Università
della
Svizzera
italiana

Facoltà
di scienze
informatiche

Bachelor Project

Proximity Alert

An Eclipse Plug-in for Software Analysis

Andrea Casarella

supervised by
Prof. Dr. Michele Lanza

Abstract

Software systems are composed of huge amounts of entities linked with each other by having different kinds of dependencies and relations between them. In this context, program comprehension plays an important role, to guide and control the development of complex software applications. We built Proximity Alert, a recommender system that assigns to every part of a software system a *"danger level"* given by static analysis and metrics extraction executed on the fly on given application.

Keywords: Object-Oriented Programming, System Analysis, Software Metrics

Acknowledgments

I would like to thank Professor Michele Lanza, my advisor and tutor who help me a lot in many situations and gives me motivations to go into deep in the beautiful world of the Computer Science. I will also thank all team supervised by the professor Lanza. In particular Robbes Romain, Mircea Lungu, Richard Wettel and Marco D'Ambros which are in my opinion the best assistants that a student could desire.

Thanks a lot to Jacopo Malnati. Class-mate, collaborator but first of all a friend that rescue me and support me during this project and sharing some of his interesting knowledges and also the framework on top of which Proximity Alert is created.

Thanks a lot to Alessandro Tettamanti, Matteo Dupraz and Cristian Caggiano, mates and friends that support me during these years and became the best adventure companions that I could find. Finally but not for last I would like also to thank my girlfriend Barbara for being so kind and patient during these years of the Bachelor degree.

Andrea Casarella

June 16, 2008

Contents

Chapter 1

Introduction and Motivation

1.1 Project Overview

In object oriented software development, a system can be considered as a collection of objects. Functionalities of the system are achieved by interactions among these objects through messages sending and receiving. Whenever, one object depends on another object to execute certain functionality, there exists a relation between those two classes. In modern object oriented system development it is recommended to build a system such that is divided in multiple layers. With this, we have objects from one layer talking to the objects of another layer.

Given an optimal abstraction of layers and appropriate relation between the entities, there are still chances that the coding process might introduce a few more vulnerability. This vulnerability is not of defective coding as such but more to do with the internal structure of the code. At this stage also object oriented metrics can be of help to identify, if we need to pay further attention to any of the code to make it more maintainable.

This is why software design and development metrics are so useful and important. They are used to ensure a better quality, maintainability and preserve from vulnerability. It is also observed that following these metrics make writing test cases easier.

Our goal is to create a recommender system as an open-source plug-in for Eclipse that assigns to every part of a software system a *"danger level"* based on a static analysis of a system. The danger level is basically represented by the Proximity Alert metric which gives also the name to our software analysis plug-in. Eclipse is an open source community whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software systems. A large community of universities, researchers and individuals are using and supporting the Eclipse framework that, therefore, has a huge audience.

We create different views based on a given project that describe and assign a Proximity Alert Metric value based on a set of metrics computed by X-Ray [Mal07] software visualization and extended by the Proximity Alert model. Proximity Alert plug-in can already be used to analyze various sized projects, offering to the user the possibility to understand, browse and evaluate any system without being forced to go trough the source code.

In this document we present and discuss our current state of research and a first implementation of the system by giving an outlook on our future work on metrics and on the Proximity Alert system.

1.2 Goal of the work

The goal of the project are in our opinion double:

- *Contributing to the Eclipse plug-in development.* Eclipse is made up by thousands of plug-ins that, together, build its development environment. Contributing to Eclipse with a plug-in is an interesting challenge, given the audience that the plug-in will have and the pleasure of providing new functionalities to such an important and very well known framework.
- *Developing an Eclipse plug-in for system analysis.*
  After understanding the Eclipse plug-in development and software metrics we build on top of our Proximity Alert plug-in. Which provide a deep analysis of the system represented by metrics and assign to each analyzed entity a Proximity Alert Metric value which can be customized as preferred directly from the user.

1.3 Structure of the Document

In the next chapters we go through software analysis concepts and techniques with theoretical and visual approaches. Moreover we focus to our case study of the software analysis and describe some related works.

We give then a description of the Eclipse Platform and its architecture, by focus then on the Proximity Alert plug-in creation and how it can provide functionalities.

After this we discuss much deeper about the Proximity Alert plug-in. There's a description of the X-Ray [Mal07] meta-model and how it is integrated in the Proximity Alert model analysis. We focus then on the software metrics and their applicability, followed by the description of the metrics defined by the Proximity Alert system and a brief description on the views provided by the system.

To better understand the Proximity Alert functioning and usage we validate it on itself and, then, draw some conclusion about the reached goals and the future work.

Chapter 2

Software Analysis

2.1 Software Complexity Analysis

Object oriented construction and design are deceptive words, because they make people think that software can be constructed as buildings or designed as cars. The fact is that a software system is complex as any other engineering artifact, think for example of what there is behind the construction of a building. Moreover, a modern software system is written by many people concurrently, leading to communication issues, compatibility issues and above all complexity issues. In addition systems are mutable and grows as plants with many interrelated parts that depend on each other, that die, that change, that are bugged and must be fixed (introducing new bugs), etc.

Software analysis can make a proactive contribution to improve the quality and reliability of a software system during all phases of the software life cycle, including the preliminary design, detailed design, implementation, test, and maintenance phases. Although the impact of an analysis is greater during the latter stages of the software life cycle, its contribution potential is maximized if it is begun early in the life cycle development of a system.

Finding an appropriate design of the system is important. Indeed it may help people understand the system and ease future changes. [LM06]

2.1.1 Static and Dynamic Analysis

The analysis of a system cab be done by using two main techniques:

- *Static analysis*: dealing only with the structure of the system, and does not require running a piece of software to begin to implement a metrics program.

- *Dynamic analysis*: measuring the efficiency and effectiveness of software testing by monitoring the software system during its execution phase. All the data collected during the execution is used in the evaluation of the thoroughness of the testing, to determine if there is adequate utilization of the testing resources, and to prioritize the test case distributions.

2.1.2 Utilization of metrics

Metrics are sets of various measures which reflect properties and aspects of a software program, either at component or system level. Metrics can be used to make quality evaluations of software, to monitor the development growth of software systems, and to estimate and predict the reliability of software.

Complex software often leads to high development costs, high maintenance costs, and decreased system reliability. By monitoring the complexity of a software program during design and development, changes can be made to reduce complexity and ensure system to remain modifiable and open to future extensions. [LD02]

2.2 Related Works & Tools

2.2.1 Law of Demeter

The Law of Demeter is a law developed during the design and implementation of the Demeter system which provide a high-level interface to class-based, object-oriented systems.

There are kinds of style rules for Object-Oriented design and programming: rules that apply to the structure of classes and rules that apply to how methods are written. The Law of Demeter focus on style rules that restrict how methods are written for a set of class definition. It restricts the message-sending structure of methods, informally the law says that each method can send messages to only a limited set of objects. [KJM89]

Goal

The goal of the Law of Demeter is to organize and reduce dependencies between classes by promoting maintainability and comprehensibility. The purpose is to guide systems to be well behaved or well formed in a sense that follows a certain style that lets them modified easily, minimizing changes required elsewhere in the programs. Easy modification in system is one important criterion that characterize a good Object-Oriented programming style.

By following the Law of Demeter programmers have also to consider other known style rules such as avoid code duplication, minimizing the number of arguments, and minimizing number of methods. This law does not restrict what a programmer can solve but restricts how he solves it.

Forms

The Law of Demeter has two main forms:

- **Class form**
  Every class is a potential supplier of any methods. However it should be limit to a set of preferred classes. In order to define such classes they introduce the concept of the acquaintance class which is a method's supplier class whose methods are called in the method. Acquaintance classes are created in order to satisfy stability, efficiency and object construction.

  - *Minimization version* : allows additional dependencies between classes but asks for minimization and acquaintance classes.
  - *Strict version*: restricts dependencies between classes

- **Object form**
  The Object form of the law says that all method may have preferred supplier objects, which are similar to preferred supplier classes. It helps to guide and serves as an additional guide for the class form.

Principles

Some of the principle covered by the Law of Demeter are listed below:

- **Coupling control**

  It is a well-known principle of software design to have minimal coupling between abstractions. The Law of Demeter effectively reduces the methods you can call inside a given method, therefore it limits the coupling of methods for the Uses relation, which is a call or a return link. It facilitates reusability of methods and raise the software's abstraction level.

- **Information hiding**

  The law enforces the structure hiding. Generally prevents a method directly retrieving a subpart of an object that lies deep in that object's hierarchy. In some Object-Oriented systems, the user can protect some of the instance variables or methods by declaring them private. This complement the law by improving modularity.

- **Information restriction**

  The law restricts the use of messages sending and complement he Information hiding. Instead of hiding certain methods you make them public but you restrict their use.

The motivation behind the Law of Demeter is to ensure that the software is modular as possible. The law reduces the occurrence of nested message sending and simplifies the methods. It has many implication for widely known software engineering principles.

Applicability of Law of Demeter

The style of modular programming encouraged by the Law of Demeter leads naturally to code that is easier to understand and maintain. The law let's redesign classes while leaving more of existing software intact. Actually by applying the law is reduced the effects of local changes to a software system can reduce many further complication in software maintenance.

The Law of Demeter leads to the development of good software when used with other well-known style rules.

2.2.2 JHawk

"Virtual Machinery's JHawk product generates a number of vital metrics relating to your Java code. Using these metric is possible improve the performance, reusability, maintainability and overall quality of Java code. Metrics are provided at overall system, package, class and method levels and separate views are available for each. Output can be saved to HTML, XML or CSV formats. A command line interface allows to integrate JHawk into the build process. The standard and command line versions are compatible with any Java environment based on JDK 1.4.2 and above. Source code up to JDK 1.5.0 can be analysed. Applications are provided for both Eclipse and file based environments."[1]

Package Level

The view at this level shows the numbers of Classes, Methods and Statements in the package. It also shows the Total and Average (per method) Cyclomatic complexity, the Total Halstead Effort and the two forms of the maintainability index. In addition there are five other metrics:

- **Instability**: This is an estimate of how susceptible the package is to change. It's values will always lie in the range 0 (not susceptible to change) to 1 (very susceptible to change). Values closer to zero are preferable. It is calculated by dividing $FanOut/(FanIn + FanOut)$ for the package.
- **FanIn or Afferent Coupling (Ca)**: This is the number of packages that contain references to this pacakge. Only packages within the System are included in the calculation.
- **FanOut or Efferent Coupling (Ce)**: This is the number of packages that this package refers to. Only packages within the System are included in the calculation.
- **Abstractness**: This indicates how abstract the package is. It is calculated as the ratio of abstract and interface classes in the package to the total number of all classes in the package.
- **Distance**: This measures the balance of a particular package between abstraction and instability.

Class Level

The view at this level shows not only metrics which measures aspects of the class but also at metrics which give information on the interaction between them in the whole system.

The following list shows some of the metrics defined for the class by JHawk:

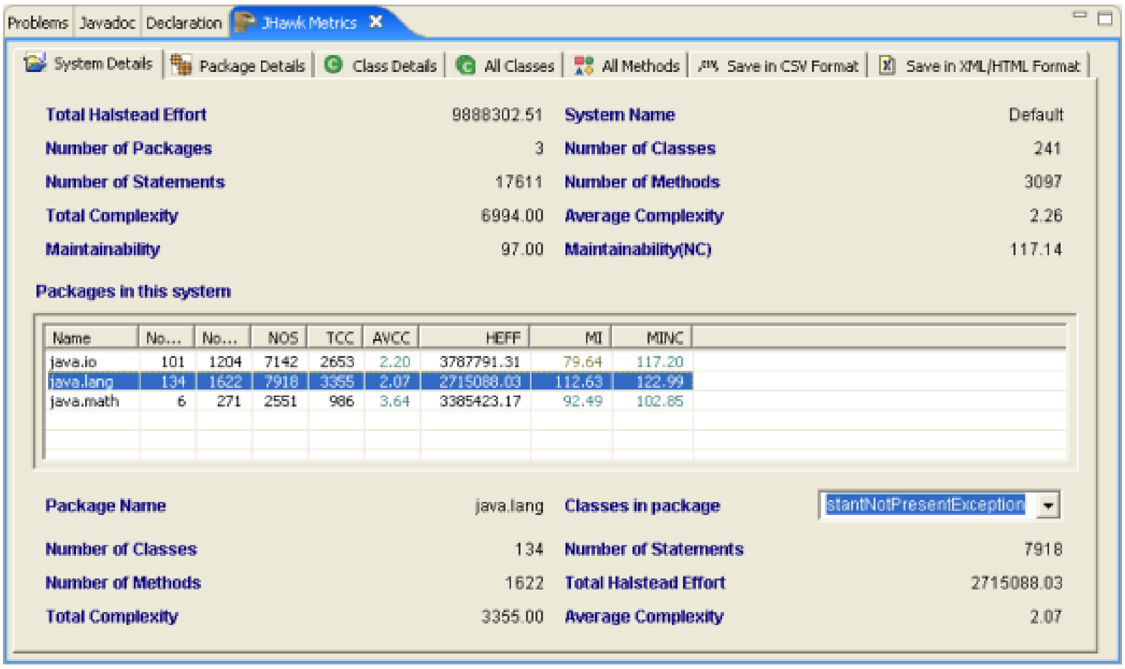
- **LCOM**: The Lack of cohesion of methods measures the correlation between the methods and the local instance variables of a class. High cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity.
- **UWCS**: The Unweighted Class Size is calculated from the number of methods plus the number of attributes of a class. Smaller class sizes usually indicate a better designed system reflecting better distributed responsibilities.

[1] http://www.virtualmachinery.com/jhawkprod.htm

- **RFC**: Response For Class measures the complexity of the class in terms of method calls. It is calculated by adding the number of declared methods in the class plus the number of distinct method calls made by the methods in the class.
- **MPC**: Message Passing Coupling measures the numbers of messages passing among objects of the class. A larger number indicates increased coupling between this class and other classes in the system.
- **CBO**: Coupling Between Objects is the total of the number of classes that a class referenced plus the number of classes that referenced the class.
- **FanIn or Afferent Coupling (Ca)**: Is the number of other classes that reference a class.
- **FanOut or Efferent Coupling (Ce)**: Is the number of other classes referenced by a class.
- **Reuse Ratio**: Is number of superclasses above this class in the class hierarchy divided by the total number of classes in the class hierarchy.

JHawk System View

Once a project, a package or a file is chosen for the analysis the System view appear populated with all of the metrics calculated at a system level. Figure 2.1 shows the JHawk System view containing a list of the packages analyzed.



Total Halstead Effort	9888302.51	System Name	Default
Number of Packages	3	Number of Classes	241
Number of Statements	17611	Number of Methods	3097
Total Complexity	6994.00	Average Complexity	2.26
Maintainability	97.00	Maintainability(NC)	117.14

Packages in this system

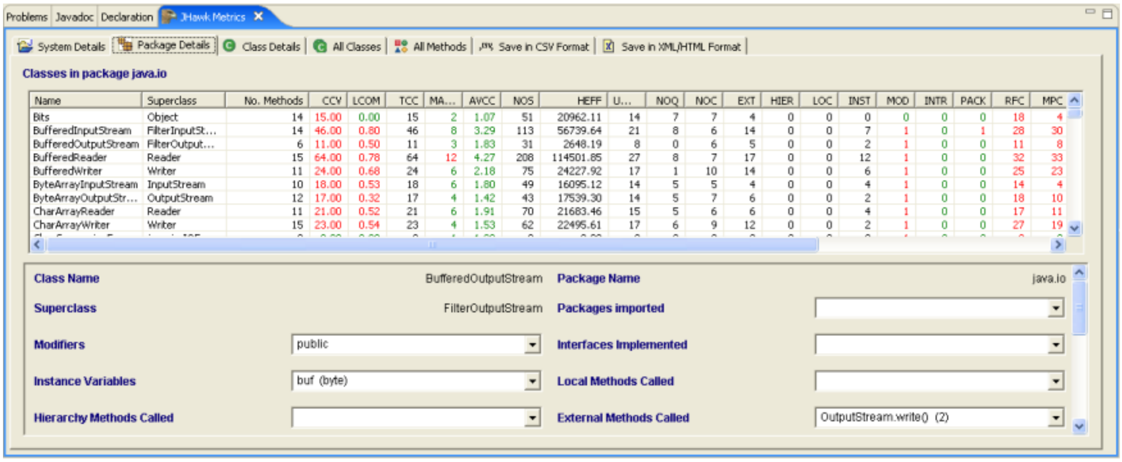
Name	No...	No...	NOS	TCC	AVCC	HEFF	MI	MINC
java.io	101	1204	7142	2653	2.20	3787791.31	79.64	117.20
java.lang	134	1622	7918	3355	2.07	2715088.03	112.63	122.99
java.math	6	271	2551	986	3.64	3385423.17	92.49	102.85

Package Name	java.lang	Classes in package	stantNotPresentException
Number of Classes	134	Number of Statements	7918
Number of Methods	1622	Total Halstead Effort	2715088.03
Total Complexity	3355.00	Average Complexity	2.07

Figure 2.1: **JHawk** System view containing a list of the packages analyzed.

JHawk Package View

When is selected a particular package on the System view then the Package view appear populated with a list of all of the classes in that package and the metrics for each class. Figure 2.2 shows the JHawk Package view containing a list of the classes analyzed of a given project.



Classes in package java.io

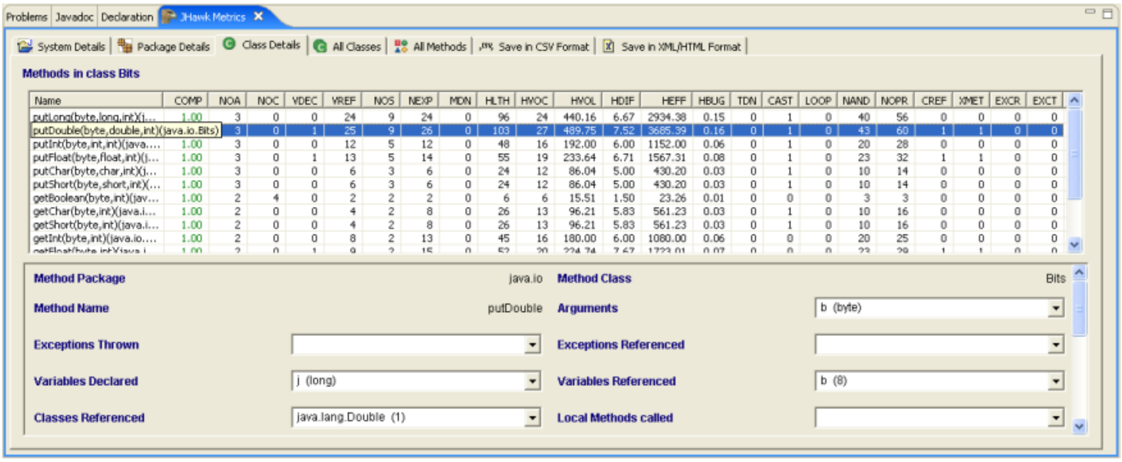
Name	Superclass	No. Methods	CCV	LCOM	TCC	MA...	AVCC	NOS	HEFF	U...	NOQ	NOC	EXT	HIER	LOC	INST	MOD	INTR	PACK	RFC	MPC
Bits	Object	14	15.00	0.00	15	2	1.07	51	20962.11	14	7	7	4	0	0	0	0	0	0	18	4
BufferedInputStream	FilterInputSt...	14	46.00	0.80	46	8	3.29	113	56739.64	21	8	6	14	0	0	7	1	0	1	28	30
BufferedOutputStream	FilterOutput...	6	11.00	0.50	11	3	1.83	31	2648.19	8	0	6	5	0	0	2	1	0	0	11	8
BufferedReader	Reader	15	64.00	0.78	64	12	4.27	208	114501.85	27	8	7	17	0	0	12	1	0	0	32	33
BufferedWriter	Writer	11	24.00	0.68	24	6	2.18	75	24227.92	17	1	10	14	0	0	6	1	0	0	25	23
ByteArrayInputStream	InputStream	10	18.00	0.53	18	6	1.80	49	16095.12	14	5	5	4	0	0	4	1	0	0	14	4
ByteArrayOutputStr...	OutputStream	12	17.00	0.32	17	4	1.42	43	17539.30	14	5	7	6	0	0	2	1	0	0	18	10
CharArrayReader	Reader	11	21.00	0.52	21	6	1.91	70	21683.46	15	5	6	6	0	0	4	1	0	0	17	11
CharArrayWriter	Writer	15	23.00	0.54	23	4	1.53	62	22495.61	17	6	9	12	0	0	2	1	0	0	27	19

Class Name BufferedOutputStream
Package Name java.io
Superclass FilterOutputStream
Packages imported
Modifiers public
Interfaces Implemented
Instance Variables buf (byte)
Local Methods Called
Hierarchy Methods Called
External Methods Called OutputStream.write() (2)

Figure 2.2: **JHawk** Package view containing a list of the classes analyzed of a particular package.

