Managing License Incompatibilities Distributing Eclipse Application Stacks

By

Samrat Dhillon

A thesis submitted to the Faculty of Graduate Studies and Research

In partial fulfillment of the requirements for the degree of

Masters of Applied Science in Technology Innovation Management

Department of Systems and Computer Engineering

Carleton University

Ottawa, Canada, K1S 5B6

July, 2008

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-44036-0
Our file Notre référence
ISBN: 978-0-494-44036-0

Abstract

This thesis proposes a technique to distribute components of application stacks that have incompatible open source licenses and operationalizes the proposed technique. The proposed technique facilitates the distribution and setup of software components that cannot be distributed due to open source license incompatibilities. The development of the solution suggests that it is easier to solve this problem on modular platforms that have interfaces to connect components during runtime.

Acknowledgements

I would like to express my gratitude to my co-supervisors Professor Dwight Deugo and Professor Tony Bailetti for their guidance, understanding and patience throughout this research. My gratitude also goes to Professor John Callahan and rest of the TIM faculty for their valuable feedback. The support received from the Talent First Network for this research is gratefully acknowledged.

Table of Contents

List of Tables

List of Figures

List of Appendices

1 Introduction

Due to incompatibility of open source software (OSS) licenses, developers cannot integrate, package and distribute all the required components of an application or stack. Consider the Eclipse C/C++ Development Tooling[1] (CDT) project where the Eclipse Public License[2] (EPL) and the GNU[3] Public License[4] (GPL) are incompatible. Due to this license incompatibility the GNU C compiler cannot be distributed with Eclipse as a stack. Since the two applications are not distributed together as a stack, this often leads to configuration problems[5].

The problem that this thesis addresses is distribution of software components covered by OSS licenses that are incompatible. The solution proposed is a technique that can be used to distribute application stacks having open source (OS) components with incompatible licenses. The value of the solution proposed is to reduce time and effort configuring software components for the user.

This chapter contains six sections. Section 1.1 describes the research objective. Section 1.2 explains the relevance of this research. Section 1.3 discusses the deliverables of this thesis. Section 1.4 discusses the contributions this research

---

[1] Eclipse C/C++ Development Tooling Project http://www.eclipse.org/cdt/

[2] Eclipse Public License version 1.0 available at http://www.eclipse.org/legal/epl-v10.html

[3] GNU is a recursive acronym for "GNU's Not UNIX", which is a free software operating system and was announced by Richard Stallman in 1983. The official website of GNU is http://www.gnu.org.

[4] GPL version 3.0 available at http://www.gnu.org/licenses/gpl.html

[5] http://max.berger.name/howto/cdt/ar01s05.jsp

makes. Section 1.5 provides definitions of frequently used terms. Section 1.6 describes how this thesis is organized.

1.1 Objective

The objective of this research is to enable the distribution and setup of components that cannot be distributed in a package due to incompatible licenses in application stacks.

1.2 Relevance

This research is relevant for at least four reasons. It is relevant to suppliers of stacks based on OSS, because the technique presented in the thesis can be used to distribute application stacks having OS components with incompatible licenses. This technique increases the options to choose components. The distributors can base their component selection criteria on quality of the component rather than being limited by license compatibility.

This research is relevant to stack users as it decreases the time spent on installing and configuring components that cannot be distributed together due to incompatible licenses.

This research is relevant to OSS foundations because it helps them to increase users' adoption of their software by managing problems related to configuration of stacks. For example the Eclipse CDT project can use this approach to distribute a complete CDT stack.

This research is relevant to researchers and academics as it highlights the importance of the incompatible OS license problem. The compatibility of license is considered to be a primary issue that inhibits the integration of code and has gained very little attention (Subramanian & Soh, 2006). Detection of incompatibilities and resolution is critical for the legally authorized use of components on a significant scale (Gangadharan, Weiss, Esfandiari & D'Andrea 2007a).

1.3 Deliverables

This research provides the following deliverables:

- A technique to manage the problem of distributing components with incompatible licenses on Eclipse platform

- A system that operationalizes the proposed technique on the Eclipse platform. This technique is operationalized using the Enhanced Package Manager (EPM) framework[6]. The EPM framework was developed in this thesis to: (i) operationalize the technique and (ii) to have a powerful package management framework available to Eclipse users.

- The identification of the attributes of a technique to manage license incompatibilities based on the lessons learned from building the system.

[6] According to Firesmith (1994) a framework is "Any portion of a software system that is designed to provide some useful services through refinement and extension by client developers".

1.4 Contributions

This thesis makes at least two contributions. First, the proposed solution will facilitate the distribution and setup of components that cannot be distributed due to open source license incompatibilities.

The second contribution of this research is an Enhanced Package Management framework for Eclipse IDE which is extendable.

1.5 Definitions

This section defines some of the terms that are used frequently in this thesis. The definitions provided are according to the context in which these terms have been used in this thesis.

- Component: - A component can be software, library or a plug-in which is distributed under a particular OS license.

- Application: - An application is complete software in itself and may or may not have an installer program associated with it. Examples of Application software are database applications like MySQL or Ingres.

- Libraries: - Libraries here refer to the libraries required by a program to compile or execute. A typical example of libraries can be the Java Server Faces (JSF) libraries, the Hibernate Object Relational Mapping (ORM) libraries and the Database drivers.

- Standalone Package: - Standalone Package refers to a software program that does not have an installer program associated with it. A typical example for this type of software is the Apache Tomcat server binary without the installer program. It is used by the Eclipse installation to execute the web-applications from inside the Eclipse.

- Plug-ins: - The term plug-ins is used specifically for Eclipse plug-ins. Eclipse is very modular software and users can extend its functionality by installing plug-ins. Eclipse also allows developers to develop Eclipse plug-ins through the Eclipse Plug-in Development Environment (PDE).

- Stack: - A stack consists of multiple applications or components, packaged and distributed together. In a stack, multiple applications or components are configured to work together. An example of a web stack is LAMP (Linux, Apache, MySQL, and PHP), in which PHP is configured to run on Apache web-server, using the MySQL database on Linux operating system.

1.6 Organization

This thesis is organized into six chapters. Chapter 1 is the introduction. Chapter 2 reviews the literature. Chapter 3 describes the approach to manage OS license incompatibilities and the architecture of EPM framework. Chapter 4 describes the operationalization aspects of the approach described in Chapter 3. Chapter 5 describes the approaches' validation. Chapter 6 provides the conclusions,

discusses the limitations of this research and identifies opportunities for future research.

2 Literature review

To find a technique to manage the problem of incompatible OS licenses, a good understanding of OS license and license incompatibility is necessary. The technique proposed in this thesis to manage incompatible licenses uses the concept of package manager on the Eclipse platform, hence the reader is provided with the background of various package managers, the Eclipse platform and Eclipse plug-in development.

Chapter 2 is organized into six sections. Section 2.1 deals with the definition of OSS and OS licenses. In section 2.2 the literature on OS license incompatibilities is reviewed. Section 2.3 examines the current methods to find and manage OS license incompatibilities. In section 2.4 the literature on package managers is reviewed and in section 2.5 the literature on the Eclipse platform is reviewed. The last section in this chapter provides the lessons learned from the literature.

2.1 Open Source License

According to the definition of OS license by Open Source Initiative[7] (OSI) the license must comply with ten criteria. The criteria are:

Free Redistribution: The license should not restrict anyone from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources.

[7] http://www.opensource.org/docs/osd

Source Code Availability: The program must include source code, and must allow distribution in source code as well as compiled form.

Allow Derived Works: The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.

Maintain Integrity of the Author's Source Code: This is a permissive, not a mandatory, part of the definition. Licenses may limit the open modification principle by requiring distributions of modified source code as original source code plus patches.

Should not Discriminate against Persons or Groups: The license must not discriminate against any person or group of persons.

Should not discriminate against Fields of Endeavour: The license must not restrict anyone from making use of the program in a specific field of endeavour.

Allow Distribution of License: This requires that licenses have legally effective provisions that give the identical rights to and enforce the generational limitations, if any, on second and subsequent generations of users.

License Must not be Specific to a Product: The rights attached to the program must not depend on the program's being part of a particular software distribution.

License Must not Restrict other Software: The license must not place restrictions on other software that is distributed along with the licensed software.

License must be technology-neutral: No provision of the license may be predicated on any individual technology or style of interface.

The OSI foundation lists 71 licenses as of June 2008, that comply with its definition[8].

The Free Software Foundation[9] (FSF) is another organisation that has a similar definition of "Free Software". It defines "Free Software" as software that allows users to run, copy, distribute, study, change and improve the software.

Lerner & Tirole (2002) divide OS licenses into three broad categories, unrestrictive, restrictive and highly restrictive. Fitzgerald (2006) groups OS licenses into four types: reciprocal, academic-style, corporate and non-approved (by FSF or OSI).

2.2 Open Source License Incompatibility

When two components with licenses A and B are combined, the following situations can happen: (i) A subsumes B, (ii) B subsumes A, (iii) A adds to B so that you must observe the requirements of both, or (iv) A and B clash — they cannot both be satisfied (Raymond & Raymond, 2002).

According to Gomez & Quinones (2008) "Two licenses are incompatible when the terms of one of them enters in conflict with the terms of the other one".

---

[8] http://www.opensource.org/licenses/alphabetical

[9] http://www.fsf.org/licensing/essays/free-sw.html

Stallman (2006) says "Two licenses are said to be compatible if the terms of both the licenses can be satisfied at once. But if there is no way to license the combination so as to satisfy the requirements of both the licenses, then the licenses are said to be incompatible".

Table 1: License Compatibility

OSI Certificate	Yes	Yes	Yes	No	No	No
GPL Compatible	Yes	No	No	Yes	No	No
Free Software	Yes	Yes	No	Yes	Yes	No
# of Licenses	15	18	27	14	15	18

Source: Ueda (2005)

The FSF states that "a license p is compatible with license q if: A work licensed under p can be distributed under the terms of q"[10]. This definition seems more specific to GPL compatibility because GPL requires derivative works to be distributed only under GPL. A specific problem with GPL license is that it is incompatible with many other licenses. That is, works under GPL can not be bundled with works under other licenses unless all rights in the other works are waived in favour of GPL (Välimäki & Oksanen, 2002). Table 1 presents a matrix that highlights the problem of GPL incompatibility. To understand the matrix

[10] http://www.fsf.org/licensing/licenses/compat.html

column three is interpreted. According to column three in the table, the number of licenses that are OSI certified, GPL incompatible and are not free software is 27.

2.2.1 Compatibilities Based on Nature of Work

Rosen defines two broad categories for OS license compatibilities based on the nature of work (Rosenlaw & Einschlag, 2004). The categories are:

- **License Compatibility for Collective Work:** Distributors of OSS often aggregate separately developed contributions onto their distribution disks as a convenience for their customers. These contributions may have been designed originally by their authors to interact with other programs in the aggregation, and the original authors or downstream aggregators may even have tested them for compatibility. The aggregator remains responsible for honouring the terms and conditions of the licenses to the individual contributions he or she has collected together including, if necessary, publishing the source code of those contributions and making available copies of the relevant licenses (Rosenlaw & Einschlag, 2004).

- **License Compatibility for Derivative Work:** This type of compatibility deals with the interaction of derivative works that have been created from multiple contributions. For example, a GPL-licensed contribution may be offered for an Apache-licensed derivative work. Or an OSL-licensed contribution may be offered for a GPL-licensed derivative work. This type

of compatibility decides which license terms should apply to the resulting derivative work (Rosenlaw & Einschlag, 2004).

2.2.2 Incompatibilities Based on License Clauses

This section presents the classification of OS licenses incompatibilities based on OS license clauses. OS license clauses can be categorized into three types: (i) independent clauses (ii) compelling clauses (iii) repelling clauses (Gangadharan et. al., 2007a). Independent licensing clauses do have an affect on the license of a derivative. Compelling clauses may restrict the clauses of a resulting license, and forces the resulting license to adhere by the compelling clauses. Repelling clauses do not allow the combination with certain other licensing clause. Gangadharan, Weiss, D'Andrea & Iannella (2007b) prepared a matchmaking algorithm to determine compatibility. Table 2 shows the compatibility of specified against unspecified license clauses. Sharealike, Non-commercial and Attribution are standard clauses of Creative Common license. The terms used in the table are described here:

Composition: Composition is the right of execution with the right of interface modification.

Derivation: Derivation is the right of allowing modifications to the interface as well as the implementation of a service.

Adaptation: Adaptation refers to the right of allowing the use of interface only (independent on the execution of services).

Table 2: Compatibility with Unspecified Scope of Right and Financial Terms

Element1	Element2	Compatibility	Rationale
Adaptation	Unspecified	Compatible	Adaptation is the right for interface reuse only
Composition	Unspecified	Incompatible	A license denying composition can not be compatible with a license allowing composition.
Composition	Adaptation	Compatible	Based on subsumption
Derivation	Unspecified	Incompatible	Derivation requires the source code of interface and implementation to be 'Open'.
Derivation	Adaptation or Composition	Compatible	Based on subsumption
Attribution	Unspecified	Compatible	The requirement for specification of attribution will not affect the compatibility when unspecified.
Sharealike	Unspecified	Compatible	Sharealike affects the composite license requiring that the composite license should be similar to the license having Sharealike element.
Non-Commercial Use	Unspecified	Incompatible	Commercial use is denied by NonCommercialUse.
Payment	Unspecified	Compatible	Payment elements do not affect compatibility directly, if unspecified. The license elements related to payment and charging are dependent on service provisioning issues.

Source: Gangadharan et. al. (2007b)

2.2.3 License Clauses that can lead to Incompatibility

This section reviews the OS license clauses which can lead to incompatibility. For the purpose of this review, six commonly used OS licenses i.e. MIT[11], Berkeley Software Distribution (BSD[12]), GPL, GNU Lesser General Public License (LGPL[13]), EPL and Creative Commons[14] are studied in detail and compared with each other on clauses that can lead to license incompatibility. This covers all the three categories of OS licenses i.e. unrestrictive, restrictive and very-restrictive. This literature helps in determining the reasons of incompatibility between the licenses. The license clauses that can lead to incompatibility are highlighted in Table 3.

[11] MIT License available at http://www.opensource.org/licenses/mit-license.php

[12] BSD License available at http://www.opensource.org/licenses/bsd-license.php

[13] LGPL License version 3 available at http://www.gnu.org/licenses/lgpl.html

[14] Creative Commons Licenses available at http://creativecommons.org/license/

Table 3: License Clauses Leading to Incompatibility

License	MIT	BSD	GPL version 3	LGPL version 3	Creative Commons[15]	EPL
Must Distribute Modified Source	No	No	Yes	Yes	*	Contributions, additions not required to be distributed. Derivatives and modifications should be.
Same License for Derivatives	No	No	Yes	LGPL can only become GPL	*	No
Same License for Combined Work	No	No	Yes	Yes	*	No
No Derivative	No	No	No	No	*	No
Commercial Use	No	No	No	No	*	No
Proprietary Software Linkage	Yes	Yes	No	Yes	Yes	Yes
Patent Distribution Along With the Program	No	No	Yes	Yes	No	Yes(except for additions which are not derivative works)
Patent Retaliation	No	No	Yes	Yes	No	Yes
Jurisdiction	No	No	Yes	Yes	*	Yes
Original Creation	No	No	No	No	No	No
Trademark	No	No	No	No	No	No

[15] Creative Commons Licenses are based on six valid combinations of 4 attributes

Must Distribute Modified Source: The MIT and BSD licenses do not require the distribution of the modified source code. This means that the modified work can be distributed as proprietary software. The GPL and the LGPL licenses require that if distributing the software with or without modifications, the distribution of the source code is must. For Creative Commons license, it depends on the particular license chosen. If Sharealike clause is chosen then the modified source code must be distributed. For EPL the distribution of the source code of the Contributions is not required. For any additions or modifications to the Program which are not Contributions but are derivative works, the source code must be made available when distributing the modified program. (Refer to Appendix B for details on Contribution, Program and derivative work specific to EPL)

The affect of this clause is that if the developer plans to distribute the derivative work as proprietary, then he cannot do that under the very-restrictive i.e. GPL and restrictive i.e. LGPL license, but can distribute it as proprietary under unrestrictive license i.e. BSD and MIT. Hence this clause makes distribution of derivatives under proprietary license, incompatible.

Must Use Same License for Derivative Work Distribution: The MIT and BSD licenses do not put any limitation on the distribution of derivative works under a particular license. The derivatives for these licenses can be distributed under proprietary licenses also. The GPL license requires that the derivative work must be distributed under GPL license. If the LGPL work is a Library, then its derivative must remain a Library. A derivative work based on LGPL license can

be licensed under GPL also. For Creative Commons, if the Sharealike clause is chosen, then the derivative work has to be distributed under a similar or compatible license, although the license can be different. The EPL does not require that the derivative work be distributed under the same license. But the EPL part of the code must remain EPL. The additions, modifications which are derivatives can be distributed under a different license. If a new license is being used for a derivative work, then the license should make it clear that only the copyright holder of the new license is responsible for any modifications.

For this clause to be implemented, it is important to understand, what exactly a derivative work is. For GPL anything that links to the GPL even during runtime is a derivative (except a program communicating at arms length). Whereas for EPL, the plug-ins developed using Eclipse are not considered as derivative works. So EPL derivatives will be incompatible with GPL, but at the same time Eclipse plug-ins may not be.

Advertising Clause: The original BSD license had a clause that required authors to include an acknowledgment of the original source in all the derivative works of BSD licensed work. This made the original BSD licensed work incompatible with GPL, because GPL does not allows any further restrictions (Wheeler, 2008). Other licenses like MIT, Creative Commons, LGPL, GPL and EPL do not have any such clause.

Commercialization: MIT, BSD, EPL, GPL and LGPL do not put any restrictions on the commercial use of the original work or its derivative works. Creative

Commons license does not allow the commercial use of the software if the clause Noncommercial is selected. Such a Creative Commons license will be incompatible with GPL because GPL does not allow more restrictions to be imposed on the derivative work.

