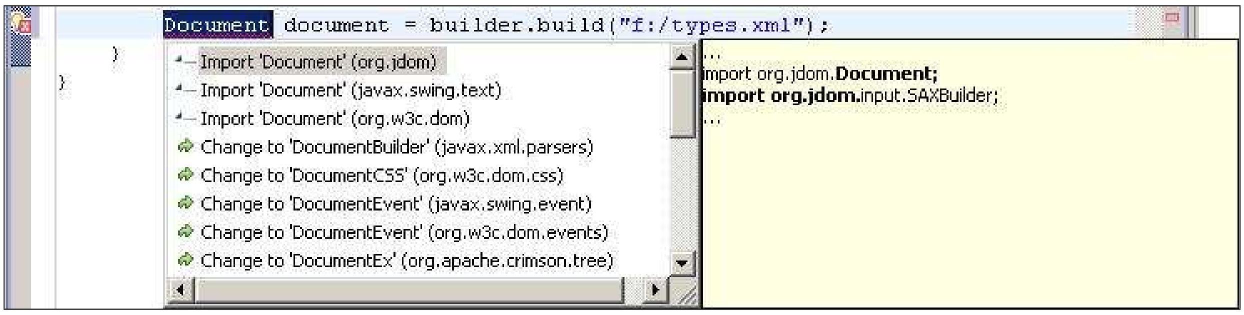
```
SAXBuilder - org.jdom.input
Test - test
```

Figure 4.19: Eclipse - Code Completion



```
Document document = builder.build("f:/types.xml");
```

Figure 4.20: Eclipse - Fixable Error

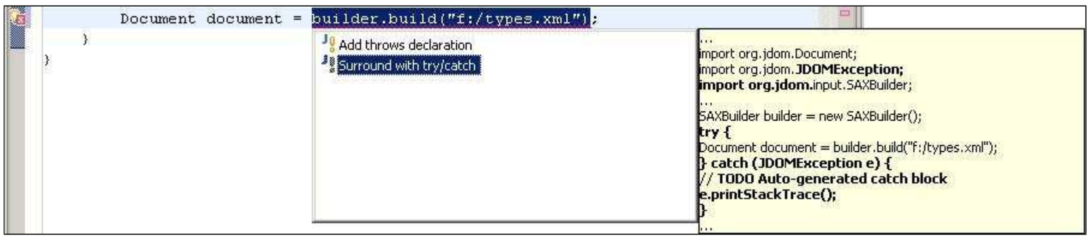


```
Document document = builder.build("f:/types.xml");
    }
}
```

```
Import 'Document' (org.jdom)
Import 'Document' (javax.swing.text)
Import 'Document' (org.w3c.dom)
Change to 'DocumentBuilder' (javax.xml.parsers)
Change to 'DocumentCSS' (org.w3c.dom.css)
Change to 'DocumentEvent' (javax.swing.event)
Change to 'DocumentEvent' (org.w3c.dom.events)
Change to 'DocumentEx' (org.apache.crimson.tree)
```

```
...
import org.jdom.Document;
import org.jdom.input.SAXBuilder;
...
```

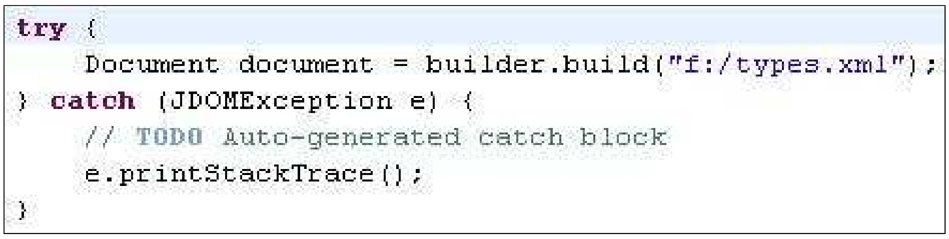
Figure 4.21: Eclipse - Solution Suggestions



```
        Document document = builder.build("f:/types.xml");
    }
}
```

```
...
import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
...
SAXBuilder builder = new SAXBuilder();
try {
Document document = builder.build("f:/types.xml");
} catch (JDOMException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
...
```

Figure 4.22: Eclipse - Solution Suggestions 2



```
try {
    Document document = builder.build("f:/types.xml");
} catch (JDOMException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

Figure 4.23: Eclipse - Added Try and Catch

Figure 4.18). The code completion of the import-archives was done automatically, but when a class is searched for, the developer must use the keyboard button combination Ctrl and Space to display the code completion. When the SAXBuilder is imported, a yellow line is displayed under the import-line. This yellow line signals that the import is not used in the java-file. This is effective for removing unused imports, but this time it is just a comment since the import will indeed be used. Creating the SAXBuilder is pretty straight forward. Code completion, such as help with what classes that could be instantiated, can be provided through pressing the Ctrl and Space keys (Figure 4.19). Adding the document that should hold the XML-data generates an error by eclipse. Placing the mouse pointer on the red lined word or on the icon to the left in the editor window pops up a tooltip that presents what kind of error it is. In this case eclipse cannot find the class Document, so that would have to be included (see Figure 4.20). Eclipse has a feature that gives a solution to such problems. Left-clicking on the icon which looks like a light bulb with a red box in front of it shows a number of solutions to this problem (Figure 4.21). The first

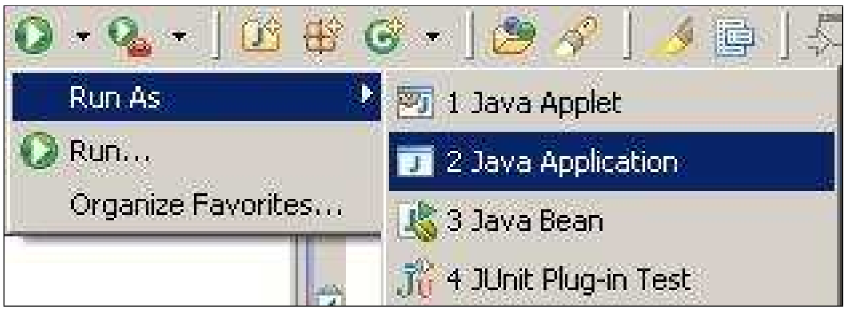
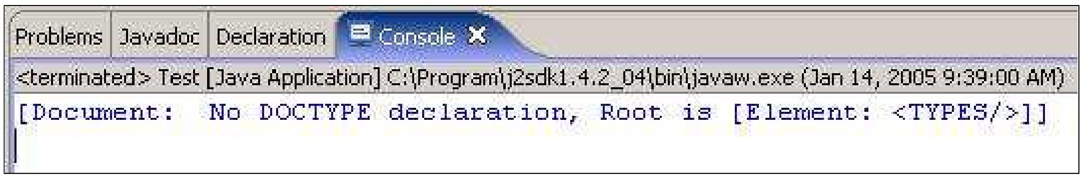


Figure 4.24: Eclipse - Execution