JHawk Class View

The Class view shows a list of all of the classes that you have chosen to analyze. The bottom panel shows all the metrics collected for the currently selected class. Figure 2.2 shows the JHawk Class view containing a list of the classes analyzed of a given project.



Methods in class Bits

Name	COMP	NOA	NOC	VDEC	VREF	NOS	NEXP	MDN	HLTH	HVOC	HVOL	HDIF	HEFF	HBUG	TDN	CAST	LOOP	NAND	NOPR	CREF	XMET	EXCR	EXCT
putLong(byte,long,int)(j...	1.00	3	0	0	24	9	24	0	96	24	440.16	6.67	2934.38	0.15	0	1	0	40	56	0	0	0	0
putDouble(byte,double,int)(java.io.Bits)		3	0	1	25	9	26	0	103	27	489.75	7.52	3685.39	0.16	0	1	0	43	60	1	1	0	0
putInt(byte,int,int)(java....	1.00	3	0	0	12	5	12	0	48	16	192.00	6.00	1152.00	0.06	0	1	0	20	28	0	0	0	0
putFloat(byte,float,int)(j...	1.00	3	0	1	13	5	14	0	55	19	233.64	6.71	1567.31	0.08	0	1	0	23	32	1	1	0	0
putChar(byte,char,int)(j...	1.00	3	0	0	6	3	6	0	24	12	86.04	5.00	430.20	0.03	0	1	0	10	14	0	0	0	0
putShort(byte,short,int)(...	1.00	3	0	0	6	3	6	0	24	12	86.04	5.00	430.20	0.03	0	1	0	10	14	0	0	0	0
getBoolean(byte,int)(jav...	1.00	2	4	0	2	2	2	0	6	6	15.51	1.50	23.26	0.01	0	0	0	3	3	0	0	0	0
getChar(byte,int)(java.i...	1.00	2	0	0	4	2	8	0	26	13	96.21	5.83	561.23	0.03	0	1	0	10	16	0	0	0	0
getShort(byte,int)(java.i...	1.00	2	0	0	4	2	8	0	26	13	96.21	5.83	561.23	0.03	0	1	0	10	16	0	0	0	0
getInt(byte,int)(java.io....	1.00	2	0	0	8	2	13	0	45	16	180.00	6.00	1080.00	0.06	0	0	0	20	25	0	0	0	0

Method Package java.io
Method Class Bits
Method Name putDouble
Arguments b (byte)
Exceptions Thrown
Exceptions Referenced
Variables Declared j (long)
Variables Referenced b (8)
Classes Referenced java.lang.Double (1)
Local Methods called

Figure 2.3: **JHawk** Class view a list of the classes analyzed of a given project.

Chapter 3

Eclipse

3.1 Foundation and Community

Eclipse began as an IBM Canada project and It was developed by OTI (Object Technology International) as a replacement for VisualAge, which itself had been developed by OTI. In November 2001, a consortium was formed to further the development of Eclipse as open source. In 2003, the Eclipse Foundation was created. Eclipse 3.3 (released on 2007) selected the OSGi Service Platform specifications as the runtime architecture.

Eclipse is also an open source community whose projects are focused on building an extensible development platform, runtimes and application frameworks for building, deploying and managing software across the entire software lifecycle.

The Eclipse open source community is composed of more than a thousand open source projects. These projects can be conceptually organized into seven different types or categories:

- 1. Enterprise Development
- 2. Embedded and Device Development
- 3. Rich Client Platform
- 4. Rich Internet Applications
- 5. Application Frameworks
- 6. Application Lifecycle Management (ALM)
- 7. Service Oriented Architecture (SOA)

The Eclipse community is also supported by a large and vibrant ecosystem of major IT solution providers, innovative start-ups, universities and research institutions and individuals that extend, support and complement the Eclipse Platform.

"The exciting thing about Eclipse is many people are using it in ways that we could never imagine. The common thread is that they are building innovative, industrial strength software and want to use great tools, frameworks and runtimes to make their job easier."[1]

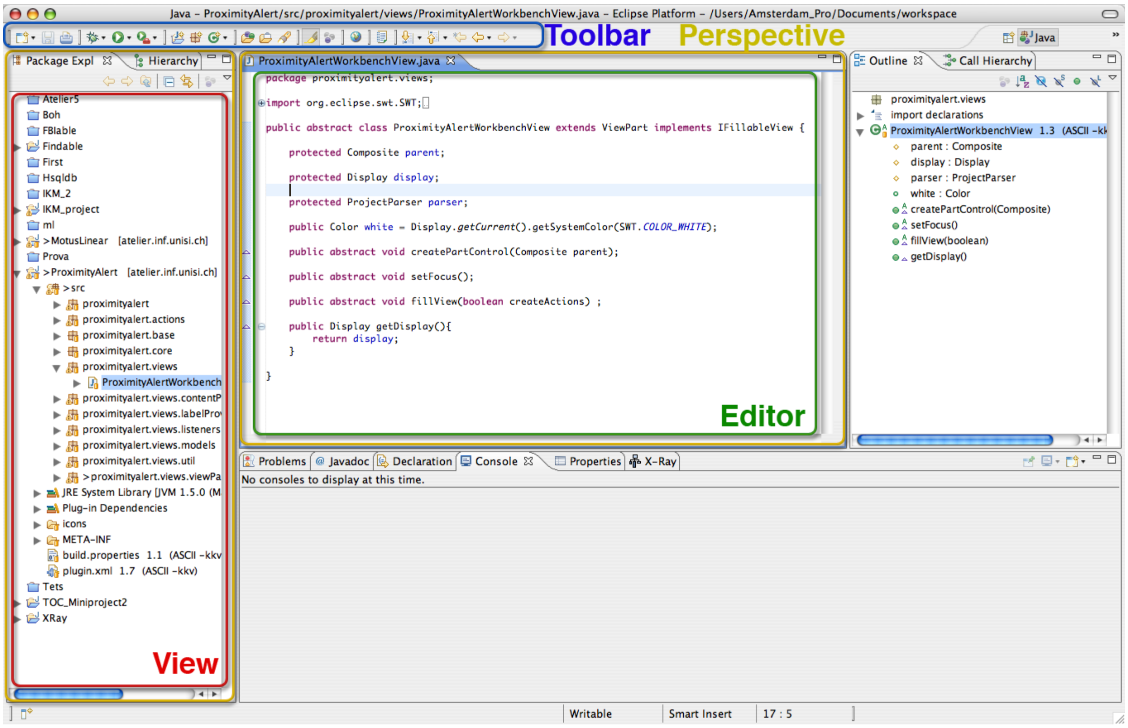
[1] http://www.eclipse.org/org/

3.2 Platform Overview

Eclipse is an integrated development environment (IDE) written primarily in Java. In its default form it is meant for Java developers, consisting of the Java Development Tools (JDT). Users can extend its capabilities by installing plug-ins written for the Eclipse software framework, such as development toolkits for other programming languages, and can write and contribute their own plug-in modules.

”Eclipse can be used to create many different kinds of content - Java files, Web content, graphics, video. These objects are stored as regular files within the Eclipse workspace. The workspace consists of one or more top level projects. Each project contains a collection of folders and files. These objects are known as resources.[2]

Figure 3.1 shows a screenshot of the main workbench user interface, with its various component within it. The Eclipse Workbench UI is a collection of windows. Each window contains a menu bar (layout OS dependent), a toolbar, a shortcut bar and one or more perspectives.


Java – ProximityAlert/src/proximityalert/views/ProximityAlertWorkbenchView.java – Eclipse Platform – /Users/Amsterdam_Pro/Documents/workspace
Toolbar
Perspective
Java
Package Expl
Hierarchy
ProximityAlertWorkbenchView.java
Outline
Call Hierarchy
View
Editor
Problems
Javadoc
Declaration
Console
Properties
X-Ray
Writable
Smart Insert
17 : 5

```
package proximityalert.views;

import org.eclipse.swt.SWT;

public abstract class ProximityAlertWorkbenchView extends ViewPart implements IFillableView {

    protected Composite parent;

    protected Display display;

    protected ProjectParser parser;

    public Color white = Display.getCurrent().getSystemColor(SWT.COLOR_WHITE);

    public abstract void createPartControl(Composite parent);

    public abstract void setFocus();

    public abstract void fillView(boolean createActions) ;

    public Display getDisplay(){
        return display;
    }

}
```

No consoles to display at this time.

Figure 3.1: **Eclipse Workbench UI** *is a collection of windows. Each window contains a menu bar (layout OS dependent), a toolbar, a shortcut bar and one or more perspectives.*

[2] http://www.eclipse.org/org/

3.2.1 Platform Architecture

The basis for Eclipse is the Rich Client Platform (RCP). The following components constitute the RCP:

- **Equinox OSGi** - a standard bundling framework
- **Core** platform - boot Eclipse, run plug-ins
- The **Standard Widget Toolkit** (SWT) - a portable widget toolkit
- **JFace** - viewer classes to bring model view controller programming to SWT, file buffers, text handling, text editors
- The Eclipse **Workbench** - views, editors, perspectives, wizards

Eclipse's widgets are implemented by a widget toolkit for Java called SWT, unlike most Java applications, which use the Java standard Abstract Window Toolkit (AWT) or Swing. Eclipse's user interface also leverages an intermediate GUI layer called JFace, which simplifies the construction of applications based on SWT.

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 via well-defined API interfaces, classes, and methods. The Platform also provides useful building blocks and frameworks that facilitate developing new tools. [CR06]

The Figure 3.2 shows the major components, and APIs of the Eclipse Platform.

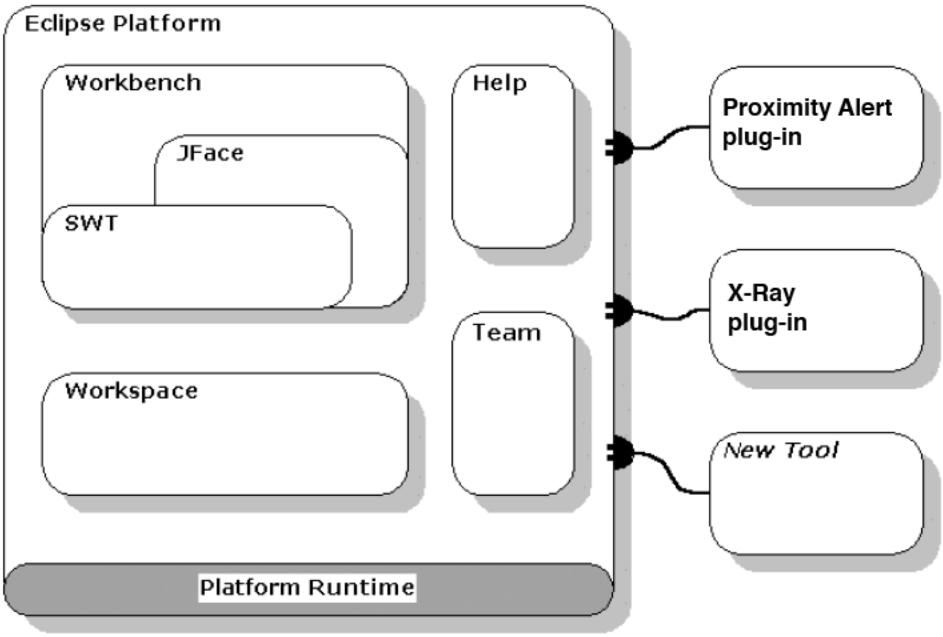


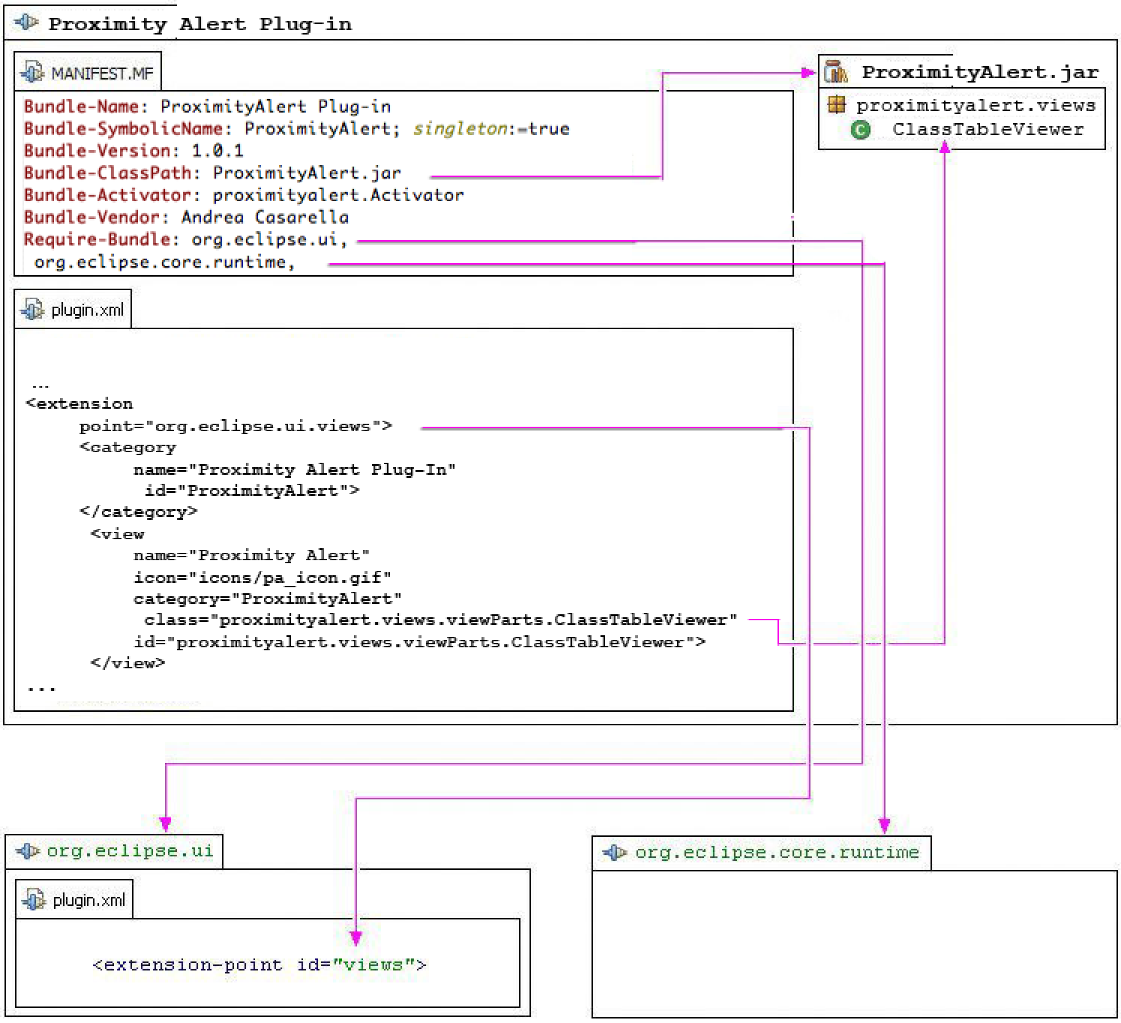
Figure 3.2: **Eclipse Platform** architecture showing its major components.

3.2.2 Plug-in structure

The behavior of every plug-in is in code, yet the dependencies and services of a plug-in are declared in the MANIFEST.MF and plugin.xml files. This structure facilitates lazy-loading of plug-in code on an as-needed basis, thus reducing both the startup time and the memory footprint of Eclipse.

On startup, the plug-in loader scans the MANIFEST.MF and plugin.xml files for each plug-in and then builds a structure containing this information. This structure takes up some memory, but it allows the loader to find a required plug-in much more quickly, and it takes up a lot less space than loading all the code from all the plug-ins all the time. [CR06]

Figure 3.3 shows the Proximity Alert plug-in declaration in the manifest with lines highlighting how the plug-in manifest references various plug-in artifacts.



```
Bundle-Name: ProximityAlert Plug-in
Bundle-SymbolicName: ProximityAlert; singleton:=true
Bundle-Version: 1.0.1
Bundle-ClassPath: ProximityAlert.jar
Bundle-Activator: proximityalert.Activator
Bundle-Vendor: Andrea Casarella
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime,
```

```
 ...
<extension
     point="org.eclipse.ui.views">
     <category
          name="Proximity Alert Plug-In"
           id="ProximityAlert">
     </category>
      <view
          name="Proximity Alert"
          icon="icons/pa_icon.gif"
          category="ProximityAlert"
           class="proximityalert.views.viewParts.ClassTableViewer"
          id="proximityalert.views.viewParts.ClassTableViewer">
      </view>
...
```

```
<extension-point id="views">
```

Figure 3.3: **Proximity Alert plug-in** declaration plug-in manifest with lines highlighting how the plug-in manifest references various plug-in artifacts

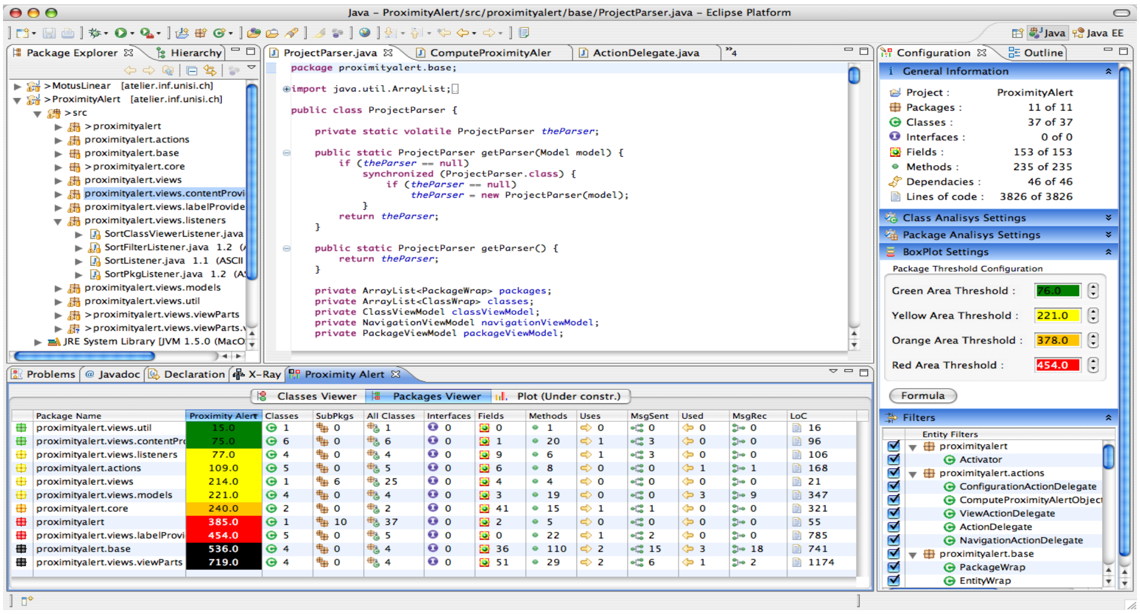
Chapter 4

Proximity Alert

4.1 The Idea

The basic idea is to create a recommender system as an open-source plug-in for Eclipse that assigns to every part of a software system a ”danger level” based on static analysis that is executed on the fly. The plug-in constantly notifies the developer about the danger level in a non-intrusive way. Using X-Ray [Mal07] plug-in meta-model it gets all entity metrics and information provided by the core of the plug-in and extend them in order to provide a larger set of metrics analysis. It allows also a fully customization by the users on *”danger level”* definition and colors depiction.