No Derivative Works: This clause was found only in the Creative Commons OS license and not in any of the other five licenses that were reviewed. If this clause is selected then the recipients are not allowed to create any derivative works. This makes Creative Commons incompatible with GPL and LGPL.

Same License for the Combined Work: The context for this section are works that are based on an open source Program but are not derivative of software under one particular license but make use of other software. MIT and BSD licenses do not put any restrictions on such software to choose a particular license. In case of GPL and LGPL if the work is distributed in isolation, then it can be distributed with any license, but if it is combined with GPL then the whole work must be distributed under GPL license. However merely aggregating software on storage medium for distribution does not require using GPL or LGPL license for distribution. Creative Commons does not put any such restrictions. EPL does not require using EPL for a work which combines EPL and some other license. However the EPL portion of the code must remain EPL.

This clause is very important for understanding the distribution of a Stack. In the case of a GPL application in the stack which does not communicate at arms length, the overall license has to be GPL. The GPL works cannot be combined

with the EPL work for distribution. This is because EPL requires that anyone distributing the work grant every recipient a license to any patent that they might hold that covers the modifications they have made. Because this is a further restriction on the recipients therefore distribution of such a combined work does not satisfy the GPL version 2.

Proprietary Software Linkage: Proprietary software linking refers to the linking of closed source applications/libraries with applications/libraries licensed under OS licenses. MIT, BSD, Creative Commons and EPL licenses allow this linkage. Hence MIT, BSD, Creative Commons and EPL are compatible with software licensed under proprietary license. GPL does not allow applications to be linked with proprietary software or libraries. Software licensed under LGPL can be linked with proprietary software. If a library under LGPL is distributed along with the application then the source code for the library must be made available. If the application requires the users to obtain the library on their own, then there is no liability to provide the source code. Hence LGPL can work successfully with proprietary software.

Patent Distribution Along with the Program: MIT, BSD and Creative Commons do not give the recipient any such rights. However EPL, GPL version 3 and LGPL version 3 grant the licensee "non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version".

The earlier versions of GPL and LGPL did not have such clause and were incompatible with EPL. This was because GPL requires that while distributing the work no additional restrictions should be imposed and the patent distribution clause was an additional restriction, hence making GPL and EPL incompatible.

Patent Retaliation: Patent retaliation means that if the recipient of the software license sues the contributor for patent infringement, then the software agreement or license is terminated. The MIT, BSD and Creative Commons do not have any patent retaliation clauses. GPL version 3, LGPL version 3 and EPL, all have patent retaliation clauses. Earlier versions of GPL and LGPL did not have this clause hence making these licenses incompatible with EPL, because this clause adds a further restriction.

Jurisdiction: The MIT and BSD licenses do not state any jurisdiction. GPL and LGPL have a universal jurisdiction. Creative Commons lists a set of 34 countries from which the jurisdiction can be chosen. EPL explicitly states that the agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to the agreement can bring a legal action under this agreement more than one year after the cause of action.

This makes EPL incompatible with GPL version 3. GPL has a universal jurisdiction, whereas EPL explicitly states that the jurisdiction shall be restricted to the laws of State of New York. This is an additional restriction, hence making it

incompatible with GPL. Therefore GPL and EPL cannot be combined and distributed.

Other Clauses: There are a few other clauses not present in the licences that were reviewed but can lead to incompatibility. Academic Free License (AFL) contains clauses related to trademark law which lead to incompatibility with GPL. The NASA Open Source Agreement requires that all the contributions have to be "original creation". This makes it incompatible with GPL (Wheeler, 2008).

This section presented the categorization of OS license incompatibilities based on license clauses. The summary of the discussion related to incompatibilities is highlighted in Table 3. The table highlights some of the key clauses of a license which should be reviewed while making decisions regarding the compatibility of OS licenses.

2.2.4 OS License and Commercial Software Practices

OS technologies create a fundamental conflict for software corporations. Business practices prefer that software corporations retain their software's intellectual property rights, hide the IP from their competitors and make profits on their IP investment. In contrast, the OS philosophy is to distribute source code and not to hide the IP but make profit from services. Licenses that are closed source restrict any reuse of the source code, while licenses such as GPL restrict the developer's freedom in terms of distribution, especially if the source code of

the component is modified or the components are statically linked. (Madanmohan & De, 2004).

Some licenses might restrict the distribution of modified source code unless it allows the distribution of patch files with the source code to modify the program at build time. Other licenses might require derived works to carry a different name or version number from the original software. From a commercial firm's perspective, the license must explicitly permit the distribution of software built from modified source code (Madanmohan & De, 2004).

The developers prefer the BSD, MIT, and X licenses in commercial products due to their unrestrictive nature. Some of the developers feel that third-party indemnification options reduce the risk associated with OS and influence them in adopting such technologies. For example, JBoss Group announced[16] that it will indemnify and defend JBoss customers from legal action alleging JBoss copyright or patent infringement. Other vendors of OSS such as Hewlett-Packard[17], Red Hat, and Novell[18] also offer indemnifications. Moreover, projects and components working under the BSD license can add closed-source versions of features/plug-ins that enhance the software's features and functionality, thus creating value for the developer and the firm (Madanmohan & De, 2004).

[16] http://www.news.com/JBoss-to-indemnify-customers/2100-7344_3-5107915.html

[17] http://www.hp.com/wwsolutions/linux/download/sco-indemnify-qa.pdf

[18] http://www.news.com/Novell-offers-legal-protection-for-Linux/2100-7344_3-5139632.html

2.2.5 Risks Associated in Commercial Environment

Major exposure occurs when integrating OSS, usually as source code, directly into a product. If the product is reproduced or integrated and distributed commercially contrary to the licensing terms and without the licensor's permission, the licensor can claim for damages or enforce to end the product's production, delivery, and sale. If the product's license doesn't comply with the OSS license terms, this leads to the breach of the original license (Ruffin & Ebert, 2004).

Another exposure is infringement of third parties' patents or other Intellectual Property Rights (IPR). Often OSS is a compilation of code from many sources, and it's not easy to detect whether some parts relate to protected IPR. The IPR's owner would seek compensation for the infringement from the OSSs' licensor or licensee. As OSS license terms generally don't provide either an indemnity for such claims or a warranty stating that the OSS doesn't infringe on third-party IPR, the licensee wouldn't have any basis for regaining such damages from the licensor. Even if the licensor or licensee quickly reacts and changes the respective code, it would still remain in delivered products and is thus subject to legal action (Ruffin & Ebert, 2004).

Another risk area is the use of OSS in development activities when OSS is a tool. Often such software tools generate output that contains tool created comments and a specific structure. The resulting artefacts and even the final product might,

in exceptional cases, be considered a derivative work, giving the copyright holder and the tool's owner certain rights to the resulting product (Ruffin & Ebert, 2004).

2.2.6 Summary

Determining license incompatibility is not trivial and is a complex task. With over a thousand possible two-way combinations, exhaustively tabulating all of them would be a job for an army of lawyers and a sheer nightmare for anyone else (Raymond, 2002). However, it is possible to reason out answers for the most important cases by thinking about what requirements licenses on individual parts export to the rest of the program (Raymond, 2002). There are debates about the compatibility of certain licenses which have not been settled. For example the OpenSSL license is viewed as being GPL incompatible by FSF lawyers while the OpenSSL developers claim it's GPL compatible (Wheeler, 2008).

2.3 Current Methods to Find and Manage License Incompatibility

There are primarily two OS projects LIDESC[19] and FOSSOLOGY[20] that help in determining license compatibility. Proprietary solutions are provided by Palamida[21] and Blackduck Software[22] .

Wheeler (2008) suggests three methods for OS projects to manage GPL incompatibility. The methods are:

[19] http://www.mibsoftware.com/librock/lidesc/

[20] http://www.fossology.com/

[21] http://www.palamida.com/

[22] http://www.blackducksoftware.com/

- Using GPL license itself for the project
- Using a license that is known compatible of GPL
- The third alternative suggested is to dual license the program (Refer to Appendix A for details on Dual Licensing)

2.4 Dependency Management

One of the problems in software distribution is the dependence of one software package on another. Dependency management makes sure that, in a distribution, packages can be installed and user installations can be upgraded when new versions of packages are produced, while taking care of the constraints imposed by the dependency metadata (Cosmo et. al., 2006). Package managers provide dependency management, package installation, upgrade and removal on the client side (Mancinelli et. al., 2006).

The package managers check the local database containing the installed software to find out whether all the software required for the current package is already provided by the system; if this check is positive, the software will be installed. If the check is negative, the software will not be installed and a message will inform the user which requirements of the software installation have not been fulfilled by the system (Reber, 2004).

The solution to this problem is that the package manager on the local system has to have a database of all packages available on the remote servers. In this way,

the package manager can resolve all the dependencies by downloading all the necessary software packages and installing them in the best order. As this is not really the right task for the package manager, the most logical solution is to write a front-end for the package manager. The front-end can then use the package manager to install new software, but it will have its own database which it can use to resolve all the dependencies (Reber, 2004).

The following sections describe the architecture and key features of some of the most widely used package management frameworks. Some of the package managers e.g. apt-get and rpm described below are based on Linux operating system. Apache Maven project's dependency feature is also reviewed because it is widely used in web-application development projects to download missing JAR files. The Eclipse based update facility i.e update manager and provisioning project are also reviewed.

2.4.1 Linux Based Package Managers

In case of Linux based package managers DEB[23] and the RPM[24] formats are the most widely used (Mancinelli, et. al., 2006). Though the binary formats of DEB and RPM packages are different, they have a lot of similarities. Both DEB and RPM packages are compressed archives. However, while RPM is actually an ad-hoc format explicitly conceived for package management while DEB packages can be produced using standard tools (i.e., ar and tar), so they can be easily

[23] The Debian Project. Debian policy manual. http://www.debian.org/doc/debian-policy/index.html.

[24] E. C. Bailey. Maximum rpm. http://www.rpm.org/max-rpm.

managed. Another difference between DEB and RPM package format is related to metadata specification. While RPM packages encode it in a binary form as a part of its ad-hoc archive format, DEB packages use a textual representation for it. This fact makes its processing easier.

2.4.2 Apache Maven

Apache Maven[25] is a project by Apache foundation that attempts to apply patterns to a project's build infrastructure in order to promote comprehension and productivity by providing a clear path in the use of best practices. Maven is essentially a project management and comprehension tool and as such provides a way to help with managing: build, documentation, reporting, dependencies, source code management systems (SCMs), release and distribution.

Dependency management is one of the features of Maven that is best known to users and is one of the areas where Maven excels. There is not much difficulty in managing dependencies for a single project, but when dealing with multi-module projects and applications that consist of tens or hundreds of modules it leads to complexities and this is where Maven can help in maintaining a high degree of control and stability.

Maven can be installed as an Eclipse plug-in and can run on any Java enabled platform. Maven is very strongly tied with the web-application development framework and can only download jar files.

---

[25] Apache Maven Project available at http://maven.apache.org/

2.4.3 Eclipse Update Manager

Eclipse Update Manager is a mechanism available inside Eclipse to download, install and upgrade Eclipse plug-ins and features. The Eclipse Platforms update manager downloads and installs new features or upgraded versions of existing features (a feature being a group of related plug-ins that get installed and updated together). The Update Manager constructs a new configuration of available plug-ins to be used the next time the Eclipse Platform is launched. If the result of upgrading or installing proves unsatisfactory, the user can roll back to an earlier configuration.

2.4.4 Eclipse Provisioning Project

Eclipse provisioning[26] (p2) is a new project aimed to replace the current Eclipse Update Manager. Before p2, many Eclipse users circumvented Update Manager and installed new plug-ins by dumping them in the eclipse/plugins/ directory and restarting with the -clean command line argument. There are many drawbacks to this approach. Although p2 will detect plug-ins added directly to the plugins folder (with an associated startup performance cost), alterations to plug-ins installed by p2 in this directory is not supported. If a plug-in that was installed by p2 is manually removed or replaced with a different version, p2 will not detect it and may be broken. The short rule of thumb is: if something is added manually, then it should be removed manually. If it is installed via p2, then it should be uninstall

[26] Eclipse p2 Project available at http://wiki.eclipse.org/Equinox_p2_Getting_Started

via p2. P2 provides a new dropins folder that is much more powerful and allows separation of content managed by p2 from content managed by other means.

Prior to p2, each Eclipse application had its own private plugins directory where the application's software was kept. This had the drawback that systems with two or more Eclipse-based applications installed ended up with significant duplication of software and other artefacts. Furthermore, the common pieces had to be upgraded separately for each application, often resulting in slow downloads of software already available elsewhere on the local system.

To escape from this duplication problem, p2 natively supports the notion of *bundle pooling*. When using *bundle pooling*, multiple applications share a common plugins directory where their software is stored. There is no duplication of content, and no duplicated downloads when upgrading software.

Section 2.4 reviewed some of the popular Package Management frameworks. Table 4 shows a brief comparison of the frameworks that were reviewed. It is evident that currently there is no fully functional Package Management framework for Eclipse.

The EPM framework on Eclipse is a step in this direction. Currently the Eclipse Update Site only provides update to the plug-ins available, because of that developers face lot of problems due to incompatible libraries, versions, different vendors, and different configuration files. EPM is designed to provide a complete set of packages that can be required on Eclipse platform. With the help of EPM,

developers can specify the version, vendor and configuration files to be used for their stack/application. This will reduce the integration time and effort and at the same time this technique can be used to manage the license conflict problem during distribution of components.

Table 4: Comparison of Package Managers

Package Managers	**Yum**	**Apt-get**	**Maven**	**Eclipse Update Manager**
Type	operating system application	operating system application	Java application	internal to Eclipse
Capability	Linux based applications and libraries	Linux based applications and libraries	only jar files for the project	Eclipse plugins
Operating System	Linux	Linux	operating system independent	operating system independent

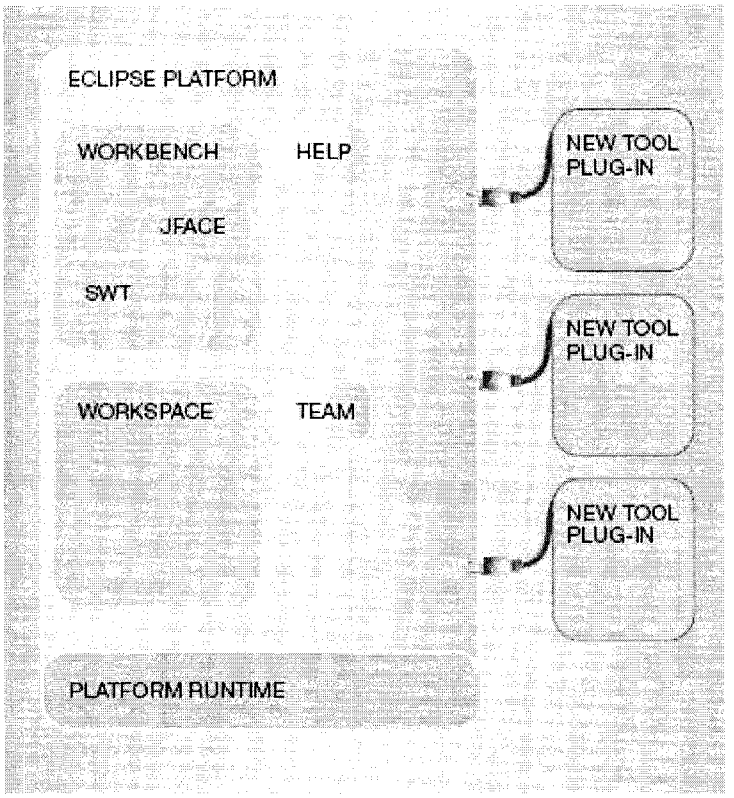
2.5 Eclipse Platform for Integration

The Eclipse Platform is an OS workbench for the integration of software development tools. Because of its extensibility and its industrial-quality user interface, Eclipse has been used as a framework for software engineering research projects. Its advanced plug-in technology allows the concurrent deployment of a large number of independently developed components. This makes Eclipse an excellent composition environment for development tools.

Although the Eclipse Platform has much built-in functionality, most of that functionality is very generic. It takes additional tools to extend the platform to

work with new content types, to do new things with existing content types, and to focus the generic functionality on something specific.

Figure 1: Eclipse Platform Architecture



Source: (Rivières & Wiegand, 2004)

The Eclipse Platform, as shown in Figure 1, is built on a mechanism for discovering, integrating, and running modules called plug-ins. A tool provider writes a tool as a separate plug-in that operates on files in the workspace and surfaces its tool-specific UI in the workbench (Rivières & Wiegand, 2004). When the platform is launched, the user is presented with an IDE composed from the set of available plug-ins. The quality of the user experience depends significantly

on how well the tools integrate with the platform and how well the various tools work with each other.

The Eclipse Platform was designed and built to meet the following requirements:

- Support the construction of a variety of tools for application development.
- Support an unrestricted set of tool providers, including independent software vendors (ISVs).
- Support tools to manipulate arbitrary content types (e.g., HTML, Java, C, JSP and XML).
- Facilitate seamless integration of tools within and across different content types and tool providers.
- Support both GUI and non-GUI based application development environments.
- Run on a wide range of operating systems, including Windows and Linux.
- Capitalize on the popularity of the Java programming language for writing tools.

The Eclipse Platform's principal role is to provide tool providers with mechanisms to use and rules to follow that lead to seamlessly integrated tools. These mechanisms are exposed through well-defined API interfaces, classes, and

methods. The platform also provides useful building blocks and frameworks that facilitate developing new tools.

2.5.1 Strengths of Eclipse

Multi-level Architecture: Most plug-in systems only provide for one level of plug-ins or components that can be plugged into the framework, but there is no generic way to allow plug-ins to be extended by plug-ins of their own. In Eclipse, though, much of the environment itself is realized in the form of plug-ins to the Eclipse core platform; and many other plug-ins are secondary to those primary plug-ins. Any plug-in can define extension points that other plug-ins can connect to. As a consequence, tool architectures can have many levels (Lüer, 2003).