```
<terminated> Test [Java Application] C:\Program\j2sdk1.4.2_04\bin\javaw.exe (Jan 14, 2005 9:39:00 AM)
[Document:  No DOCTYPE declaration, Root is [Element: <TYPES/>]]
```

Figure 4.25: Eclipse - Console

solution listed is to import the org.jdom.document which is the one the developer would want in this case. To the right of the suggested method menu there is a presentation of what the suggested method would actually do if executed. After that action is performed, eclipse marks the creation of the XML-building with the red lines. Clicking on the light bulb icon this time generates the suggestion list found in Figure 4.22. The adding of the try and catch brackets automatically adds a printout of the exception error if caught. A new import has also been made during the adding of the try and catch code. The last thing added is the debug presentation of the document.

Execution of the application is done through the green play button on the toolbar (Figure 4.24). Clicking on the side-arrow shows numerous ways to execute the application. Clicking on "Run As->Java Application" makes eclipse ask the developer if he or she wishes to save the file first, and if the developer choose to save, the application is run. It should be noted that the printing is done under the "console" tab, which is not displayed as default (Figure 4.25).

4.3.6 Conclusion

The comparison has followed a certain scenario and the comparison will therefore only touch the aspects which were evaluated. There are other aspects which also would have been interesting to compare. One such ascpect is how the different development environments handle version handling and a possible integration with the Concurrent Versions System (CVS) [4, 9]. Another aspect is how easy extensions of the development environment can be made and how they would work. But due to time limitations, these aspects have not been reviewed. The comparison was also done from a default configuration without much knowledge of how the actual development environment functioned in detail. This was as intended, as the review was supposed to look at how JBuilder and eclipse handled the creation of the XML-debug-writing application from default values and settings. It could be argued that some of the features that in this review are seen as "less good" could be changed in the settings to that of the reviewers liking, but this review does not take such "after changeable settings" into account.

The project creation of the two IDEs are very similar in that both require that a project exist. The advantage here is leaning towards JBuilder since if a developer would try to create a class in JBuilder, the project creation page launches so that the creation of the project can be done. In comparison, eclipse launches the class-creation page but demands a folder, which at start does not exist. There was no obvious way to create a folder from the class-creation wizard, so that wizard had to be closed and a project had to be created manually.

The way `jdom.jar` was included as a library in the class path was equally easy comparing the two development environments. What could have been interesting to see was how JBuilder and eclipse handled dependencies between projects and if code completion had worked in that case. Another interesting comparison would have been to see how easy the deployment of plug-ins and the like would have been for JBuilder, especially when the plug-in would have external dependencies, since this was a bit of a hassle in eclipse (see

Section 7.1).

Error handling was similar in the environments but not identical. Both used the red underlining technique to convey errors to the developer. But in order to specify where in the file the error was, eclipse used a little red box near the scrollbar. JBuilder used a part in a window where the line number was written as well as a text describing the problem. The author of this thesis preferred the eclipse solution to the JBuilder solution since it could possibly give a better overview of the amount of errors located in a certain area in the code. For example, imports of external code often would need some adapting to the current code. Using the error conveying on the right side of the editor window would give the developer a feel of where the errors are located and what needs to be fixed and adapted. JBuilder would in that respect have problems conveying information about the density of errors in a certain area, since an amount of numbers is harder to quickly translate into positions then actually seeing where the errors are located. It should be noted that both development environments used a real-time Java compilation in order to get these errors. Receiving the errors directly when developing is certainly good, but the downside is that the programming environment demands more resources from the workstation.

Code completion had some minor differences between the two applications. One relevant difference was the fact that code completion often came automatically when using JBuilder where in eclipse the developer in most cases had to use the Ctrl + Space combination. For beginners in the Java development world, the key combination might have been news to them. If that would have been the case they would have missed a great deal of help from eclipse. In the case when a developer is unfamiliar with the key-combination JBuilder would have been preferred, but considering that code completion could be unnecessary in certain scenarios (because the developer knows what he or she wants to write and the code completion takes some resources) the author of this thesis consider neither JBuilder or eclipse the winner of the code completion comparison.

When running the application, JBuilder had no seemingly fast way to do this. The developer had to configure how the project should be run, whereas in eclipse only two easy clicks were needed to get the application to run. The downside was that in this case (with output to the console) it felt like nothing happened in eclipse when the program was executed. The developer had to find the console tab to find that something had actually happened. In JBuilder, on the other hand, there was clear output in the message windows as well as sound effects being played at execution and termination of the application. The sound effects were experienced as something unnecessary and could easily be frustrating (as there is a sound effect when the program crashes too). The advantage here is leaning towards eclipse.

Even though this application was very simple, it shows certain key features in each of the development environments. The winner in this particular case is eclipse, but not with a large margin, and this is the subjective statement by the reviewer. The development environments were very similar in usage. It should be noted that with the development of new IDEs, the developers has to upgrade their computers. The JBuilder 2005 Enterprise edition has a minimum memory requirement of 512Mb RAM which could be seen as high. The memory requirement of eclipse could not be found by the reviewer but an qualified guess would be that the requirements of eclipse are equal to those of JBuilder. The memory requirement is probably due to the real-time compilation done by the development environments.

Feature	**Advantage**
Creation of the Java-class	JBuilder
Error conveying	eclipse
Code Completion	-
Execution	eclipse

Table 4.1: Comparison between JBuilder and eclipse

4.4 Development Environment Survey

To collect information about different development environments at Incomit, a survey was produced. The survey guidelines used is described in "Vetenskaplig metod"[6] by Rolf Ejvegård. These guidelines describe how questions should be written, how the layout of the survey should be designed and if any significant statistical information can be derived from it.

An e-mail was sent out to the developers' mailing list which redirected the mail to nineteen people. Fifteen developers at Incomit answered the survey. Four persons could not answer because of legitimate reasons; one person was on parental leave, one person was the author of this thesis, and another two had occupations not related to code creation. Of the remaining fifteen people all answered. This means that there was a 100% response rate which, of course, is the best response rate one can have.

Below are the questions shown in the form they were asked. The questions were asked in Swedish, but are translated in this thesis into English. The results of the answers are discussed after each question.

4.4.1 Awareness of Development Environments

1. Which of the following development environments have you heard about?
(Place an X after the option which suites you the best)
eclipse ( )
JBuilder ( )
Netbeans ( )
None of the above ( )

By using the question above, an overview of the employees general knowledge could be identified. The result of this question is shown in Figure 4.26 and shows that the employees had a good knowledge of what kind of development environments that are

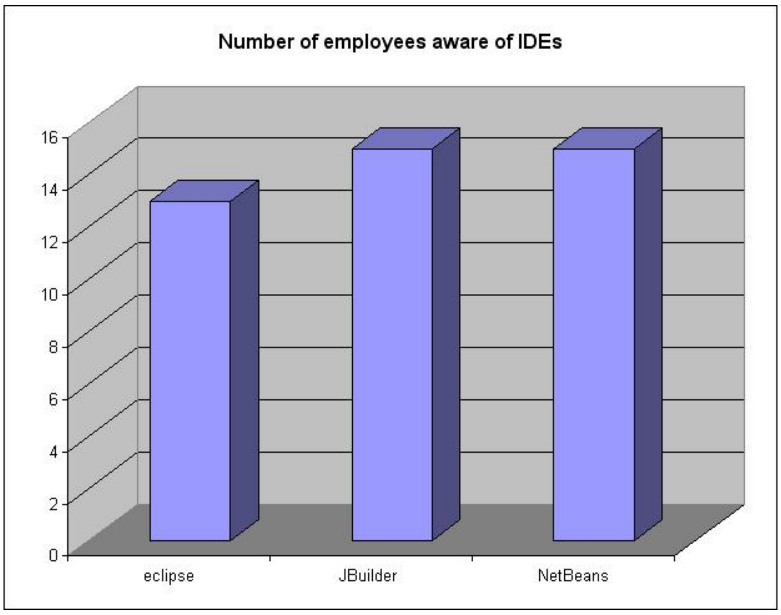


Figure 4.26: Awareness of IDEs

available. Surprisingly, the only IDE which not everybody knew of was eclipse which had the highest priority to be integrated with the Extension SDK. It should noted that none of the employees answered "None of the above".

4.4.2 Evaluation of Development Environments

2. Which of the following development environments have you tried?
(Place an X after the option which suites you the best)
eclipse ( )
JBuilder ( )
NetBeans ( )
None of the above ( )

Only three out of fifteen developers had used eclipse while thirteen had used NetBeans (see Figure 4.27). Although nearly half had used JBuilder, only three felt confident enough

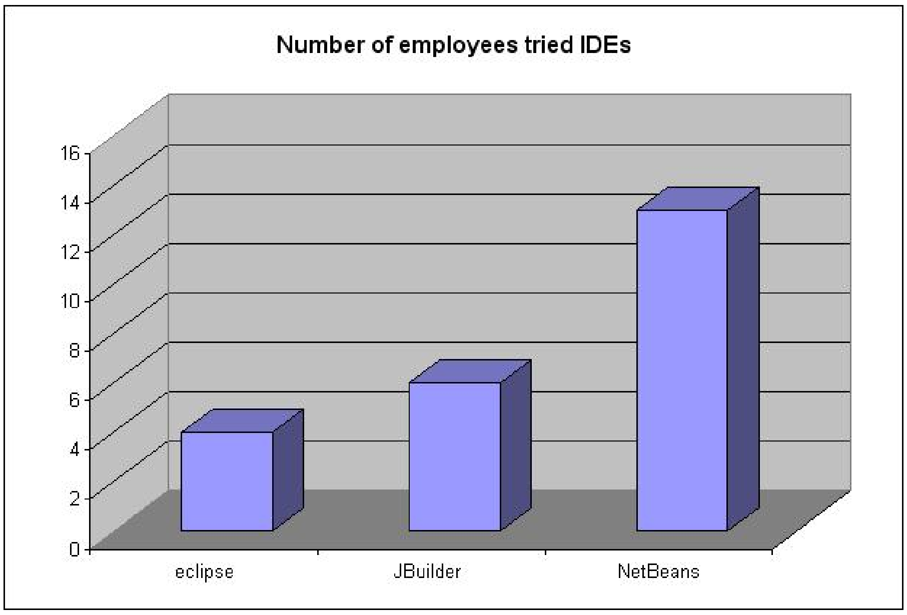


Figure 4.27: Tried IDE

to grade it (see Figure 4.28).

4.4.3 Grading of eclipse

3. If you should grade the development environment 'eclipse', what grade should it receive?
On a scale from 1 - 5 (where 1 is 'bad' and 5 is 'excellent')
If you haven't tried it enough to grade it put an X as answer.
Please give a short motivation for your answer: ( )

In the third to fifth questions the developers answered questions on how they like different development environments. Only three people graded eclipse but the grades eclipse were given were clearly the highest in the survey (see Figure 4.28 and Figure 4.29). The only negative comment about eclipse was that the learning threshold for eclipse was a bit high. Some of the positive feedback was that it was easy to use, easy to integrate

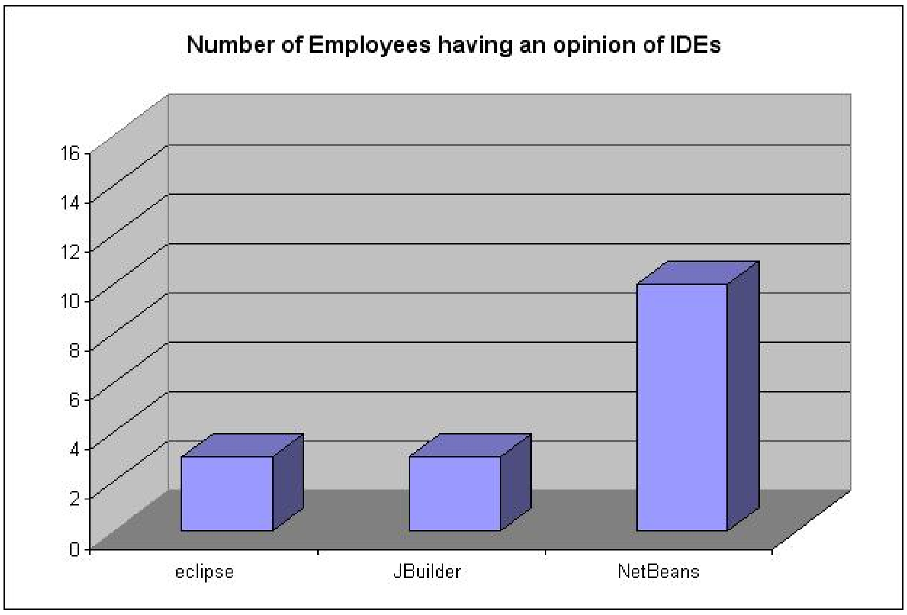


Figure 4.28: Experience handling IDE

with other products and stable. It should be noted that with such a low response rate this information cannot be seen as a general opinion of eclipse. However, what this result does indicate is that the developers at Incomit have little experience of eclipse, but those who do have positive remarks about the IDE.

4.4.4 Grading of JBuilder

> 4. If you should grade the development environment 'JBuilder', what grade should it receive?
> On a scale from 1 - 5 (where 1 is 'bad' and 5 is 'excellent')
> If you haven't tried it enough to grade it put an X as answer.
> Please give a short motivation for your answer: ( )

JBuilder had poor responses. Only three people graded it, and although the grades were not that low (see Figure 4.29), the responses insinuated that JBuilder had some integration

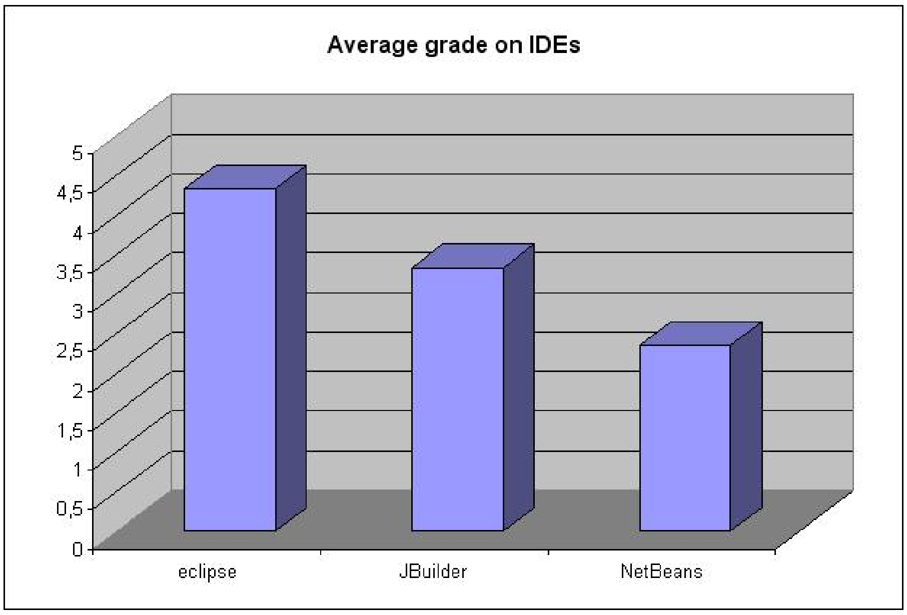


Figure 4.29: Average Grade of IDEs

issues. The developers claimed that using JBuilder restrained them to that development environment and thus it was hard to integrate it with other products. If these are old issues that have been addressed or if this is still the case has not investigated in this thesis.

4.4.5 Grading of NetBeans

5. If you should grade the development environment 'NetBeans', what grade should it receive?
On a scale from 1 - 5 (where 1 is 'bad' and 5 is 'excellent')
If you haven't tried it enough to grade it put an X as answer.
Please give a short motivation for your answer: ( )

Ten developers graded NetBeans which meant that they had used it at least enough to get a proper opinion about it. Most of the grades were low and the aspect which most people complained about was the performance of NetBeans. They claimed that

NetBeans was slow and demanded a lot of computer resources to work with. NetBeans.org performed a survey about the performance of netbeans 3.6 [25], which seem to contradict the statement about NetBeans beeing slow in performance. It should be observed that NetBeans' survey was not done by an unbiased corporation. Another downside a few people complained about was the GUI-creation ability in NetBeans. One person argued that there was a poor connection between what the GUI looked like in the design phase and what it actually looked like when running. However, there were also some positive comments about NetBeans. One of these comments concerned the flexibility of NetBeans. NetBeans uses a modular perspective[24] (not unlike the plug-in approach eclipse uses. See more in Section 4.3.2) which facilitates expanding and integration with other software.

4.4.6 Summary from Development Environment Grading

Figure 4.29 shows the average grades the different development environments received. It should be said that this diagram does not depend on how many of the employees actually gave grades. Therefore, the NetBeans result could be seen as the result best showing how the employees of Incomit feel about the IDE as compared to JBuilder and eclipse where few developers could answer. Since so few have graded JBuilder and eclipse, the result is not significant and can only give an indication of the company's acceptance towards those specific IDEs. Even if this result is just a hint about what the employees prefer it should be taken seriously and be properly evaluated.

4.4.7 Development Environment Preference

6. Which development environment would you prefer to use?
eclipse ( )
JBuilder ( )
No opinion ( )

If you have answered 'No opinion' on question number 6 then you can skip question number 7.
7. What is the main reason you prefer the development environment you marked in question number 6 over the other?
Please motivate your answer: ( )

In the final two questions the participants were asked to decide which of JBuilder and eclipse they would choose and why. NetBeans was ruled out here because of expressed interest in JBuilder and eclipse from Incomit. Even though there were only three persons who graded eclipse, five persons preferred eclipse over JBuilder. One reason for this result was the ease with which third-party software could be integrated into eclipse (see Section 4.3.2). Another reason was the fact that eclipse is open source software. One participant who had not used eclipse thought it better than JBuilder because of positive rumours. This opinion could indicate that eclipse has a good reputation, or that JBuilder has a bad reputation.

4.5 Development Environment Features Survey

A second survey was also performed and once again it was the developers at Incomit who were the participants. This survey covered features in development environments and facilitating tools for the development of Incomit specific services. The employee gave his or her opinion on how important each aspect was, where the grades were "not so important", "A bit important", "Important", "Very Important" and "I couldn't live without it".

The way this survey was distributed was different from that in the first survey. This time a web based survey was used after finding a homepage[3] which provided free survey functionality. Using the web based survey technique had several benefits: it was easier to get an overview of the results at any time in the survey collection phase. It was also easier

[3]http://www.freeonlinesurvey.com/

to create the survey and for the developers to answer the survey as they only had to surf to the web page and check a number of checkboxes (compared to the complexity of using e-mail). However, the technique had some negative sides too. It was impossible in the free version to check the origin of a submitter, and the results from the survey could only be checked within ten days from posting the survey.

The questions were created through person-to-person interviews and written conversations with Incomit developers using an Instant Messenger software. Finally, the authors personal views were used in order to produce the questions.

Below, the questions are first printed as they were stated in the survey. After that, a deeper explanation of the question is given. Then the results of the question and the conclusions that can be drawn from this answer are discussed.

4.5.1 Code Templates

> How important do you feel code templates are for your development of a SLEE service?
>
> *Code templates provide necessary steps which are mandatory in the type of service you are creating. It could for example be registration when you are developing a new plug-in.*

Code templates describe certain steps which are necessary for some types of services. For example, a service in the SLEE (see Section 2.4) needs to implement certain methods in order to achieve communication with the SLEE. Using a code template could make this easier and specify how a clean SLEE-service should look like without any or little functionality. Thus a SLEE-service code template provides the user with a quick start where he or she can start implementing the functionality without the mandatory work around it. The Java-code below demonstrates how a code template could look like, where the class `CodeTemplate` is part of the code template and the interface `MandatoryMethods` defines the methods which are mandatory.

```
public interface MandatoryMethods {
    // this method is called by some other instance
    public int start();
    public int stop();
}