Figure 4.1 shows the Proximity Alert Workbench view on Eclipse running the analysis on itself. The view displayed is the *Package Viewer* table with the *Configuration View* on the right.



Java – ProximityAlert/src/proximityalert/base/ProjectParser.java – Eclipse Platform
Java
Java EE
Package Explorer
Hierarchy
>MotusLinear [atelier.inf.unisi.ch]
>ProximityAlert [atelier.inf.unisi.ch]
>src
>proximityalert
proximityalert.actions
proximityalert.base
>proximityalert.core
proximityalert.views
proximityalert.views.contentProvi
proximityalert.views.labelProvide
proximityalert.views.listeners
SortClassViewerListener.java
SortFilterListener.java 1.2
SortListener.java 1.1 (ASCII
SortPkgListener.java 1.2 (A
proximityalert.views.models
proximityalert.views.util
>proximityalert.views.viewParts
>proximityalert.views.viewParts.
JRE System Library [JVM 1.5.0 (MacO
ProjectParser.java
ComputeProximityAler
ActionDelegate.java
Configuration
Outline
Problems
Javadoc
Declaration
X-Ray
Proximity Alert
Classes Viewer
Packages Viewer
Plot (Under constr.)
Formula

```
package proximityalert.base;

import java.util.ArrayList;

public class ProjectParser {

    private static volatile ProjectParser theParser;

    public static ProjectParser getParser(Model model) {
        if (theParser == null)
            synchronized (ProjectParser.class) {
                if (theParser == null)
                    theParser = new ProjectParser(model);
            }
        return theParser;
    }

    public static ProjectParser getParser() {
        return theParser;
    }

    private ArrayList<PackageWrap> packages;
    private ArrayList<ClassWrap> classes;
    private ClassViewModel classViewModel;
    private NavigationViewModel navigationViewModel;
    private PackageViewModel packageViewModel;
```

Package Name	Proximity Alert	Classes	SubPkgs	All Classes	Interfaces	Fields	Methods	Uses	MsgSent	Used	MsgRec	LoC
proximityalert.views.util	15.0	1	0	1	0	0	1	0	0	0	0	16
proximityalert.views.contentPr	75.0	6	0	6	0	1	20	1	3	0	0	96
proximityalert.views.listeners	77.0	4	0	4	0	9	6	1	3	0	0	106
proximityalert.actions	109.0	5	0	5	0	6	8	0	0	1	1	168
proximityalert.views	214.0	1	6	25	0	4	4	0	0	0	0	21
proximityalert.views.models	221.0	4	0	4	0	3	19	0	0	3	9	347
proximityalert.core	240.0	2	0	2	0	41	15	1	1	0	0	321
proximityalert	385.0	1	10	37	0	2	5	0	0	0	0	55
proximityalert.views.labelProvi	454.0	5	0	5	0	0	22	1	2	0	0	785
proximityalert.base	536.0	4	0	4	0	36	110	2	15	3	18	741
proximityalert.views.viewParts	719.0	4	0	4	0	51	29	2	6	1	2	1174

General Information

Project :	ProximityAlert
Packages :	11 of 11
Classes :	37 of 37
Interfaces :	0 of 0
Fields :	153 of 153
Methods :	235 of 235
Dependacies :	46 of 46
Lines of code :	3826 of 3826

Class Analisys Settings

Package Analisys Settings

BoxPlot Settings

Package Threshold Configuration

Green Area Threshold :	76.0
Yellow Area Threshold :	221.0
Orange Area Threshold :	378.0
Red Area Threshold :	454.0

Filters

Entity Filters

- ☑ proximityalert
  - ☑ Activator
- ☑ proximityalert.actions
  - ☑ ConfigurationActionDelegate
  - ☑ ComputeProximityAlertObject
  - ☑ ViewActionDelegate
  - ☑ ActionDelegate
  - ☑ NavigationActionDelegate
- ☑ proximityalert.base
  - ☑ PackageWrap
  - ☑ EntityWrap

Figure 4.1: **Proximity Alert plug-in** analyzing itself. The view displayed is the *Package Viewer* table and the *Configuration View*.

4.2 Meta-Models

Meta-modeling is the analysis, construction and development of the frames, rules, constraints, models and theories applicable and useful for the modeling in a predefined class of problems.

In computer science field meta-modeling is the construction of a collection of "concepts" inside a certain domain. A model is an abstraction of phenomena in the real world, and a metamodel is yet another abstraction, highlighting properties of the model itself. This model is said to conform to its meta-model similar to a program conforms to the grammar of the programming language in which it is written.

We create a meta-model using and extending the X-Ray meta-model which maps java source code to a internal code representation. Both meta-models are described in the next sections.

4.2.1 X-Ray Internal Code Representation

Figure 4.2 shows the entities involved in the internal code representation provided by the X-Ray [Mal07] plug-in. It shows how every Java entity is modeled by an `Entity Representation`.

Moreover, packages and classes are contained within another entity (respectively, a project and a package), therefore use the `ContainedEntityRepresentation` class. The `ModelExtractor` class contains a `ProjectRepresentation` that is made up by zero or more `PackageRepresentations`,which are composed by zero or more `Class Representations`.

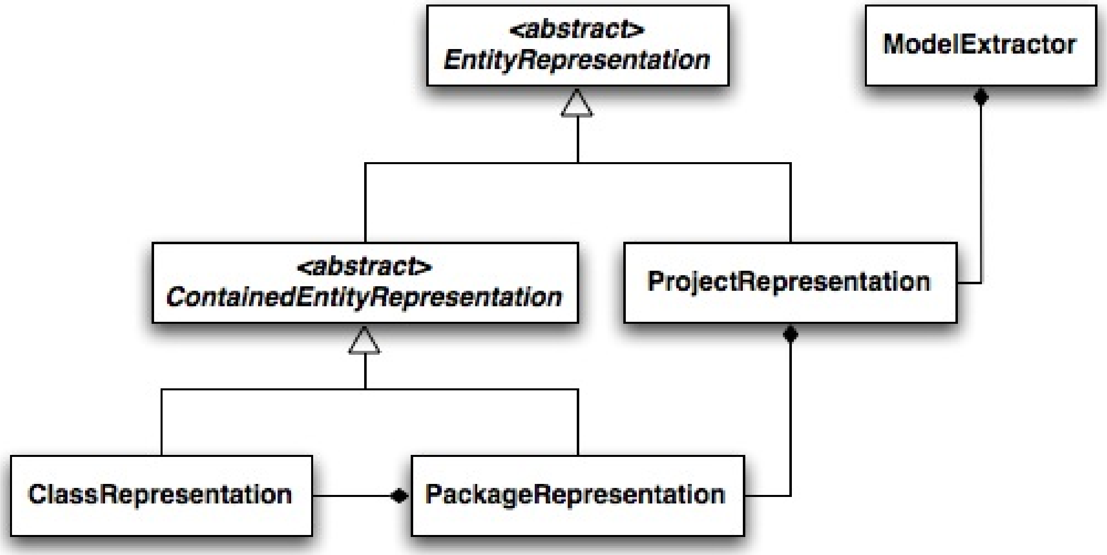


Figure 4.2: A simplified view of the **X-Ray Internal Code Representation** meta-model.

One of the most interesting part of this excellent tool is the model extractor which is responsible for creating an internal code representation which reflect he underlying source code. The model extractor is composed of two phases [Mal07]:

- **Hierarchy builder:** parses the project and collects information about the inheritance hierarchy of every class in the system. Exploiting interfaces and functionalities provided by the org.eclipse.core and org.eclipse.jdt.core plug-ins and libraries, the Hierarchy Builder fills the internal code representation (ICR) of X-Ray by gathering information from the one used by Eclipse itself. X-Rays internal code representation is a set of data structures containing meaningful data about the project, packages and classes; moreover it stores every metric and dependency.

- **Dependency Builder:** scans the source code of the system and collects information about dependencies between classes. These dependencies will be used by the Class and Package Dependency Views while creating dependency edges (arrows) between entities.

Figure 4.3 shows how is possible to get and use the described model from the X-Ray plug-in.

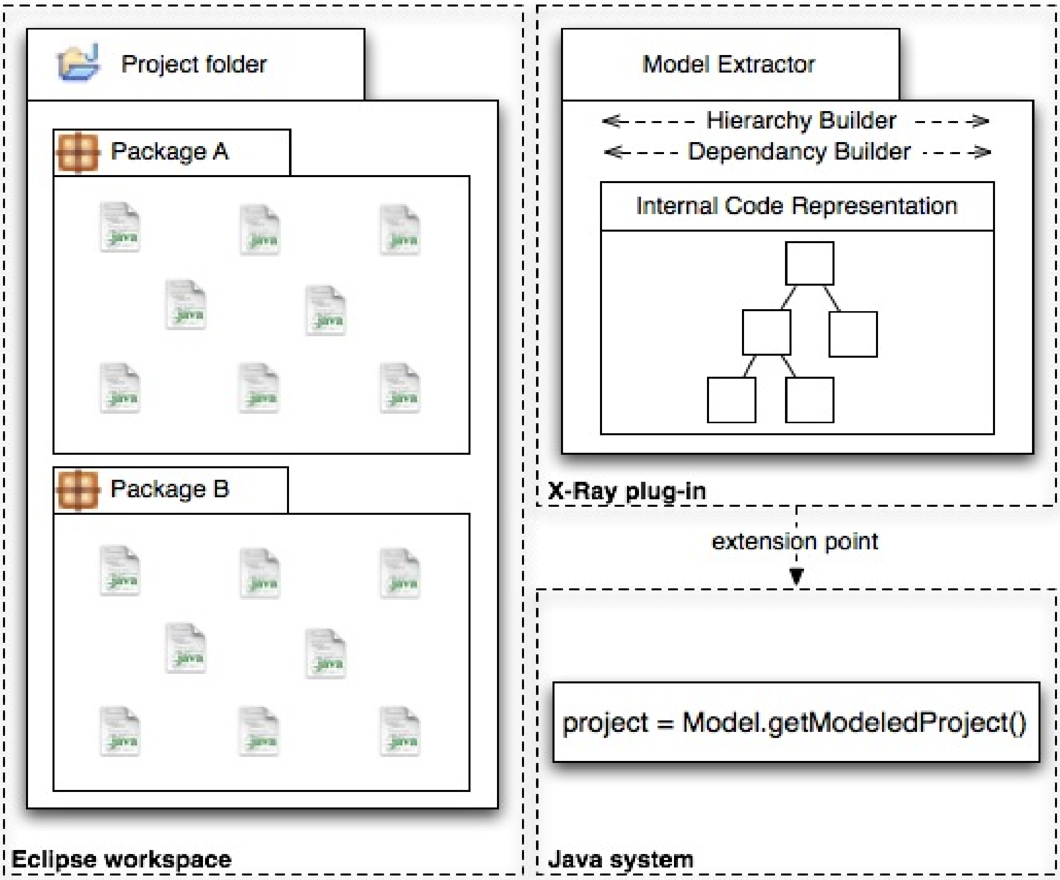


Figure 4.3: **X-Ray plug-in** allows to have access to the internal code representation of a modeled project.

4.2.2 Proximity Alert Model

Given the support of the X-Ray model extractor we can use the internal code representation and wrap it inside the Proximity Alert core model which provides metrics extension for each entity, proximity alert value and danger area depiction.

Figure 4.4 shows how Proximity Alert model uses and extends the X-Ray meta-model. Entities involved in the internal code representation provided by the X-Ray plug-in such as the `ClassRepresentation` and `PackageRepresention` and are wrapped inside concrete wrappers that extends the abstract class `EntityWrap`. The `ProjectWrap` entity is the core of the system which is composed of the model given by X-Ray merged with the Proximity Alert model.

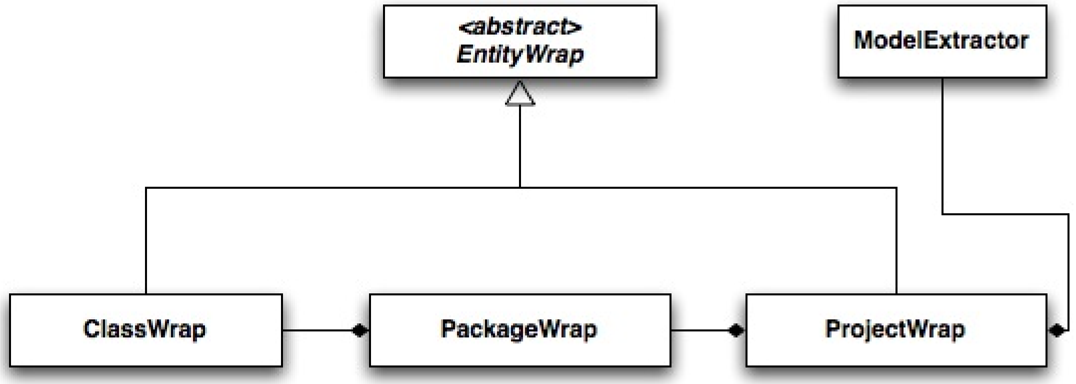


Figure 4.4: A simplified view of the **Proximity Alert** model using the `ModelExtractor` given by the X-Ray plug-in.

4.3 Software Metrics

Sometimes it's hard to tell how solid is a design really is, this because every design is subjective. That's where metrics takes role, in fact they can be helpful in pointing out strengths,weakness and potential problems while they don't provide a complete picture of the entire system. Metrics are more than just numbers, actually they can measure how well is used abstraction in the code. Good design will use abstract classes and interfaces so that other classes can program to those interfaces rather than specific implementations [BDW06].

What is a metric ?
It is the mapping of a particular characteristic of a measured entity to a numerical value. An entity and its characteristics can be anything.

Why is it useful to measure ?
In software engineering it is important and useful to measure systems, otherwise we risk losing control because of their complexity. Losing control in such a case could make us ignore the fact that certain parts of the system grow abnormally or have a bad quality.

Software metrics can be divided into two major groups [LM06]:

- *Project metrics*: They deal with the dynamics of a project, with what it takes to get to a certain point in the development life cycle and how to know you are there. They can be used in a predictive man- ner, e.g., to estimate staffing requirements. Being at a higher level of abstraction, they are less prescriptive, but are more important from an overall project perspective.

- *Design metrics*: These metrics are used to assess the size and in some cases the quality, size and complexity of software. They look at the quality of the projects design at a particular point in the development cycle. Design metrics tend to be more locally focused and more specific, thereby allowing them to be used effectively to directly examine and improve the quality of the products components.

4.3.1 Proximity Alert plug-in metrics

Proximity Alert plug-in provide an extension of the metrics given by the internal code representation of X-Ray and use them to perform additional features that are related to those measures. First of all we decide to integrate all the already available metrics and use them for doing additional analysis that can increase the description of the system design, its components and the relation between them.

The extension is done dynamically by the entity wrapper, because the metrics could be inherent to a single or to multiple entities we define three phases of metrics creation:

- **Single entity analysis:** this phase creates and extends all metrics that concern only the analysis of a class (i.e the Constr. metric which represent the number of constructors).

- **Multiple entities analysis:** this phase compute all metrics that somehow depend on other entities (i.e I-Fields which represent the number of inherited fields), or concern entity that are composition of other entities like for example packages.

- **Proximity Alert Metric analysis:** this phase compute for each entity in the analysis the Proximity Alert metric which involves all the products of the previous phases.

Since metrics could describe an infinite number of things in different forms, we decide to group our metrics by taking into account their description and affection type.

Package Metrics

At the Package level there are two main type of metrics that are the ones that describe the entity from the size point of view and the ones that describe the entity in term of complexity. An additional type of metrics is defined which consider not only the system but also the interaction with Proximity Alert system.

Package Metrics		
Type	**Name**	**Description**
Size	Classes	Number of classes declared in a package.
	SubPkgs	Number of sub-packages.
	All Classes	Number of classes in a package and its sub-packages.
	Interfaces	Number of interfaces declared in a package.
	Fields	Number of fields declared in all classes of a package.
	LoC	Number of lines of code implementation.
Complexity	Uses	Number of classes used by any class of a package.
	MsgSent	Number of messages sent to external classes.
	Used	Number of classes which uses any class of a package.
	MsgRec	Number of messages received by external classes.
Other	PAvg	Proximity Alert average of all entities of a package.

Table 4.1: **Package Metrics** provided by Proximity Alert analysis.

Class Metrics

At the class level we look not just at metrics which measure aspects of the class we also look at metrics which give us information on the interaction between classes. Metrics which measure these class interactions tell us far more about our design than about our code. Some of the metrics tell us how good our design is between our methods and inheritance dependencies while others tell us how much a change to a particular class will affect code in other class. The ideal is that changes to one class should have minimal effects on other classes, and that the number of other classes affected should be minimal.

Class Metrics		
Type	**Name**	**Description**
Size	Fields	Number of declared fields.
	Methods	Number of declared methods.
	Constr.	Number of declared constructors.
	LoC	Number of lines of code implementation.
Inheritance	I-Fields	Number of inherited fileds.
	I-Methods	Number of inherited methods.
	I-Constr.	Number of inherited constructors.
Complexity	Uses	Number of external class which are used by a class.
	MsgSent	Number of messages sent to external classes.
	Used	Number of external class which uses a class.
	MsgRec	Number of messages received by external classes.

Table 4.2: **Class Metrics** provided by Proximity Alert analysis.

4.3.2 Proximity Alert Metric

The main functionality of the system that gives also the name to our system is the Proximity Alert Metric. Most particularly it is a metric that measure and describe how an entity is involved in the system compared also with the other entities.

Proximity Alert Metric is a measure which correspond to the product of a formulation on a set of given metrics, their relevance, and on a set of entities.

By taking into account that designs are subjective, we gives the possibility to the users to customize the formulation of the Proximity Alert Metric by allowing *entity metrics selection/exclusion*, *relevance customization* and *entities filtering*.

Proximity Alert Formula

The basic formula for the calculation of the Proximity Alert Metric could be summarized in a sum of a set of given metrics multiplied by their respectively relevance and could be represented as follows:

$$ProximityAlert(entity) = \sum_{i=0}^{n}(m_i \cdot r_{m_i})$$

where n is the size of a set of given metrics, m_x represent a metric in the given set, and r_{m_x} is the relevance index of a metric.

The scope of the Proximity Alert Metric is to represent the relation of an entity given set of metrics by highlighting not only the importance of it but also the role that plays in the whole system. The relevance index is a bounded number that could have a value between 0 and 1, actually this index is a percentage of the value that has to be considered in the total sum for a given metric. This allows to customize the formula such that the metrics could be weighted.

The problem was how to represent such an important metric as the Proximity Alert of an entity, and we decide to adopt a depiction in group of similar entities by using the box-plot technique.

4.3.3 Box-Plot

In descriptive statistics, a boxplot is a convenient way of graphically depicting groups of numerical data through their five-number summaries which are the smallest observation, the first quartile, the median, the third quartile and the largest observation of a set (as shown in Table 4.3).

Figure 4.5 shows a graphical representation of the Box-plot.

Box-Plots are useful to display differences between populations without making any assumptions of the underlying statistical distribution. The spacings between the different parts of the box help indicate the degree of dispersion and skewness in the data, and identify outliers[1].