Explicit Ports: Each plug-in can define any number of extension points, to which other plug-ins can connect themselves. Extension points are specified in the manifest file of a plug-in. They function as specifications of optional requirements of a component, or requirement ports. The client (i.e., the component defining the extension point) declares that it can support any extension that adheres to a given interface. By making extension points explicit, it is easy to see where and how components may be plugged into a system (Lüer, 2003).

Self-Description: Eclipse manifest files provide description of plug-ins. The information given is partly redundant to the information in the code, and partly extends it. Redundant information is given for performance optimization; it allows

the platform to be aware of the essential properties of a plug-in before it has loaded its Java classes. Non-redundant information includes the definition of extension points as requirement ports, version numbers, and user interface information (Lüer, 2003).

Containers to Encapsulate Components: Eclipse puts effort into effectively encapsulating plug-ins from each other. A plug-in can access another plug-in only if it is declared as "required" in its manifest. To make this possible, each Eclipse plug-in has a Java class loader of its own; this class loader can verify whether an attempted access to another class is legal according to the manifest. The class loaders thus realize a container concept – each component runs in a container of its own that regulates communication with other components. In this way, Eclipse manages to avoid unintended or undocumented dependencies between plug-ins (Lüer, 2003).

2.5.2 Weakness of Eclipse

Strict Requirements are not Possible: All Eclipse plug-ins are optional: they can extend the functionality of existing components, but are never required. It may be desirable, however, to strictly require the presence of certain components. For example, a word processing component may require a language library that includes functions such as hyphenation and spell-checking, which depend on the natural language employed. A word processor without such a component does not make much sense; on the other hand, it should be possible to replace this component independently from the rest of the word

processor. In this case, the plug-in does not constitute an optional extension of the system, but a required component that has multiple, alternative implementations. If a plug-in is installed manually, Eclipse will not check if the dependent plug-in is installed or not (Lüer, 2003).

No Explicit Connectors: The Eclipse plug-in architecture is based on the assumption that each plug-in extends specific extension points in specific components. It is not possible to have alternative implementations of the same functionality, since every component that provides a certain functionality (i.e., connecting to a given extension point) will be hooked up to the system (Lüer, 2003).

2.5.3 Eclipse Plug-in Development

The standard configuration of the Eclipse workbench is, indeed, mostly realized as collection of plug-ins. Each plug-in has a file system directory of its own, in which its code and a manifest file are located. The manifest is a text file that provides information about the plug-in. When the Eclipse platform is started up, all the plug-in manifests are read, and the associated plug-ins are hooked up with the system. For example, a plug-in might define an additional toolbar button, and after the manifest has been read, this toolbar button will be created (Rivières & Wiegand, 2004).

Platform Runtime and Plug-in Architecture: A plug-in is the smallest unit of Eclipse Platform function that can be developed and delivered separately.

Usually a small tool is written as a single plug-in, whereas a complex tool has its functionality split across several plug-ins. Except for a small kernel known as the Platform Runtime, all of the Eclipse Platforms functionality is located in plug-ins (Gamma & Beck, 2003 & D'Anjou et. al., 2004).

Plug-ins are written in the Java language. A typical plug-in consists of Java code in a JAR (Java archive) library, some read-only files, and other resources such as images, Web templates, message catalogues, native code libraries, and so forth. Some plug-ins do not contain code at all. One such example is a plug-in that contributes on-line help in the form of HTML pages. There is also a mechanism that permits a plug-in to be synthesized from several separate fragments, each in its own directory. This is the mechanism used to deliver separate language packs for an internationalized plug-in (Clayberg & Rubel, 2004).

On start-up, the Platform Runtime discovers the set of available plug-ins, reads their manifest files, and builds a plug-in registry. The platform matches extension declarations by name with their corresponding extension point declarations. The problems, such as extensions to missing extension points, are detected and logged. The resulting plug-in registry is available through the platform API. Plug-ins cannot be added after start-up.

An extension point may declare additional specialized XML element types for use in the extensions. This allows the plug-in supplying the extension to communicate arbitrary information to the plug-in declaring the corresponding

extension point. Moreover, manifest information is available from the plug-in registry without activating the contributing plug-in or loading any of its code. This property is critical to supporting a large base of installed plug-ins, only some of which are needed in any given user session. Until a plug-in's code is loaded, it has a negligible memory footprint and impact on start-up time. Using an XML-based plug-in manifest file also makes it easier to write tools that support plug-in creation. The Plug-In Development Environment (PDE), which is included in the Eclipse SDK (software development kit), is such a tool.

A plug-in is activated when its code actually needs to be run. Once activated, a plug-in uses the plug-in registry to discover and access the extensions contributed to its extension points. For example, the plug-in declaring the user preference extension point can discover all contributed user preferences and accesses their display names to construct a preference dialog. This can be done by using only the information from the registry, without having to activate any of the contributing plug-ins. The contributing plug-ins will be activated when the user selects a preference from a list. Activating plug-ins in this manner does not happen automatically; there are a small number of API methods for explicitly activating plug-ins. Once activated, a plug-in remains active until the platform shuts down. Each plug-in is furnished with a subdirectory in which to store data specific to the plug-in; this mechanism allows a plug-in to retain important state information between runs.

2.6 Lessons Learned

The key lessons learned from the literature are:

- Very restrictive and restrictive OS licenses affect packaging and distribution more than the unrestrictive OS licenses do. Hence extra care should be taken when using software distributed with very restrictive and restrictive OS license.

- Detecting OS license incompatibilities is a complex task. While there are tools to detect OS license incompatibilities, the outcome of these tools have not been tested in courts.

- Currently there is no comprehensive package management framework for Eclipse. The Eclipse Update Manager can only install Eclipse plug-ins.

- Software with plug-in interface can be easily extended and distributed to provide value added tools. In case of OSS, this helps in easier contribution to the project and also helps in creating business around the OS project where developers can build and commercialize custom components around the main project.

- During deployment a significant amount of time is spent configuring different software to work together.

3 Approach

This chapter describes the technique to manage the license incompatibilities in Eclipse application stacks using the EPM framework. It also describes the technique to develop Eclipse plug-ins that can make use of the EPM framework. This chapter is organized into four sections. Section 3.1 describes the technique to manage the license incompatibilities advanced in this thesis. Section 3.2 describes the deletion criteria for this technique. Section 3.3 describes the architecture of the proposed technique along with the meta-data for the repositories. Section 3.4 describes how to develop plug-ins using the proposed technique.

3.1 Technique to Manage License Incompatibilities

This thesis proposes to manage the incompatibilities of OS licenses in Eclipse application stacks by following the steps outlined below:

- The first step involves identifying the components whose license causes the incompatibility in integration, packaging and distribution. There can be multiple components whose licenses can cause the incompatibility. Identification of the license conflicts is a complex task and requires thorough understanding of the license of the component being used. Some of the OS license clauses that lead to license incompatibilities were discussed in section 2.2.

- The second step involves the deletion of those components identified in the first step from the Eclipse application stack and preparing a dependency file which contains the information about the components which have been removed.

- The third step involves the usage of EPM during the distribution of the Eclipse application stack, so that the missing components can be downloaded and installed, before the execution of the application stack.

3.2 Deletion Criteria

As discussed in section 3.1, the components whose license causes incompatibility during integration, packaging and distribution have to be deleted. The deletion criteria can vary in different situations. Some of the situations are discussed in section 3.2.1 to 3.2.3.

3.2.1 Architecture

In the case where the application stack has a component that will first install itself and then download the rest of the missing components, the components whose license is incompatible with this component's license must be deleted.

3.2.2 Type of License Being Used

Some OS licenses require that the overall license of the combined work must be the same as that of the component's license if that component is being used. In such a case, if there are incompatible licenses in the combined work, then

components whose licenses are incompatible with the license of the combined work, should be deleted.

3.2.3 Size of the Component

In the case where there is no license that enforces the license of the combined work, one of the deletion criteria is to delete the component with the smallest size, whose license is incompatible with other components. The deletion of the component with smallest size will decrease the download and setup time because component with smaller size can be downloaded quickly.

3.2.4 Approach used for EPM

In EPM approach the criteria for deletion is based on the "Architecture". This is because the EPM framework is a prerequisite on the client system to download deleted components. Components whose license is incompatible with EPL must be deleted and their information should be added to a dependency file.

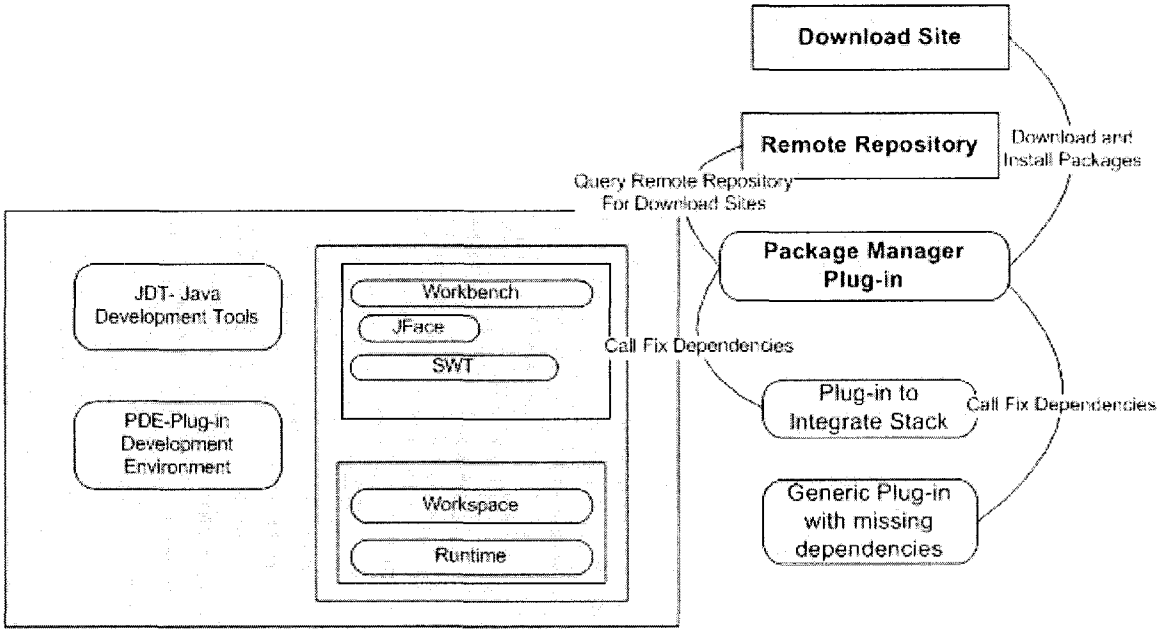
The deletion criteria presented in the sections above is not a comprehensive list. These were some of the criteria that were noted during the course of research.

3.3 System Design

This section describes the design of EPM. The architecture of the proposed solution is presented in Figure 2. The EPM framework consists of the Local Repository, Remote Repository, the PackageManager plug-in, Dependency file, and the mirror servers. The EPM framework can download and install libraries,

Eclipse plug-ins, applications with installers and standalone packages. Figure 7 in Chapter 4 shows a sequence diagram of EPM framework.

Figure 2: Package Manager Architecture



3.3.1 Package

EPM uses the term "package" to define a software component and its attributes. The attributes of a package include the following:

- Package Name: This is the name of the software or the name of the component. For example the package name for Apache Tomcat 5.5 Server Runtime is "Apache Tomcat".

- Package Version: This attribute is equivalent of the version of the component. For example the package version for Apache Tomcat 5.5 Server Runtime is "5.5".

- Package Type: This attribute specifies whether this component is a library, plug-in, installer or a standalone package. This information is required, as EPM will take different actions for different type of components. For example, Eclipse plug-ins should be copied at a particular location for installation and an application with installer will require the execution of the installer file.

- Installer-file: For applications with installers, this attribute specifies the installer file and its location that should to be executed, once the package has been downloaded.

3.3.2 Local Repository

For EPM to work, every Eclipse installation must have a Local Repository. A Local Repository is an XML file that does not contain any information about the packages but contains URLs for Remote Repositories which contain the package information. It has been designed in this way so that there is minimal update required at the user installation, if there is any change in the package information. The Local Repository XML file should be present in the root directory of the Eclipse installation with the name "repository.xml". Figure 3 shows a sample of Local Repository.

Figure 3: Local Repository

```
<?xml version="1.0" encoding="UTF-8"?>
<repository xmlns="http://www.samratdhillon.com/pkmgrLocalRepXMLSchema">
        <site>
                <meta-url>http://www.ingres.com/repo.xml</meta-url>
        </site>
        <site>
                <meta-url> http://freshrpms.net/</meta-url>
        </site>
</repository>
```

In this repository file, there are URLs for two Remote Repositories. The EPM expects the Remote Repository URL should return an XML file by the name of repo.xml.

The Local Repository has three different tags:

- The <repository> tag marks the beginning and the end (</repository>) of the Local Repository.
- The <site> tag is used to differentiate the URLs of different Remote Repositories.
- The <meta-url> specifies the actual URL of the Remote Repository.

3.3.3 Remote Repository

A Remote Repository is an XML file that contains the information about the packages and their download servers.

Figure 4: Remote Repository

```
<?xml version="1.0" encoding="UTF-8"?>
<repository
xmlns="http://www.samratdhillon.com/pkmgrRemoteRepXMLSchema">
        <package type="standalone">
                <package-name>Apache Tomcat</package-name>
                <package-version>5.5</package-version>
                <download-url>
                        http://apache.mirror.rafal.ca/tomcat/tomcat-
                        5/v5.5.23/bin/apache-tomcat-5.5.23.zip
                </download-url>
                <installer-file></installer-file>
        </package>
        <package type="installer">
                <package-name>Ingres database</package-name>
                <package-version>5.5</package-version>
                <download-url>
                  http://localhost:8080/PackageRepository/ingres.zip
                </download-url>
                <installer-file>/ingres/setup.exe</installer-file>
        </package>
        <package type="library">
                <package-name>iijdbc</package-name>
                <package-version>0</package-version>
                <download-url>
                    http://localhost:8080/PackageRepository/iijdbc.jar
                </download-url>
                <installer-file></installer-file>
        </package>
</repository>
```

Figure 4 shows a sample of a Remote Repository. It specifies three different types of packages i.e. a library (iijdbc.jar), an installer for Ingress database with its installer file location and a standalone package (Apache Tomcat 5.5 Server). Remote Repositories do not physically host the components, but point to the actual download locations. This design was chosen because mirror servers for these packages are already present and EPM Remote Repositories need not replicate the information.

The different tags used in the Remote Package Repository are:

- The <repository> tag is used to mark the beginning and end (</repository>) of the Remote Repository.

- The <package> tag is used to define each new component that is made available by the Remote Repository. Defining the type attribute for the <package> tag is mandatory as it is required for processing. The type attribute can have value “library”, ”installer”, “standalone” and “plugin”.

- The <package-name> tag specifies the name of the component.

- The <package-version> tag specifies the version of the component.

- The <download-url> tag specifies the URL of the mirror sever from where the component can be downloaded.

- The <installer-file> tag specifies the path and filename of the installer in case the package type is an installer. In case the package type is not an installer the tag can be left blank.

3.3.4 Dependency File

The dependency file, as shown in Figure 5, is an XML file that contains the list of components that have not been distributed with the Eclipse plug-in due to license incompatibility with the distribution license of the stack application. The dependency file should be named “dependency.xml”.

Figure 5: Dependency File

```
<?xml version="1.0" encoding="UTF-8"?>
<dependency xmlns="http://www.samratdhillon.com/pkmgrDepXMLSchema" >
        <package type="library">
                <package-name>iijdbc</package-name>
                <package-version>1</package-version>
        </package>
</dependency>
```

The dependency file contains the package name, version and its type. This information is sufficient for EPM to correctly map a component from the Remote Repositories. The developer is responsible for determining the license compatibility and preparing the dependency file. The tags used in the dependency file are:

- The <dependency> tag is used to mark the beginning and the end (</dependency>) of the dependency file.

- Each dependency or the missing component is specified inside the <package> tag. The type attribute is also required.

- The <package-name> tag specifies the name of the package.

- The <package-version> specifies the version of the package.

3.4 Technique to Develop Plug-ins

This thesis proposes a technique to develop Eclipse plug-ins that allows the Eclipse plug-ins to make use of EPM to download missing components that are

required by the plug-ins. The following are the requirements on Eclipse plug-ins to use EPM:

- The plug-in should have two directories named - library and dependency. All the libraries required by the plug-in should be present in the library folder. In case where there is a license incompatibility between the library and the stack application distribution license, then that library should be removed from the library folder and the package information for that library should be entered in the dependency.xml file.

- If there are components other than the libraries required by the plug-in, which cannot be distributed due to license incompatibility, the missing component's package information should be added into dependency.xml file and it should be present in the dependency folder. The developer is responsible for determining the license compatibility and preparing the dependency file.

- During the plug-in development, the PackageManager plug-in should be added to the dependencies list of the plug-in being developed. This dependency list is not the same as dependencies in the dependency.xml file. This list contains the list of prerequisite plug-ins required to compile and execute this plug-in. Since the plug-in being developed needs to make an API call to the PackageManager plug-in, it must be present in the list of required plug-ins.

- All the plug-ins making use of EPM must have an Activator class. The developer can choose to include this from the plug-in development wizard. The Activator class extends either org.eclipse.ui.plugin.AbstractUIPlugin (for UI plug-ins) or org.eclipse.core.runtime.Plugin (for non-UI plug-ins) class. From the start method of the Activator class the plug-in should call the DependencyManager.fixDependency method passing the PLUGIN_ID as the parameter.

- EPM requires that the plug-in should not be distributed as JAR file but as a directory. This is required because the PackageManager plug-in will read the dependency.xml file and when there are missing libraries, the libraries are downloaded and saved in the library folder of the plug-in. Since reading and writing into JAR files creates an overhead, the EPM has been designed only to read and write into normal directory file structure and not JAR files.