public class CodeTemplate implements MandatoryMethods {
    public CodeTemplate() {
    }
    public int start() {
        initiate();
        register(this);
    }
    public int stop() {
        unregister(this);
        destroy();
    }
}
```

Over 50% of the developers thought that code templates were "Very important". Obviously people regarded them as something which they could find useful, which is why this feature was prioritised at the highest level possible. (See Figure 4.30)

4.5.2 Code Examples

How important do you feel code examples are for your development of a SLEE service?

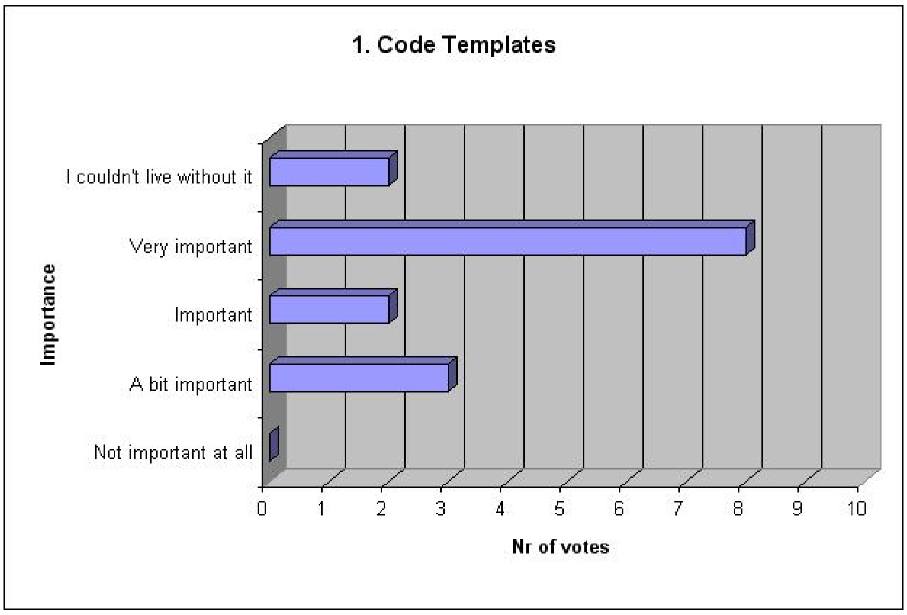


Figure 4.30: Importance of Code Templates

Code examples range from complete examples of SLEE services to parts of code which explain key areas.

Code examples can be compared to Code templates. They both simplify the creation of a service. Code templates have little or no functionality but code examples show a specific functionality or a range of functionalities. A code template provides the user with stubs whereas a code example shows a working flow of code. From a developers point of view the templates has commented rows that says "Put your code here" or similar, while the example is often a code snippet or code not needing any user manipulation at all to work. An example of Code Examples can be found in Figure 6.8.

Compared to Code Templates, people found Code Examples to be even more important (see figure 4.31). It should be noted that some people who voted "Very important" about the Code Templates now voted "Important" in the Code Examples. This means that some people think that Code Templates are more important than Code Examples, but the majority thinks otherwise.

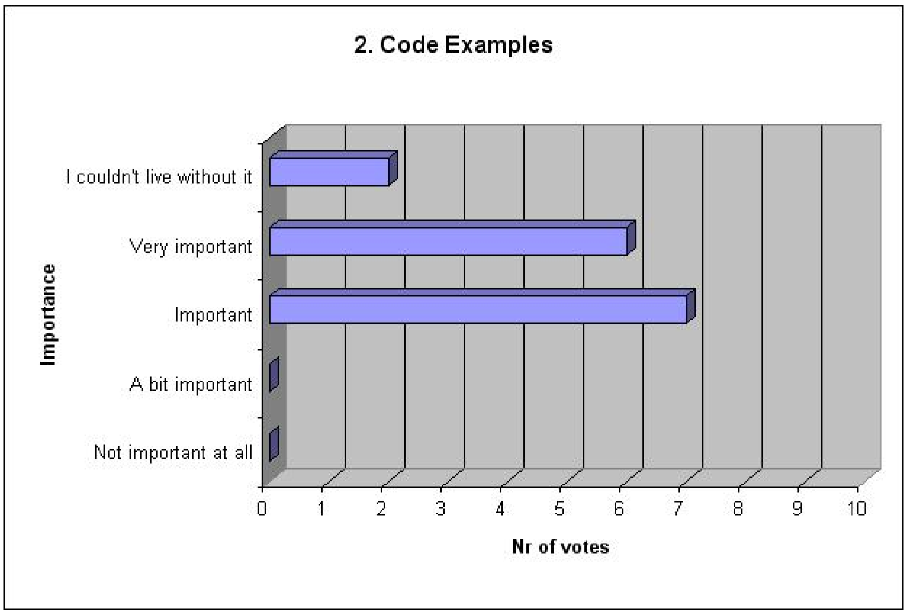


Figure 4.31: Importance of Code Examples

4.5.3 Tutorials

> How important do you feel tutorials are for your development of a SLEE service?

and

> How important do you feel tutorials are for beginners (people who are not familiar with Incomit's APIs) development of a SLEE service?
>
> *A tutorial guides you through the steps of creating a service. It is very basic but shows the whole flow of the process - from creation to completion.*

Tutorials are generally a set of lessons which the user are guided through in order to learn how to use a software product. In the case of the Extension SDK the tutorial could for example walk the user through how to create a SLEE-service. The user follows the steps provided by the tutorial and performs the various tasks he or she is ordered to do.

The developers at Incomit found tutorials to be of less importance to them as can be seen in Figure 4.32. It should be noted that none of the developers thought it to be totally

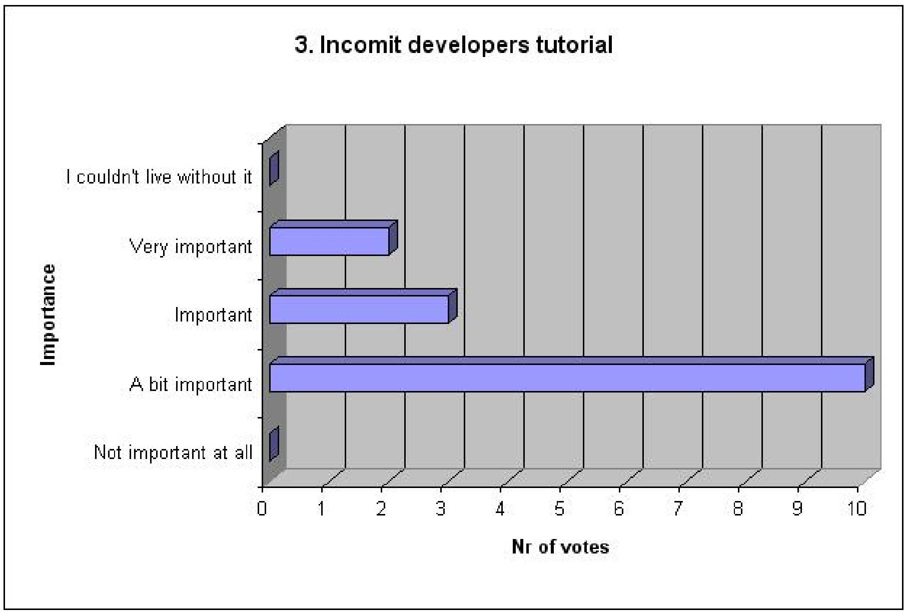


Figure 4.32: Importance of Incomit Tutorials

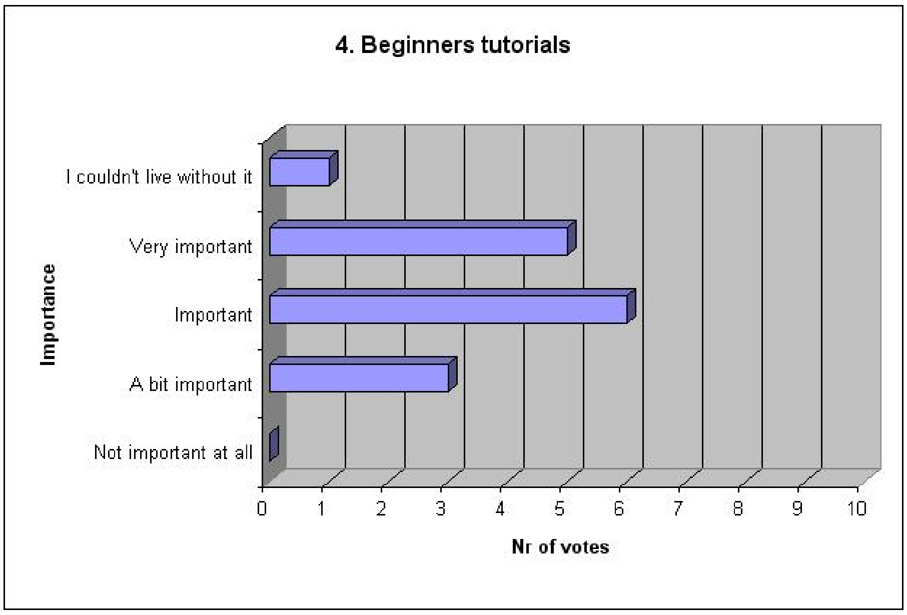


Figure 4.33: Importance of Beginner Tutorials

unimportant, but most of them rated it as next to that. However, they thought that beginners could benefit from tutorials and hence saw it as important in respect to people outside of Incomit.

4.5.4 Wizards

> How important do you feel wizards in the Integrated Development Environment (IDE) i.e. Eclipse are for the development of SLEE services?
>
> *A wizard is a number of dialogs shown for the user in order to easy step-by-step configure what kind of SLEE services you would like to create.*

Wizards are a set of dialogs shown to the user aiming towards an end-goal. The user can then change the outcome of the wizard by selecting or deselecting checkboxes provided by a graphical interface. Questions could be regarding what kind of SLEE services you would like to create. The result from the wizard in that case would be template code. Pictures how a wizard can look like can be seen in Figure 4.1 and Figure 4.14.

The result of this question had only some divergence as most developers considered wizards to be "Important" or "A bit important". From Figure 4.34 it seems like the developers do not find wizards to be that important.

4.5.5 Performance

> How important is the performance of the SDK in order to develop SLEE services?
>
> *Performance - the responsiveness of the SDK. The lesser the performance the longer it takes to process certain key-thing such as code creation, deployment etc.*

The performance is evaluated from the following criteria: how long development environment tasks take (could for example be the amount of time it takes from a keypress to

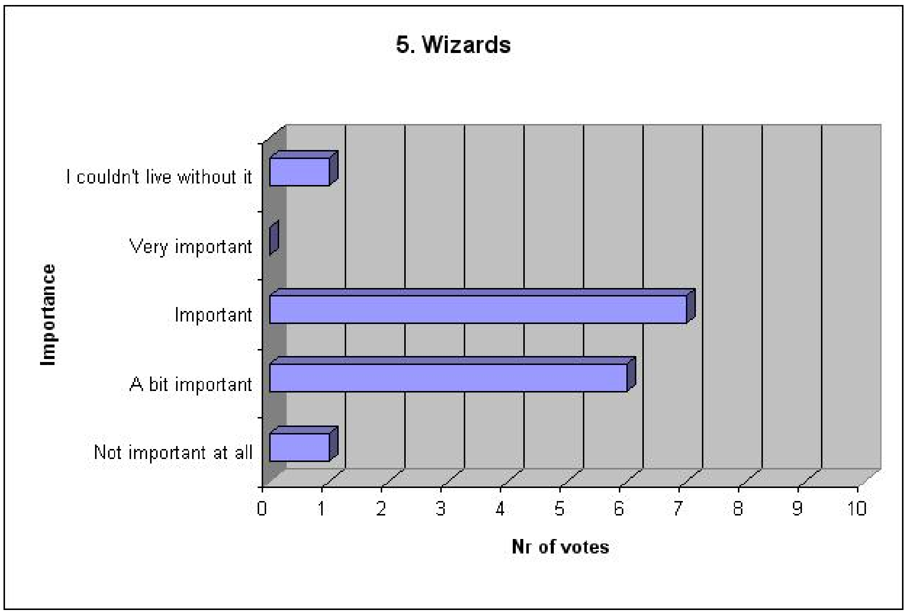


Figure 4.34: Importance of Wizards

the action actually taking place) and how long certain tasks take (for example generating code from a template). The performance depends not only on the software but on the hardware as well.

The results of the performance question was the most divergent result of the survey. It seems that the developers at Incomit have a very varying view of the importance of performance. The result could perhaps depend on how they viewed the question. If they viewed it as if these answers would prioritise the requirements of the Extension SDK they could very well have answered a lower importance level. The lower importance level would make the priority of optimising the code of the Extension SDK lower. The developers might have compared the importance of what they needed and thought that performance was something they did not crave that much, as performance often is dependent on other factors like hardware. Other developers might not have thought about it that much and just remembered situations of frustration over low performance and thus have given this query a higher importance. (See Figure 4.35)

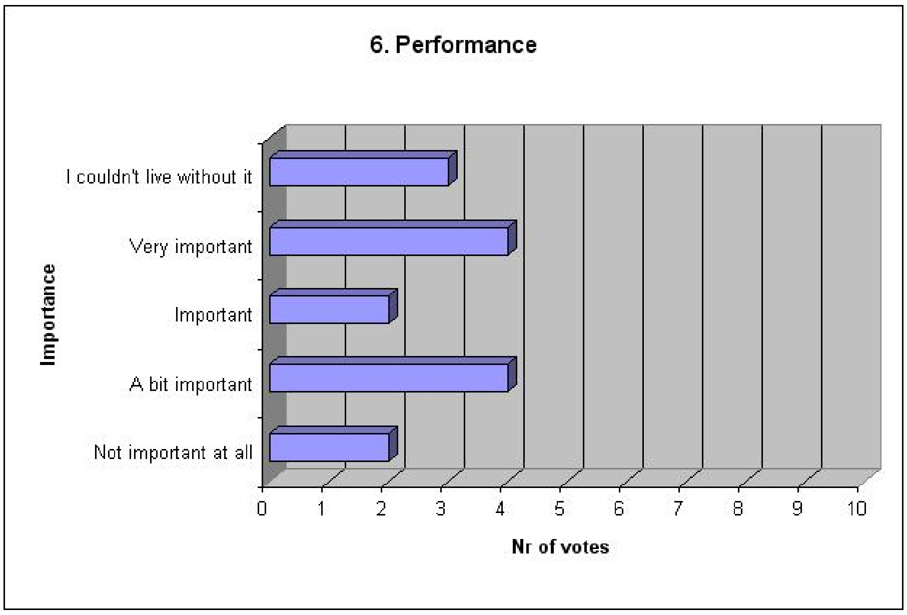


Figure 4.35: Importance of Performance

4.5.6 Method Generation

How important do you feel method generation from interface-definition-files is?

Consider having an IDL file and using this method generation feature. Then the java-files and code-stubs would be automatically created from this file.

Interface files, such as CORBA IDL files or Web Services WSDL files, can be parsed through and used to create stubs for future development. Most of the developers consider the automatic creation of code stubs from interface files to be important. It was not something they absolutely needed to do their job but at the same time it was not considered to be unimportant. (See Figure 4.36)

4.5.7 Code Completion

How important do you feel code-completion is for the development of SLEE services?

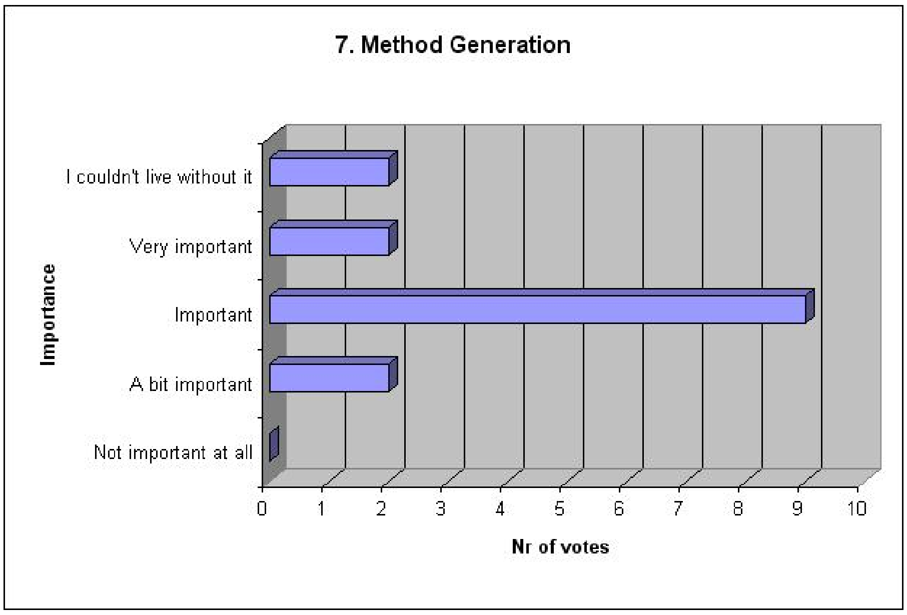


Figure 4.36: Importance of Method Generation from Interface files

The IDE suggests and fills in a possible function call, variable or constant.

Code completion was viewed as the most important feature (excluding Javadoc) investigated in the survey. The result shown in Figure 4.37 shows how much the developers believe code completion facilitates programming and should thus be considered a very important feature for developing environments. Examples of how code completion looks like can be found in Figure 4.6 and 4.19.

4.5.8 Automatic Code Packaging

How important do you feel automatic code packaging is?

Creating a deployable jar-file which needs no manual intervention (xml files are created automatically)

Incomit uses ant (see Section 3.3) to compile their source code. The application ant follows commands written in a `build.xml` file just like the compiling utility make[8] follows

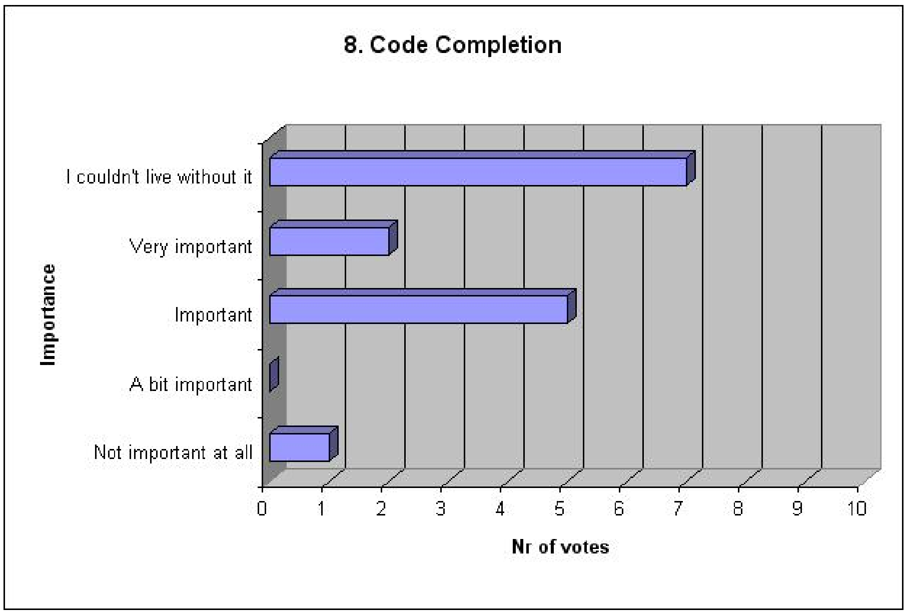


Figure 4.37: Importance of Code Completion

the commands in `makefile`. The automatic code packaging means that the jar file is created automatically and can be deployed and run instantly from creation.

Automatic code packaging had a normalised grade of importance around the "important" level. The conclusion that can be drawn from this result is that it is more important to some developers than others but the overall view is that it is fairly important. (See Figure 4.38)

4.5.9 Automatic Deployment

How important do you feel automatic deployment into the SLEE is?

The IDE automatically deploys the service into the SLEE without user intervention.

The ordinary way to integrate compiled source code with a SLEE is to transfer the jar-file to the computer running the SLEE and configure the SLEE to integrate that executable

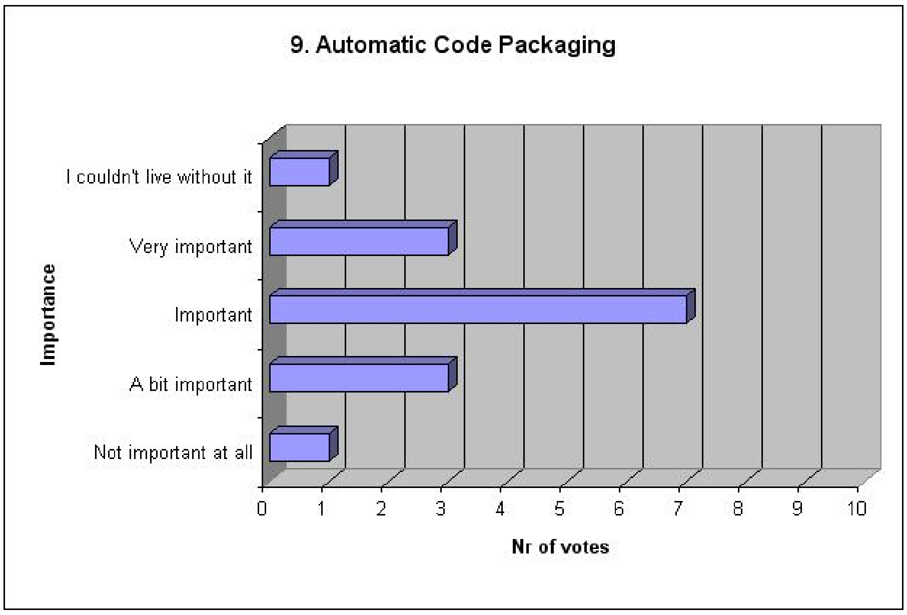


Figure 4.38: Importance of Automatic Code Packaging

code. There are ways to automatically accomplish the integration from the network. One way is to start a Web Server distributing the jar-file. The SLEE can then use its install functionality to connect to the Web Server, download the jar-file and execute it during runtime.

The developers at Incomit clearly did not think automatic deployment into the SLEE was a feature which had significant importance, as can be seen from Figure 4.39. This could mean that the developers feel that the manual intervention needed to perform the deployment is not that difficult and time consuming.

4.5.10 Errorless Compilation

> How important is it for you that the service should be compilable without errors directly from creation without any further additions by the user?

After using a creation wizard, there are two possible states the code could be in, non-

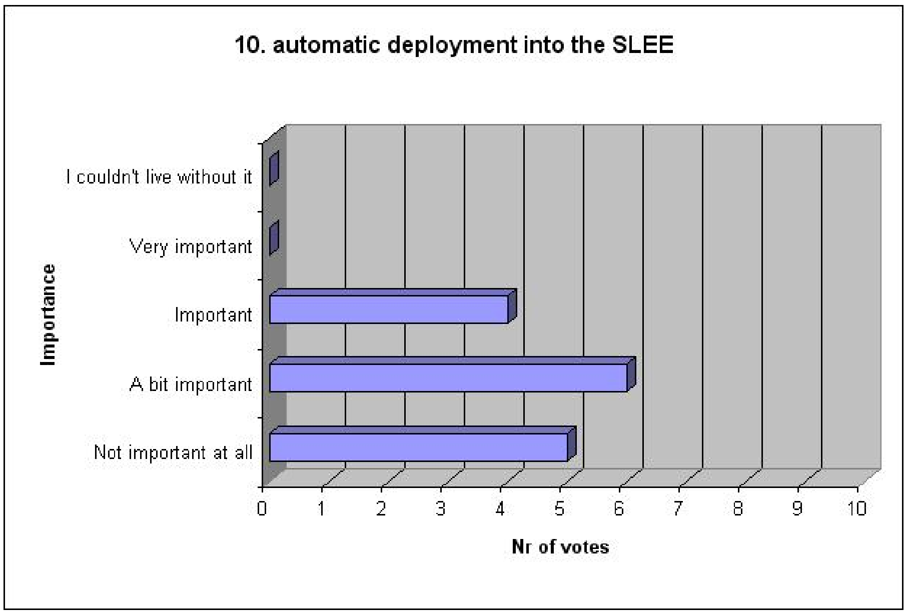


Figure 4.39: Importance of Automatic Deployment

compilable or compilable. The non-compilable code needs some sort of manual intervention from the user to get it to compile. Compilable code is compilable and runnable directly from creation. The compilable code has little (maybe a dummy-function) or no functionality.

Regarding the question about whether or not code should be error-free directly from creation there was no clear indication from the developers. The average grade lies just under "important" but considering that a third of the answering people listed it as very important or higher makes the topic not that uninteresting. (See Figure 4.40)

4.5.11 Javadoc

How important is it for you to have the documentation javadoc for the APIs?

Documentation of the javadoc API provides information about method calls, which parameters they require and what they do. Javadoc was voted as the most important of all features, and since it is more or less standard to have javadoc documentation it should

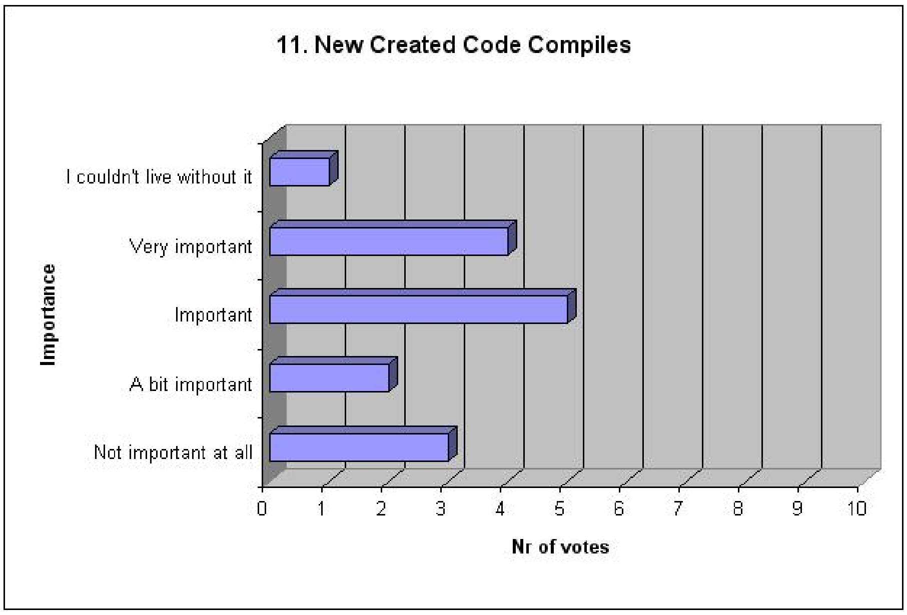


Figure 4.40: Importance of New Created Code Compilable

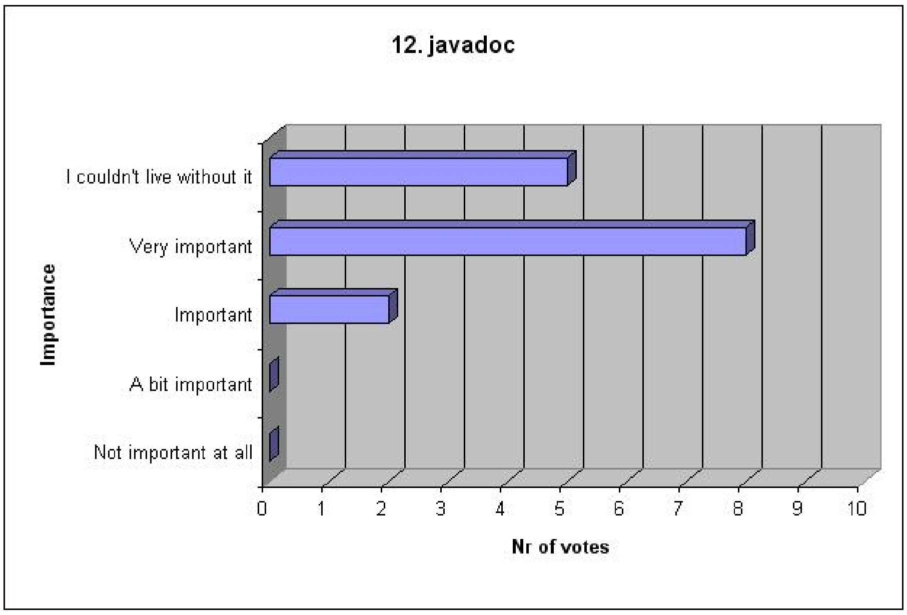


Figure 4.41: Importance of Javadoc

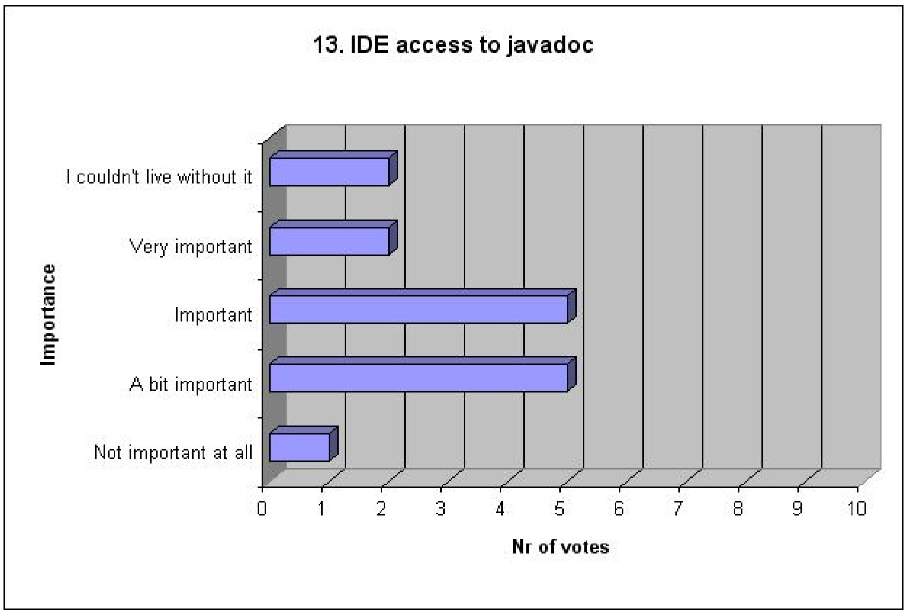


Figure 4.42: Importance of Accessing documentation from IDE

come as no surprise. Good documented code ease the work tremendously for the developer using that code. (See Figure 4.41)

4.5.12 Accessible Documentation

How important is it for you to have access to documentation (java doc and development guide) from the IDE?

This means that you could easily reach the doc. from a button or menu.

Being able to access the documentation of classes and methods fast and easily from within the development environment is a priority of most developers. A quick way to reach documentation reduces development time.

Accessing the documentation was not something that the developers at Incomit saw as something that would help them to a greater extent. A developer commented that this could be a feature more sought after when demonstrating the Extension SDK to customers.

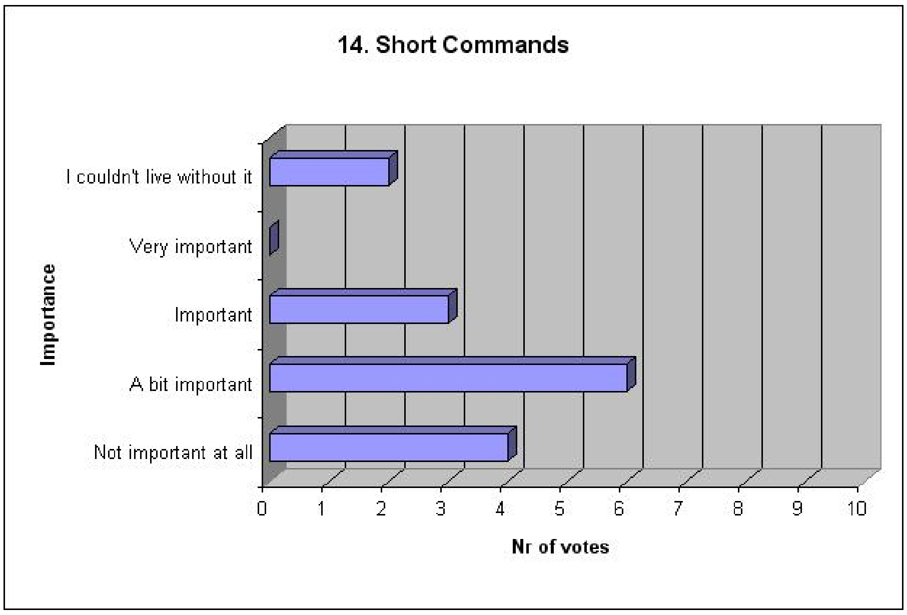


Figure 4.43: Importance of Short Commands

(See Figure 4.42)

4.5.13 Short Commands

> How important is it to have short commands in order to insert code such as trace-support?

There is a feature in some IDEs which enables the user to program his or her own scripts which converts a text combination to something else. An example of this is eclipse which can show a for-loop stub when the word "for" are written in the editor.

Using short commands was something that most of the developers saw as less important, although there were two persons who could not live without it. (See Figure 4.43)

4.6 Summary

This chapter briefly discussed the requirement specification that existed prior to this thesis, evaluated the two development environments JBuilder and eclipse, and covered two surveys targeted at the developers of Incomit. The core requirements from the old requirement was listed. In the comparison between JBuilder and eclipse, eclipse emerged as the victor even if it was not by a large margin. The last two parts showed the results of two surveys done at Incomit. In the first survey, which touched the subject of which development environment that was preferred, the developers at Incomit seemed to prefer eclipse. The second survey concerned different features in development environments as well as facilitating features in a future SDK. In that survey there were some features which stood out from the rest in terms of importance. These features were Code Completion, Javadoc, Code Examples and Code Templates.

Chapter 5

Requirement Specification

5.1 Introduction

All the requirement gathering from the previous chapter is evaluated and summarized in this chapter. The old requirement specification was the base on which this new requirement specification was built. The evaluation between JBuilder and eclipse as well as the first survey supported the decision on what development environment to support. The second survey in the previous chapter, about the development environment features, helped prioritizing the requirements and created some of the them.

5.2 Requirement Evaluation

This section will go through the results from the previous chapter and discuss how these are integrated into the SDK or discarded.

The old requirement specification suggested that the SDK should support IDEs from other vendors (as well as their own IDE which will not be discussed in this thesis) which lead to the surveys about eclipse and JBuilder. From this survey and the evaluation on the two IDEs, it was decided that eclipse would be the IDE which the SDK would support.

From the second survey we have the following requirements:

Code Templates was suggested by the old requirement specification to facilitate code creation. This was the foundation for the question about the importance of code templates in the second survey. The developers at Incomit graded code templates very high and thus it was included as a top priority requirement.

Code Examples was included by the same reasons as code templates.

Tutorials was not included as a top priority requirement because of time restraints and the fact that it did not end up that high on features the developers of Incomit regarded as important.

Wizards was regarded as being not that important by the developers of Incomit. Nonetheless, it was included in the SDK under "eclipse support". The motivation for this is that Wizards is a good way to integrate the ease of creating a Service with a Graphical User Interface.