[1] http://en.wikipedia.org/wiki/Box_plot

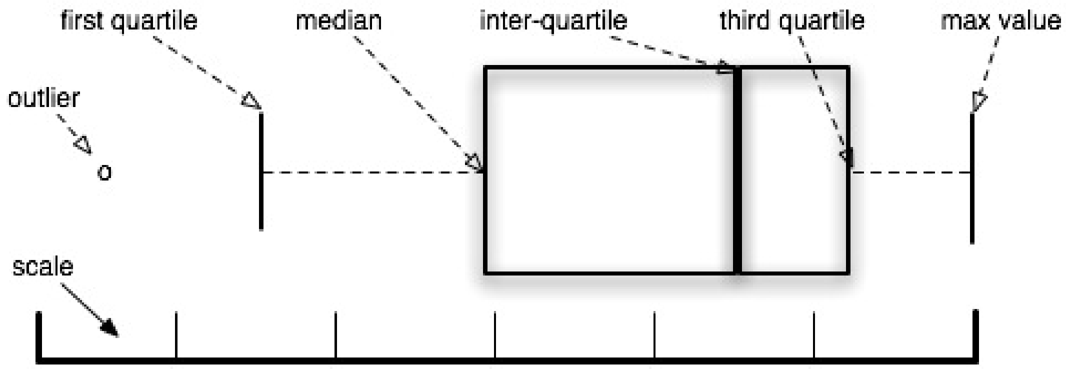


Figure 4.5: **Box-Plot** technique.

Box-Plot element	**Definition**
median (m)	middle-ranked item in the data set
first-quartile (q_1)	median of the values that are less than m
third-quartile (q_2)	median of the values that are more than m
inter-quartile (iq)	$iq = q_2 - q_1$
max-value	last item of the data set

Table 4.3: **Box-Plot** thresholds.

4.3.4 Proximity Alert Metric and Box-Plot

Using the Box-plot technique we are able to represent not only the Proximity Alert as a numerical value but also in a graphical way which contribute on the visualization of it. We decide to depict elements of a set, in our case the set of package and classes in analysis, and we assign a color area which is represented by the interval of the box-plot elements.

As shown in Figure 4.6 to each interval is equivalent a color area which represent how the proximity alert of a entity is placed respect to the all set of entities.

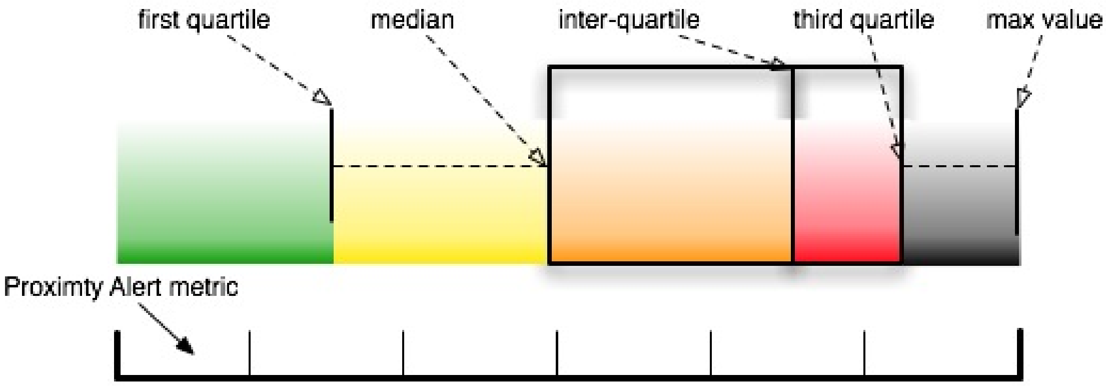


Figure 4.6: **BoxPlot** depiction in area color of the Proximity Alert metric.

We define therefore five color groups which are represented as colors. These groups collect subset of entities that, once the analysis is performed and the Proximity Alert metrics has a value, are in the same area of danger. This level of danger or risk is related to the color to which is assigned.

Most particularly area-groups are a scale of danger levels represented by colors:

- *Green area* - very low danger
- *Yellow area* - low danger
- *Orange area* - medium danger
- *Red area* - high danger
- *Black area* - very high danger

The danger level is intended to show for each entity the relation with entire set and also how much is risky affect changes in it based on the performed analysis.

4.4 Views

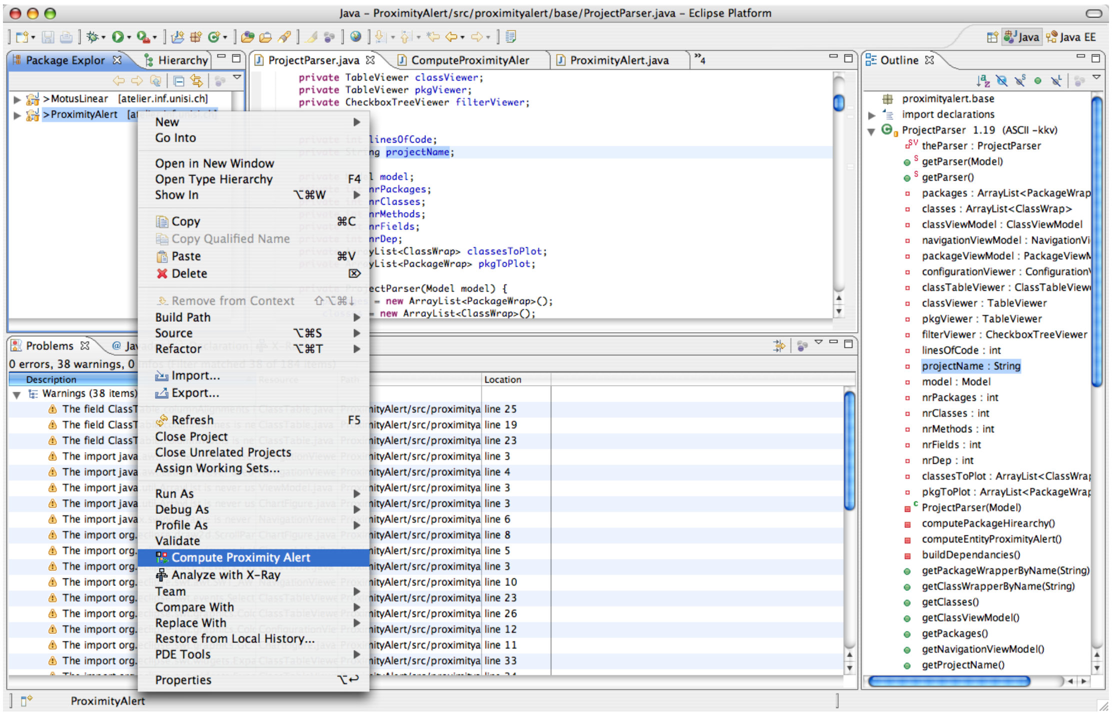
The Proximity Alert plug-in provide complete overview of the system analyzed by contributing to the Eclipse platform with its own views which allows the user to display, browse, and customize them. The two views provided by the system are the *Proximity Alert View* and the *Configuration View*.

Many plug-ins either add a new Eclipse view or enhance an existing one as a way to provide information to the user. Views must implement the `org.eclipse.ui.IViewPart` interface. Typically, views are subclasses of `org.eclipse.ui.part.ViewPart` and thus indirectly subclasses of `org.eclipse.ui.part.WorkbenchPart`, inheriting much of the behavior needed to implement the `IViewPart` interface.

4.4.1 Views Initialization

The Proximity Alert views appear by clicking on the **Window - Show View** dialog, **Other...**, then **Proximity Alert plug-in** and finally on the **Proximity Alert icon** or the **Configuration icon**.

This is the default procedure to open a view part, but it is not really aimed for the main functionality of our plug-in. Proximity Alert aim to visualize a specific project selected by the user, therefore we contributed, as shown in Figure 4.7, to the projects context menu an action (Compute Proximity Alert) that will activate the plug-in and analyze the currently selected project.


Java – ProximityAlert/src/proximityalert/base/ProjectParser.java – Eclipse Platform
Java
Java EE
Package Explor
Hierarchy
>MotusLinear [atelier.inf.unisi.ch]
>ProximityAlert [at
ProjectParser.java
ComputeProximityAler
ProximityAlert.java
»4
New
Go Into
Open in New Window
Open Type Hierarchy F4
Show In ⌥⌘W
Copy ⌘C
Copy Qualified Name
Paste ⌘V
Delete
Remove from Context ⇧⌥⌘↓
Build Path
Source ⌥⌘S
Refactor ⌥⌘T
Import...
Export...
Refresh F5
Close Project
Close Unrelated Projects
Assign Working Sets...
Run As
Debug As
Profile As
Validate
Compute Proximity Alert
Analyze with X-Ray
Team
Compare With
Replace With
Restore from Local History...
PDE Tools
Properties ⌥↩
Problems
Ja
0 errors, 38 warnings, 0
Description
Location
Warnings (38 items)
The field ClassT
The field ClassT
The field ClassT
The import java
The import java
The import java
The import java
The import java
The import org.
The import org.
The import org.
The import org.
The import org.
The import org.
The import org.
The import org.
The import org.
ityAlert/src/proximitya line 25
ityAlert/src/proximitya line 19
ityAlert/src/proximitya line 23
ityAlert/src/proximitya line 3
ityAlert/src/proximitya line 4
ityAlert/src/proximitya line 3
ityAlert/src/proximitya line 3
ityAlert/src/proximitya line 6
ityAlert/src/proximitya line 8
ityAlert/src/proximitya line 5
ityAlert/src/proximitya line 3
ityAlert/src/proximitya line 10
ityAlert/src/proximitya line 23
ityAlert/src/proximitya line 26
ityAlert/src/proximitya line 12
ityAlert/src/proximitya line 11
ityAlert/src/proximitya line 33
Outline
proximityalert.base
import declarations
ProjectParser 1.19 (ASCII -kkv)
theParser : ProjectParser
getParser(Model)
getParser()
packages : ArrayList<PackageWrap
classes : ArrayList<ClassWrap>
classViewModel : ClassViewModel
navigationViewModel : NavigationVi
packageViewModel : PackageViewM
configurationViewer : Configuration
classTableViewer : ClassTableView
classViewer : TableViewer
pkgViewer : TableViewer
filterViewer : CheckboxTreeViewer
linesOfCode : int
projectName : String
model : Model
nrPackages : int
nrClasses : int
nrMethods : int
nrFields : int
nrDep : int
classesToPlot : ArrayList<ClassWrap
pkgToPlot : ArrayList<PackageWrap
ProjectParser(Model)
computePackageHirearchy()
computeEntityProximityAlert()
buildDependancies()
getPackageWrapperByName(String)
getClassWrapperByName(String)
getClasses()
getClassViewModel()
getPackages()
getNavigationViewModel()
getProjectName()
ProximityAlert

```
private TableViewer classViewer;
private TableViewer pkgViewer;
private CheckboxTreeViewer filterViewer;

linesOfCode;
ng projectName;

l model;
nrPackages;
nrClasses;
nrMethods;
nrFields;
nrDep;
yList<ClassWrap> classesToPlot;
yList<PackageWrap> pkgToPlot;

ectParser(Model model) {
 = new ArrayList<PackageWrap>();
 = new ArrayList<ClassWrap>();
```

Figure 4.7: **Proximity Alert plug-in** adds an action that will be visible after right-clicking on a project in the Package Explorer (In this example, the Proximity Alert project itself). Clicking on **Compute Proximity Alert** action, Proximity Alert views will be activated and shown.

After this the two views provided by Proximity Alert are shown in the standard Eclipse perspective. By adding this specific action we removed the difficulty of deciding which project to analyze, given that the user might have multiple projects opened concurrently as far as different files belonging to different projects in the workbench editor. In this way we force to select which project the plug-in will visualize and analyze.

4.4.2 Proximity Alert View

The Proximity Alert View provides a tabbed panel which allows to switch between two type of tables. The tables contained represent the list of the entities analyzed by giving the value for each of the available metrics and shows the Proximity Alert Metric computed.

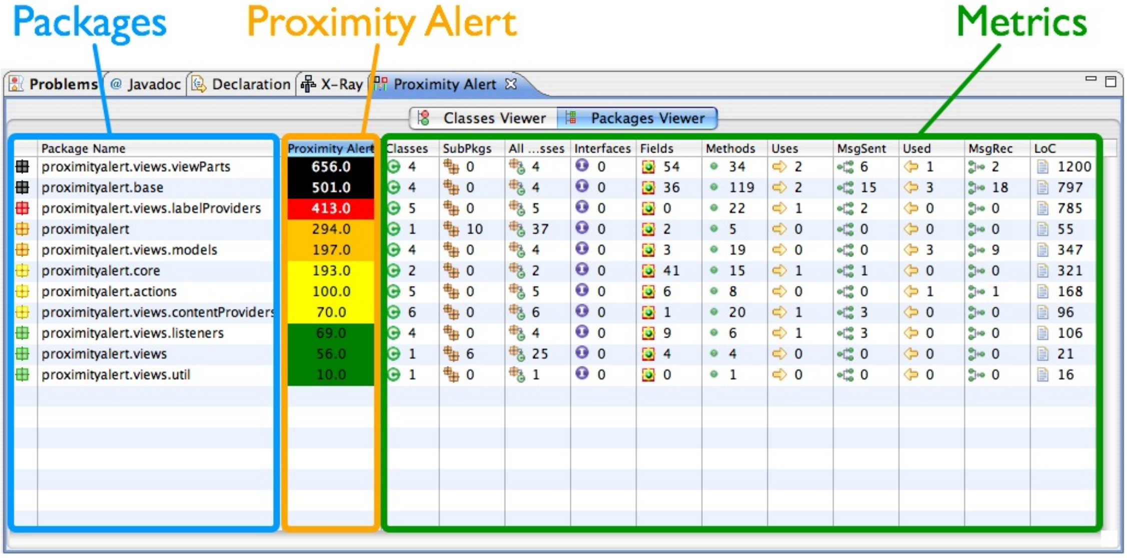
The two tables are called respectively *Package Viewer* and the *Class Viewer*.

Package Viewer

The package viewer provides a deep view of entities analyzed and shows in the left part the name of the packages, then the Proximity Alert Metric followed by all other metrics provided by the analysis described in the Proximity Alert Metrics section.

The table provides all operation of selection and sort for each column in order to give he users the possibility to browse more easily the content.

Figure 4.8 shows the Package Viewer on the Proximity Alert project and how its composed.



Package Name	Proximity Alert	Classes	SubPkgs	All ...sses	Interfaces	Fields	Methods	Uses	MsgSent	Used	MsgRec	LoC
proximityalert.views.viewParts	656.0	4	0	4	0	54	34	2	6	1	2	1200
proximityalert.base	501.0	4	0	4	0	36	119	2	15	3	18	797
proximityalert.views.labelProviders	413.0	5	0	5	0	0	22	1	2	0	0	785
proximityalert	294.0	1	10	37	0	2	5	0	0	0	0	55
proximityalert.views.models	197.0	4	0	4	0	3	19	0	0	3	9	347
proximityalert.core	193.0	2	0	2	0	41	15	1	1	0	0	321
proximityalert.actions	100.0	5	0	5	0	6	8	0	0	1	1	168
proximityalert.views.contentProviders	70.0	6	0	6	0	1	20	1	3	0	0	96
proximityalert.views.listeners	69.0	4	0	4	0	9	6	1	3	0	0	106
proximityalert.views	56.0	1	6	25	0	4	4	0	0	0	0	21
proximityalert.views.util	10.0	1	0	1	0	0	1	0	0	0	0	16

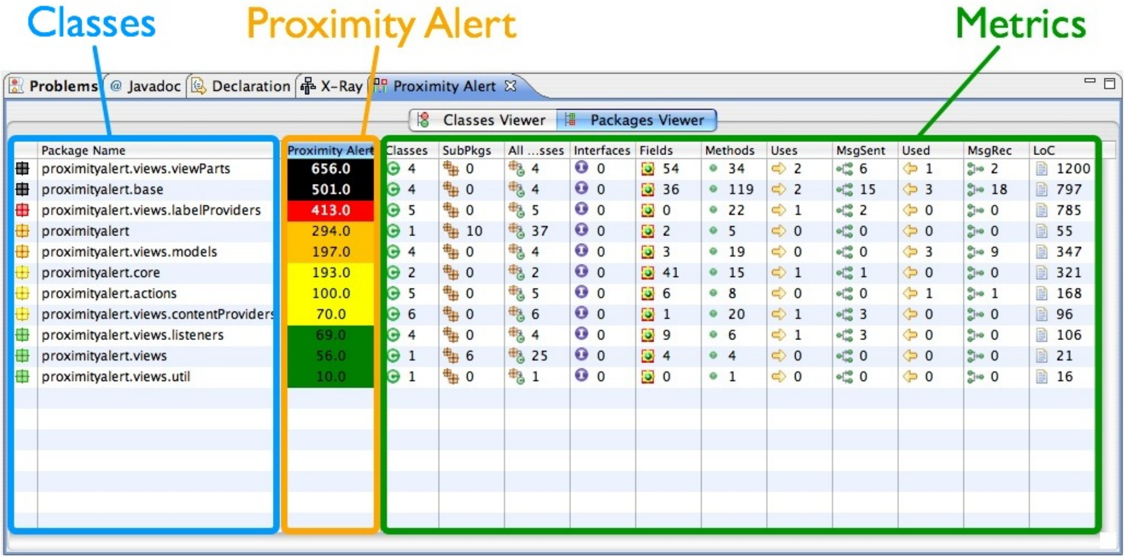
Figure 4.8: **Proximity Alert view** showing the **Package Viewer** of itself.

Class Viewer

The class viewer provides a deep view of entities analyzed and shows in the left part the name of the classes, then the Proximity Alert Metric followed by all the metrics provided by the analysis described in the Proximity Alert Metrics section.

The table provides all operation of selection and sort for each column in order to give he users the possibility to browse more easily the content. In this table is also possible to select classes and by double-clicking on them its possible to open the respectively *.java* file in the Editor View Part.

Figure 4.9 shows the Class Viewer on the Proximity Alert project and how its composed.



	Package Name	Proximity Alert	Classes	SubPkgs	All ...sses	Interfaces	Fields	Methods	Uses	MsgSent	Used	MsgRec	LoC
⊞	proximityalert.views.viewParts	656.0	4	0	4	0	54	34	2	6	1	2	1200
⊞	proximityalert.base	501.0	4	0	4	0	36	119	2	15	3	18	797
⊞	proximityalert.views.labelProviders	413.0	5	0	5	0	0	22	1	2	0	0	785
⊞	proximityalert	294.0	1	10	37	0	2	5	0	0	0	0	55
⊞	proximityalert.views.models	197.0	4	0	4	0	3	19	0	0	3	9	347
⊞	proximityalert.core	193.0	2	0	2	0	41	15	1	1	0	0	321
⊞	proximityalert.actions	100.0	5	0	5	0	6	8	0	0	1	1	168
⊞	proximityalert.views.contentProviders	70.0	6	0	6	0	1	20	1	3	0	0	96
⊞	proximityalert.views.listeners	69.0	4	0	4	0	9	6	1	3	0	0	106
⊞	proximityalert.views	56.0	1	6	25	0	4	4	0	0	0	0	21
⊞	proximityalert.views.util	10.0	1	0	1	0	0	1	0	0	0	0	16

Figure 4.9: **Proximity Alert view** showing the **Class Viewer** of itself.

4.4.3 Configuration View

The *Configuration View* provides an expand bar which allows to show or hide different items. The view provide five different expandable item which are titled: *General Information*, *Class Analysis Settings*, *Package Analysis Settings*, *BoxPlot Settings* and *Filters*.

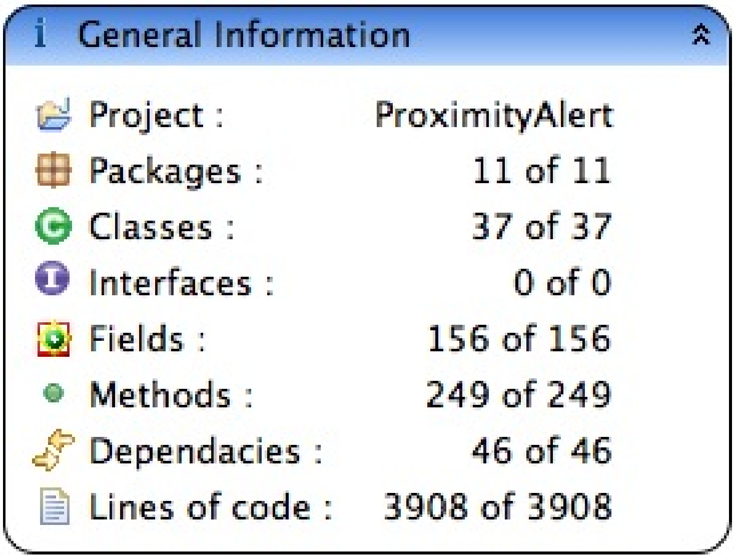
For each changes in these panels effects could be seen dynamically on the analysis of the project.