3.5 Summary

The architecture of the EPM framework offers flexibility to update the component's information. Since the component information is stored in Remote Repository, components can be added and updated without any updates on the clients. For adding new Remote Repository only a new URL has to be added to the Local Repository. The Remote Repositories do not have to host the components locally. This architecture allows leveraging the existing download servers and hence saves disk space on the Remote

Repository servers. The XML format of the Remote Repository and the dependency file is also very intuitive.

The key elements in the technique described in this section are the dependency.xml file and the PackageManager plug-in. As already stated, the format of the dependency.xml file is very simple and only requires name, version and the type of the component. The usage of the PackageManager plug-in is very easy. The plug-in that needs to download dependencies has to call only one method (DependencyManager.fixDependencies) with PLUGIN_ID as the parameter. Hence using this technique is not a complex task.

Using this technique is straight forward for Eclipse developers who have a foundation in plug-in development environment. Even for the developers without a background in plug-in development, this technique is not difficult as the plug-in required to download missing components requires only one function call from the start method of the *Activator* class.

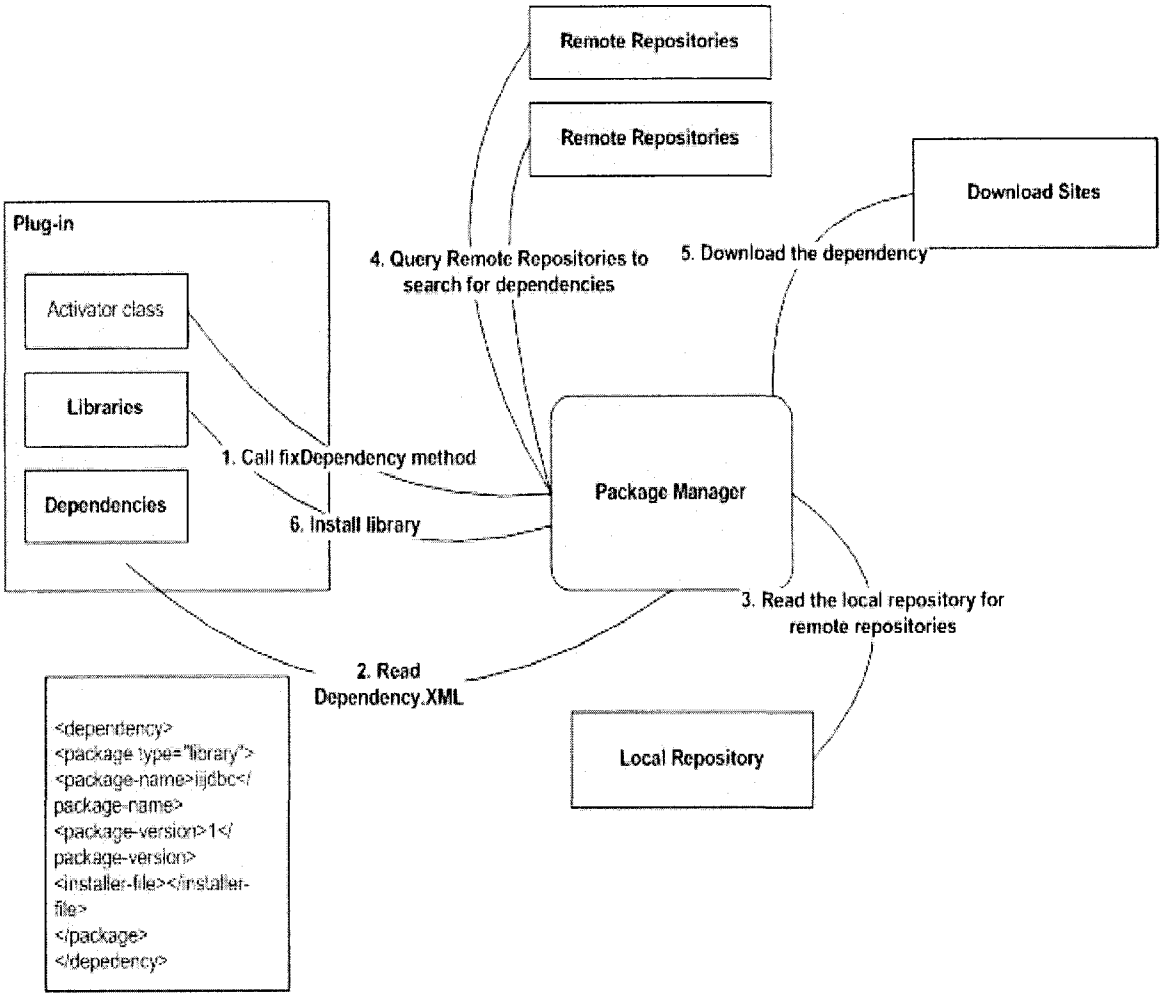
4 Operationalization

This chapter describes the operationalization of the technique proposed in Chapter 3. Chapter 4 is organized into five sections. Section 4.1 describes how PackageManager plug-in communicates with components of EPM. Section 4.2 describes the technique used by the PackageManager plug-in to dynamically refresh components. Section 4.3 and 4.4 describe some of the important configuration files. Section 4.5 describes some of the constraints on plug-in development due the use of the PackageManager plug-in.

4.1 PackageManager Plug-in

Central to the EPM's functioning is the PackageManager plug-in. The PackageManager plug-in connects all the different entities (e.g. Local Repository, Remote Repository and dependency file) of the EPM. Whenever a new plug-in is making use of EPM to get missing components, it calls the DependencyManager's fixDependency method. The PackageManager plug-in then reads the dependency.xml file and prepares the list of packages that have to be downloaded. It then reads the Local Repository present in the root directory of the Eclipse installation for the URLs of the Remote Repositories. It then reads the Remote Repositories and prepares the list of packages available. The next step involves the matching between the missing components and components available from the Remote Repositories. Missing components, whose package-name, package-version and package-type matches with the ones from the Remote Repositories are selected for download and setup.

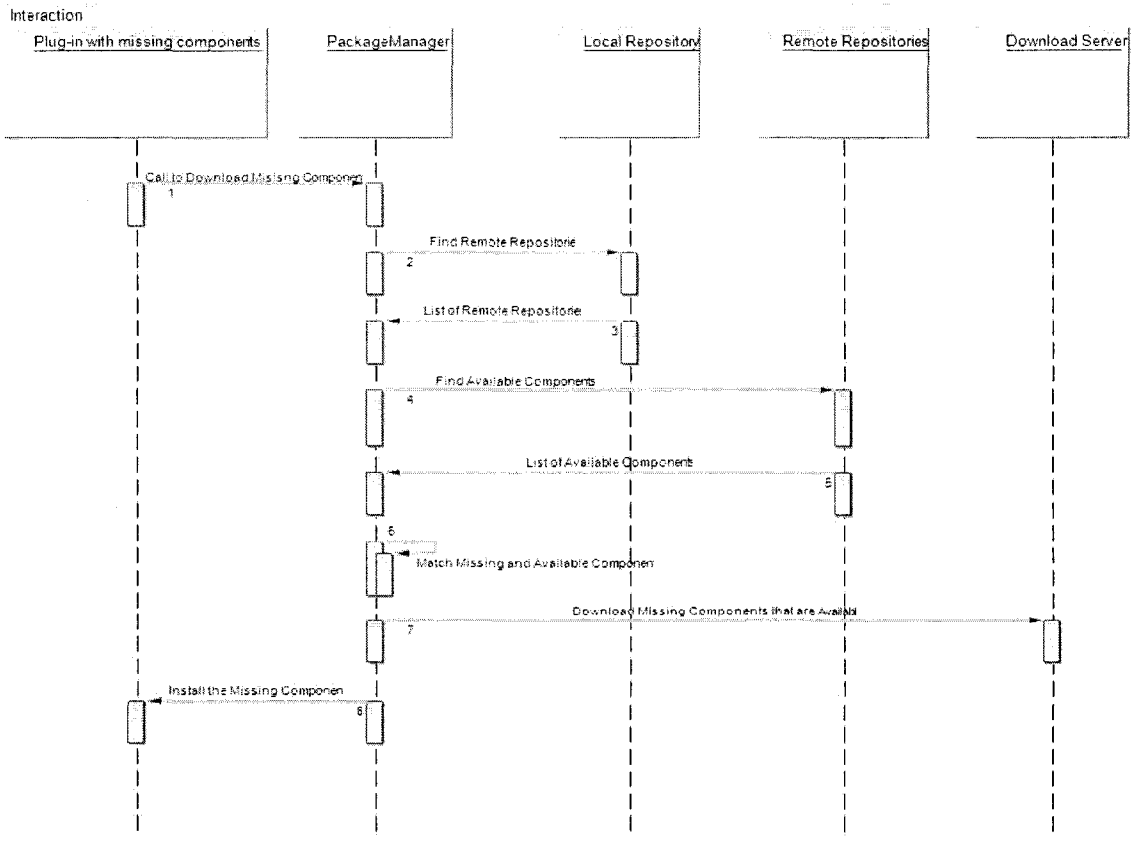
Figure 6: Working of the PackageManager



After the PackageManager plug-in matches and finds the components that are to be downloaded and installed, it tries to download them from the download server. The PackageManager plug-in is currently capable of only downloading using http and all the components are expected to be zip files. This is not a major restriction as in almost all the repositories the components are distributed using http or ftp and they are packaged as a zip or tar files. Figure 7 shows the operation of the PackageManager plug-in.

After the zip file is downloaded, the PackageManager, checks the component's package type. If the package type is a library then the zip file is extracted in the library folder of the plug-in which called the PackageManager plug-in. If the package type is an installer, then the zip file is extracted in the packages folder under the root directory of the Eclipse installation. After the zip file has been extracted, the PackageManager, queries the package list for the installer file and then launches it. When the package type is standalone package, the zip file is extracted in the package folder present under the root directory of the Eclipse installation. Figure 7 presents the sequence diagram that shows the interaction between different modules of EPM.

Figure 7: Sequence Diagram



4.1.1 PackageManager Plug-in Details

The PackageManager plug-in depends on three core Eclipse plug-ins. The plug-ins and their brief descriptions are given below.

- org.eclipse.ui: This plug-in provides application programming interfaces for interaction with and extension of the Eclipse Platform User Interface.

- org.eclipse.core.runtime: This plug-in provides the API related to running the platform. This includes area such as the definition and management of plug-ins and the starting, stopping and maintaining the platform. In addition, this API supplies various utility types such as Path, IPath and various progress monitors.

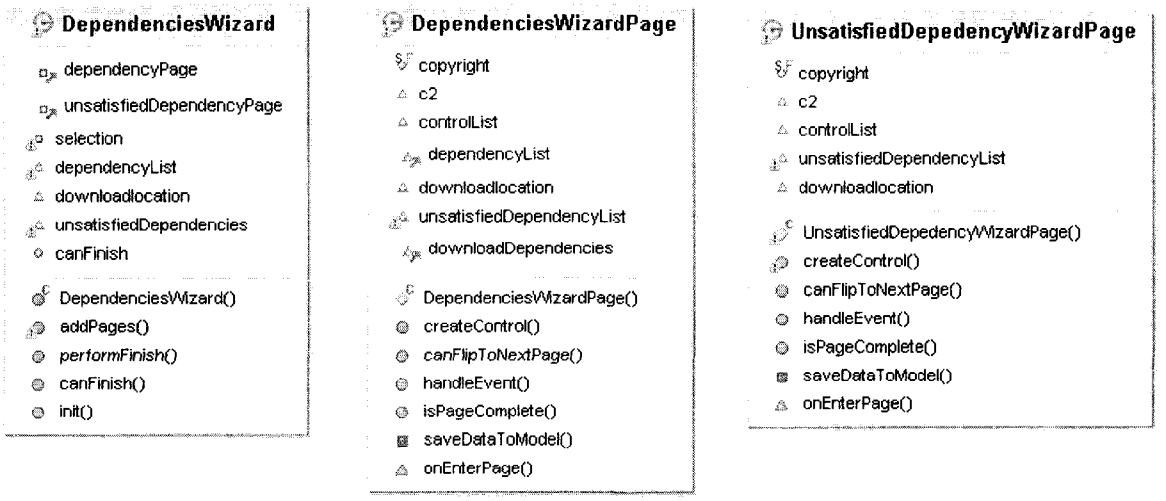
- org.eclipse.core.filesystem: This package specifies the client API for the file system. This API can be used to query and manipulate files stored in some arbitrary backing store.

The PackageManager plug-in consists of the following five java packages. The java packages and classes are described below:

- com.packagemanager.ui: This package provides the wizard interface so that when the PackageManager plug-in detects missing dependencies, it initializes a wizard which shows the missing components and guides the user to download them. This package consists of three classes described in the class diagram in Figure 8. The wizards are created by extending the

Wizard class and implementing the *INewWizard* interface provided by the org.eclipse.ui plug-in. The wizards further require wizard pages for display. These wizard pages are created by extending the *WizardPage* class provided by org.eclipse.ui plug-in.

Figure 8: Class Diagram of com.packagemanager.ui

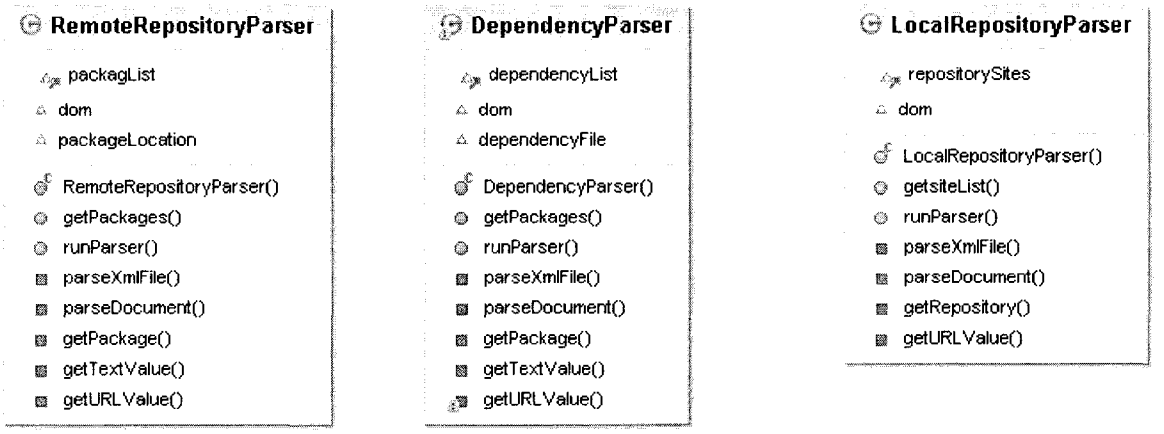


- com.packagemanager.utilities: This package contains two important utility classes *HTTPFileDownloader* for downloading components using HTTP and *Unzip* to unzip the downloaded components. The *HTTPFileDownloader* class uses *URLConnection* class to download components using HTTP from download servers. The *UnZip* class makes use of java.util.zip.ZipFile and java.util.zip.ZipEntry classes to perform the unzip operation.

- com.packagemanager.parser: This package contains three classes. The *DependencyParser* is, responsible for parsing the dependency file. The *LocalRepositoryParser* is responsible for parsing the Local Repository.

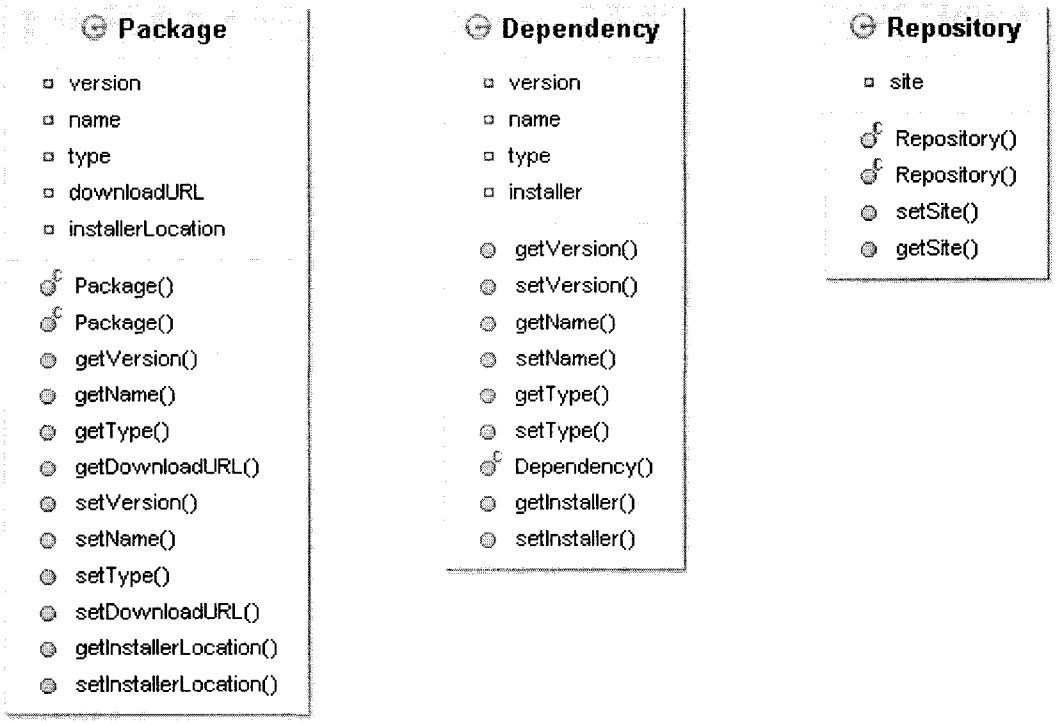
The *RemoteRepositoryParser* is responsible for parsing the Remote Repositories. The parser classes make use of the Document Object Model (DOM) parsing interface. The DOM interface is perhaps the easiest to understand. It parses an entire XML document and constructs a complete in-memory representation of the document. Figure 9 shows the class descriptions of the three parser classes.