Performance was not explicitly included into the SDK, because the scope of the SDK had to be limited.

Method generation from interface files was included but is a part of the requirement "Facilitate Development...".

Automatic Code Packaging was included since this was something that felt relevant for the SDK.

Automatic Deployment into the SLEE was not included, since the importance factor was pretty low.

New Created Code Compiles was included and renamed "Automatically Generated Code Compiles".

Javadoc was included into the requirement, since it was in the original requirement specification and the fact that it was considered very important by the developers of Incomit.

IDE access to Javadoc was not included at the time of this thesis, but would not be too hard to implement.

Short commands is entirely up to the development environment and not something that the SDK is planning to help with.

There are some additional requirements from the old requirement specification listed here:

Facilitating Development of Network Plug-ins from the old requirement specification is still there since this is the core feature of the SDK.

Documentation is a relevant requirement which came from the old requirement specification and got top priority in the requirement specification below.

Installation Wizard was not included in the top priortity requirements because of time restraints.

5.3 Requirements Listing

This section lists all the top priority requirements on which the SDK was built. The SDK is called "Extension SDK" in the requirement listing because it was the development name and it is used for creating extensions to the Application Hub.

5.3.1 Code Templates

The Extension SDK shall provide code templates for creating SLEE services and thus help with registration into the SLEE (see in Section 2.4). This means that the code templates shall implement the mandatory interfaces which are needed for SLEE registration and access.

5.3.2 Code Examples and Class Libraries

The Extension SDK shall provide code (examples and utility classes where necessary) for a number of functionalities used by SLEE services. Below is a list of functionalities that should be implemented (by examples and utility classes) into the Extension SDK.

High Availability is the ability to keep an application or service operational and usable by clients most of the time. This is done by providing redundant resources and handovers when one resource is unavailable. One of the requirements when using High Availability is that traffic should not be lost when a resource breaks down and another resource takes that place.

Load Balancing comes into play when a resource is under load and more connections would like to use that resource. Load Balancing will then distribute traffic to other resources that experiences less load. It's basically balancing load over several resources.

Alarm Generation is used when the Incomit SLEE convey errors to the user. The SLEE service can fire alarms at will that can be handled by the user in a apropriate way.

Event Generation means an event is sent to all SLEEs which are building up the Application Hub or the OSA Gateway. This is often used when updates are made which affect a number of SLEE services. An example is when a SLEE service changes a value in a database and has to inform everyone dependent on this value about the change. This can then be done using an event.

Load Measurement and Reporting targets the ability of a SLEE service to report its status. An internal SLEE service can find out its own load by asking the SLEE. The

SLEE calculates the load of the SLEE services by checking the amount of memory used and the number of active threads. External plugins have to find out their own load and convey it when asked.

Overload Protection involves the functionality to avoid SLEE services getting overloaded. This is done by restricting access to services that are overloaded. There are three stages of overload. Normal, Overloaded and Severely Overloaded. When a SLEE services becomes overloaded, services above it will try to find a different one with less load. If the SLEE services becomes severely overloaded it will throw exceptions and stop answering.

Event Channel Usage for Receiving Events concerns the functionality of receiving events. SLEE services that want to listen to events of a certain type has to register a listener in order to obtain information about that event.

Time Service Usage describes how SLEE services can use the Time service utility. The SLEE services can then perform tasks at a given time or periodic tasks which execute at given intervals.

Data Base Access involves the ability of getting SLEE services to use databases to store data in.

Trace Service Usage provides debugging functionality for SLEE services by printing information to a file.

Task Scheduling is used when tasks is sent between SLEE services. The Task scheduler handles threads and avoids deadlocks.

Supervised List Usage handles services and checks their status. The supervised list is used with High Availability and Load Balancing. It makes sure that requests are not being sent to non-responding services. If a service does not respond then it is put into a zombie list where no external requests will be sent to it. This zombie list is checked with discrete intervals. If the service in the zombie list suddenly starts to respond it will be put back.

Charging Data Record (CDR) usage is responsible for saving information so the appropriate billing and charging measures can be taken.

Configuration Management is the ability to manage the SLEE service from external sources by setting parameters. An example of an external source is the SLEE manager.

5.3.3 Facilitate Development of Network Plug-ins

The Extension SDK shall facilitate the development of the Network Plug-ins by providing code examples which describes the mandatory registrations which need to be done in order to get the plug-in integrated with the SLEE. It should also provide the following code templates:

Generic plug-in. Allows creation of a generic network plug-in which conforms to the SLEE.

Call Control plug-in. Allows creation of a call control plug-in, which enables setting up, monitoring, managing and terminating calls.

Messaging plug-in. Allows creation of a messaging plug-in, which enables the sending and receiving of messages (for instance SMS).

User Location plug-in. Allows creation of user location plug-in, which enables the retrieval of the location a mobile or fixed terminal.

User Status plug-in. Allows creation of user status plug-in, which enables the retrieval of the status of a terminal.

Charging plug-in. Allows creation of charging plug-in, which enables content based charging and billing functionality.

Subscriber Profile plug-in. Allows creation of subscriber profile plug-in, which enables the retrieval and storing of a subscriber's information (such as name, address and payment method).

5.3.4 Facilitate Development of Net Simulator Support

The Extension SDK shall facilitate the development of the Network Simulator support by providing code examples and necessary utility classes. The SDK shall also provide templates for registration with the net simulator and support for the Network Simulator management.

5.3.5 Standalone Extension SDK

The Extension SDK shall be possible to use standalone. All dependencies the SDK has shall be included in the Extension SDK package. This means that the Extension SDK shall be possible to install and use on a clean machine without having to download or install any additional tools or sources.

5.3.6 Separation of Logic and GUI

The Extension SDK shall be designed so that the logic is separated from the development environment. This means that it should be possible to run the Extension SDK without an development environment and using the basic Extension SDK logic via scripts instead. This approach simplifies the conversion to another development environment, should that be necessary in the future.

5.3.7 Operating Systems

The Extension SDK shall support Windows XP.

5.3.8 Documentation

The Extension SDK shall provide the following documentation:

- Incomit Service Extension SDK Developer Guides.
- Javadoc
- Development Environment User's Guide

The Incomit Service Extension SDK Developer Guides explaines how the Extension SDK works with its different helping utilities. The Developer Guides describes how the scripts work and gives examples on how to use them.

The Javadoc provides documentation over methods and classes from different parts:

- Class libraries
- SLEE
- Plug-in interface

The Development Environment User's Guide is a manual over how to use the development environment with the Extension SDK. It describes how to perform the different actions provided by the Extension SDK functionality.

5.3.9 Eclipse Support

The Extension SDK shall be possible to use in combination with the eclipse IDE.

5.3.10 Automatic Code Packaging

The Extension SDK shall automaticaly create the `build.xml` file which will have the functionality of creating a deployable jar-file.

5.3.11 Automatically Generated Code Compiles

The code which is generated from the Extension SDK shall be able to compile directly after creation.

Chapter 6

Design of the SDK

6.1 Introduction

This chapter describes the design of the SDK derived from the requirements summarized in Chapter 5. First, the aspect of modularity is discussed since requirements such as Separation of logic and GUI (see Section 5.3.6) demands a design which is thoroughly thought through. Second, the classes which form the SDK's logic and the graphical user interface are briefly described. Third, the code templates[1] are reviewed. These helping code pieces are used not only to create a uniform code look of each service that is created using the SDK, but also to create the framework which the developer will work in. Fourth, code examples are briefly discussed with an example to clarify what they are. The central role of the SDK is to facilitate the creation of Network Plug-ins, using code templates, code examples, javadoc and more. What a Network Plug-in consists of is discussed there after. The template files which are used as the foundation for creating code with the SDK has certain special tokens in them which are discussed in the fifth part. Finally, the integration into the development environment eclipse is explained.

The main idea of the SDK is that it should facilitate the design of Incomit related

[1]see Section 4.5 under code templates for more details

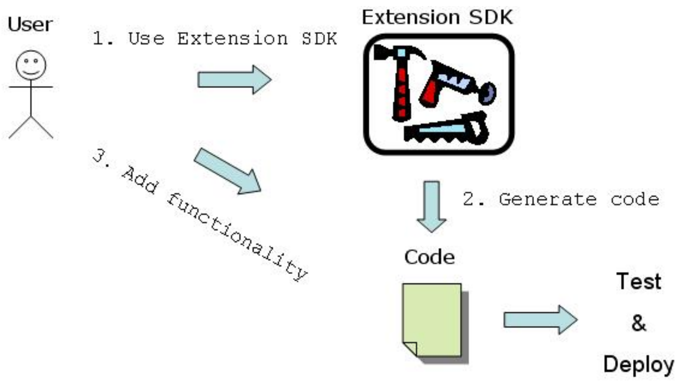


Figure 6.1: Overview

software development. This is done mainly through providing code templates and code examples which the developers can use. If the users intends to create a SLEE service they input the name and properties of the SLEE service they would like to have and the Extension SDK generates the code according to the users' wishes.

6.2 Modularity

One of the requirements was that the Extension SDK should be developed in such a way that the GUI part of the Extension SDK was separated from the Extension SDK logic. This makes the whole Extension SDK more modular in its architecture. It should also be easy to change the GUI part from, for example, eclipse to JBuilder without rewriting large amounts of code. The Extension SDK was designed so that scripts could be used to create code. Using scripts means that the whole GUI part is bypassed. (See Figure 6.2)

What this means is that a Extension SDK logic part would have to be created which consists of all logic needed in the SDK. Features like automatic code deployment and generating code from code templates (code which compiles on first try) are implemented in this logic. On top of this logic, different layers could be made, for instance console based access or eclipse GUI.

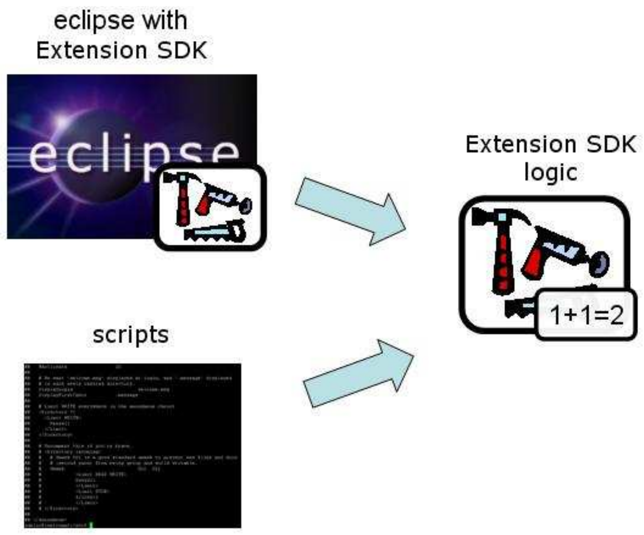


Figure 6.2: Separation of Logic and GUI

6.3 Parsing Specific Classes

This section briefly describes the classes which add to the parsing functionality of the Extension SDK. A class diagram of the "parsing" classes can be found in Figure 6.3.

JarResources. This class contains functionality for extracting files from .zip and .jar files.

HandleProperties. Retrieving values from a property file can be done through this class.

Parser. This class has the core functionality. The parsing through the template files is done in this class.

InterfaceToStubs. This class creates stub-java code from Java-interface files.

HandleTypes. Used to get values from a specific XML file.

HandleDir. Handles the location of the directories of the Extension SDK.

Dir. Small class that retrieves the location of the Extension SDK from a property file.

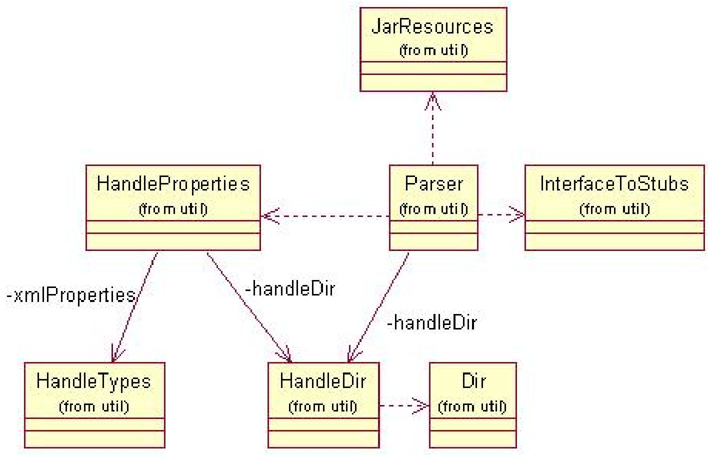


Figure 6.3: Parser Class Diagram

6.4 Eclipse Specific Classes

In this section there is a brief description of the classes which are eclipse related which provide the graphical user interface of the Extension SDK. A class diagram of the eclipse classes can be found in Figure 6.4.

ExtSDKWizard. This is a base class for an Extension SDK wizard. If a new wizard should be added to the Extension SDK, a new class should be created which inherits from this class.

NSPluginWizard. The Network Simulator wizard.

PluginWizard. The class which contains the functionality of the wizard for network plug-ins.

PropertyWizardPage. This class is used by wizards and dynamically creates a GUI from a property file.

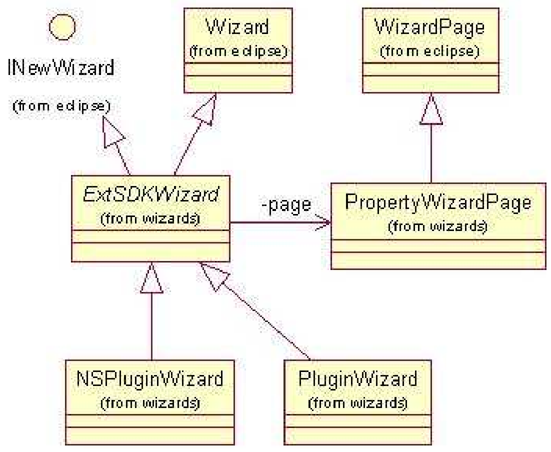


Figure 6.4: Eclipse Class Diagram

6.5 Code Templates

There are numerous advantages of the template [2] approach. First, less sources make changes easier. Should it be necessary to change generated code in future releases, changes only has to be done in the template code. Second, the code looks the same and is more uniform. Code that has the same structure and style is easier to understand over time since it is easier to follow code one is familiarized with. Third, problems with performance and bugs in the template code can be taken care of once and for all and future code will be free from those problems. The downside of using code templates is that errors or performance issues that exist within them will propagate to code generated from the templates. Hopefully, the fact that there is only one place to fix the error will in most cases be an advantage instead of an disadvantage. The code generated from code templates has none or little functionality other than registration into the SLEE (see Section 2.4), it is then up to the user to add the functionality that makes the SLEE service unique. (See Figure 6.1)

The Extension SDK logic's job is to generate the code that the developer wants from

[2]see Section 4.5 under code templates for more details