General Information

Figure 4.10 shows general information about the project analyzed such as the name and the number of packages, classes, interfaces, fields, methods, dependencies and lines of code contained in it.

The value are represented as a ratio of the number of considered in the analysis on the total available.

Therefore during filters configuration of the user it can has the possibility to see what is the portion of the component considered in the analysis on the maximum available.



General Information

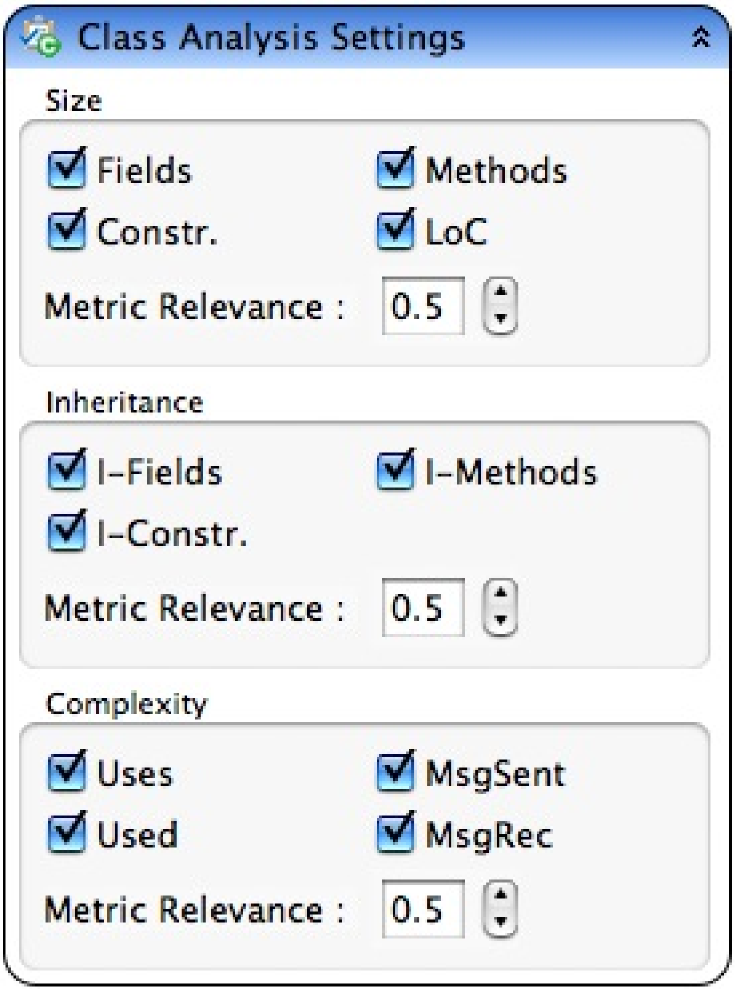
Project :	ProximityAlert
Packages :	11 of 11
Classes :	37 of 37
Interfaces :	0 of 0
Fields :	156 of 156
Methods :	249 of 249
Dependacies :	46 of 46
Lines of code :	3908 of 3908

Figure 4.10: **General Information** item of the Configuration View shows general information about the project analyzed such as the name and the number of packages, classes, interfaces, fields, methods, dependencies and lines of code contained in it. In this case shows information about the Proximity Alert project itself.

Class Analysis Settings

Figure 4.11 shows all the available class metrics that could be taken into account for the analysis of the project. The user has the possibility to check or uncheck metrics in order to add or exclude from the analysis.

The metrics are divided into types such as *size*, *inheritance* and *complexity*, for each of them is possible to change their relevance in the formula. The relevance is a number bounded from 0 to 1.



Class Analysis Settings

Size

☑ Fields ☑ Methods

☑ Constr. ☑ LoC

Metric Relevance : 0.5

Inheritance

☑ I-Fields ☑ I-Methods

☑ I-Constr.

Metric Relevance : 0.5

Complexity

☑ Uses ☑ MsgSent

☑ Used ☑ MsgRec

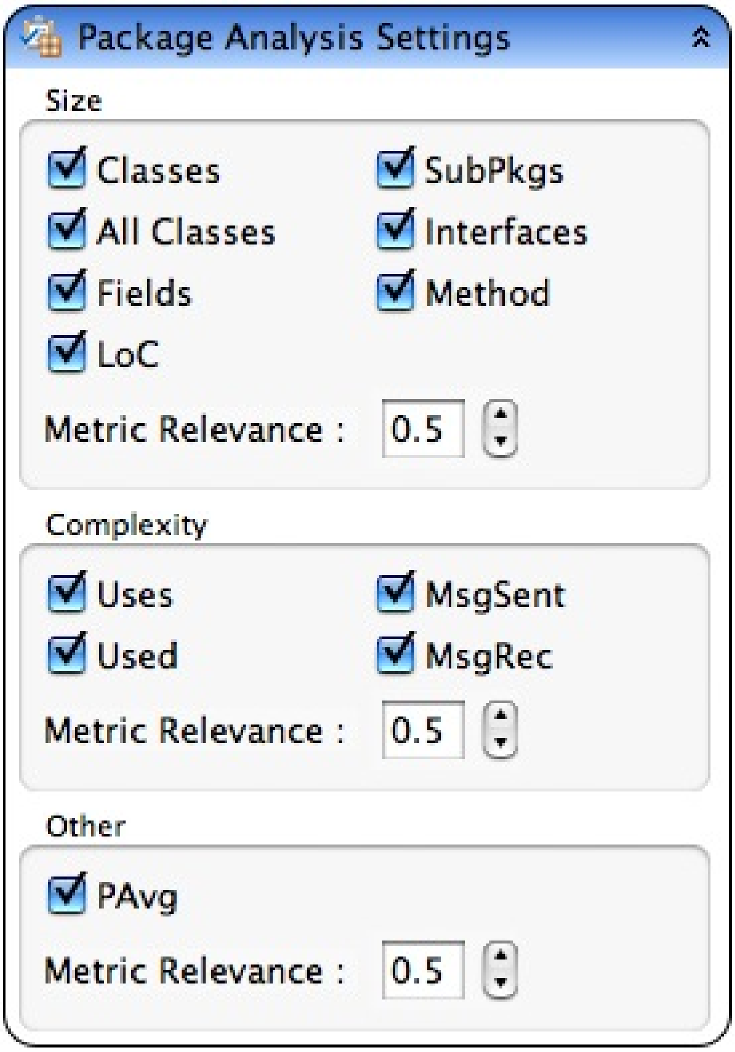
Metric Relevance : 0.5

Figure 4.11: **Class Analysis Settings** items shows all the available class metrics that could be taken into account for the analysis of the project. The user has the possibility to check or uncheck metrics in order to add or exclude from the analysis.

Package Analysis Settings

Figure 4.12 shows all the available package metrics that could be taken into account for the analysis of the project. The user has the possibility to check or uncheck metrics in order to add or exclude from the analysis.

The metrics are divided into types such as *size*, *complexity* and *other*, for each of them is possible to change their relevance in the formula. The relevance is a number bounded from 0 to 1.



Package Analysis Settings

Size

☑ Classes ☑ SubPkgs
☑ All Classes ☑ Interfaces
☑ Fields ☑ Method
☑ LoC

Metric Relevance : 0.5

Complexity

☑ Uses ☑ MsgSent
☑ Used ☑ MsgRec

Metric Relevance : 0.5

Other

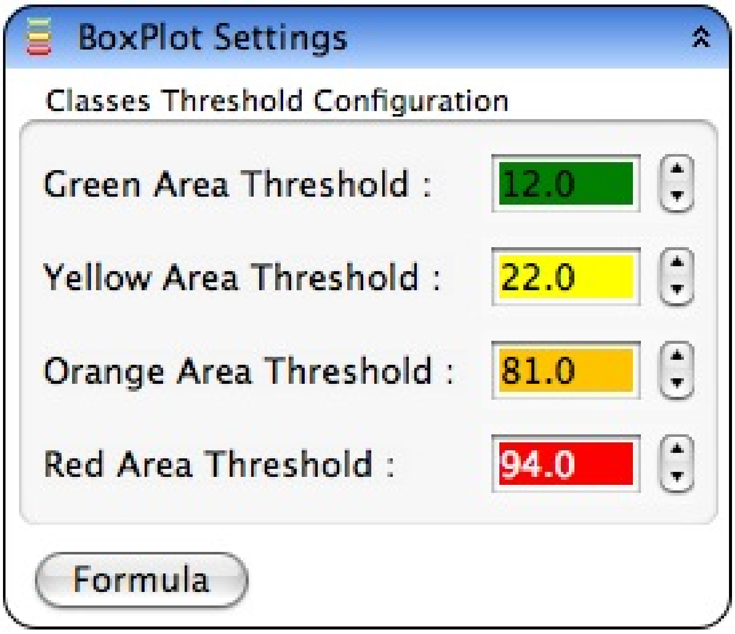
☑ PAvg

Metric Relevance : 0.5

Figure 4.12: **Package Analysis Settings** item shows all the available package metrics that could be taken into account for the analysis of the project. The user has the possibility to check or uncheck metrics in order to add or exclude from the analysis.

Box-Plot Settings

Figure 4.13 shows the settings of the box-plot. Most particularly shows the current thresholds of the *green*, *yellow*, *orange* and *red* area and allows user to modify them and recalculate them with the available button *formula*.



BoxPlot Settings

Classes Threshold Configuration

Green Area Threshold : 12.0

Yellow Area Threshold : 22.0

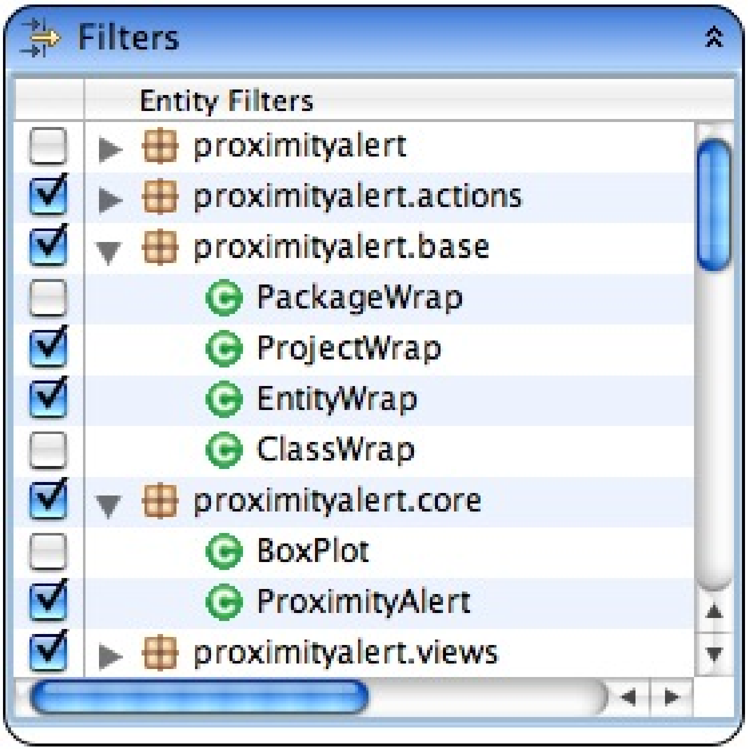
Orange Area Threshold : 81.0

Red Area Threshold : 94.0

Figure 4.13: **BoxPlot Setting** item shows the current thresholds of the *green*, *yellow*, *orange* and *red* area and allows user to modify them and recalculate them with the available button *formula*

Filters

Figure 4.14 shows the FIlters item which is composed by a navigable tree that contains all project packages and classes with the possibility to check or uncheck them such that the only the selected entities are taken into account in the analysis. If a user check or uncheck a package contained in the project automatically all the classes contained reflect the container visibility. Consider that for any checked class is taken into account also the package which contains them.



Filters

Entity Filters

- □ ▶ proximityalert
- ☑ ▶ proximityalert.actions
- ☑ ▼ proximityalert.base
  - □ PackageWrap
  - ☑ ProjectWrap
  - ☑ EntityWrap
  - □ ClassWrap
- ☑ ▼ proximityalert.core
  - □ BoxPlot
  - ☑ ProximityAlert
- ☑ ▶ proximityalert.views

Figure 4.14: **Filters** item shows a navigable tree that contains all project packages and classes with the possibility to check or uncheck them such that the only the selected entities are taken into account in the analysis. The entity displayed are the one of the Proximity Alert project.

4.5 The Plug-in

The plugin requires Java 1.5 and the Eclipse framework 3.*. It is written in Java, it's free and open-source.

The plug-in can be downloaded directly from the official website[2] of the Proximity Alert plug-in , and is also registered in the Eclipse Plug-In Center[3].

This document can be found within the Proximity Alert documentation on the official web-page. Source code and Javadocs can be found on the web site and also in the jar file.

[2] http://atelier.inf.unisi.ch/∼casarela/ProximityAlert

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

Chapter 5

Validation

5.1 Introduction

During the creation of the Proximity Alert plug-in we initiated a collaboration with the X-Ray plug-in's creator that allowed us to validate and create our system. From this exiting collaboration we have been able also to perform usage and extension of a pre-existent system and model it as an extendible framework. Thanks also to this great experience and partnership in the Eclipse platform contribution we've been able to build our Proximity Alert analysis tool.

In this chapter we validate our work and its applicability in the system design development. By using the Proximity Alert plug-in cooperating with X-Ray software visualization, we guide users trough all the operations available and drawing some observation about the current design and the potential changes that could be done of the Proximity Alert project in order to validate also the usage and realibility of the system.

First we describe how we extracted the source code from the system and loaded the data for an analysis in our tool. In the second part we present an informal overview of the results we gathered while we analyzed the software itself using the plug-in. The results should present useful informations which we are able to extract from the source code and how the data can be interpreted. We talk then about user interactions and analysis customization that our system provides.

5.2 From Source Code to Proximity Alert

5.2.1 X-Ray integration and extension

Once is given a project to analyze all the sources are scanned using the X-Ray meta-model. Used as an external framework Proximity Alert extends the model given and use it to recreate an extended internal code representation. The code is represented as a collection of entities, which are concretely defined as project, package and class representations. The wrap strategy allows the system to use the preexistent model and extends each entities by performing additional analysis of their composition and their role in the system.

Figure 5.1 shows how Proximity Alert is executed and how use the X-Ray model in order to perform extension if it.

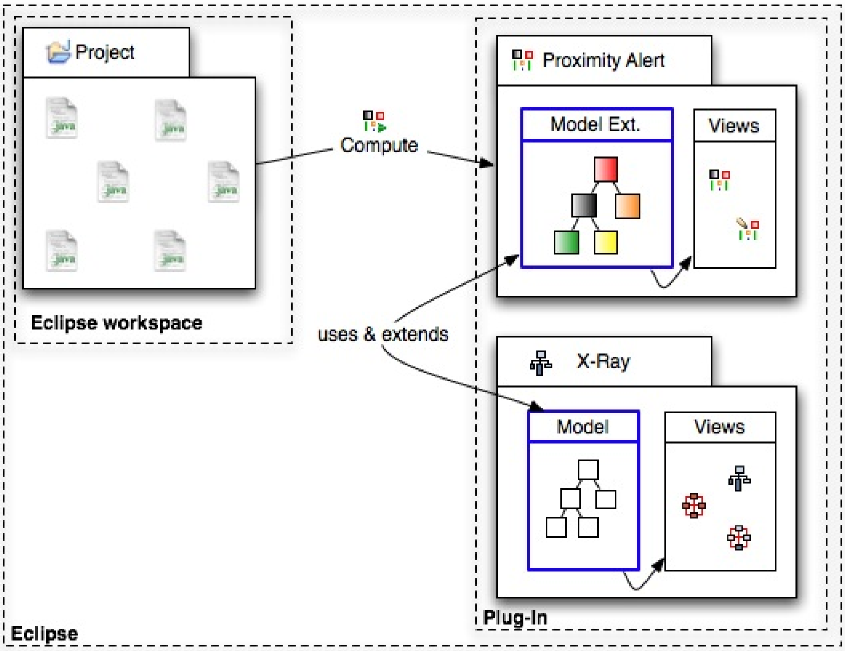


Figure 5.1: Proximity Alert system schema shows how Proximity Alert is executed and how use the X-Ray model in order to perform extension if it.

5.2.2 Metrics Definition

Consider that since informations are extracted directly from the source code using the Eclipse JDT there are a lot of details that X-Ray meta-model does not extract and once the model is given they are not more accessible. For this reason we've decide for now to define metrics that could be derived from the available informations given by X-Ray. Actually we've already planned a review of the X-Ray meta-model and the amount of information that would be useful to access.

Proximity Alert metrics are provided at overall system, package and class level. Metrics defined by the plug-in are listed and described in the previous chapter in Table 4.1 for the package level and in Table 4.2 for the class level.

5.2.3 Proximity Alert Views

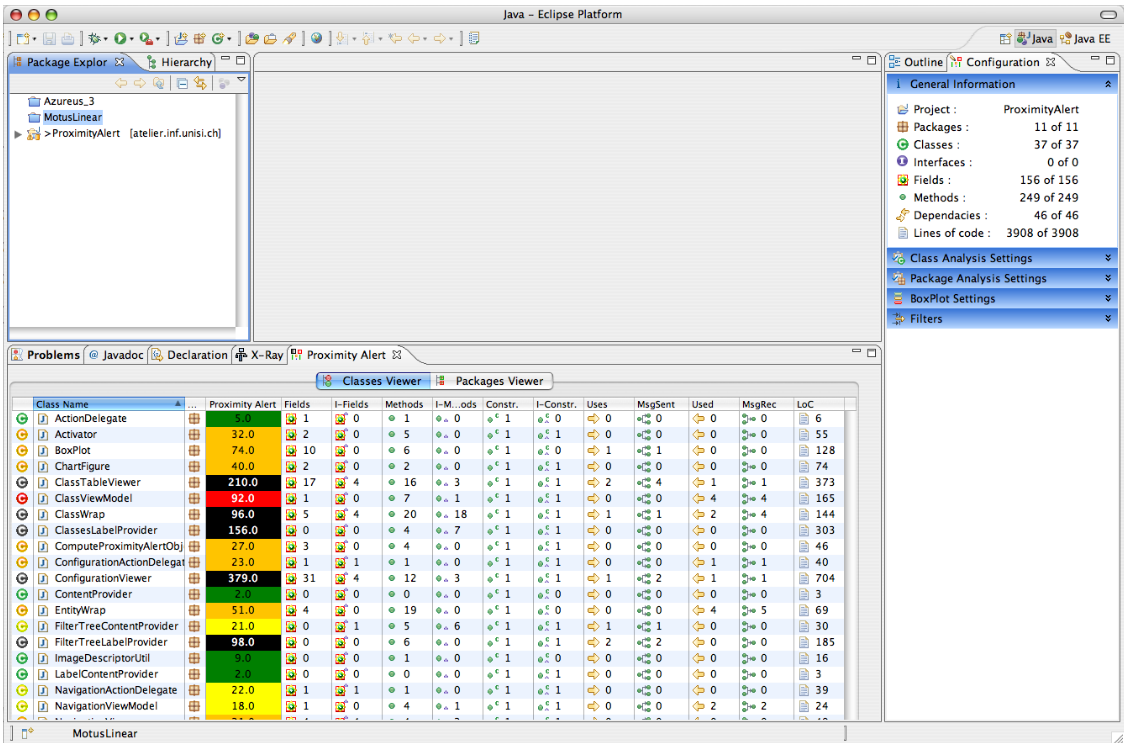
The views provided by the plug-in allows to browse entities and perform sort actions directly from the table viewers which contain them. From the class viewer is also possible to directly double-click on a class and open it in the editor part of Eclipse such that if user want to look fast at the code of it is able do it in a simple and fast way.