Figure 9: Class Diagram of com.packagemanager.parser



- com.packagemanager.model: This package contains the *Dependency*, *Package* and *Repository* classes that define the model used by the *PackageManager* class. The *Package* class represents the component definition as in the Remote Repositories. The *Dependency* class represents the component requirement as specified in the dependency.xml. The *Repository* class models the structure of a Remote Repository. Figure 10 shows the class descriptions of the three model classes.

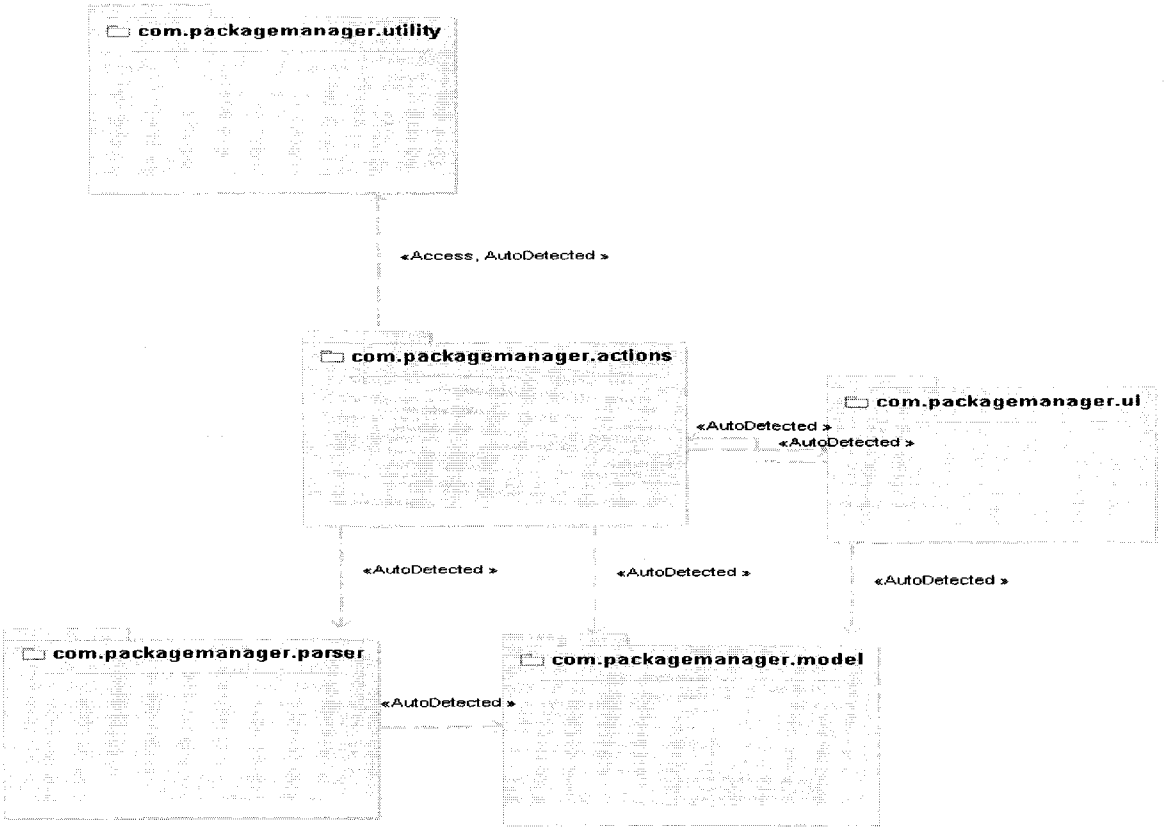
Figure 10: Class Diagram of com.packagemanager.model



- com.packagemanager.actions: This package contains the classes that do the actual processing. The details of this package are discussed in the next section 4.1.2.

Figure 11 shows the dependency diagram between the different Java packages.

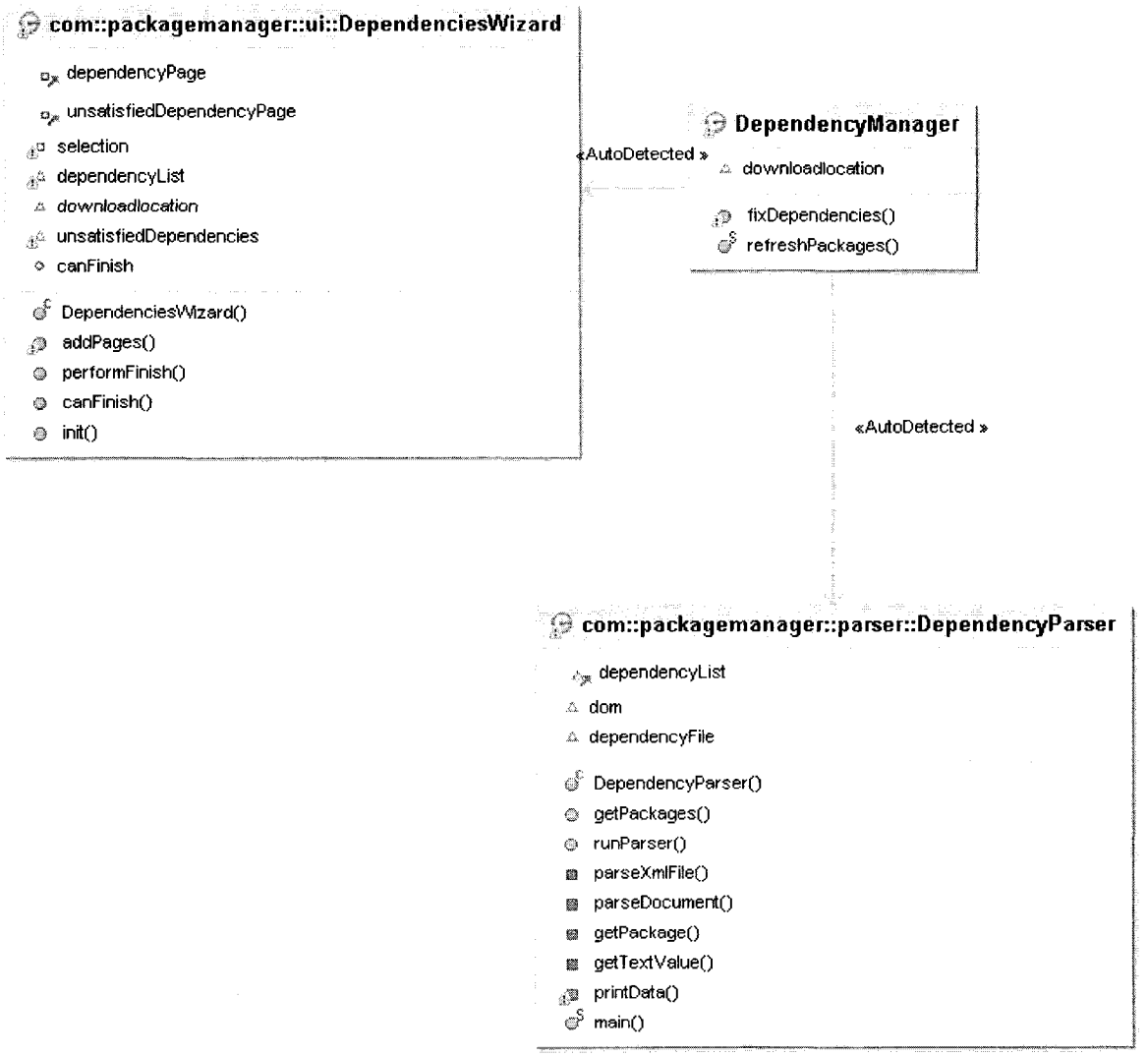
Figure 11: Dependency Diagram



4.1.2 Function of the PackageManager Plug-in

The PackageManager plug-in exposes its functionality to other plug-ins through the *DependencyManager* class. Plug-ins with missing dependencies should call the fixDependencies method of the *DependencyManager* class with their PLUGIN_ID as parameter. This will trigger the PackageManager plug-in to fix the dependencies i.e. to download the missing components.

Figure 12: Dependency Diagram for *DependencyManager*



The *DependencyManager* class uses the provided PLUGIN_ID parameter to read the dependency.xml file located inside the dependency folder. The dependency folder should be present under the root directory of the plug-in. The *DependencyManager* then calls the *DependencyParser* to parse the dependency.xml file and prepare a list of missing components. The missing component list is an *ArrayList* consisting of objects of *Dependency* class. The conversion from the XML file to an *ArrayList* of *Dependency* objects is done by

the *DependencyParser* class. The *DependencyManager* class then calls the *DependenciesWizard* class passing the list of *Dependency* objects. The dependency diagram for the *DependencyManager* class is shown in Figure 12. This brings up the user interface wizard to download the missing components. The wizard is shown in Figure 14.

Figure 13: Dependency Diagram for *DownloadDependencies*

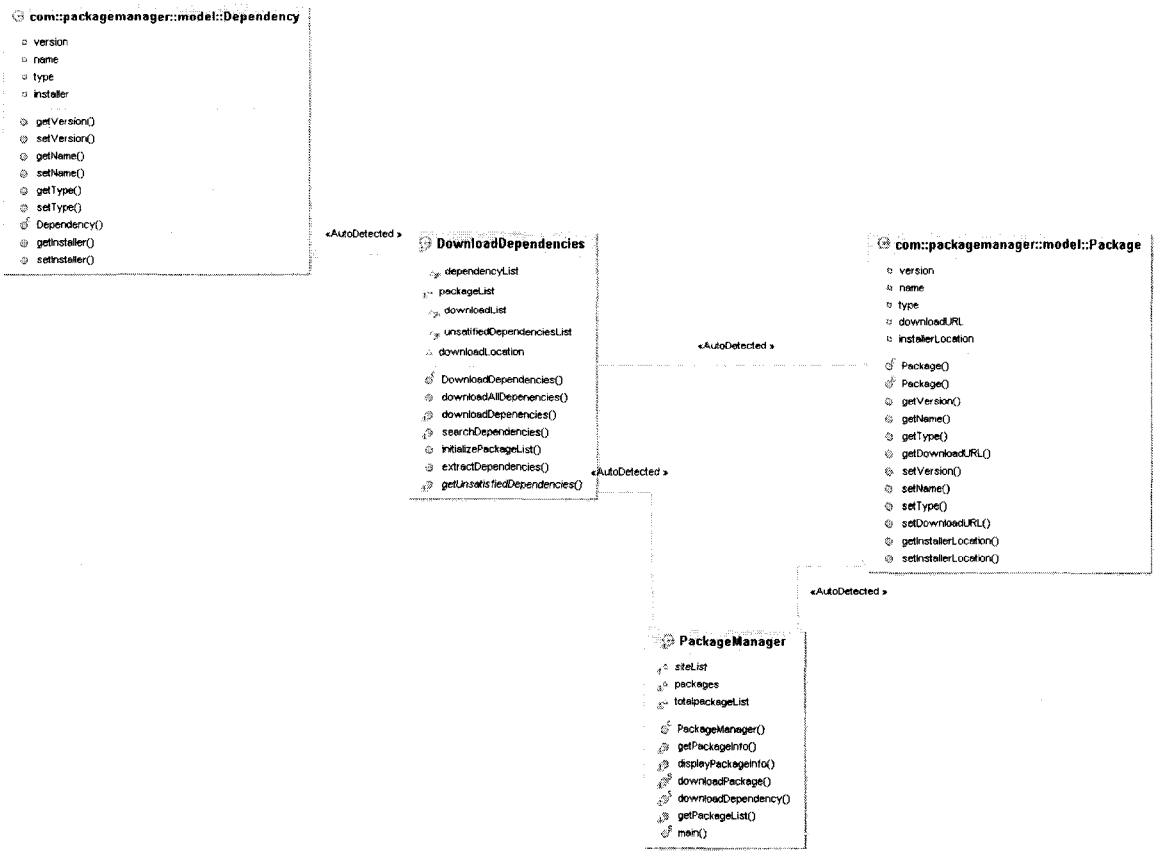
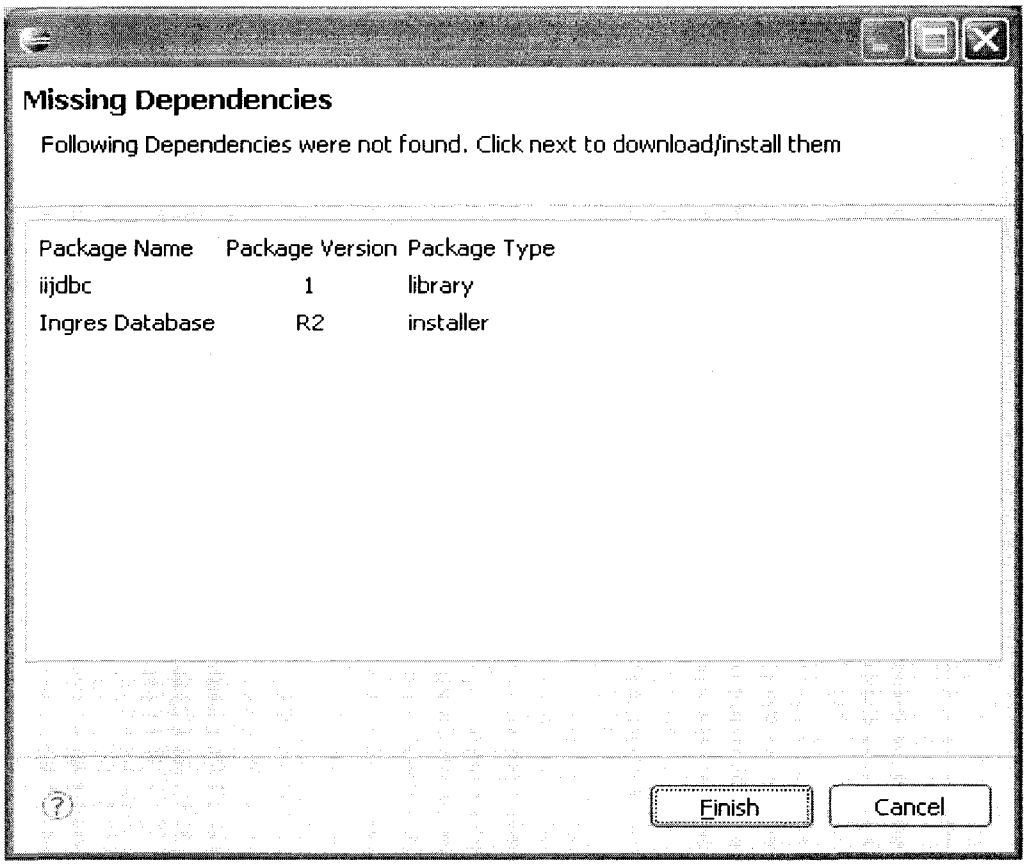


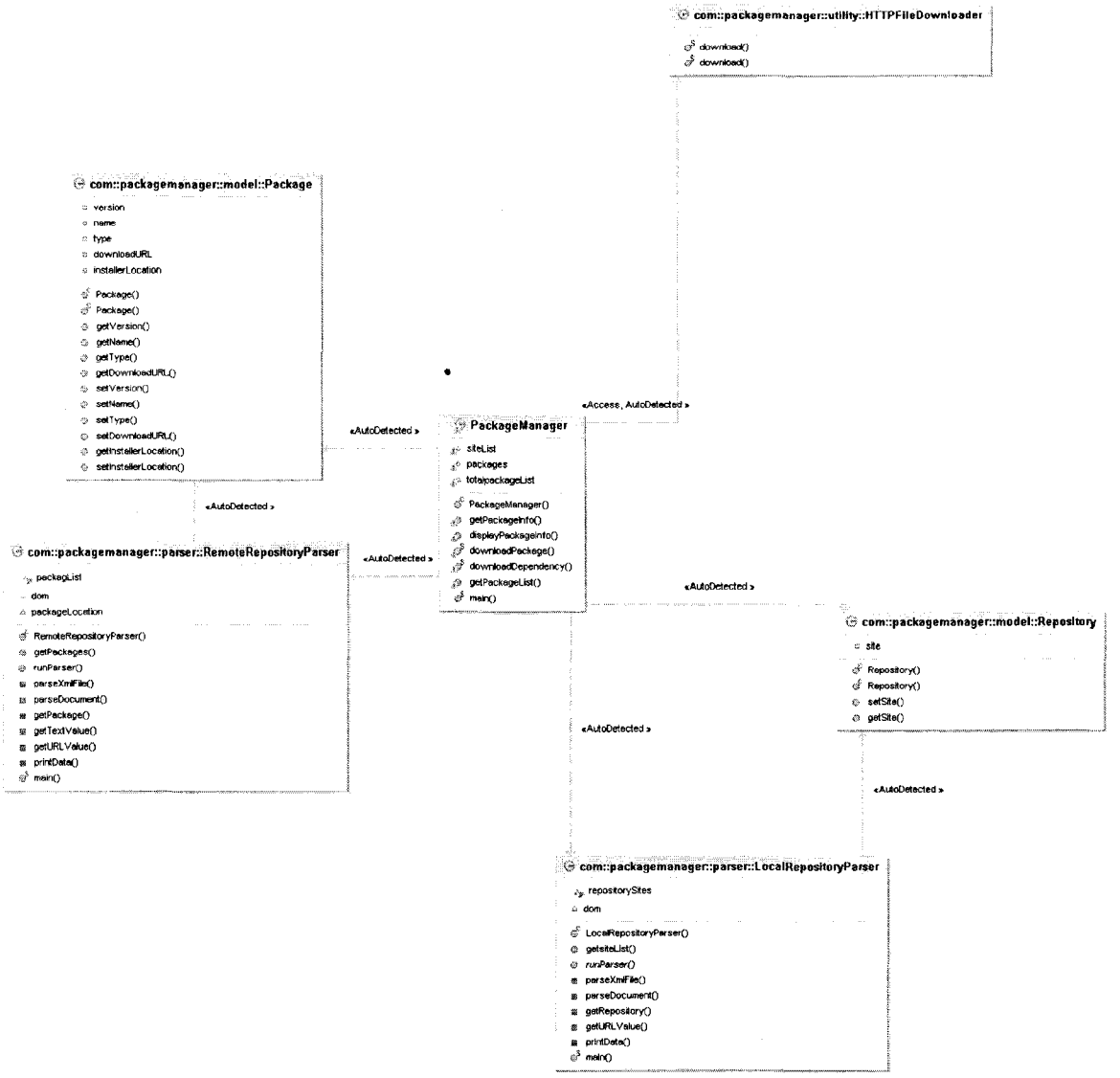
Figure 14: Dependencies Wizard



When the user clicks the finish button of the wizard page, the *DependenciesWizard* calls the *DownloadDependencies* class with the list of Dependency objects. Figure 13 shows the dependency diagram for the *DownloadDependencies* class. The *DownloadDependencies* class first calls the *PackageManager* class to initialize the list of components that are available through different Remote Repositories. The PackageManager plug-in in turn calls the *LocalRepositoryParser* class to get the list of Remote Repositories and then calls the *RemoteRepositoryParser* to get the list of components available from the Remote Repositories. The *RemoteRepositoryParser* returns a list of *Package* objects to the *PackageManager* class. The *DownloadDependencies*

class then uses this list of *Package* objects to match it with the list of *Dependency* objects. The *Dependency* objects whose package name, package version and package type matches with the *Package* objects are selected for download. This is performed by *downloadPackage* method of the *PackageManager* class. Figure 15 shows the dependency diagram for the *PackageManager* class.