```
hidden.OPERATION_STUBS_FROM_INTERFACE_DIR=
        c:/incomit/ext_sdk/idl_generated/lib/idl_generated.jar
hidden.OPERATION_STUBS_FROM_INTERFACE_FILE=
        ${hidden.XML_TYPE_INTERFACE}Operations.java
extsdk.plugin.combobox.XML_TYPE=Messaging

extsdk.plugin.MOD_PATH=com.mycompany.myplugin
extsdk.plugin.JAVADOC=${extsdk.plugin.NAME}_javadoc.zip
extsdk.plugin.NAME=Name
extsdk.plugin.PROJECT_NAME=${extsdk.plugin.NAME}
extsdk.plugin.JAR_FILENAME=${extsdk.plugin.NAME}.jar
```

Figure 6.5: Property File Example

the templates available. In order to make the Extension SDK as generic and modular as possible it works in the following way: The code templates have tokens in them which are replaced by actual text strings during the generation process. An example of a token could be `%[TITLE]` which, whenever discovered, is changed to the title entered by the user. Information about special tokens are found in Section 6.8. The Extension SDK parses a file which has values that correspond to each token, at the beginning of the code generation. This file is unique for each type of service available for creation in the Extension SDK. An example of such a file is the `extsdk_plugin.properties` which can be seen in Figure 6.5.

The property file above requires some explanation in order to be understood. It is based on standard property files[20] with the addition of ant-like property assignment evaluation. By property assignment evaluation is meant that in ant it is possible to assign a value to a property and then retreive this value using `${ }` around the property name. The property segment below could clarify how this works:

```
property=hello
another=${property}
```

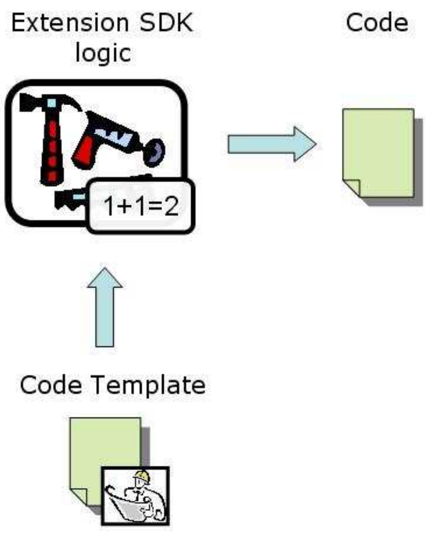


Figure 6.6: Code Templates

The above property-example would after parsing evaluate `property`'s value as `hello` and `another`'s value also as `hello`.

The `extsdk_plugin.properties` file above decides which text strings the parser should search for and also decides the default value the text token should be replaced with. Text strings such as `%[TITLE]`, `%[MYRESOURCE]`, `%[RESOURCE_IF]` and so on are searched for by the parser. The parser searches through all related files. For example, if call control plug-in stubs are generated, the call control plug-in template files are parsed through. If the parser finds "`%[TITLE]`" in the template file it will change it to "`name`". The default value "`name`" can be changed through editing `extsdk_plugin.properties`. Another example would be that "`%[MYRESOURCE]`" is changed in the template file to "`name_res`" in the generated file. A second way to change the title would be to use eclipse in collaboration with the Extension SDK. A wizard will then prompt the properties and their default values for the user to change. More information on the eclipse plug-in can be found in Section 6.9.

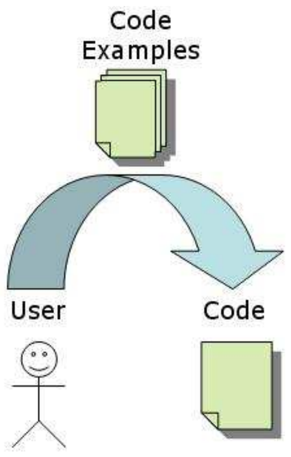


Figure 6.7: Code Examples

6.6 Code Examples

Code examples give examples of how a certain task could be performed. The language the code examples are written in is Java, since the whole Extension SDK and everything around it is written in Java. The example will help them to get the insight or the knowledge they need in order to overcome the problem with implementing their part of the source code. Code examples are integrated into the code by the user and not externally by the Extension SDK.

Figure 6.8 shows an example of an code example. That particular code example shows the user how to use the trace service provided in the Incomit SLEE. Trace logs are put in a file on a machine that runs the SLEE and are used for debugging.

6.7 Network Plug-ins

A network plug-in consists of at least two code classes which provide the interface to the Incomit Application Hub. What this means is that these classes connect the protocol specific code to the functionality of the Application Hub. The classes can of course