Consider that most of the functionalities that could be done in the table viewers are now under construction. Actually our decision was first to create a draft of the system and collect feedback in order to gather all the impressions of the first users and use them to perform additional features. The *Configuration View* allows to customize completely the analysis, such as *metrics inclusion/exclusion*, *entities filtering* and *edit box-plot thresholds*.

5.3 Case Study : Proximity Alert

Let us analyze using the Proximity alert plug-in the project itself, As shown in the Figure 4.7 we select the project and choose the action **Compute Proximity Alert**. Immediately a bar inform the progress for each entity analyzed in the project so far. Once is finished opens all the views provided by the plug-in and are shown in the Eclipse UI. For a graphical convection we expose the ideal position of the views in order to have best control and view of the plug-in.

As shown in Figure 5.2 the Proximity Alert project is analyzed and the plug-in is ready for user customization and use. In the bottom of the Eclipse UI we have the *Proximity Alert View* which allows to browse the entities analyzed contained as described in the previous chapter, while in the right side we have the *Configiration View* which allows to customize settings and to get general information about the current system analysis.



General Information

Project :	ProximityAlert
Packages :	11 of 11
Classes :	37 of 37
Interfaces :	0 of 0
Fields :	156 of 156
Methods :	249 of 249
Dependacies :	46 of 46
Lines of code :	3908 of 3908

Class Analysis Settings

Package Analysis Settings

BoxPlot Settings

Filters

Class Name	...	Proximity Alert	Fields	I-Fields	Methods	I-M...ods	Constr.	I-Constr.	Uses	MsgSent	Used	MsgRec	LoC
ActionDelegate		5.0	1	0	1	0	1	0	0	0	0	0	6
Activator		32.0	2	0	5	0	1	1	0	0	0	0	55
BoxPlot		74.0	10	0	6	0	1	0	1	1	0	0	128
ChartFigure		40.0	2	0	2	0	1	1	0	0	0	0	74
ClassTableViewer		210.0	17	4	16	3	1	1	2	4	1	1	373
ClassViewModel		92.0	1	0	7	1	1	1	0	0	4	4	165
ClassWrap		96.0	5	4	20	18	1	1	1	1	2	4	144
ClassesLabelProvider		156.0	0	0	4	7	1	1	0	0	0	0	303
ComputeProximityAlertObj		27.0	3	0	4	0	1	1	0	0	0	0	46
ConfigurationActionDelegat		23.0	1	1	1	0	1	1	0	0	1	1	40
ConfigurationViewer		379.0	31	4	12	3	1	1	1	2	1	1	704
ContentProvider		2.0	0	0	0	0	1	0	0	0	0	0	3
EntityWrap		51.0	4	0	19	0	1	0	0	0	4	5	69
FilterTreeContentProvider		21.0	0	1	5	6	1	1	1	1	0	0	30
FilterTreeLabelProvider		98.0	0	0	6	0	1	1	2	2	0	0	185
ImageDescriptorUtil		9.0	0	0	1	0	1	0	0	0	0	0	16
LabelContentProvider		2.0	0	0	0	0	1	1	0	0	0	0	3
NavigationActionDelegate		22.0	1	1	1	0	1	1	0	0	0	0	39
NavigationViewModel		18.0	1	0	4	1	1	1	0	0	2	2	24

Figure 5.2: **Proximity Alert plug-in** in action on its source code immediately after launched the analysis by selecting the project from the Eclipse workbench.

After this let us we are able to draw some general observation about the project we've just analyzed. Using X-Ray plug-in we have also a graphical support which will allows us to analyze the source not only based on numbers but also represented as shapes. In order to analyze the project using he X-Ray plug-in just select the **Analyze with X-Ray** action by selecting the same project as the one analyzed with Proximity Alert. Figure 5.3 shows the Complexity View of the Proximity Alert project.

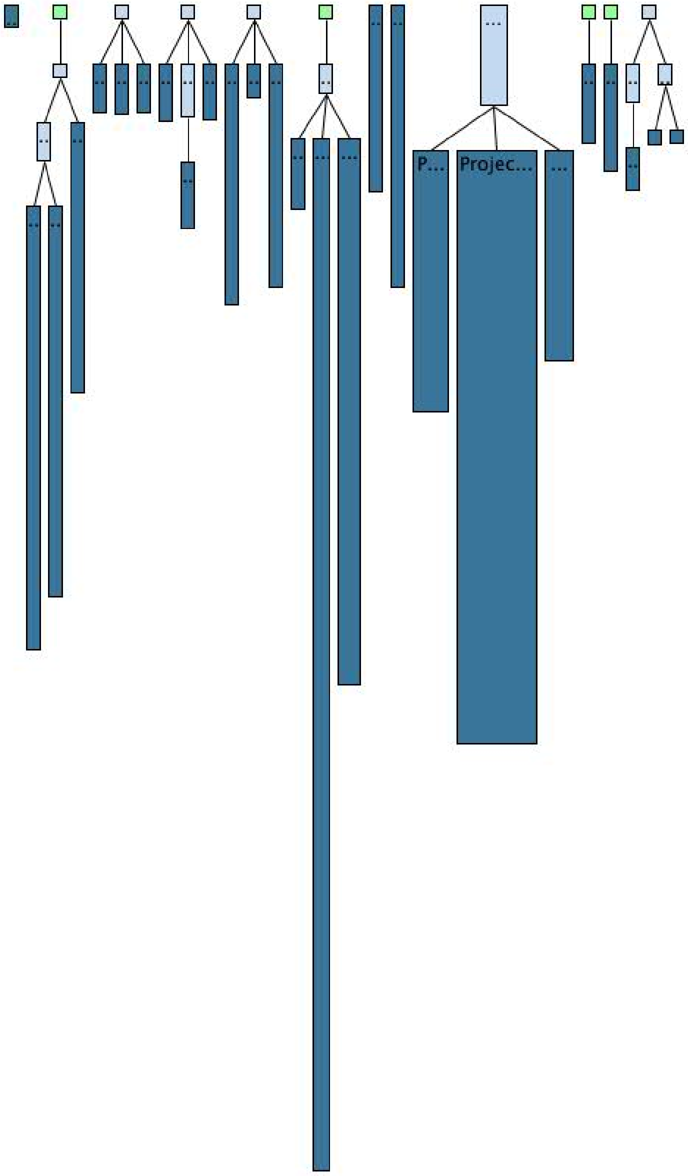
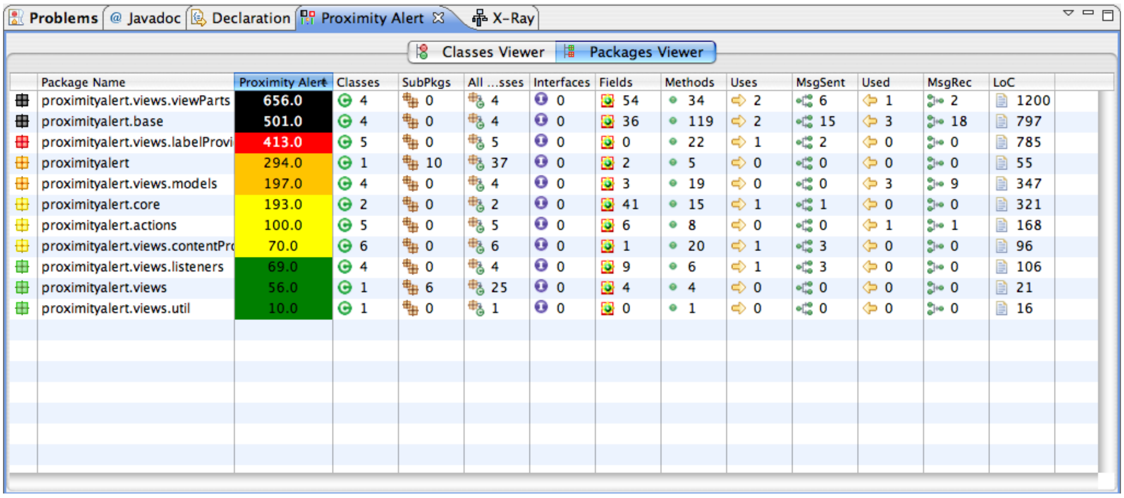


Figure 5.3: **X-Ray** plug-in system. Figure shows the *Complexity View* of the Proximity Alert project.

5.3.1 Packages Analysis

Let us analyze a bit the project given by having a look at the packages defined in it. In order to do that we simply select the *Packages Viewer* table available in the *Proximity Alert View*.

Figure 5.4 shows the packages contained in the Proximity Alert project.



	Package Name	Proximity Alert	Classes	SubPkgs	All ...sses	Interfaces	Fields	Methods	Uses	MsgSent	Used	MsgRec	LoC
▦	proximityalert.views.viewParts	656.0	4	0	4	0	54	34	2	6	1	2	1200
▦	proximityalert.base	501.0	4	0	4	0	36	119	2	15	3	18	797
▦	proximityalert.views.labelProvi	413.0	5	0	5	0	0	22	1	2	0	0	785
▦	proximityalert	294.0	1	10	37	0	2	5	0	0	0	0	55
▦	proximityalert.views.models	197.0	4	0	4	0	3	19	0	0	3	9	347
▦	proximityalert.core	193.0	2	0	2	0	41	15	1	1	0	0	321
▦	proximityalert.actions	100.0	5	0	5	0	6	8	0	0	1	1	168
▦	proximityalert.views.contentPr	70.0	6	0	6	0	1	20	1	3	0	0	96
▦	proximityalert.views.listeners	69.0	4	0	4	0	9	6	1	3	0	0	106
▦	proximityalert.views	56.0	1	6	25	0	4	4	0	0	0	0	21
▦	proximityalert.views.util	10.0	1	0	1	0	0	1	0	0	0	0	16

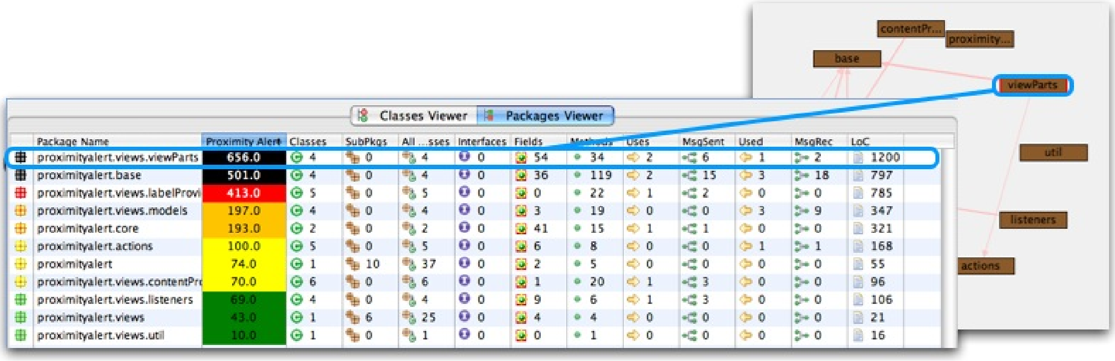
Figure 5.4: **Package Viewer** of the Proximity Alert project. By simply clicking on the table headers available the table dynamically sorts component allowing faster and simpler view of the entities contained.

Thanks to the *Package Viewer* table we can make some observation about the packages that compose the project. We see that the project is composed by several packages and sub-packages each of them represented in a row of the table and having a set of metrics which describe much deeper their composition. As shown in the previous chapter the most important metric defined in this table and in the one of the class entities is the Proximity Alert Metric which is computed by the formula and depicted with the Box-plot technique.

We see that the project is composed by three packages which have a very low danger level, three with a low level, two with a medium level, one with an high and finally two with a very high danger level. If we look also at all the metrics we could also see why those entities have those values and why are they under a certain level of danger. In this case we consider all the available metrics defined for each entity, therefore the Proximity Alert Metric as described in the previous chapter is computed taking into account all of them. Given the list of entities we assume which are the most involved and which have a very high danger level, which means that affecting changes in those entities could affect behavior, relation with other entities directly dependent or just linked. If we came back to the table we can interpret better this concept. We also use the X-Ray plug-in to give a graphical support to the tables.

Consider the `views.viewParts` package which has a very high danger level. This very high risk is given in fact because contains classes with an high number of lines of coded and an high number of fields defined inside them.

Figure 5.5 shows the entity in the table and its representation in the *Package Dependency View* of the X-Ray pug-in.



	Package Name	Proximity Alert	Classes	SubPkgs	All ...sses	Interfaces	Fields	Methods	Uses	MsgSent	Used	MsgRec	LoC
⊞	proximityalert.views.viewParts	656.0	4	0	4	0	54	34	2	6	1	2	1200
⊞	proximityalert.base	501.0	4	0	4	0	36	119	2	15	3	18	797
⊞	proximityalert.views.labelProvi	413.0	5	0	5	0	0	22	1	2	0	0	785
⊞	proximityalert.views.models	197.0	4	0	4	0	3	19	0	0	3	9	347
⊞	proximityalert.core	193.0	2	0	2	0	41	15	1	1	0	0	321
⊞	proximityalert.actions	100.0	5	0	5	0	6	8	0	0	1	1	168
⊞	proximityalert	74.0	1	10	37	0	2	5	0	0	0	0	55
⊞	proximityalert.views.contentPr	70.0	6	0	6	0	1	20	1	3	0	0	96
⊞	proximityalert.views.listeners	69.0	4	0	4	0	9	6	1	3	0	0	106
⊞	proximityalert.views	43.0	1	6	25	0	4	4	0	0	0	0	21
⊞	proximityalert.views.util	10.0	1	0	1	0	0	1	0	0	0	0	16

Figure 5.5: **Package Viewer** of the Proximity Alert project. Analysis and view of the `views.viewParts` package.

By looking at the metrics values of the `views.viewParts` package we observe that this entity is does not have too many relations, in fact it communicates only with two other entities that are in the `base` and with the `views.actions`. Given these dependencies and the names with the related ones it becomes clear that this entity contains classes which are responsible for creating and maintaining the views provided by our plug-in.

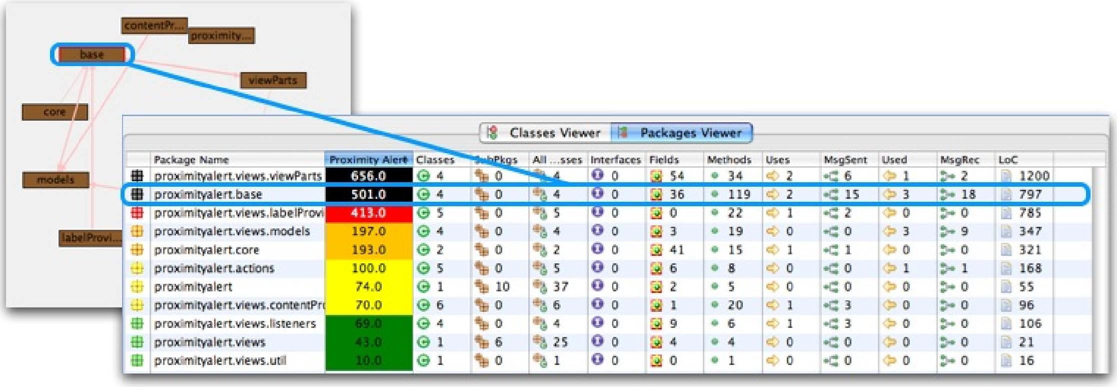
Given this first observation we also deduce why it contains lot of lines of codes, in fact those are given because views in SWT require a lot of customization of the widgets and layout configurations. For that reason we can also make other relevant observation about the other metrics defined, for example why there're so many fields defined inside the package.

Consider now the `base` package which is the second entity with the higher danger level. In this case while there are less lines of code and field declared it results havin a very high risk because of other reason that we could notice from the package composition.

Figure 5.6 shows the entity in the table and its representation in the *Package Dependency View* of the X-Ray plug-in.

By looking at the metrics values of the `base` package we can observe that this entity has many relations, in fact it communicates with four other entities which are the `core`, the `views.models`, the `views.labelProviders` and the `views.viewParts` packages. Also in this case given those dependencies we can make some assumption about the composition of the considered entity and why it results to have such an high danger level.

Given the fact that this entity has many relations it's clear that contains classes which are in a sense the most involved in the system and the ones that form the intelligence of whole system. In fact the package contains the model of the internal code represen-



	Package Name	Proximity Alert	Classes	SubPkgs	All ...sses	Interfaces	Fields	Methods	Uses	MsgSent	Used	MsgRec	LoC
▦	proximityalert.views.viewParts	656.0	4	0	4	0	54	34	2	6	1	2	1200
▦	proximityalert.base	501.0	4	0	4	0	36	119	2	15	3	18	797
▦	proximityalert.views.labelProvi	413.0	5	0	5	0	0	22	1	2	0	0	785
▦	proximityalert.views.models	197.0	4	0	4	0	3	19	0	0	3	9	347
▦	proximityalert.core	193.0	2	0	2	0	41	15	1	1	0	0	321
▦	proximityalert.actions	100.0	5	0	5	0	6	8	0	0	1	1	168
▦	proximityalert	74.0	1	10	37	0	2	5	0	0	0	0	55
▦	proximityalert.views.contentPr	70.0	6	0	6	0	1	20	1	3	0	0	96
▦	proximityalert.views.listeners	69.0	4	0	4	0	9	6	1	3	0	0	106
▦	proximityalert.views	43.0	1	6	25	0	4	4	0	0	0	0	21
▦	proximityalert.views.util	10.0	1	0	1	0	0	1	0	0	0	0	16

Figure 5.6: **Package Viewer** of the Proximity Alert project. Analysis and view of the `base` package.

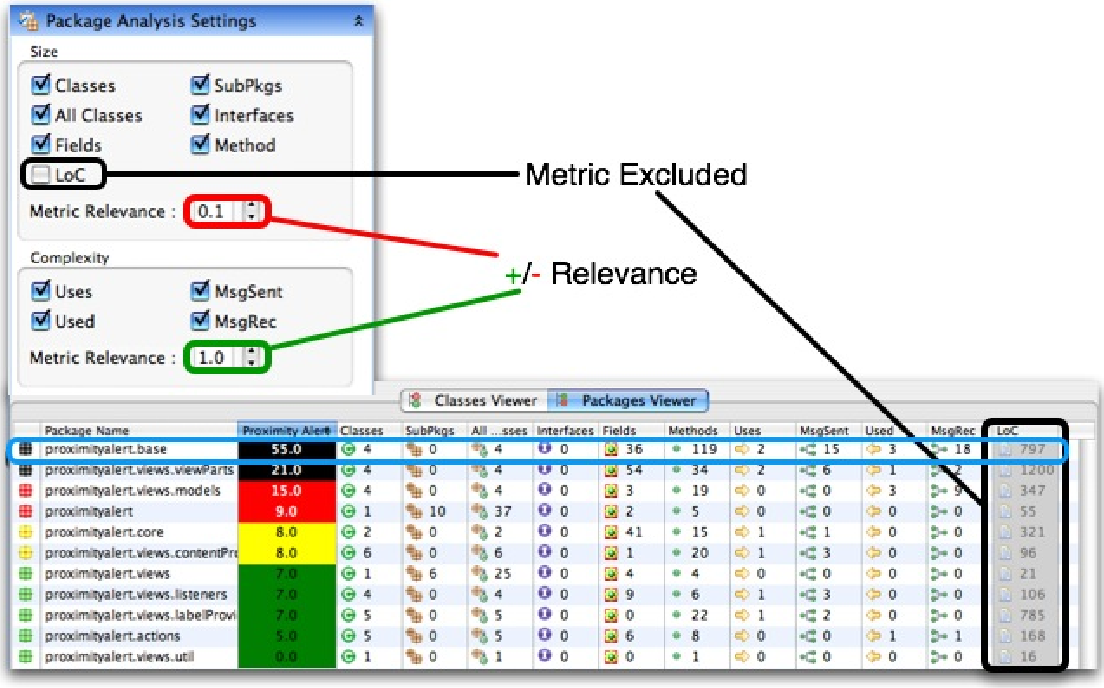
tation builded by the Proximity Alert. Since the model is used in more than one part of the system it's clear why it has so many relation with the external entities. If we look much deeper entity's metrics we can see that it provides a large number of methods but more over it has many incoming and outgoing calls. That's could be the main cause of why it results to have a very high risk. Consider that if we want to change something in this entity we have to pay attention also to the related entities.