Figure 15: Dependency Diagram for *PackageManager* Class



When all the components in the dependency.xml are available from the Remote Repositories and are successfully downloaded and installed, then the *DependencyManager* class deletes the dependency.xml file. Therefore, next time when the plug-in is loaded and makes a call to the *DependencyManager* class, the *DependencyManager* will try to check if the dependency.xml file is present and not find it, then the *DependencyManager* class will return control back to the plug-in without any further processing.

4.1.3 Selective Processing Based on Component Type

The *PackageManager* class takes selective action based on the component's type that is being downloaded. This decision is based on the value of the variable "type" of the Package object.

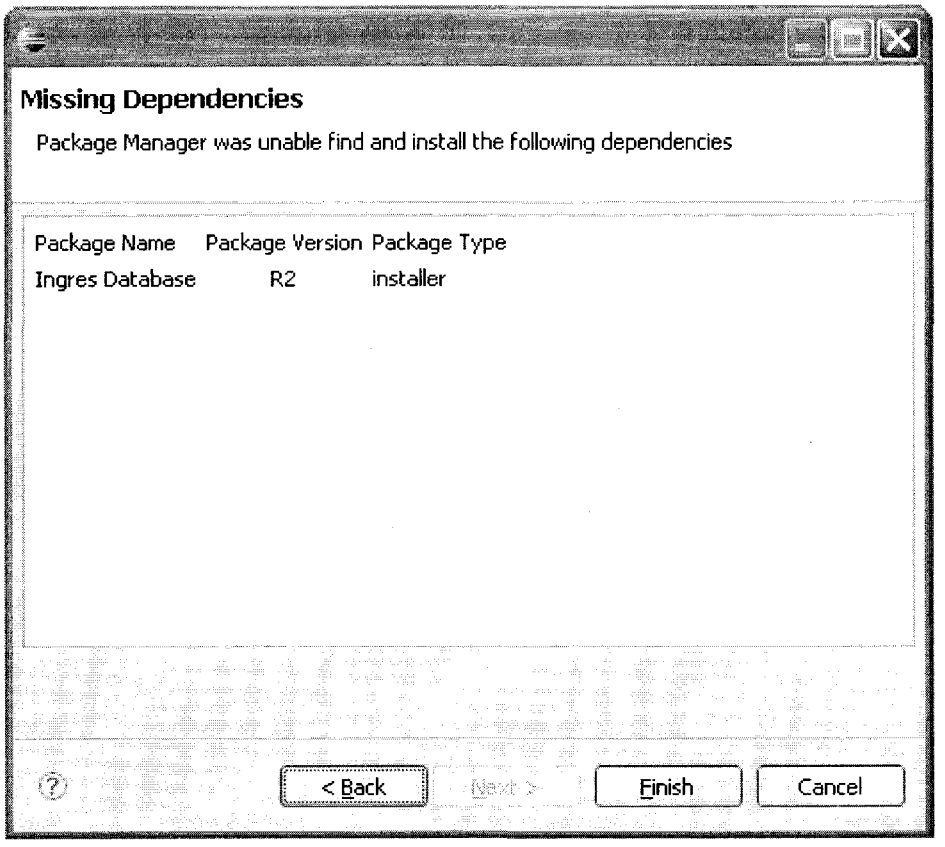
If the type is "library" then the component is downloaded and unzipped in library folder of the plug-in that called the PackageManager plug-in. This action is taken because the plug-in developer was supposed to add the libraries in the library folder of the plug-in. The developer was also required to set the classpath of the plug-in to the libraries in the library folder. So when the missing libraries are downloaded and placed in the library folder, they are automatically included in the classpath of the plug-in.

For the component type "plugin", the Eclipse plug-ins are downloaded and unzipped in the *plugins* folder of the Eclipse installation.

The *downloadPackage* function reads the download URL of the component from the downloadURL attribute of the *Package* object. The *downloadPackage* then calls the *HTTPFileDownloader* class to download the component at a particular location. The download location depends on the component type.

Components in the dependency.xml file for which no matching components are found in the Remote Repositories are added to a list by the *DownloadDependencies* class. The *DownloadDependencies* class then calls the *UnsatisfiedDependenciesWizard* class to notify these components. Figure 16 shows the wizard.

Figure 16: Unavailable Components



If the component type is "installer", then the *PackageManager* class will download them into the packages directory under the root of Eclipse installation. After downloading is complete, the component is unzipped. The *PackageManager* class then reads the installer file location from the *Package* object and executes the file using the *Runtime* class.

If the component type is "standalone", then the component is downloaded and unzipped in the package folder under the root of the Eclipse installation. It is the responsibility of the plug-in developer to extend an extension point in Eclipse and then hook it into the downloaded packages. This concept is explained more in the Validation Chapter 5.

4.2 Dynamically Refreshing the Plug-in

Once the components specified for the plug-in have been downloaded, the plug-in cannot immediately recognize the new components. This is because these components do not get loaded into the memory immediately after download. This situation is very specific to the libraries required by the plug-in. If the libraries required by the plug-in are downloaded after the plug-in has been loaded in the memory, then the libraries will not be in the classpath of the plug-in, as these libraries were not present in the classpath when the plug-in was loaded.

A method to solve the above problem is to force the restart of the Eclipse workbench. This way the plug-ins will be loaded again with the newly

downloaded libraries in the classpath. But this method requires the reboot of the workbench and the user has to save the working state and is not user friendly.

A better solution is for the PackageManager plug-in to make use of the OSGi framework to dynamically refresh the plug-in so that the newly downloaded libraries are made available without the restart of the workbench. The OSGi Framework implements a complete and dynamic component model, something that is missing in standalone Java/VM environments. Applications or components (coming in the form of bundles for deployment) can be remotely installed, started, stopped, updated and uninstalled without requiring a reboot. The code in Figure 17 shows how to dynamically refresh a plug-in.

Figure 17: Dynamic Refresh of Plug-in

```
public static void refreshPackages(Bundle[] bundles, BundleContext context) {
        ServiceReference packageAdminRef =
        context.getServiceReference(PackageAdmin.class.getName());
        PackageAdmin packageAdmin = null;
        if (packageAdminRef != null) {
                packageAdmin =
                (PackageAdmin)context.getService(packageAdminRef);
                if (packageAdmin == null)
                        return;
                }

                final boolean[] flag = new boolean[] { false };
                FrameworkListener listener = new FrameworkListener() {
                        public void frameworkEvent(FrameworkEvent event) {
                                if (event.getType()   ==FrameworkEvent.
                                PACKAGES_REFRESHED)
                                synchronized (flag) {
                                        flag[0] = true;
                                        flag.notifyAll();
                                }
                        }
                };
```

```
            context.addFrameworkListener(listener);
            packageAdmin.refreshPackages(bundles);
            synchronized (flag) {
                  while (!flag[0]) {
                        try {
                              flag.wait();
                        }
                        catch (InterruptedException e) {
                        }
                  }
            }
            context.removeFrameworkListener(listener);
            context.ungetService(packageAdminRef);

      }
```

4.3 PackageManager Manifest File

This section gives a brief description of the MANIFEST.MF file of the PackageManager plug-in. MANIFEST.MF file is present under the META-INF folder of a plug-in. The manifest file describes the content of the plug-in to the Eclipse runtime. In addition to basic plug-in information such as plug-in identifier, version, etc., this file contains:

- Dependencies section listing all the plug-ins required by the plug-in. A plug-in must list as dependencies all the plug-ins needed for its code to compile.

- Runtime section declaring the libraries where the plug-in code is packaged. At runtime, the class loader searches these libraries when loading the plug-in's classes.

Figure 18: MANIFEST.MF file for the PackageManager plug-in

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: PackageManager Plug-in
Bundle-SymbolicName: PackageManager;singleton:=true
Bundle-Version: 1.0.0
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.filesystem
Export-Package: com.packagemanager.actions;
  uses:="org.eclipse.jface.action,
   org.eclipse.jface.resource,
   org.eclipse.ui,
   org.eclipse.jface.viewers,
   org.eclipse.ui.plugin,
   org.osgi.framework",
 com.packagemanager.model,
 com.packagemanager.parser;uses:="org.w3c.dom",
 com.packagemanager.ui;
  uses:="org.eclipse.jface.wizard,
   org.eclipse.ui,
   com.packagemanager.actions,
   org.eclipse.jface.viewers,
   org.eclipse.swt.widgets,
   org.eclipse.jface.dialogs",
 com.packagemanager.utility
Bundle-Activator: com.packagemanager.actions.Activator
```

Figure 18 shows the MANIFEST.MF file for the PackageManager plug-in. The Export-Package section specifies the java packages of the plug-in that should be visible to the downstream plug-ins. This will make the Java package com.packagemanager.actions available to the plug-in that declares its dependency on the PackageManager plug-in. Through this package the dependent plug-ins can then call the *fixDependency* method of the *DependencyManager* class.

The Singleton attribute="true" specifies that there is only one instance of the PackageManager throughout the workbench.

The Bundle-SymbolicName is the human readable name by which other plug-ins can make a reference to the PackageManager plug-in.

The Require-Bundle section specifies the plug-ins that the PackageManager plug-in is dependent on. In other words, these plug-ins are prerequisites for the PackageManager to execute.

The Bundle-Activator specifies the class that is called when the plug-in is loaded and in this case it is com.packagemanager.actions.Activator.

4.4 plugin.xml for PackageManager Plug-in

The plugin.xml file defines the extension aspects of a plug-in. This file is present under the root directory of a plug-in.

The PackageManager plug-in does not extend any extension points provided by the Eclipse workbench to provide the download and install facility to other plug-ins. It only uses the interface provided by other plug-ins. However the goal of this plug-in is not just in managing the dependencies of the plug-ins but to become a complete Package Management framework for Eclipse. As a step in this direction the PackageManager plug-in provides a menu option so that all the components available through the Remote Repositories are visible. The user has the option to download these components directly from the Eclipse workbench. For the menu

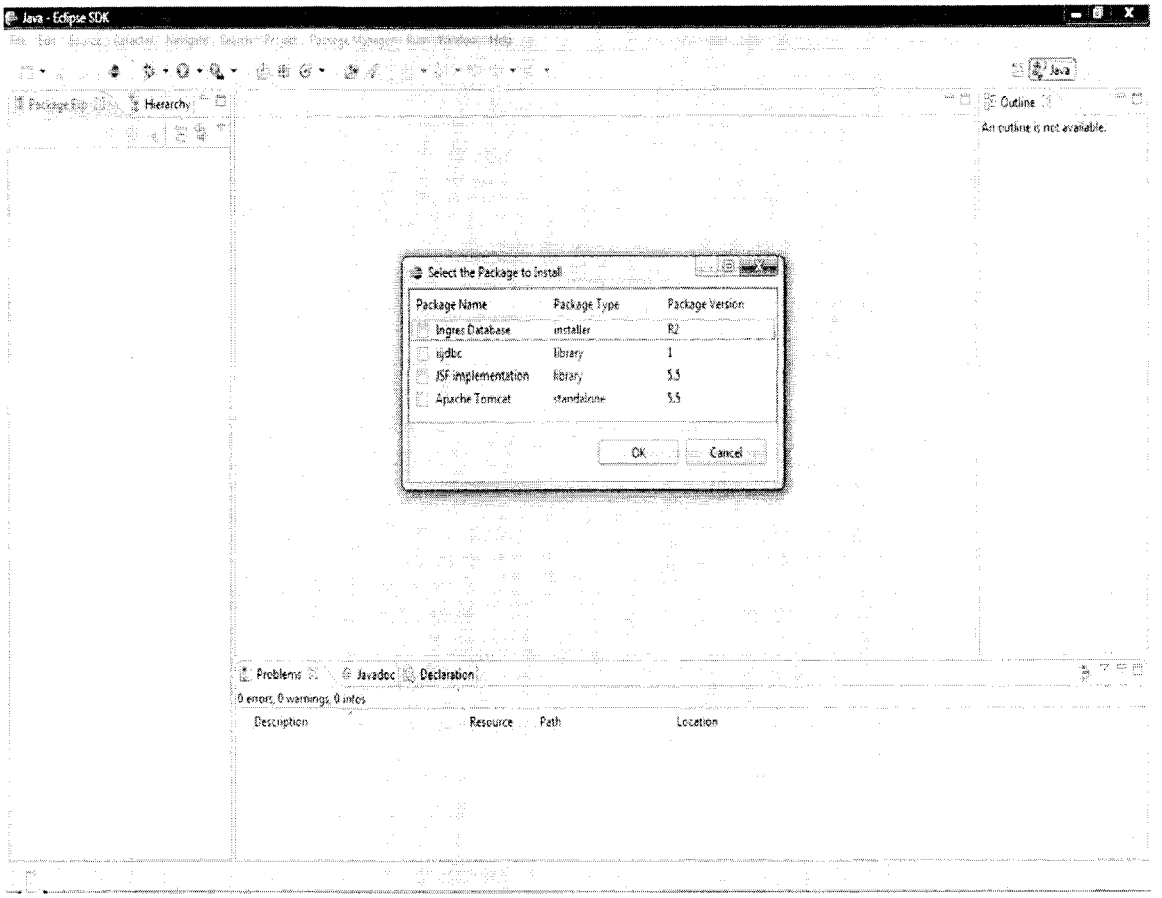
contribution the PackageManager plug-in extends the org.eclipse.ui.actionSets extension point. The plugin.xml for the actionSets contribution is shown in Figure 19.

Figure 19: plugin.xml for the PackageManager plug-in

```
<plugin>
  <extension
      point="org.eclipse.ui.actionSets">
    <actionSet
        label="Install Package"
        visible="true"
        id="PackageTest.actionSet">
      <menu
          label="Package Manager"
          id="packageMenu">
        <separator
            name="packageGroup">
        </separator>
      </menu>
      <action
          label="Install Packages"
          icon="icons/sample.gif"
          class="com.packagemanager.actions.PackageAction"
          tooltip="Package Manager"
          menubarPath="packageMenu/packageGroup"
          toolbarPath="packageGroup"
          id="com.packagemanager.actions.PackageAction">
      </action>
    </actionSet>
  </extension>
</plugin>
```

The plugin.xml specifies that a menu with the label “Package Manager” should be present in the main menubar and action “Install Packages” in the main toolbar. The action “Install Packages” brings up a dialog, shown in Figure 20 that shows the components available from all the remote repositories for direct download.

Figure 20: Install Package action Dialog



4.5 Constraints

The usage of the PackageManager plug-in has some constraints as follows:

- The plug-ins that make use of the PackageManager plug-in to download missing libraries should have a "library" folder under the root directory of the plug-in. During the compilation of the plug-in, the classpath should be set to the libraries in the "library" folder. The PackageManager plug-in will download the libraries only into the library folder of the plug-in.

- The plug-in that uses the PackageManager plug-in, cannot be distributed as a JAR file. The PackageManager is not capable of reading and writing into JAR files. Hence it requires the plug-ins to be distributed as directories.

- During the compilation of the plug-in, the PackageManager plug-in is required. This is because the plug-in has to call the fixDependency method of DependencyManager class.

- Plug-ins that use the PackageManager plug-in should have an Activator class, because the PackageManager plug-in should be called at the very start of the execution of the plug-in.

5 Validation

This chapter describes the validation performed on the framework. For the purpose of validating the framework a web-application development stack was developed (Refer to Appendix C for testing performed on this framework). This chapter is organized into four sections. Section 5.1 describes the components chosen for web-application development stack. Section 5.2 describes the technique to manage license incompatibilities in web-application development stack. Section 5.3 describes how to configure a web-application development stack using Eclipse extension point mechanism. Section 5.4 describes the execution during the install and configuration of stack components.

5.1 Developing a Web-Application Development Stack

This section describes how EPM framework is used to develop a stack with incompatible OS licenses. This stack development approach is not generic, but specific to a web-application development stack. This specialization is because the techniques and extension points required for a particular configuration are different. For example the extension point to configure JSF libraries is different from the extension point required to configure a default database driver in a DTP application.

5.1.1 Components Used in Stack

The logical components required for the proposed web-application development stack include the following:

- Development IDE
- Database
- Web Application Development Framework
- Object Relational Mapping (ORM)
- Database communication libraries
- Server Runtime

The physical components chosen for the web-application development stack include the following:

- Eclipse: Eclipse is one of the most popular OS IDE and is very widely used by web-application developers. Eclipse is distributed under the EPL license. The EPM framework has been developed on Eclipse, so Eclipse IDE WTP edition 3.3 has to be used in this stack. Other popular OS IDE looked at was NetBeans, but Eclipse was chosen because of its better PDE that enabled easier stack development.

- Ingres Database: Ingres is an OS database which is distributed under GPL version 2 license. The database driver for Ingres is also licensed under GPL which is also incompatible with EPL. So using Ingres database becomes a good example where EPM framework can be used to overcome the problem of incompatible licenses.

- JavaServer Faces (JSF): JSF technology simplifies building user interfaces for JavaServer applications. Developers of various skill levels can quickly build web applications by: assembling reusable UI components in a page; connecting these components to an application data source; and wiring client-generated events to server-side event handlers. JSF has currently two implements, one by Sun called SUN-RI and other by Apache foundation called MyFaces. In this stack, MyFaces implementation version 1.1 which is distributed under Apache License will be used. Other OS Web-application development frameworks considered for the stack were Struts and Spring. Struts is almost a dead project, while Spring was not chosen due to its complexity in configuration.

- Hibernate: Hibernate is an ORM library for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database. Hibernate solves object-relational impedance mismatch problems by replacing direct persistence-related database accesses with high-level object handling functions. Hibernate is OSS that is distributed under the LGPL license. Hibernate was chosen due its maturity and popularity.

- Apache Tomcat: The server runtime used in the stack is Apache Tomcat version 5.5. Apache Tomcat provides a container to run Java based web-applications and is one of the most widely used server runtimes for Java.

5.2 Managing Incompatible Licenses in Stack

A stack with the components identified in section 5.1.1 cannot be directly distributed due to incompatible OS licenses. Eclipse is distributed under EPL which is incompatible with LGPL (distribution license for Hibernate) and GPL (distribution license for Ingres database and Ingres database driver). This section describes how to manage the problem of incompatible licenses in this stack using EPM framework.

Applying the technique described in section 3.1 to manage the incompatible licenses, the first step is to find the licenses that are incompatible. As discussed above, the licenses of Hibernate, Ingres database driver and Ingres database are incompatible with the Eclipse license. The second step involves the deletion of these components from the stack and preparing a dependency.xml file that contains the information of these components. Figure 21 shows the dependency.xml file for these components.

Figure 21: Dependency.xml File to Download Missing Components