```
.
.
.
TraceLogService myTraceService = myServiceContext.getTraceService();
.
int a = 42;
int b = getRandomNumber();
a = a + b;
if(myTraceService.isTraceActive()){
    myTraceService.logTrace("MyClass",
                            "methodAPlusX",
                            TraceLogService.USERDEF_1,
                            ("Result of stupid calculation:" + a));
}
.
.
.
```

Figure 6.8: Trace Service Code Example

have any name but for simplicity reasons we name them `SleeService_impl.java` and `SleeServiceOAM_impl.java` (even a plug-in is a SLEE Service). OAM stands for Operation, Administration and Maintenance. These two files should only contain methods which are included from interfaces. Protocol specific methods should preferably be separated to another file.

When a network plug-in is created using the SDKs code auto generation feature the directories below are created.

- build - Contains the ant build file build.xml.
- doc - This is where the javadoc is placed.
- generated - Idl2java places the java-files generated from the idl-files here.
- idl - Contains the idl-files which specifies the service's corba interfaces.
- lib - This is the directory where the service jar-file is placed after compilation.

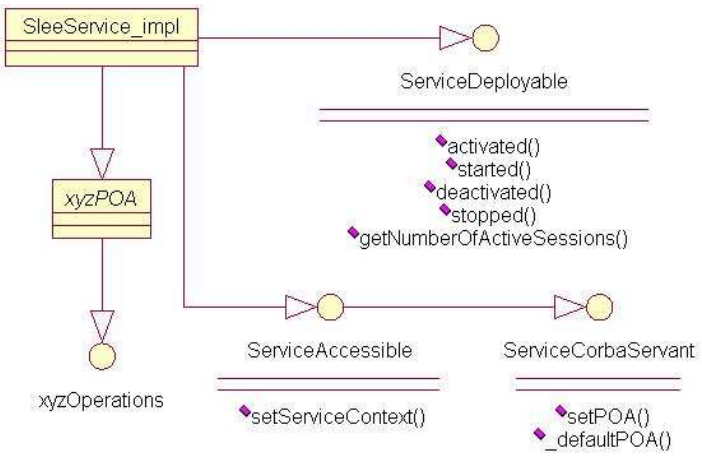


Figure 6.9: SleeService Class Diagram

- src - This is the source directory where the source code is located.

6.7.1 SleeService

The classes which the class `SleeService_impl` inherits from are `ServiceAccessible`, `ServiceDeployable` and a third class that for now will be called `xyzPOA`. `ServiceAccessible` provides (as described in Chapter 2.4) the functionality of accessing specified methods from external sources. The `ServiceDeployable` class enables the SLEE Service to be deployable into the Incomit SLEE. A deployed service can use all the SLEE utilities such as trace-service for debugging purposes (see Figure 6.8 for an code example), database-access and more. The third class `xyzPOA` is a class generated from a CORBA Interface Definition Language (IDL) file. This IDL file specifies the plug-in specific interface and therefore consists of method declarations and method variable definitions. This interface, which is defined in the program language independent IDL file, is converted to Java specific classes through an idl-to-java mapping. More information about Java specific CORBA usage is available at [21].

The mapping from IDL to Java creates a number of files of which most have little interest to a developer using CORBA in Java. If the IDL-file was named `xyz.idl` there is a file created called `xyzPOA.java` which is of greater importance. If there is a need to distribute the methods declared in `xyz.idl` then a class (in this case that class is `SleeService_impl`) should extend (inherit) the `xyzPOA`-class, and further implement all methods specified through the `xyzPOA`-class.

A requirement stated in Section 5.3 is the facilitation development of network plug-ins. A network plug-in has the structure described above and is depicted in Figure 6.9. A central part of the network plug-in is the registration of the plug-in to the Incomit Application Hub. This part is already implemented in the `SleeService_impl.java` after creation and needs no interaction from the developer in order to function. A problem arises when specific plug-ins are designed, such as a Call Control (see Section 5.3.3) plug-in.

The first attempt to solve the problem with how to deal with specific plug-ins was discarded. The idea was to create a template for each type of Network plug-in. The positive side of that approach was that certain code which was repeated throughout a type of network plug-in could be included in the template and developers could ignore those parts. Due to the fact that creating such a template for every single one of the types would be enormously time consuming this approach was discarded.

After that a second design approach was made. This approach was more generic and based on the fact that the specific network plug-in `SleeService_impl.java` used an IDL-interface which tied it to that specific network plug-in. For example, the Call Control plug-ins all extend `CallControlResourcePOA` or at least a POA-class which inherits from that particular class. If one could somehow access the IDL-interface and print the stubs of each method into the `SleeService_impl.java` this would help the user to see which methods he or she had to implement.

Since one requirement from the surveys was that the code should be compilable from creation, there is more functionality than simply changing the interface to code-stubs. All return values have to be set so that the compiler does not complain about missing return

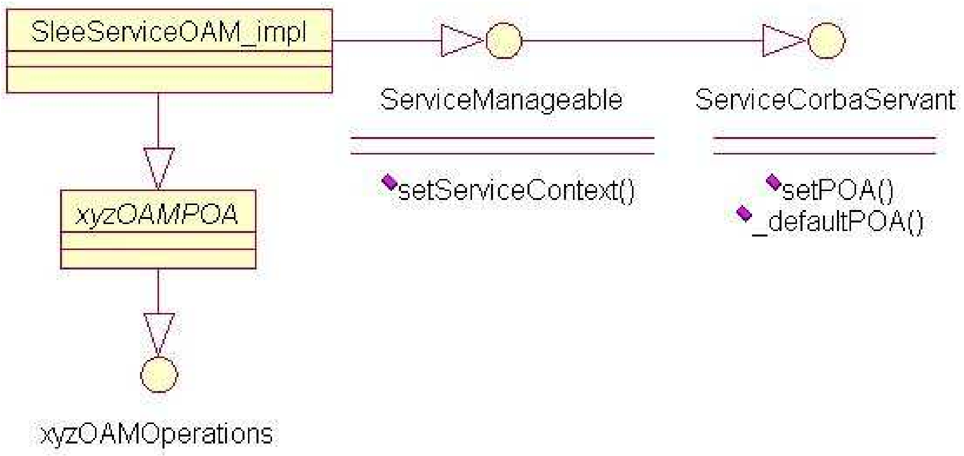


Figure 6.10: SleeServiceOAM Class Diagram

values. Inheritance is also an issue, since the interfaces may inherit from other classes, so a traversal through a number of classes had to be made, instead of just parsing one class.

Since parsing through the IDL and generating Java Code from that parsing would evolve into too many hours work, a closer inspection of the .java-files that the idl-to-java mapping created was made. This inspection resulted in the discovery of `xyzOperations.java` which was an interface file that the `xyzPOA.java` file inherited from. The `xyzOperations.java` included all the methods declared in the `xyz.idl`. The solution of creating stubs from the interface was done through parsing the `xyzOperations.java`, locating the methods, changing the declaration to definition and printing them into `SleeService_impl.java`.

6.7.2 SleeServiceOAM

The `SleeServiceOAM_impl` file has similar architectural dependencies as `SleeService_impl`, but this file is for management purposes only. The `SleeServiceOAM_impl` file is used to retrieve and set values through a managing interface. It could for example be to retrieve the value of when the SLEE service becomes overloaded or to set that value. `SleeServiceOAM_impl` inherits from the interface `ServiceManageable` which makes the SleeService manageable. `SleeServiceOAM` also inherits from another class, a `SleeServiceOAMPOA` (the name is depending on the IDL-file) which is generated from an idl-to-java mapping. The source is `SleeServiceOAM.idl` where the developer self declares all methods that the SLEEmanager (see Section 2.4) can manage externally. This is thus something that it is hard to facilitate, since the management methods depends very much on what kind of service that is being created.

A number of approaches were considered when trying to facilitate the OAM for the user. One of these was to let the user design his own OAM IDL and use IDL-to-Java to create the `Operations.java` file, and then use `Operations.java` to generate stubs which are entered into the `SleeServiceOAM.java`. This would be a nice approach since it at least facilitates the stubs for the developer (like in the `SleeService.java`-case), but at the same time it demands that the user should have designed his own IDL before running the Extension SDK. Another approach that was considered was to have a very simple IDL file as a default. If the developer did not have an IDL, the Extension SDK would use the default IDL. This default IDL would provide a generic get and a generic set method. The advantage of this approach is that the developer can use the Extension SDK from start and get and set a number of values. The downside would be that the developer would have to keep a record of the names of the values he or she sets. For example `set("overloadValue","70")` would set the overloadValue to 70 and `get("overloadValue")` would thus return `"70"`. The developer has to know about the `"overloadValue"` since he or she can only see the `set` and `get` method in the manager interface compared to a normal `setOverLoadValue(70)` which would show `setOverLoadValue`. The final approach, which is how the Extension SDK works, is a thin OAM IDL which gives the developer an example of how to use the IDL and extend it on his or her own. The OAM IDL provides methods for setting and getting the overload percentages and severe overload percentages of the SLEE service.

6.8 Special Tokens

Some of the tokens found in the code template file have unique functionality. Normal tokens are just found and translated to their respective value as described in Section 6.5. Below follows a description of these special tokens and how they are used.

The token `XML_TYPE_IMPORT` outputs a java import line, such as `"import " + XML_TYPE_IMPORT` (where `XML_TYPE_IMPORT` is decremented in every iteration). The string `"XML_TYPE_IMPORT=java.util.vector;"` could be an example on how `XML_TYPE_IMPORT` is set in the property file. It should be noted that tokens which start with `XML_TYPE_` are special and not listed in the property file but instead listed in `types.xml` (more on this later in this section).

The string above in the property file would change `XML_TYPE_IMPORT` to `"import java.util.vector;"`. A question that could be asked here is why the import-token was implemented like this. Why not just use `"import XML_TYPE_IMPORT"` directly in the template file and then the `XML_TYPE_IMPORT` would not be a special case. If it would be implemented without the special case what would happen if there had to be several imports in the java-file? One intuitive solution would be to use a range of numbers of `XML_TYPE_IMPORT`, for instance `"import XML_TYPE_IMPORT_1"` and `"import XML_TYPE_IMPORT_2"`. Using such a solution would mean that the number of imports always have to be the same but, in reality the number could vary depending on what type (for example messaging, call control) of service that is going to be created. Another solution would be to use `"import XML_TYPE_IMPORT"` and if there is more than one import, then just extend the token `XML_TYPE_IMPORT` with another import line. However, what would happen if there is no import needed? Then there would still be a `"import "` which is not supposed to be there.

If a property in the property file has "hidden." before its name then that property will not be included when the Graphical User Interface (GUI) is created (see Section 6.9). Two of the hidden properties are `OPERATION_STUBS_FROM_INTERFACE_FILE` and

`OPERATION_STUBS_FROM_INTERFACE_DIR`. Both these tokens are used when the token `OPERATION_STUBS_FROM_INTERFACE` is found in a template file. This token is not included in the property file since it is a special token that points to a location in the template file where the generated stubs from the interfaces are input. There are some points where the Extension SDK could be seen as a bit static. The fact that there is currently no way of chosing another name for the token and the fact that there is no way to use more than one "stubs from interfaces" makes the modularity of the Extension SDK less dynamic. This static behaviour was motivated because of time restraints and because there were so few cases where more than one interface should be translated into stubs.

When the token `OPERATION_STUBS_FROM_INTERFACE` is found, the java file which contains the interface is located through the `OPERATION_STUBS_FROM_INTERFACE_FILE` and `OPERATION_STUBS_FROM_INTERFACE_DIR` properties. Then this file is parsed and the methods are output where the `OPERATION_STUBS_FROM_INTERFACE` was found. In order to clarify how this works an example is provided below:

```
hidden.OPERATION_STUBS_FROM_INTERFACE_DIR=
       c:/incomit/ext_sdk/idl_generated/lib/idl_generated.jar
hidden.OPERATION_STUBS_FROM_INTERFACE_FILE=
       ${hidden.XML_TYPE_INTERFACE}Operations.java
```

When the token `OPERATION_STUBS_FROM_INTERFACE_FILE` is found, the Extension SDK will look for the `${hidden.XML_TYPE_INTERFACE}Operations.java` file in the location specificed by `OPERATION_STUBS_FROM_INTERFACE_DIR`. `OPERATION_STUBS_FROM_INTERFACE_DIR` can be either a .jar-file or a directory.

The `${hidden.XML_TYPE_INTERFACE}` is generated from `types.xml`.

The XML-file could look as below (this is an example).

```
<?xml version='1.0'?>
<!-- THIS IS THE PLUGIN SIMULATOR TYPES -->
```

```
<TYPES>
    <TYPE value="Messaging">
        <!-- name of interface to use -->
        <INTERFACE value="MessagingResourceExt"/>
        <!-- path where interface lies -->
        <IMPORT value="com.incomit.resources.messaging"/>
        <TYPE value="MESSAGING_TYPE"/>
    </TYPE>
    <TYPE value="Call Control">
        <!-- name of interface to use -->
        <INTERFACE value="CallControlResource"/>
        <!-- path where interface lies -->
        <IMPORT value="com.incomit.resources.callcontrol"/>
        <TYPE value="CALL_CONTROL_TYPE"/>
    </TYPE>
</TYPES>
```

`XML_TYPES_INTERFACES` is directly generated from the XML-files depending on what kind of TYPE that is chosen. If "Messaging" is chosen than `XML_TYPES_INTERFACES` will be set to "`MessagingResourceExt`". `XML_TYPES_IMPORT` will be mapped to "com.incomit.resources.messaging" and `XML_TYPES_TYPE` will be mapped to "`MESSAGING_TYPE`". It is possible to add other tags such as XYZ in the XML-file and then use the `XML_TYPES_XYZ`-tag in the template-file.

The `MOD_PATH` tag is used to specify the package path of the service being created. An example in the template file could be "`package $[MOD_PATH];`" which could be translated to "`package com.acme.services.myspecificservice`". `MOD_PATH` is not only used to insert the package path but also used to create the appropriate directories in order to create the directory structure for the service. For example, the `MOD_PATH` above would generate the directory "com" and under that one "acme" would be created and so on. The final directory structure would in this example look as follows:

```
com/acme/services/myspecificservice
```

It would be under the leaf directory (myspecificservice in this case) that the generated files are created.

The suffixes `_START` and `_END` of tokens create the `MOD_PATH` structure in an IDL-fashion. These tags should only exist inside an IDL template file. It creates indentation as well as module structure. An IDL file with the associated module path "com.acme.myproject" could looks as follows:

```
module com {
  module acme {
    module myproject {


    };
  };
};
```

Consider that `MOD_PATH` is set to "com.acme.myproject". Then the follwing template file would be identical after parsing as the above IDL-file.

```
$[MOD_PATH_START]


$[MOD_PATH_END]
```

The IDL file above provides no functionality but it shows how an IDL-structure look like. The `_SLASH` tag suffix converts all dots to slashes when used. If `MOD_PATH` is "com.acme.myproject" then `MOD_PATH_SLASH` would be "com/acme/myproject". The `_LC` tag suffix converts the tag content to lowercase. If `XML_TYPE_TAG` is "MSG" then `XML_TYPE_TAG_LC` is "msg".

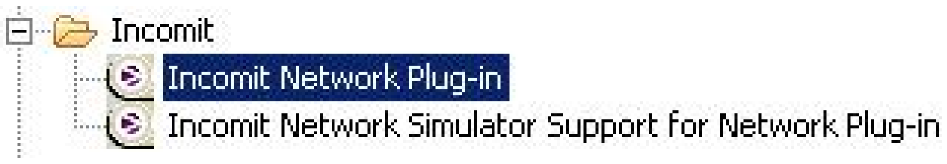


Figure 6.11: Eclipse Project Folder

6.9 Graphical User Interface

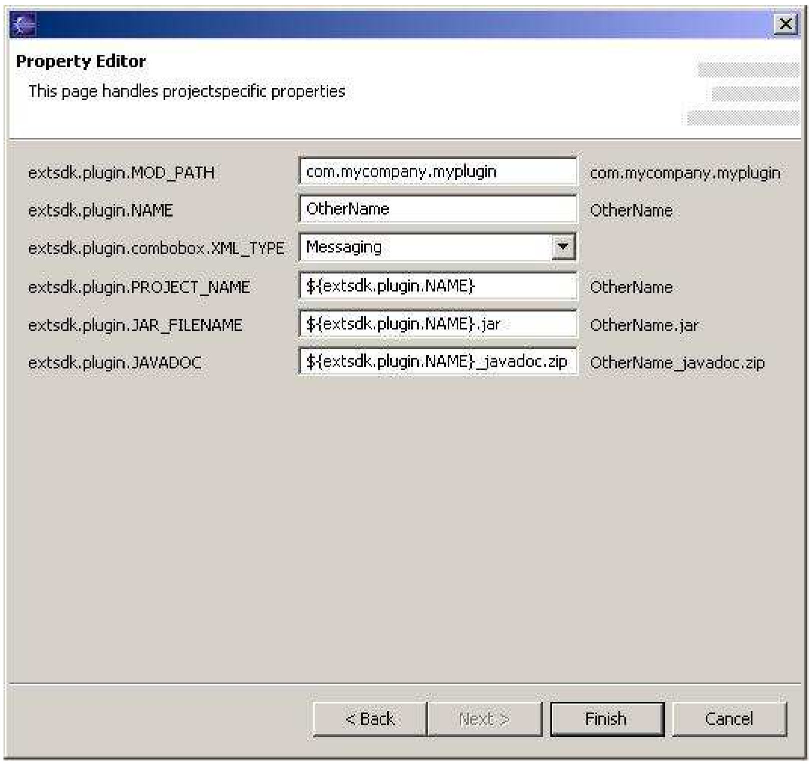
It is stated in the requirement specification (see Section 5.3) that the Extension SDK should have support for eclipse. The Extension SDK provides an eclipse plug-in which collaborates with the Extension SDK's logic. This plug-in provides only minor additional functionality in comparison to the scripted Extension SDK.

After installation, eclipse should have a new Incomit folder in the Graphical User Interface under the project creator. The Extension SDK facilitates creation of various SLEE services which are listed under this folder. As of now, there are two services listed here, the Incomit Network Plug-in and the Network Simulator Plug-in (see Figure 6.11)

When a service is chosen, the developer enters the name of the project he or she wants to create and then clicks next. The second page of the creation wizard is an automatically generated GUI which is generated from a property file (see Figure 6.5) that can be different for each service (see Figure 6.12).

The Property Editor is divided into three columns. The first column is the name of the property, the second is the value of the property and the third is the resolved value of the property. If text is written in the second column, then all dependent rows in the third column are updated. If the property name has "`combobox.XML_`" somewhere in its name then instead of a textbox a combobox is shown. The options from this combobox is read from the `types.xml` file.

After the Property Editor, which finishes when the button "Finished" is pressed, the project is created. Which files that are included depend on what type of service the

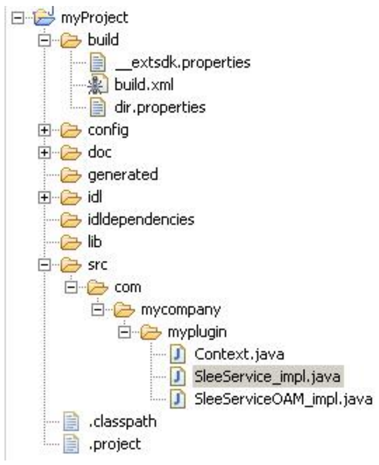


Property Editor

This page handles projectspecific properties

extsdk.plugin.MOD_PATH	com.mycompany.myplugin	com.mycompany.myplugin
extsdk.plugin.NAME	OtherName	OtherName
extsdk.plugin.combobox.XML_TYPE	Messaging	
extsdk.plugin.PROJECT_NAME	${extsdk.plugin.NAME}	OtherName
extsdk.plugin.JAR_FILENAME	${extsdk.plugin.NAME}.jar	OtherName.jar
extsdk.plugin.JAVADOC	${extsdk.plugin.NAME}_javadoc.zip	OtherName_javadoc.zip

Figure 6.12: Property Editor