5.3.2 Package Analysis Customization & Filtering

Assume now we want to modify the analysis of the packages and recompute the Proximity Alert metric by changing the metrics considered in the computation and by modifying their relevance in order to increase their impact in the results. The customization of the analysis allows to completely edit the calculation, but in the next releases it will be possible to edit the entire formula. This allows user to build their own Proximity Alert formula and apply it to any future analysis.

Once we decide what kind of analysis to perform we switch to the *Configuration View* and we select the *Package Analysis Settings* item which is described in the previous chapter. We decide to do an analysis that exclude the lines of code and reduce the relevance of the size metrics such as the number of classes, sub-packages, methods and increase the relevance of the complexity metrics. In this way we want to see more precisely which entity as the higher danger level considering the complexity of it, this means how is involved in term of relations with other entities and vice-cersa. As we can see in Figure 5.7 the things changes a bit from the previous analysis.

Actually we've performed a kind of analysis in order to focus on the complexity of an entity for still confirm the general analysis that we've done before. In fact the `base` package result to be the entity most complex of the system and our intuition that it may contains classes which play central role on the system is now confirmed.



Size

☑ Classes ☑ SubPkgs
☑ All Classes ☑ Interfaces
☑ Fields ☑ Method
□ LoC

Metric Relevance : 0.1

Complexity

☑ Uses ☑ MsgSent
☑ Used ☑ MsgRec

Metric Relevance : 1.0

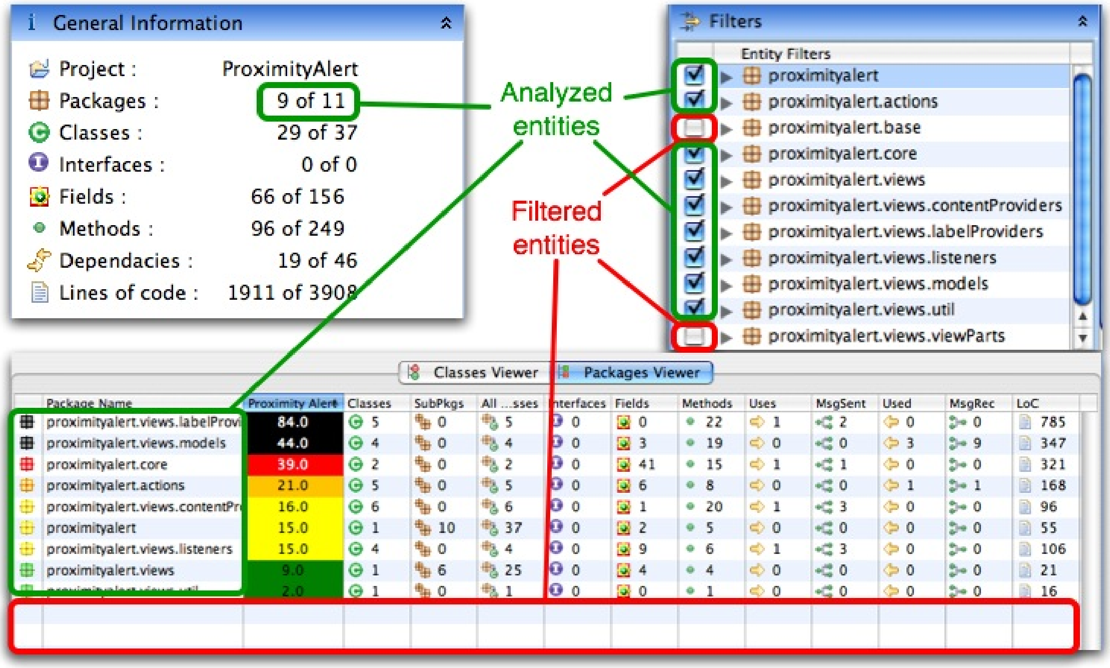
Package Name	Proximity Alert	Classes	SubPkgs	All ...sses	Interfaces	Fields	Methods	Uses	MsgSent	Used	MsgRec	LoC
proximityalert.base	55.0	4	0	4	0	36	119	2	15	3	18	797
proximityalert.views.viewParts	21.0	4	0	4	0	54	34	2	6	1	2	1200
proximityalert.views.models	15.0	4	0	4	0	3	19	0	0	3	9	347
proximityalert	9.0	1	10	37	0	2	5	0	0	0	0	55
proximityalert.core	8.0	2	0	2	0	41	15	1	1	0	0	321
proximityalert.views.contentPr	8.0	6	0	6	0	1	20	1	3	0	0	96
proximityalert.views	7.0	1	6	25	0	4	4	0	0	0	0	21
proximityalert.views.listeners	7.0	4	0	4	0	9	6	1	3	0	0	106
proximityalert.views.labelProvi	7.0	5	0	5	0	0	22	1	2	0	0	785
proximityalert.actions	5.0	5	0	5	0	6	8	0	0	1	1	168
proximityalert.views.util	0.0	1	0	1	0	0	1	0	0	0	0	16

Figure 5.7: **Package Viewer** and **Package Analysis Settings** views.

We uncheck the LoC metric in the *Package Analysis Settings* panel and we decrease the relevance for the size metrics equal to 0.1, so that the formula compute the new Proximity Alert Metric value for each entity based on the new settings. Our intent is to see the entity which has the higher alert value based on its complexity, therefore we increase the relevance of the complexity metrics equal to 1. The table is refreshed dynamically and show the new Proximity Alert Metric values.

Assume now we want to filter some entities so that we can see in the lower risk level entities which one is the most complex, for example we want to consider all packages excluding the `proximityalert.base` and the `proximityalert.views.viewParts`. If we want to filter some entity so that is not included in the analysis we have to select the *Filters* item contained in the *Configuration View* and uncheck the desired entities.Table is rebuild with new alert value and risk area.

Total of entities considered in the analysis are visible in the *General Information* panel of the same view. Figure 5.8 will show general settings provided in order to perform the analysis described.



General Information

Project :	ProximityAlert
Packages :	9 of 11
Classes :	29 of 37
Interfaces :	0 of 0
Fields :	66 of 156
Methods :	96 of 249
Dependacies :	19 of 46
Lines of code :	1911 of 3908

Filters

Entity Filters

- ☑ proximityalert
- ☑ proximityalert.actions
- □ proximityalert.base
- ☑ proximityalert.core
- ☑ proximityalert.views
- ☑ proximityalert.views.contentProviders
- ☑ proximityalert.views.labelProviders
- ☑ proximityalert.views.listeners
- ☑ proximityalert.views.models
- ☑ proximityalert.views.util
- □ proximityalert.views.viewParts

Classes Viewer | Packages Viewer

Package Name	Proximity Alert	Classes	SubPkgs	All ...sses	Interfaces	Fields	Methods	Uses	MsgSent	Used	MsgRec	LoC
proximityalert.views.labelProvi	84.0	5	0	5	0	0	22	1	2	0	0	785
proximityalert.views.models	44.0	4	0	4	0	3	19	0	0	3	9	347
proximityalert.core	39.0	2	0	2	0	41	15	1	1	0	0	321
proximityalert.actions	21.0	5	0	5	0	6	8	0	0	1	1	168
proximityalert.views.contentPr	16.0	6	0	6	0	1	20	1	3	0	0	96
proximityalert	15.0	1	10	37	0	2	5	0	0	0	0	55
proximityalert.views.listeners	15.0	4	0	4	0	9	6	1	3	0	0	106
proximityalert.views	9.0	1	6	25	0	4	4	0	0	0	0	21
proximityalert.views.util	2.0	1	0	1	0	0	1	0	0	0	0	16

Figure 5.8: **Package Viewer**, **General Information** and **Filters** views.

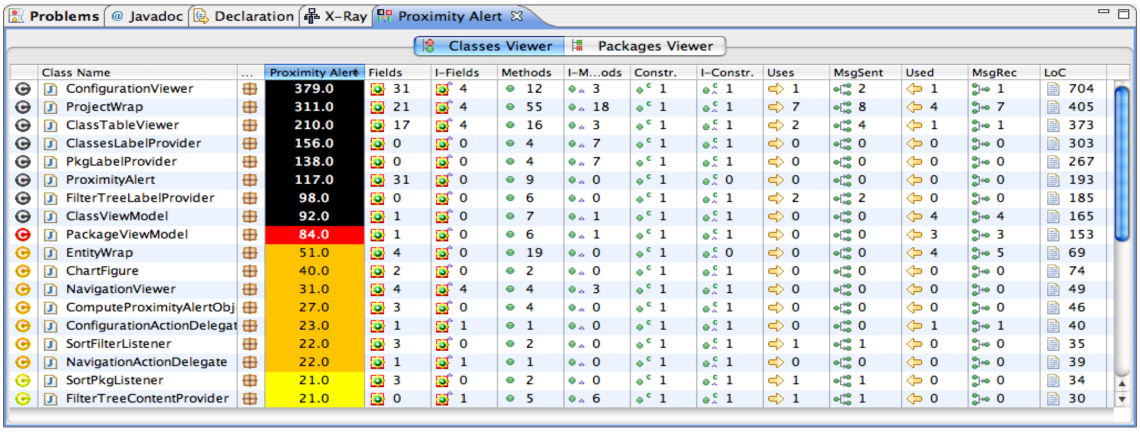
5.3.3 Classes Analysis

Consider now we want to perform at class level our analysis. In order to do that we simply select the *Classes Viewer* table available in the *Proximity Alert View*. Figure 5.9 shows the classes contained in the Proximity Alert project.

Using the *Classes Viewer* table we can further investigate on the composition of the project. It visible that the project is composed by a set of classes each of them represented in a row of the table, showing the set of metrics that describe much deeper their composition. As before the most important metrics defined in the table is the Proximity Alert Metric which is computed by the system based on the analysis settings.

By browsing the entities inside the table we've the possibility to have general impression about the composition of them. In this case all the metrics available are taken into account in the analysis. Thanks to the Box-plot technique we have color depiction which help us visualize better entities and their danger level. Given the list of entities and the depiction we can assume which are the most involved in the system and which have an high risk level, which means that affecting changes in those entities could affect behaviors and relations with the other entities directly dependent or just linked. As before we came back to the table to interpret better this concept. We also use the X-Ray plug-in to give a graphical support to the tables.

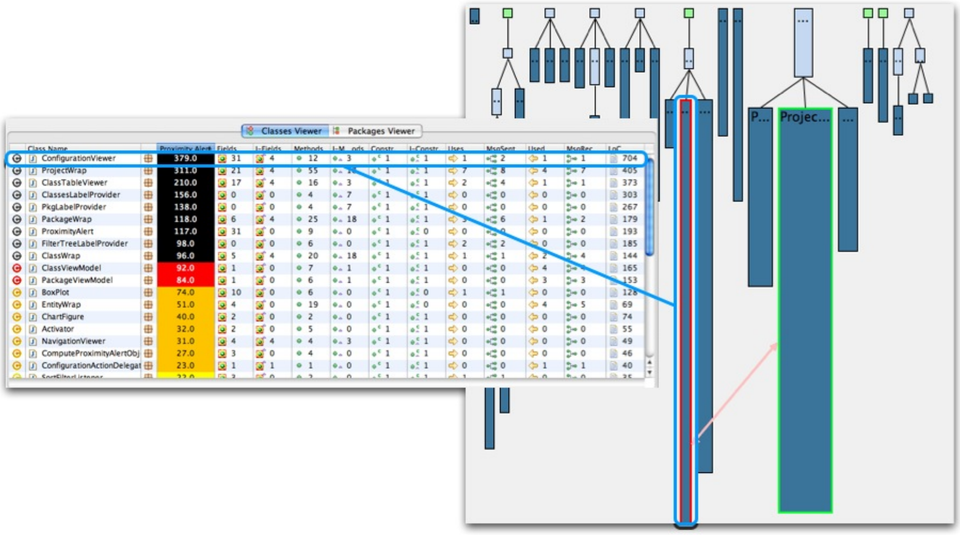
Let us analyze a class which has a very high proximity alert value, for example the `ConfigurationViewer` class. The very high danger level is given, in this case, because of the values of the metrics considered in the analysis, more over by looking at the value of them it visible that this class has a large number of lines of code and also a large number of fields declared.



Class Name	...	Proximity Alert	Fields	I-Fields	Methods	I-M...ods	Constr.	I-Constr.	Uses	MsgSent	Used	MsgRec	LoC
ConfigurationViewer		379.0	31	4	12	3	1	1	1	2	1	1	704
ProjectWrap		311.0	21	4	55	18	1	1	7	8	4	7	405
ClassTableViewer		210.0	17	4	16	3	1	1	2	4	1	1	373
ClassesLabelProvider		156.0	0	0	4	7	1	1	0	0	0	0	303
PkgLabelProvider		138.0	0	0	4	7	1	1	0	0	0	0	267
ProximityAlert		117.0	31	0	9	0	1	0	0	0	0	0	193
FilterTreeLabelProvider		98.0	0	0	6	0	1	1	2	2	0	0	185
ClassViewModel		92.0	1	0	7	1	1	1	0	0	4	4	165
PackageViewModel		84.0	1	0	6	1	1	1	0	0	3	3	153
EntityWrap		51.0	4	0	19	0	1	0	0	0	4	5	69
ChartFigure		40.0	2	0	2	0	1	1	0	0	0	0	74
NavigationViewer		31.0	4	4	4	3	1	1	0	0	0	0	49
ComputeProximityAlertObj		27.0	3	0	4	0	1	1	0	0	0	0	46
ConfigurationActionDelegat		23.0	1	1	1	0	1	1	0	0	1	1	40
SortFilterListener		22.0	3	0	2	0	1	1	1	1	0	0	35
NavigationActionDelegate		22.0	1	1	1	0	1	1	0	0	0	0	39
SortPkgListener		21.0	3	0	2	0	1	1	1	1	0	0	34
FilterTreeContentProvider		21.0	0	1	5	6	1	1	1	1	0	0	30

Figure 5.9: **Classes Viewer** of the Proximity Alert project. By simply clicking on the table headers available the table dynamically sorts component allowing faster and simpler view of the entities contained.

Figure 5.10 shows the entity in the table and its representation in the *Complexity View* of the X-Ray plug-in.



Class Name	...	Proximity Alert	Fields	I-Fields	Methods	I-M...ods	Constr.	I-Constr.	Uses	MsgSent	Used	MsgRec	LoC
ConfigurationViewer		379.0	31	4	12	3	1	1	1	2	1	1	704
ProjectWrap		311.0	21	4	55	[illegible]	1	1	7	8	4	7	405
ClassTableViewer		210.0	17	4	16	3	[illegible]	1	2	4	1	1	373
ClassesLabelProvider		156.0	0	0	4	7	1	1	0	0	0	0	303
PkgLabelProvider		138.0	0	0	4	7	1	1	0	0	0	0	267
PackageWrap		118.0	6	4	25	18	1	1	3	6	1	2	179
ProximityAlert		117.0	31	0	9	0	1	0	0	0	0	0	193
FilterTreeLabelProvider		98.0	0	0	6	0	1	1	2	2	0	0	185
ClassWrap		96.0	5	4	20	18	1	1	1	1	2	4	144
ClassViewModel		92.0	1	0	7	1	1	1	0	0	4	[illegible]	165
PackageViewModel		84.0	1	0	6	1	1	1	0	0	3	3	153
BoxPlot		74.0	10	0	6	0	1	0	1	1	0	0	128
EntityWrap		51.0	4	0	19	0	1	0	0	0	4	5	69
ChartFigure		40.0	2	0	2	0	1	1	0	0	0	0	74
Activator		32.0	2	0	5	0	1	1	0	0	0	0	55
NavigationViewer		31.0	4	4	4	3	1	1	0	0	0	0	49
ComputeProximityAlertObj		27.0	3	0	4	0	1	1	0	0	0	0	46
ConfigurationActionDelegat		23.0	1	1	1	0	1	1	0	0	1	1	40
SortFilterListener		22.0	3	0	2	0	1	1	1	1	0	0	35

Figure 5.10: **Classes Viewer** of the Proximity Alert project. Analysis and view of the `ConfigurationViewer` class.

By looking at the metrics values of the `ConfigurationViewer` we can observe that this entity is does not have too many relations, in fact it communicates only with one other entity the `ProjectWrap` class. Given this dependency and the names it becomes clear that this entity is responsible for creating and maintaining the *Configuration View* of our plug-in. It is therefore legal a collaboration with the class mentioned and its very high danger level is given because it means that changes inside this class could be dangerous. Metrics in this case play a central role, in fact they can help us to understand the entity composition and by following design rules we can detect identity disharmonies [LM06].

If we look also at the code, by just double-clicking on the class, it's visible that the code is not so permissive for changes and ease addition of new features. Our analysis highlighted that this class can become potentially a God Class, which tends to centralize the intelligence of the system performing too much work on its own, delegating only minor details to a set of trivial classes and using the data from other classes. This has a negative impact on the reusability and the understandability of that part of the system [LM06].

Therefore, to have a better design, we should consider to re-design this class and divide it into multiple classes,each of them with its responsibility and behavior.

5.3.4 Classes Analysis Customization & Filtering

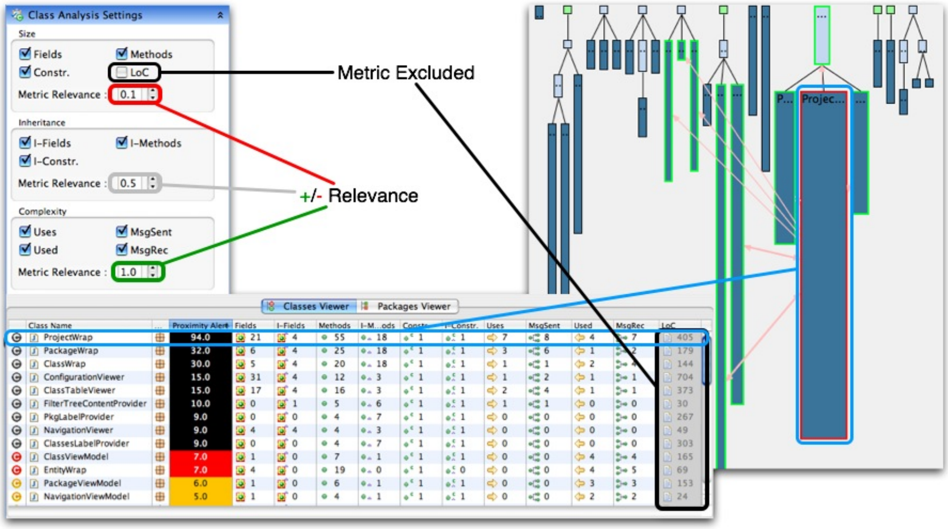
Assume now we want to modify the analysis of the classes and recompute the Proximity Alert Metric by changing the metrics considered in the computation and by modifying their relevance in order to increase their impact on the results.

Once we've decide what kind of analysis we want to perform we switch to the *Configuration View* and we select the *Classes Analysis Settings* item which is described in the previous chapter. We decide to do an analysis that exclude the lines of code and reduce the relevance of the size metrics such as the number of fields, methods and increase the relevance of the complexity metrics. In this way we want to see more precisely which entity as the higher danger level considering its complexity, meaning how it is involved in term of relation white the other entities and vice-cersa. As we can see in Figure 5.11 the things changes again from the previous analysis.

Actually we perform a kind of analysis focusing on the complexity of an entity in order to confirm the general analysis that we've done in the package analysis. In fact the `ProjectWrap` class is an entity of the `base` package which in the similar analysis done previously results to be the entity most complex of the system, therefore our intuition that the `base` package may contains classes which play central role on the system is again confirmed.