```
<?xml version="1.0" encoding="UTF-8"?>
<dependency xmlns="http://www.samratdhillon.com/pkmgrDepXMLSchema" >
        <package type="library">
                <package-name>iijdbc</package-name>
                <package-version>1</package-version>
        </package>
        <package type="library">
                <package-name>Hibernate</package-name>
                <package-version>3.2.6</package-version>
        </package>
        <package type="installer">
                <package-name>Ingres Database</package-name>
                <package-version>R2</package-version>
```

```
        </package>
</dependency>
```

The third step involves using the EPM framework so that the missing components can be downloaded by the Eclipse workbench. To make use of the EPM framework an Eclipse plug-in must be developed that will have the dependency.xml file described in Figure 21. The plug-in can be distributed as a part of Eclipse workbench, but it requires the plug-in to be distributed under a license compatible with EPL. In this case the plug-in is linked with a GPL component i.e. the Ingres database driver. GPL requires that if any program is linked with a GPL component, then the overall distribution license must be GPL. Hence the plug-in cannot be distributed as a part of Eclipse workbench because GPL and EPL are incompatible with each other. In the current scenario the plug-in can be distributed through the Eclipse update manager or can be manually downloaded and installed by the end user.

To develop a plug-in that makes use of the EPM framework to download missing components, the technique for plug-in development described in section 3.4 is applied. The plug-in will have the JSF, Ingres database driver and Hibernate libraries in the library folder under the root of the plug-in during the build process. Once the build process is complete, the Ingres database driver and Hibernate libraries are removed. A folder named dependency is created and the dependeny.xml file placed in that folder. The PackageManager plug-in is added to the dependency list of the plug-in being developed. From the *Activator* class of the plug-in, the fixDependencies method of the DependencyManager class will

be called. This method will trigger the download of the missing components i.e. the Hibernate JAR files and the Ingres database driver.

The next step involves setting up of Remote Repository for the components that have been removed. Figure 19 shows the Remote Repository used for the missing components in this stack. The Eclipse workbench's Local Repository should also contain a URL that points to the Remote Repository described in Figure 22.

Figure 22: Remote Repository for the Stack

```
<?xml version="1.0" encoding="UTF-8"?>
<repository
xmlns="http://www.samratdhillon.com/pkmgrRemoteRepXMLSchema">
        <package type="installer">
                <package-name>Ingres database</package-name>
                <package-version>R2</package-version>
                <download-url>
                   http://www.ingres.com/PackageRepository/installer/ingres.zip
                </download-url>
                <installer-file>/ingres/setup.exe</installer-file>
        </package>
        <package type="library">
                <package-name>iijdbc</package-name>
                <package-version>1</package-version>
                <download-url>
                   http://www.ingres.com/PackageRepository/library/iijdbc.zip
                </download-url>
                <installer-file></installer-file>
        </package>
        <package type="library">
                <package-name>Hibernate</package-name>
                <package-version>3.2.6</package-version>
                <download-url>
                http://www.ingres.com/PackageRepository/library/hibernate.zip
                </download-url>
                <installer-file></installer-file>
        </package>
</repository>
```

5.3 Configuring Stack with Eclipse Plug-ins

This section describes some of the techniques that can be used to configure a web-application development stack by using Eclipse extension point mechanism. This involves the configuration of the components that have been supplied with the plug-in as well as the components that are downloaded by the plug-in after initialization.

Configurations provided through Eclipse plug-ins are persisted through different workspaces while the configurations done manually are not persisted through different workspaces. This means that once a developer has created a configuration in a workspace manually, when the workspace is changed the developer looses the configurations of the previous workspace. This problem can be overcome locally, by using the option "Copy Settings", which will copy the settings from the previous workspace. However this method is not effective if the developer intends to distribute the configuration. This requires the creation of a plug-in that can extend particular extension points related to the configuration that the developer wants to persist over workspaces.

The setup required for a JSF web-application project include the JSF, Hibernate JAR files and the database driver libraries to be included in the classpath and buildpath so that the web-application can compile and execute. The JSF editor in Eclipse requires a reference to the JSF implementation libraries to validate JSF tags and to provide context sensitive help. The Web-application also requires defining a server runtime which can run the web-application. The Server runtime

definition is also used to compile the web-application and to validate JSP tags in a webpage.

Configuring JSF Implementation Libraries: To configure the JSF implementation libraries, a plug-in uses the "org.eclipse.jst.jsf.core.pluginProvidedJsfLibraries" extension point. Figure 23 is a sample of plugin.xml that shows how to extend the extension point "org.eclipse.jst.jsf.core.pluginProvidedJsfLibraries" so that the JSF libraries can be automatically configured.

Figure 23: plugin.xml to Provide Default JSF Libraries

```
<extension
      id="id1"
      point="org.eclipse.jst.jsf.core.pluginProvidedJsfLibraries">
    <jsfLibrary
        archiveFilesDelegate="com.ingres.stack.MyJSFImpl"
        isImplementation="true"
        label="JSF Implementation"
        name="com.ingres..jsfLibraryImpl">
    </jsfLibrary>
  </extension>
```

In Figure 23, the archiveFilesDelegate tag is used to specify the Java class that extends the *PluginProvidedJSFLibraryArchiveFilesDelegate* class. In this example the plugin.xml specified that *com.ingres.stack.MyJSFImpl* class will extend the *PluginProvidedJSFLibraryArchiveFilesDelegate* class. The class that extends the *PluginProvidedJSFLibraryArchiveFilesDelegate* class has to override getArchiveFiles method. In this method the developer can specify the physical location of the libraries.

Figure 24: Adding JSF Implementation Libraries

```
public void getArchiveFiles() {

        addArchiveFile("library/myfaces-api-1.1.5.jar");
        addArchiveFile("library/myfaces-impl-1.1.5.jar");
}
```

The isImplementation tag specifies if the libraries are the JSF implementation libraries. Figure 24 shows the code required in the implementation class to specify the location of the libraries.

Configuring Other Required Libraries: The other libraries required for a web-application project can also be specified using the same extension point. The only change required is to set the isImplementation tag false. The plug-in can then specify the class that will extend the *PluginProvidedJSFLibraryArchiveFilesDelegate* class. The class that extends *PluginProvidedJSFLibraryArchiveFilesDelegate* can then override the getArchiveFiles method to specify the physical location of the required libraries. These libraries can be any libraries that are required in the buildpath and the classpath of the web-application project.

Configuring Server Runtime: Figure 25 shows the code to create a default server runtime.

Figure 25: Configuring Server Runtime

```
public void setRuntime(){
    IRuntimeType[] runtimes= ServerCore.getRuntimeTypes();
    IRuntimeType tomcat55=null;
    for(int i=0;i<runtimes.length;i++){
```

```
            if(runtimes[i].getVendor().equals("Apache") &&
            runtimes[i].getVersion().equals("5.5"))
            tomcat55=runtimes[i];
        }

        try {
            IRuntimeWorkingCopy tomcatcopy=
            tomcat55.createRuntime("Apache Tomcat v5.5", null);
            IPath serverloc = new Path("C:\\Eclipse\\packages\\apache-tomcat-
            5.5");
            tomcatcopy.setLocation(serverloc);
            tomcatcopy.validate(null);
            tomcatcopy.save(true, null);
        }