```
myProject
  build
    __extsdk.properties
    build.xml
    dir.properties
  config
  doc
  generated
  idl
  idldependencies
  lib
  src
    com
      mycompany
        myplugin
          Context.java
          SleeService_impl.java
          SleeServiceOAM_impl.java
  .classpath
  .project
```

Figure 6.13: Project Folders

developer chose in the Property Editors combobox. Depending on that type the according source directory is set and all template files under that source directory is parsed and sent to the project directory (see Figure 6.13).

After the project is created, with all the necessary dependencies made by the plug-in, compiling it without errors should be possible. The plug-in does not use the basic built-in eclipse compiler but the external ant functionality that eclipse has support for. Compiling the service is done by running the `build.xml`-file in the build directory. Output from the compilation is shown in eclipse console window.

The GUI part is quite thin, but this is as intended as it makes porting the Extension SDK to other IDEs easier. Had the GUI part been heavier that functionality would have been lost in the transition to another IDE.

Chapter 7

Results and Evaluation

All requirements mentioned in Section 5.3 save two has been implemented in The Extension SDK. The first is the "Code Examples and Class Libraries"-requirement. Although that requirement looks massive on paper, it would be sufficient with a code sample and explaining documentation for each segment. There is currently only one code example in the Extension SDK. Therefore, the requirement is not fullfilled, but there exists parts which show how to complete the Code Examples requirement. The second requirement that has not been implemented is the Documentation requirement. The javadoc does exist, but the Incomit Service Extension SDK Developer Guide and the Development Environment User's Guide have not been created. However, some of the information in those two documents is available in this thesis.

The Extension SDK is a functional SDK. If we recollect the parts in Section 2.5 that listed what a SDK normally consisted of and compare with the SDK created here, we can see that there are some small differences. The largest difference is that in order to use this SDK the developer must have access to the runtime files, which means that these are not included into the SDK. The SDK is a stand-alone product (see Section 5.3) and should be bought by customers who want to extend the functionality of their Incomit Application Hub.

7.1 Problems

The largest obstacle when developing the SDK was to understand the system that Incomit has and all the terminology surrounding that system. This was a continuous process that was still in progress as the final touches to this thesis were placed. The information about the system mainly came from large documents within Incomit. These documents were created for a telecommunication community and therefore had quite a number of abbreviations and acronyms in them. That, along with the fact that they often were quite large and directed towards a certain part rather than describing a larger overview of the system, made it a bit like a solving a big jigsaw puzzle.

Another large task was to understand how eclipse's plug-in environment worked and how to create an eclipse plug-in. This was pretty straightforward at first, but when external libraries should be added some issues came up. When testing of a plug-in is done in eclipse another instance of eclipse is started. The new instance has the plug-in installed automatically. The problem was that the testing and actually deploying of the plug-in was two different things. Another problem with the eclipse plug-in was the usage of a property file tied to the plug-in. This property file could be used to change strings in the code, in the same way the Extension SDK works. The problem was that the property file was included in the jar-file which made modification of that property-file a hassle. If the property file was extracted outside of the jar-file, the plug-in was unable to locate it. After a lot of testing it would seem as the property-file had to be named "plugin.property", and this made it work. It was quite hard to find good help on these issues even though the eclipse community is large. A forum[1] was found and proved to be of help.

In order to fullfill the requirement with different types of network plug-ins there was an initial attempt to create a template for each service capability (messaging, call control and more). This attempt was abandoned after a reassessment of time left at that stage. The solution instead was to create stubs from the services capability interfaces described

[1]http://www.eclipseplugincentral.com/

in Chapter 6.

When testing the created network plug-ins an error occurred in the SLEE. The SLEE claimed that it could not recognize the type of the network plug-in. It was later found that the Application Hub the plug-in was deployed into was a light version of a real Application Hub and that this light version lacked reference to the plug-in interface. After adding this plugin reference no error was displayed.

7.2 Future work

Future work on the Extension SDK has to be done if it should be released as a product. The testing phase of the Extension SDK has been too short and too limited to validate its functionality.

The Extension SDK has been created in such a way that it is very easy to extend. It is encouraged to extend the toolkit further, with services mentioned in the Requirement Specification in the Appendix.

The parsing of the `xyzOperations.java` files is done in a restricted way and should possibly be addressed to enable for more general parsing of that file - or better yet, parsing from IDL to java stubs according to the java-to-idl standard.

References

[1] Etsi. Website, 21th April 2005. http://www.etsi.org/.

[2] The 3rd Generation Partnership Project. Jslee and the jain initiative. Website, 9th December 2004. http://www.3gpp.org/.

[3] L Klostermann A Moerdijk. Opening the networks with parlay/osa apis: standards and aspects behind the apis. Website, 21th April 2005. http://www.3gpp.org/ftp/tsg_cn/WG5_osa/_Pesentation_Tutorial_Press-Release/Opening_the_networks_with_Parlay_OSA.PDF.

[4] Inc. CollabNet. Cvs home. Website, 30th Januray 2005. https://www.cvshome.org/.

[5] Bruce Eckel. Comparing c++ and java. Website, 2nd May 2005. http://www.javacoffeebreak.com/articles/thinkinginjava/comparingc++andjava.html.

[6] Rolf Ejvegård. *Vetenskaplig metod*. Studentlitteratur, 2nd edition, 1993.

[7] Steve Loughran Erik Hatcher. *Java Development with Ant*. Manning, 4th edition, 2003.

[8] Inc. Free Software Foundation. Gnu make. Website, 29th October 2004. http://www.gnu.org/software/make/.

[9] Inc. Free Software Foundation. Cvs gnu project. Website, 30th Januray 2005. http://www.gnu.org/software/cvs/.

[10] The Parlay Group. The parlay group. Website, 1st December 2004. http://www.parlay.org/.

[11] Jean-Pierre Hubaux, Constant Gbaguidi, Shawn Koppenhoefer, and Jean-Yves Le Boudec. The impact of the Internet on telecommunication architectures. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(3):257–273, 1999.

[12] Jason Hunter and Brett McLaughlin. Jdom project. Website, 13th January 2005. http://www.jdom.org/.

[13] Incomit. Incomit application hub 5.1 product description.

[14] Incomit. Incomit osa gateway 5.1 product description.

[15] Incomit. Incomit application hub 5.1 product description. *Comput. Archit.*, pages 85–87, 2004.

[16] ISO. Iso16387. Website, 30th Januray 2005. `http://www.iso.ch/cate/d16387.html`.

[17] Cynthia Andres Kent Beck. *Extreme Programming Explained*. ADDISON-WESLEY, 2nd edition, 2004.

[18] Microsoft. Web services home. Website, 29th March 2005. `http://msdn.microsoft.com/webservices/`.

[19] Sun microsystems. Jslee and the jain initiative. Website, 14th November 2004. `http://java.sun.com/products/jain/`.

[20] Sun microsystems. Properties (java api). Website, 28th November 2004. `http://java.sun.com/j2se/1.4.2/docs/api/java/util/Properties.html`.

[21] Sun microsystems. Tutorials and codestamp: Corba. Website, 17th December 2004. `http://java.sun.com/developer/onlineTraining/Programming/JDCBook/corba.html`.

[22] Sun microsystems. Enterprise javabeans technology. Website, 29th March 2005. `http://java.sun.com/products/ejb/`.

[23] Sun microsystems. The java hotspot performance engine architecture. Website, 2nd May 2005. `http://java.sun.com/products/hotspot/whitepaper.html`.

[24] NetBeans. Netbeans platform. Website, 14th October 2004. `http://www.netbeans.org/products/platform/howitworks.html`.

[25] netBeans.org. Netbeans 3.6 performance survey. Website, 29th Januray 2005. `http://performance.netbeans.org/survey/nb36-performance-survey.html`.

[26] 3GPP Technical Specification Group Core Network. Open service access (osa). Website, 21th April 2005. `http://www.3gpp.org/ftp/Specs/html-info/23198.htm`.

[27] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, 2.5 edition, September 2001.

[28] Inc. O'Reilly Media. Xml. Website, 30th Januray 2005. `http://www.xml.com/`.

[29] Ravi Rajagopulan. The impact of open service access on network services. *Lucent white paper*, pages 1–7, 2002.

[30] Wikipedia. Definition of service level agreement. Website, 29th Januray 2005. `http://www.mywiseowl.com/articles/Service_Level_Agreement`.

Appendix A

Appendix

A.1 Original Requirement Specification

A.1.1 Movade ExtSdk-R1: Facilitate development of SCSs

The Extension SDK shall facilitate the development of the following components:

- Internal Plug-ins
- SCS Plug-ins
- SCS-proxies
- JAIN SPA services
- ESPA services providing the following APIs:
- JESPA
- WESPA
- ATE support for a new SCS

A.1.2 Movade ExtSdk-R2: Code templates

The Extension SDK shall facilitate the development of the components by providing code templates.

A.1.3 Movade ExtSdk-R3: Code examples and class libraries

The Extension SDK shall facilitate the development of the components by providing code examples and class libraries containing utility classes (if necessary) for:

- High Availability
- Load Balancing
- Alarm generation
- Event generation
- Overload protection
- Load measurement and generation
- Event channel usage
- Time service usage
- Data Base access
- Trace service usage
- Task scheduling
- Supervised list usage

A.1.4 Movade ExtSdk-R4: Facilitate development - Network Plug-ins

The Extension SDK shall facilitate the development of the Networ Plug-ins by providing:

- Code examples and/or utility classes for:
  - Plug-in manager registration

A.1.5 Movade ExtSdk-R5: Facilitate development - SCS Plug-ins

The Extension SDK shall facilitate the development of the SCS Plug-ins by providing:

- Code examples and/or utility classes for:
  - Plug-in manager registration

A.1.6 Movade ExtSdk-R6: Facilitate development - SCS-proxies

The Extension SDK shall facilitate the development of the SCS-proxies by providing:

- Policy repository templates
- Policy rule file templates (In case the user does not have access to the RDT)
- Code examples and/or utility classes for:
- Parlay Framework registration
- Policy initialisation and evaluation
- Statistics generation
- Charging generation
- Plug-in manager and plug-in access
- SCS manager registration

A.1.7 Movade ExtSdk-R7: Facilitate development - JAIN SPA Services

The Extension SDK shall facilitate the development of the JAIN SPA services by providing:

- Code examples and/or utility classes for:
- JAIN SPA Framework registration
- Statistics generation
- Charging generation

A.1.8 Movade ExtSdk-R8: Facilitate development - ESPA Services

The Extension SDK shall facilitate the development of the ESPA services by providing:

- Code examples and/or utility classes for:
    - ESPA Service registration

A.1.9 Movade ExtSdk-R9: Facilitate development - ATE support

The Extension SDK shall facilitate the development of the ATE support by providing:

- TBD

A.1.10 Movade ExtSdk-R10: Movade DS support

The Extension SDK shall be possible to use standalone and in combination with the Movade DS.

A.1.11 Movade ExtSdk-R11: Other IDE support

The Extension SDK shall be possible to use in combination with Integrated Development Environments from other vendors.

A.1.12 Movade ExtSdk-R12: Facilitate deployment - General

The Extension SDK shall facilitate the deployment of the developed components. Generally for the developed components this includes:

- Deployment into SLEE
- TBD

A.1.13 Movade ExtSdk-R13: Facilitate deployment - Internal plug-ins

The Extension SDK shall facilitate the deployment of developed internal plug-ins. This includes:

- TBD

A.1.14 Movade ExtSdk-R14: Facilitate deployment - SCS plug-ins

The Extension SDK shall facilitate the deployment of developed SCS plug-ins. This includes:

- TBD

A.1.15 Movade ExtSdk-R15: Facilitate deployment - SCS-proxies

The Extension SDK shall facilitate the deployment of developed SCS-proxies. This includes:

- Provisioning of new service types into the Parlay FW.
- Provisioning of new statistics type into the SLEE Statistics service
- Provisioning of policy rules

A.1.16 Movade ExtSdk-R16: Facilitate deployment - JAIN SPA services

The Extension SDK shall facilitate the deployment of developed JAIN SPA services. This includes:

- Provisioning of new service types into the Parlay FW

A.1.17 Movade ExtSdk-R17: Facilitate deployment - ESPA services

The Extension SDK shall facilitate the deployment of developed ESPA services. For the actual ESPA service this includes:

- Provisioning of new service types into ESPA
- TBD

For the APIs provided by ESPA this includes:

- JESPA
- Provisioning of the developed components into the DS
- WESPA
- Publication of the WSDL files

A.1.18 Movade ExtSdk-R18: Standalone Extension SDK

The SCS SDK shall be possible to use standalone, i.e. all sources and development tools required shall be included in the SCS SDK. This means that the SCS SDK shall be possible to install and use on a clean machine without having to download or install any additional tools or sources.

A.1.19 Movade ExtSdk-R19: Documentation

The following documentation shall be available:

- Movade Service Extension SDK Developer Guides
- Javadoc for:
- class libraries
- SLEE
- JAIN SPA Framework (Service registration)
- Parlay Framework (Service registration)
- ESPA Access
- Plug-in interface
- DS Usert́s Guide

A.1.20 Movade ExtSdk-R20: Windows

The Extension SDK shall support windows 2000.

A.1.21 Movade ExtSdk-R21: Possible to install on top of Application development SDK

The Extension SDK shall be possible to install on top of an installed Application development SDK. This means that the installation shall be possible to skip and only the templates e.t.c. should be installed in the existing DS.

A.1.22 Movade ExtSdk-R22: Installation wizard

It shall exist an installation wizard. It shall be possible to choose if DS shall be installed or not. E.g. in some cases the Movade Application SDK might already be installed, and in some cases the user only wants to have the ext_sdk templates library.