        catch (CoreException e) {
            e.printStackTrace();
        }

}
```

This method should be executed before the web-application project has been created. This can be done by creating a web-application project facet.

Extending WTP Using Project Facets: Faceted Project Framework provides a powerful mechanism for extending the capabilities of the Web Tools Platform. Project facets are typically used as a way of adding functionality to a project. When a facet is added to the project it can perform any necessary setup actions such as copying resources, installing builders, and adding natures. Facets can also be used as markers for enabling user interface elements.

Figure 26: WTP facets to configure project

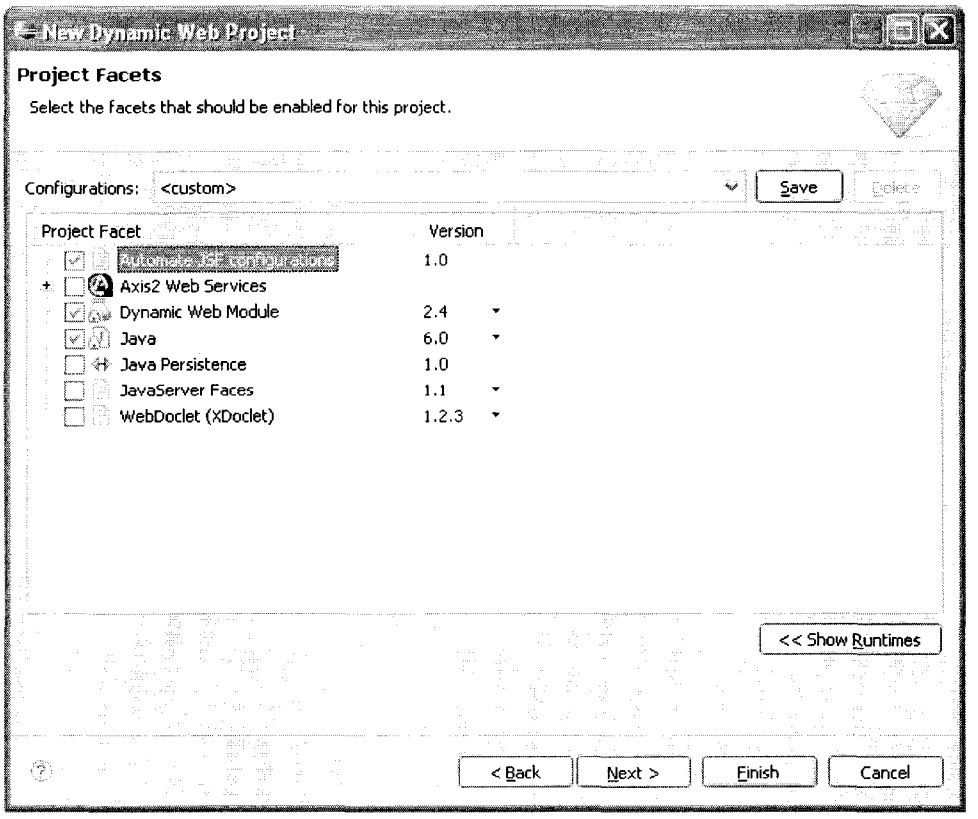


Figure 26 shows a custom WTP facet that is contributed by the plug-in. The method to create default server runtime can be placed inside the class that implements the IDelegate interface. The class that implements the IDelegate interface should override the execute method. The execute method is called before a project is created with the selected facet.

5.4 Sample Usage

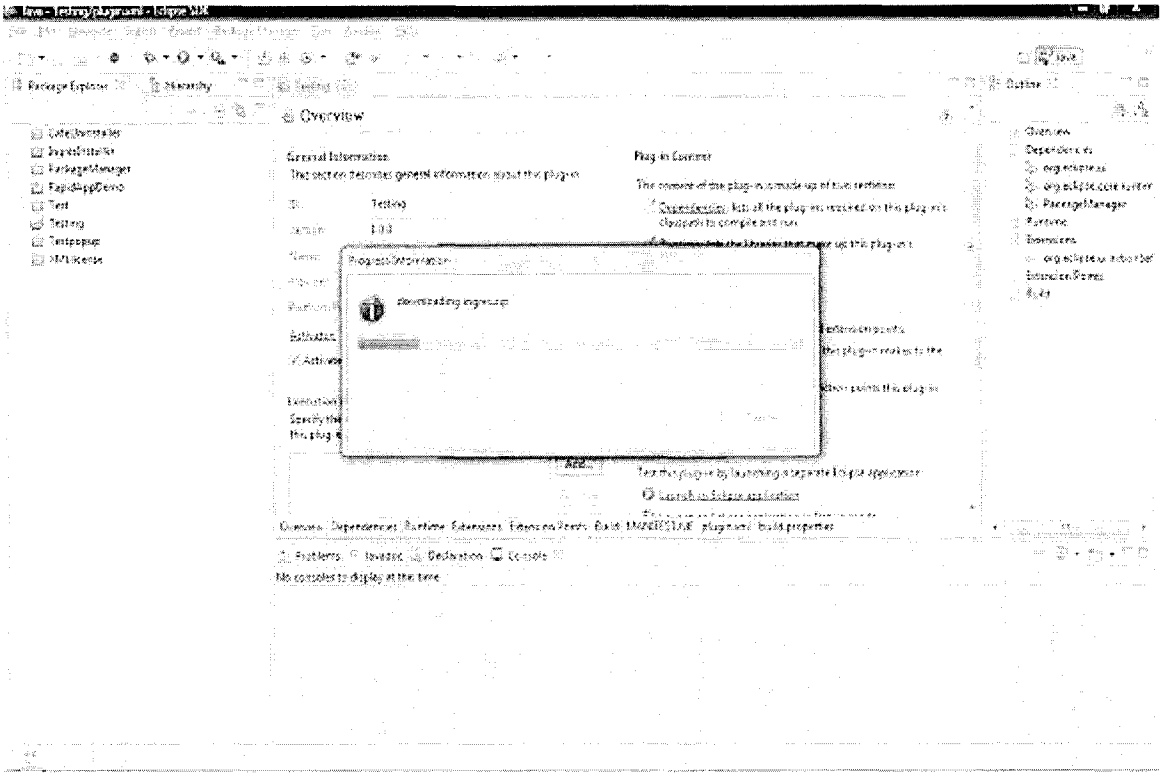
This section describes the sample execution during the install and setup of web application development stack. The components of the web application development stack have been described in section 5.1.1. When the plug-in described in section 5.2 and section 5.3 is installed and loaded for the first time,

the *Activator* class of the plug-in makes a call to the PackageManager plug-in. The PackageManager plug-in then reads the dependency.xml file (shown in Figure 21) and brings up the install wizard as shown in Figure 14. When the user clicks the finish button of the wizard the PackageManager plug-in starts downloading the components specified in the dependency.xml file (Ingres Database, Ingres Database driver and Hibernate JAR files) and brings up the download and install progress bar as shown in Figure 27.

The Hibernate JAR files and the Ingres Database driver library are downloaded and extracted into the library folder of the plug-in (developed in section 5.1.2 and 5.1.3). The Ingres Database version R2 is downloaded to the packages folder under the root directory of the Eclipse installation. After the Ingres Database has been downloaded the installer file associated with the database setup is executed by the PackageManager plug-in to install the Ingres Database.

When all the missing components of the web application development stack have been downloaded the dependency.xml file is deleted so that when the next time the PackageManager plug-in is called, it will simply return the control back to the plug-in (plug-in that configures the web application stack) rather than downloading the components in the dependency.xml file.

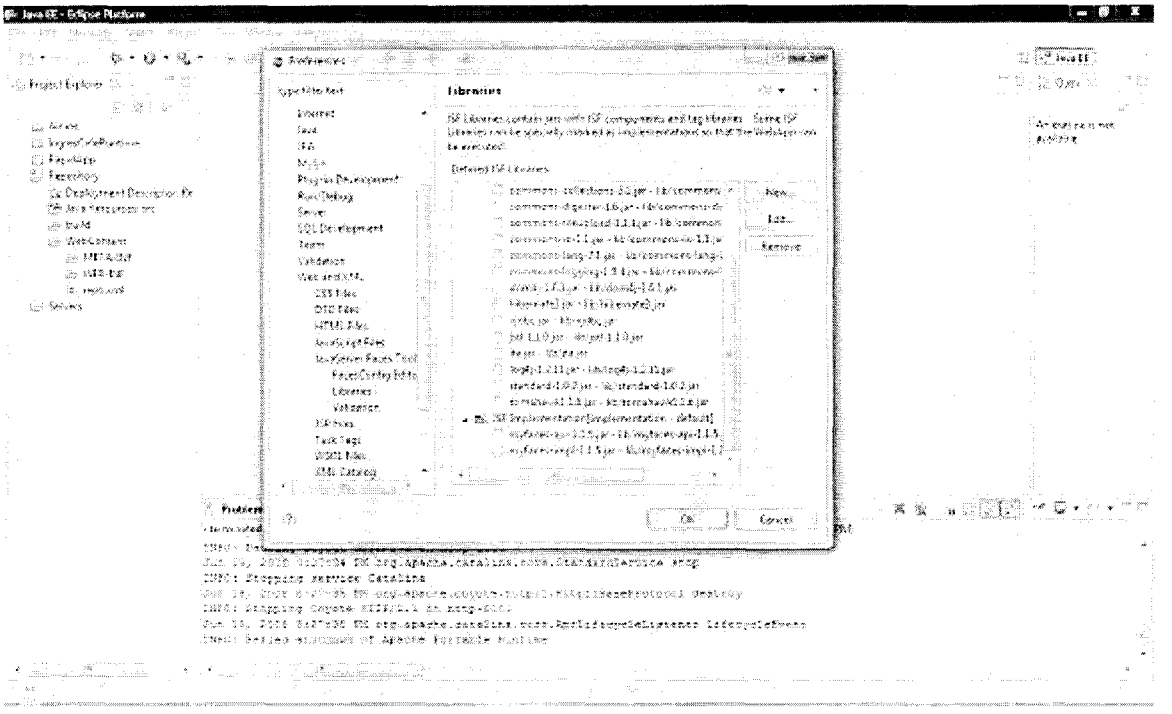
Figure 27: Download and Setup Progress Bar



After all the components have been downloaded and installed, they are automatically configured by the Eclipse extension point mechanism. Now when the user creates a new dynamic web application project and chooses the automate JSF configuration facet as shown in Figure 26, the libraries for web application development (JSF, Hibernate and Ingres Database driver JAR files) and the server runtime (Apache Tomcat 5.5) are automatically configured. The JSF web application development libraries and other common required libraries are configured by the extension point "org.eclipse.jst.jsf.core.pluginProvidedJsfLibraries". The Apache Tomcat server runtime is configured by the code shown in Figure 25. Figure 28 shows the resulting configuration of JSF and other common required libraries after

download and setup. The user is now ready to start developing web applications using JSF framework, Ingres database and Hibernate ORM.

Figure 28: Configuration of JSF and Other Libraries



5.5 Summary

This chapter described a technique to manage the problem of incompatible licenses in an Eclipse based web-application development stack. The web-application development stack contained three components that could not be distributed due to incompatible OS licenses. This chapter successfully demonstrated how to distribute such an Eclipse application stack. This chapter also described how Eclipse extension point mechanism can be leveraged to configure components. In the web-application development stack that was developed using four components i.e. JSF libraries, Hibernate libraries, Ingres

database driver and Apache Tomcat server runtime were configured automatically using the extension point mechanism. Using the extension point mechanism to configure components makes the configuration persist over different workspaces.

6 Conclusions, Limitations and Suggestion for Future Research

This chapter is organized into three sections. Section 6.1 describes the conclusions of the research. Section 6.2 describes the limitations of the research. Section 6.3 provides suggestions for future research.

6.1 Conclusions

The objective of this research was to enable the distribution and setup of components that cannot be distributed together due to incompatible licenses in application stacks. This thesis prescribed a technique to manage incompatible OS license in application stacks in section 3.1. The usage of this technique is described in section 3.4 and validated in chapter 5. Chapter 5 also explains how to manage issues related to configuration.

Solutions to distribution and configuration problems are easier solved if the software components which need to be distributed are modular with well defined interfaces.

The approach adopted in this thesis to manage the incompatibilities was successfully operationalized. This approach can be further improved by having metadata that is compatible across existing software package repositories and package managers.

The area of research undertaken in this thesis is relatively new as evidenced by the following:

- Ambiguities still remain about incompatibilities between OS licenses. This is evident from the example of OpenSSL community, where developers of OpenSSL claim it to be compatible with GPL, while FSF claims OpenSSL to be incompatible with GPL (Wheeler, 2008).

- Scant literature that discusses OS licenses incompatibility and potential solutions to manage the problem

6.2 Limitations

There are a few limitations associated with the research in this thesis.

- The system can handle only one level of dependencies of software components, it cannot handle multiple levels of dependencies.

- The dependency file is erased after all the dependencies have been downloaded, so no history is maintained.

- The solution has a few low level operational limitations such as handling system restart.

- The process to identify license incompatibilities is not automatic. It requires human intervention and hence prone to errors due to different skill sets.

- The solution developed in this thesis is specific to Eclipse and not a generic solution. The stack development approach described in Chapter 5 is also very specific to a web-application development stack.

- This technique cannot be used to manage all type of incompatibilities. For example this technique cannot be used to distribute proprietary components that link with GPL licensed components.

6.3 Suggestions for Future Research

Future research can be done to develop a solution that handles the full life-cycle including version control and uninstall for different components that get installed.

Second area can be the classification of the license incompatibilities between all the OS licenses. This research reviewed the clauses of only 6 OS licenses.

The third area can be to find a technique if GPL licensed software can be used with proprietary software. GPL states that a proprietary software can be linked with GPL software if the communication between these two is at an "arm's length". No such formal technique exists to determine the "arm's length". Such a technique can be used in conjunction with this framework to determine if GPL software used in the product leads to license incompatibilities.

The fourth area of research can be to convert the existing techniques to detect license incompatibilities (LIDESC or FOSSOLOGY libraries) into Eclipse plug-ins

and use them in combination with the EPM framework to automate packaging and distribution of Eclipse based stacks.

7 References

Clayberg, E. & Rubel, D. 2004. Eclipse: Building Commercial-Quality Plug-Ins (The Eclipse Series). Addison-Wesley Professional.

Cosmo, R. Di, Durak, B., Leroy, X., Mancinelli F. & Vouillon, J., 2006. Maintaining large software distributions: new challenges from the FOSS era. In Proceedings of the FRCSS 2006 workshop. Available at: http://gallium.inria.fr/~xleroy/publi/edos-frcss06.pdf (retrieved Nov. 20, 2007).

D'Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J. & McCarthy, P., 2004. The Java Developer's Guide to Eclipse. 2nd Edition. Addison-Wesley Professional.

Firesmith, D.G., 1994. ASTS Abbreviations and Glossary. ASTS-G01 Version 2.0. Fort Wayne, IN: Advanced Software Technology Specialists.

Gangadharan, G. R., Weiss, M., Esfandiari, B. & D'Andrea, V., 2007a. A Feature Interaction View of License Conflicts, International Conference on Feature Interactions, 25-37.

Gamma, E. & Beck, K. 2003. Contributing to Eclipse: Principles, Patterns, and Plug-Ins (The Eclipse Series). Addison-Wesley Professional.

Gangadharan, G.R., Weiss, M., D'Andrea, V. & Iannella, R., 2007b. Service License Composition and Compatibility Analysis. Fifth International Conference on Service-Oriented Computing, Vienna, Austria, 17-20.

Gomez, F.P. & Quinones, K.S., 2008, Legal Issues Concerning Composite Software. Proceedings of the Seventh International Conference on Composition-Based Software Systems.

Lüer, C., 2003. Evaluating the Eclipse Platform as a Composition Environment, 3rd International Workshop on Adoption-Centric Software Engineering , Portland, Oregon. Available at: http://www.cs.bsu.edu/homepages/chl/Eclipse.pdf (retrieved Oct. 20, 2007)

Mancinelli, F., Boender, B., Cosmo, R. di, Vouillon, J., Durak, B., Leroy, X. & Treinen, R., 2006. Managing the Complexity of Large Free and Open Source Package-Based Software Distributions. 21st IEEE International Conference on Automated Software Engineering (ASE'06). pp. 199-208.

Madanmohan, T.R. & De, Rahul., 2004. Open Source Reuse in Commercial Firms. IEEE Software. 21(6): 62-69.

Raymond, E. S. & Raymond, C.O., 2002. Licensing HOWTO. Draft Available at : http://www.catb.org/~esr/Licensing-HOWTO.html. (retrieved Oct. 20, 2007)

Reber, A., 2004. Distributed Software Installation for Free Software. [M.Sc. thesis]: Department of Electronic & Computer Engineering, Brunel University.

Rivières, J. des & Wiegand, J., 2004. Eclipse: A platform for integrating development tools. IBM Systems Journal, 43(2). Available at: https://www.research.ibm.com/journal/sj/432/desrivieres.pdf. (retrieved Oct. 20, 2007)

Rosenlaw, L., & Einschlag, M.B., 2004, Open Source Licensing Software Freedom and Intellectual Property Law. Prentice Hall.

Ruffin, M. & Ebert, C., 2004. Using Open Source Software in Product Development: A Primer. IEEE Software. 21(1): 82-86.

St. Laurent, A. M., 2004. Understanding Open Source and Free Software Licensing. O'Reilly Media, Inc.

Stallman, R., 2006. Transcript of Richard Stallman at the 2nd International GPLv3 Conference. Available at: http://fsfeurope.org/projects/gplv3/fisl-rms-transcript#licence-compatibility. (retrieved Mar. 20, 2008)

Subramanian, A. M. & Soh, P.H., 2006. Knowledge Integration and Effectiveness of Open Source Software Development Projects. The Tenth Pacific Asia Conference on Information Systems

The Debian Project. Debian policy manual. http://www.debian.org/doc/debian-policy/index.html. (retrieved Sep. 20, 2007)

Tirole, J. & Lerner, J., 2005. The Scope of Open Source Licensing. Journal of Law, Economics, and Organization. 21(1): 20-56.

Ueda, M., 2005. Licenses of Open Source Software and their Economic Values. Proceedings of the 2005 Symposium on Applications and the Internet Workshops. 381-383.

Välimäki, M. & Oksanen, V., 2002. Evaluation of Open Source Licensing Models for a Company Developing Mass Market Software. The Proceedings of International Conference on Law and Technology.

Wheeler, D.A., 2008. Make Your Open Source Software GPL-Compatiable. Or Else. Available at: http://www.dwheeler.com/essays/gpl-compatible.html. (retrieved May. 30, 2007)

Appendices

Appendix A: Dual Licensing

Dual-licensing is the practice of distributing identical software under two different licenses. This is commonly done to support OS business models. In this model, one option is a proprietary software license, which allows the possibility of creating proprietary applications derived from it, while the other option is an OS license requiring any derived works to be released under the same license. The copyright holder of the software distributes the OS version of the software at no cost, and profits by selling licenses to commercial operations looking to incorporate the software into their own business.

Such licensing allows the holder to offer customisations, early releases, generate other derivative works or grant rights to third parties to redistribute proprietary versions all the while offering everyone an OS version of the software.

Appendix B: EPL Specific Definitions

Program:

According to EPL the term Program refers to the software that is being licensed under the Eclipse Public License.

Contribution

Contributions refers to the software being distributed under EPL and also any artifacts produced by changing or making additions to the software .But Contributions do not include additions to the Program which are:

- Separate modules of software distributed in conjunction with the Program under their own license agreement
- Not derivative works of the Program.

Derivative Works

EPL is governed under U.S. law. Under the U.S. Copyright Act, a "derivative work" is defined as "...a work based upon one or more preexisting works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which a work may be recast, transformed, or adapted. A work consisting of editorial revisions, annotations, elaborations, or other modifications which, as a whole, represent an original work of authorship, is a "derivative work"." The Eclipse Foundation interprets the term "derivative work" in a way that

is consistent with the definition in the U.S. Copyright Act, as applicable to computer software.

Most of the Eclipse plug-ins are not derivative works, because they only link to the Eclipse libraries.

Appendix C: Testing

To test the functionality of the framework following test cases were executes:

Different Combinations of Four Different Types of Components

The framework was tested for all the different combinations of four different types of components, i.e. "library", "installer", "standalone" and "plug-in". The total combinations for four different types of components are fifteen. These fifteen test cases are: - {library}, {installer}, {standalone}, {plug-in}, {library, installer}, {library, standalone}, {library, plug-in}, {installer, standalone}, {installer, plug-in}, {standalone, plug-in}, {library, installer, standalone}, {installer, standalone, plug-in}, {standalone, plug-in, library}, {library, plug-in, installer}, {library, installer, standalone, plug-in}. These fifteen different combinations of components were specified in dependency.xml files to test the functionality to process each component. So fifteen different dependency.xml files were used to test the framework and its capability to download and process four different types of components.

Expected results for different types of components:

- Installer: For the component type "installer", the component should be downloaded to the packages folder under the root directory of the Eclipse installation. After the downloading is complete, the component should be unzipped and the installer file should be executed.

- Standalone: For the component type "standalone", the component should be downloaded to the packages folder under the root directory of the Eclipse installation. After the downloading is complete, the component should be unzipped.

- Plug-in: For the component type "plug-in", the component should be downloaded and unzipped into the plugins directory under the root folder of the Eclipse installation.

- Library: For the component type "library", the component should be downloaded and unzipped into the library folder of the plug-in that made call to the PackageManager plug-in.

Figure 29 shows a sample dependency file used to test all the component types.

Figure 29: Sample of dependency.xml File Used for Testing

```
<?xml version="1.0" encoding="UTF-8"?>
<dependency xmlns="http://www.samratdhillon.com/pkmgrDepXMLSchema" >
        <package type="library">
                <package-name>iijdbc</package-name>
                <package-version>1</package-version>
        </package>
        <package type="standalone">
                <package-name>Apache Tomcat</package-name>
                <package-version>5.5</package-version>
        </package>
        <package type="installer">
                <package-name>mysql</package-name>
                <package-version>5.5</package-version>
        </package>
        <package type="installer">
                <package-name>mysql</package-name>
                <package-version>5.5</package-version>
        </package>
```

```
        <package type="plug-in">
                <package-name>Ingres Cafe</package-name>
                <package-version>0.5</package-version>
        </package>
</dependency>
```

Testing Remote Repositories

To test that this framework can download and install components from Remote Repositories in correct manner, the following test cases were executed.

- Single Remote Repository and the missing components specified in the dependency.xml file are available.

- Multiple Remote Repositories and the missing components specified in the dependency.xml file are available, but not from one Remote Repository i.e. using two or more Remote Repositories.

- Multiple Remote Repositories and the missing components specified in the dependency.xml file are available, but one or more components are available from more than one Remote Repository. In such a case the framework should download and process the component from only one Remote Repository.

Testing the Download Location

- Single Remote Repository and the components specified in dependency.xml files are hosted on the same server as the Remote Repository.

- Single Remote Repositories and the components specified in the dependency.xml are not present on the same server as the Remote Repository, but on some other server.

- Single Remote Repository and the components specified in the dependency.xml are downloaded in such a way that at least one component is downloaded from the Remote Repository and at least one component is downloaded from a server other than the Remote Repository.

- Multiple Remote Repositories and the components specified in the dependency.xml are downloaded from multiple download servers.

Incomplete/Complete Dependencies

- In case all the components in the dependency.xml file are available from the Remote Repositories and are successfully downloaded, the dependency.xml file should be deleted.

- If for some components in the dependency.xml file, no matching component is available from Remote Repositories, then the dependency.xml file should be modified to remove the entry for the components which have been successfully downloaded. The components which have not been found should be notified to the user.

Testing Menu bar "Package Manager"

- Updates to the Local Repositories and the Remote Repositories should be reflected in the "Install Packages" dialog i.e. each time "Install Packages" action is called, it should reflect the latest list of components that are available for download.

- Selective processing should work even if the components are installed through the "Install Packages" action i.e. standalone components should be downloaded into the packages folder under the root of Eclipse installation, installer components should be downloaded to the packages folder and their install file should be executed, plug-in components should be downloaded to the plugins folder under the root of the Eclipse installation.

Appendix D: XML Schema

Figure 30: XML Schema for Local Repository

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
targetNamespace="http://www.samratdhillon.com/pkmgrLocalRepXMLSchema"
xmlns:tns="http://www.samratdhillon.com/pkmgrLocalRepXMLSchema"
elementFormDefault="qualified">

        <xsd:complexType name="site">
                <xsd:sequence minOccurs="1" maxOccurs="unbounded">
                        <xsd:element name="meta-url" type="xsd:anyURI"/>
                </xsd:sequence>
        </xsd:complexType>
        <xsd:element name="repository" type="tns:site"/>
</xsd:schema>
```

Figure 31: XML Schema for Dependency File

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
targetNamespace="http://www.samratdhillon.com/pkmgrDepXMLSchema"
elementFormDefault="qualified"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://www.samratdhillon.com/pkmgrDepXMLSchema">
   <xsd:complexType name="package">
        <xsd:sequence minOccurs="1" maxOccurs="unbounded">
                <xsd:element name="package-name" type="xsd:string"/>
                <xsd:element name="package-version" type="xsd:string"/>
        </xsd:sequence>
        <xsd:attribute name="type">
                <xsd:simpleType>
                        <xsd:restriction base="xsd:string">
                                <xsd:enumeration value="installer"/>
                                <xsd:enumeration value="plugin"/>
                                <xsd:enumeration value="library"/>
                                <xsd:enumeration value="standalone"/>
                        </xsd:restriction>
                </xsd:simpleType>
        </xsd:attribute>
   </xsd:complexType>
   <xsd:element name="dependency" type="tns:package"/>
</xsd:schema>
```

Figure 32: XML Schema for Remote Repository

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
targetNamespace="http://www.samratdhillon.com/pkmgrRemoteRepXMLSchem
a" elementFormDefault="qualified"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:pkmgr="http://www.samratdhillon.com/pkmgrRemoteRepXMLSchema">
   <xsd:complexType name="package">
      <xsd:sequence minOccurs="1" maxOccurs="1">
            <xsd:element name="package-name" type="xsd:string"/>
            <xsd:element name="package-version" type="xsd:string"/>
            <xsd:element name="download-url" type="xsd:anyURI"/>
            <xsd:element name="installer-file" type="xsd:string"/>
      </xsd:sequence>
      <xsd:attribute name="type" use="required">
            <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="standalone"/>
                        <xsd:enumeration value="installer"/>
                        <xsd:enumeration value="plugin"/>
                        <xsd:enumeration value="library"/>
                  </xsd:restriction>
            </xsd:simpleType>
      </xsd:attribute>
   </xsd:complexType>
   <xsd:element name="repository" type="pkmgr:package"/>
</xsd:schema>
```