We uncheck the LoC metric in the *Classes Analysis Settings* panel and we decrease the relevance for the size metrics to 0.1, so that the formula will compute the new Proximity Alert Metric value for each entity based on the new settings. Our intent is to see the entity which has the higher danger level based on its complexity, therefore we increase the relevance of the complexity metrics to 1. The table is refreshed dynamically and shows the new Proximity Alert values.

Assume now we want to analyze only a certain set of classes, let say we want to analyze only the entities contained in the `base` and `core` packages. If we want to filter some entity so that is not included in the analysis we have to select the *Filters* item contained in the *Configuration View* and uncheck the desired entities.Table is rebuild with new alert value and risk area. Total of entities considered in the analysis are visible in the *General Information* item on the same view. Figure 5.12 shows the *Classes Viewer*, *General Information* and *Filters* with the performed settings involved in the analysis.



Class Analysis Settings

Size

☑ Fields ☑ Methods
☑ Constr. □ LoC
Metric Relevance : 0.1

Inheritance

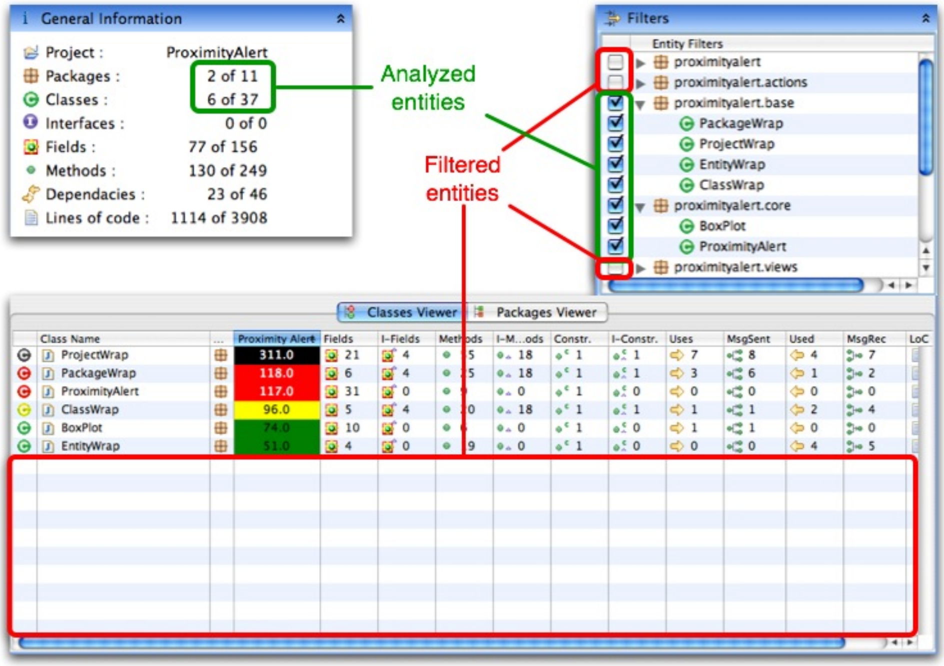
☑ I-Fields ☑ I-Methods
☑ I-Constr.
Metric Relevance : 0.5

Complexity

☑ Uses ☑ MsgSent
☑ Used ☑ MsgRec
Metric Relevance : 1.0

Class Name	...	Proximity Alert	Fields	I-Fields	Methods	I-M...ods	Constr.	I-Constr.	Uses	MsgSent	Used	MsgRec	LoC
ProjectWrap		94.0	21	4	55	18	1	1	7	8	4	7	405
PackageWrap		32.0	6	4	25	18	1	1	3	6	1	2	179
ClassWrap		30.0	5	4	20	18	1	1	1	1	2	4	144
ConfigurationViewer		15.0	31	4	12	3	1	1	1	2	1	1	704
ClassTableViewer		15.0	17	4	16	3	1	1	2	4	1	1	373
FilterTreeContentProvider		10.0	0	1	5	6	1	1	1	1	0	0	30
PkgLabelProvider		9.0	0	0	4	7	1	1	0	0	0	0	267
NavigationViewer		9.0	4	4	4	3	1	1	0	0	0	0	49
ClassesLabelProvider		9.0	0	0	4	7	1	1	0	0	0	0	303
ClassViewModel		7.0	1	0	7	1	1	1	0	0	4	4	165
EntityWrap		7.0	4	0	19	0	1	0	0	0	4	5	69
PackageViewModel		6.0	1	0	6	1	1	1	0	0	3	3	153
NavigationViewModel		5.0	1	0	4	1	1	1	0	0	2	2	24

Figure 5.11: **Classes Viewer** of the Proximity Alert project. Analysis and view of the `ProjectWrap` class.



General Information

Project :	ProximityAlert
Packages :	2 of 11
Classes :	6 of 37
Interfaces :	0 of 0
Fields :	77 of 156
Methods :	130 of 249
Dependacies :	23 of 46
Lines of code :	1114 of 3908

Filters

Entity Filters

□ proximityalert
□ proximityalert.actions
☑ proximityalert.base
☑ PackageWrap
☑ ProjectWrap
☑ EntityWrap
☑ ClassWrap
☑ proximityalert.core
☑ BoxPlot
☑ ProximityAlert
□ proximityalert.views

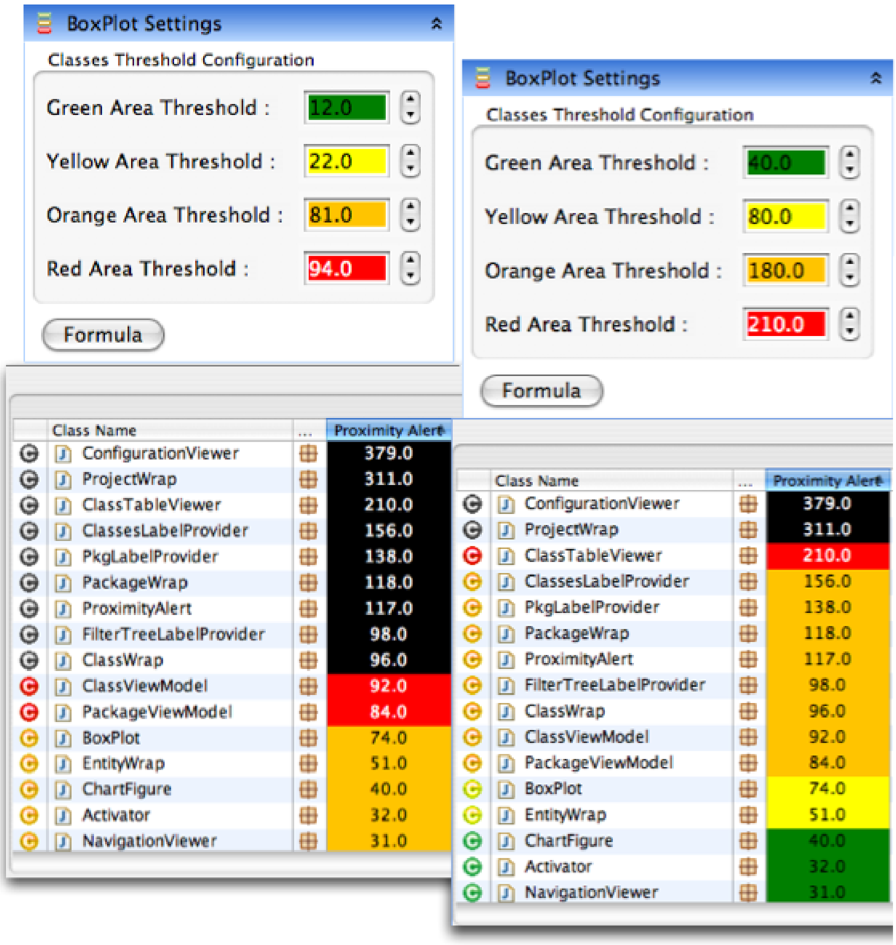
Class Name	...	Proximity Alert	Fields	I-Fields	Methods	I-M...ods	Constr.	I-Constr.	Uses	MsgSent	Used	MsgRec	LoC
ProjectWrap		311.0	21	4	[illegible]	18	1	1	7	8	4	7	
PackageWrap		118.0	6	4	[illegible]	18	1	1	3	6	1	2	
ProximityAlert		117.0	31	0	[illegible]	0	1	0	0	0	0	0	
ClassWrap		96.0	5	4	[illegible]	18	1	1	1	1	2	4	
BoxPlot		74.0	10	0	[illegible]	0	1	0	1	1	0	0	
EntityWrap		51.0	4	0	[illegible]	0	1	0	0	0	4	5	

Figure 5.12: **Classes Viewer**, **General Information** and **Filters** views.

5.3.5 BoxPlot Settings

Is possible that Proximity Alert Metric value differs a lot because of the computation and the value of the metrics taken into account. This can cause big gap between values of classes and the box-plot technique could depict in high danger level also classes which are indeed very easy and does not have a big involvement in the system, such Data classes. We decide therefore to allowing modification on the box-plot's thresholds in order to group class also following customization of the user.

Figure 5.13 shows how the table changes when new values in the *BoxPlot Settings* panel are inserted for each color area thresholds, in this case we consider the class level.



BoxPlot Settings

Classes Threshold Configuration

Green Area Threshold : 12.0

Yellow Area Threshold : 22.0

Orange Area Threshold : 81.0

Red Area Threshold : 94.0

Class Name	...	Proximity Alert
ConfigurationViewer		⬛ 379.0
ProjectWrap		⬛ 311.0
ClassTableViewer		⬛ 210.0
ClassesLabelProvider		⬛ 156.0
PkgLabelProvider		⬛ 138.0
PackageWrap		⬛ 118.0
ProximityAlert		⬛ 117.0
FilterTreeLabelProvider		⬛ 98.0
ClassWrap		⬛ 96.0
ClassViewModel		🟥 92.0
PackageViewModel		🟥 84.0
BoxPlot		🟧 74.0
EntityWrap		🟧 51.0
ChartFigure		🟧 40.0
Activator		🟧 32.0
NavigationViewer		🟧 31.0

BoxPlot Settings

Classes Threshold Configuration

Green Area Threshold : 40.0

Yellow Area Threshold : 80.0

Orange Area Threshold : 180.0

Red Area Threshold : 210.0

Class Name	...	Proximity Alert
ConfigurationViewer		⬛ 379.0
ProjectWrap		⬛ 311.0
ClassTableViewer		🟥 210.0
ClassesLabelProvider		🟧 156.0
PkgLabelProvider		🟧 138.0
PackageWrap		🟧 118.0
ProximityAlert		🟧 117.0
FilterTreeLabelProvider		🟧 98.0
ClassWrap		🟧 96.0
ClassViewModel		🟧 92.0
PackageViewModel		🟧 84.0
BoxPlot		🟨 74.0
EntityWrap		🟨 51.0
ChartFigure		🟩 40.0
Activator		🟩 32.0
NavigationViewer		🟩 31.0

Figure 5.13: **Classes Viewer** of the Proximity Alert project. On the left there's the initial table and on the right the new one. Settings of the Box-plot threshold are visible on top of them.

Chapter 6

Conclusions

6.1 Achieved Goals

So far the plug-in provides the following features:

- *System analysis and metrics extension*
- *Proximity Alert view*
- *Configuration view*

At this time these functionalities result to be useful for analyzing small and medium sized projects. Thanks to the integration and extension of the X-Ray meta-model our system give a basic and relevant support of system analysis.

With the *Proximity Alert View* the user can access to the project content such as classes and packages and perform set of action which could be useful to browse easier entities analyzed. By using the *Configuration View* the user can manually configure the analysis such as check/uncheck metrics and modify their relevance index, customize the box-plot thresholds and also filters the set of entities for the analysis. Products of each action performed by the user are shown dynamically in order to improve user responsiveness and interactions.

6.2 Future Work

For the future releases of the system, we will refactor the code, fix the known bugs and manage visualization limitations, and obviously add more features. Our intent is to give to the user the most detailed model of a given project, therefore we decide not only to extend the Proximity Alert plug-in but also collaborate with the extension of the model of X-Ray by performing an active cooperation between the two models.

6.2.1 Model Analysis

In the next releases the model will perform deeper analysis and more metrics definition which could be useful for the scope of our system. The intention is also to extend the model by not only giving an analysis of the package and class entities but also at the method layer. By performing such a deep scan could be also possible to consider the body implementation of the methods.

6.2.2 Metrics

By having a more deeper analysis we are able to define more metrics measures by also integrating the most common software metrics. We want also integrate a full customizable system of metrics creation in which user can define metrics store them and reuse by applying them to any analyzed system. This new features will certainly increase the analysis details.

6.2.3 View Functionalities

The views in the next releases the views will also have more functionalities which allows users to perform any kind of operations from any point. All tables will perform more selection actions in order to simplify plug-in usage. The *Configuration View* will give more customization actions.

By collaborating with the creator of the X-Ray plug-in our intent is to reach a fully contribution not only for the models but also by interacting with the views.

6.2.4 Analysis Customization

The most interesting feature that we want to add in the future releases is also the full customization of the Proximity Alert formula which define the value of the Proximity Alert Metric. The user will be unlimited to perform any kind of operation on the defined metrics and also define new ones which can be integrated in the calculation.

6.3 Bugs & Limitation

So far we know just these bugs and limitation of our system and we can summarize them in these categories:

- *Scalability* - Since this is a first implementation we need to work more on the scalability by performing a deeper analysis and performing more test that can increase reliability and can be helpful to analyze high-sized projects.
- *Visualization* - With very big project view often get stuck because of refreshing problems which is managed by SWT in a non always comprehensive way. Also scroll bars, for unknown reasons and under certain situations, doesn't appear but by refreshing the whole Eclipse UI all get its normal behavior.

Appendix A

Implementation

A.1 Proximity Alert core

The core of the project is composed by the Proximity Alert model which is composed by some classes which is composed by the internal code representation of X-Ray plug-in extended by wrapper which perform deeper description of the entities.

A.1.1 Class description

- `EntityWrap`

  Abstract class which represents an abstract entity. It contains all attributes which can be abstract for any possible concrete entity. It use the `Proximity Alert` class in order to compute the value of the metric.

- `ClassWrap`

  Concrete class which extends `EntityWrap` class. In the internal code representation it has the role of the class entity which perform metric extensions.

- `PackageWrap`

  Concrete class which extends `EntityWrap` class. In the internal code representation it has the role of the package entity which perform metric extensions. It is responsible to perform analysis in the classes which compose it by considering entity visibility.

- `ProjectWrap`

  Concrete class which extends `EntityWrap` class. In the internal code representation it has the role of the project entity which perform metric extensions, handle entities visibility and keep trace of the whole internal code representation. It collaborate with the `BoxPlot` class in oder to perform depiction in danger areas of each entity considered in the analysis.

- `BoxPlot`

  This class define the Box-plot techinique which performs operations in order to depict entities in group areas based on the Proximity Alert Metrics of a given set of entities considered for the analysis.

- `ProximityAlert`

  This class takes an `EntityWrap` as input and perform calculation on all the set of considered metrics and their relevances and the result will be set to the given entity as proximity alert metric value.

A.1.2 UML

Class diagram

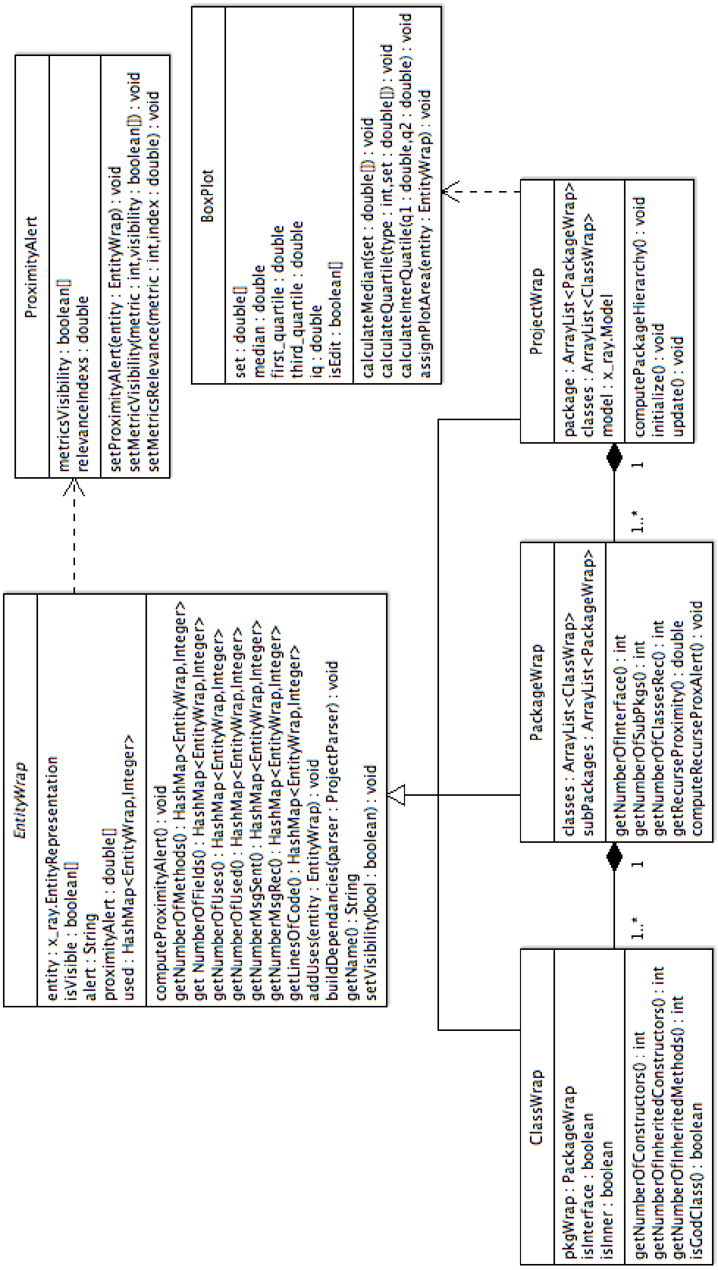

ProximityAlert
metricsVisibility : boolean[]
relevanceIndexs : double
setProximityAlert(entity : EntityWrap) : void
setMetricVisibility(metric : int,visibility : boolean[]) : void
setMetricsRelevance(metric : int,index : double) : void
BoxPlot
set : double[]
median : double
first_quartile : double
third_quartile : double
iq : double
isEdit : boolean[]
calculateMedian(set : double[]) : void
calculateQuartile(type : int,set : double[]) : void
calculateInterQuatile(q1 : double,q2 : double) : void
assignPlotArea(entity : EntityWrap) : void
ProjectWrap
package : ArrayList<PackageWrap>
classes : ArrayList<ClassWrap>
model : x_ray.Model
computePackageHierarchy() : void
initialize() : void
update() : void
1
1..*
EntityWrap
entity : x_ray.EntityRepresentation
isVisible : boolean[]
alert : String
proximityAlert : double[]
used : HashMap<EntityWrap,Integer>
computeProximityAlert() : void
getNumberOfMethods() : HashMap<EntityWrap,Integer>
get NumberOfFields() : HashMap<EntityWrap,Integer>
getNumberOfUses() : HashMap<EntityWrap,Integer>
getNumberOfUsed() : HashMap<EntityWrap,Integer>
getNumberMsgSent() : HashMap<EntityWrap,Integer>
getNumberMsgRec() : HashMap<EntityWrap,Integer>
getLinesOfCode() : HashMap<EntityWrap,Integer>
addUses(entity : EntityWrap) : void
buildDependancies(parser : ProjectParser) : void
getName() : String
setVisibility(bool : boolean) : void
PackageWrap
classes : ArrayList<ClassWrap>
subPackages : ArrayList<PackageWrap>
getNumberOfInterface() : int
getNumberOfSubPkgs() : int
getNumberOfClassesRec() : int
getRecurseProximity() : double
computeRecurseProxAlert() : void
1
1..*
ClassWrap
pkgWrap : PackageWrap
isInterface : boolean
isInner : boolean
getNumberOfConstructors() : int
getNumberOfInheritedConstructors() : int
getNumberOfInheritedMethods() : int
isGodClass() : boolean

Figure A.1: **Class Diagram** of the Proximity Alert plug-in core model.

Sequence diagram

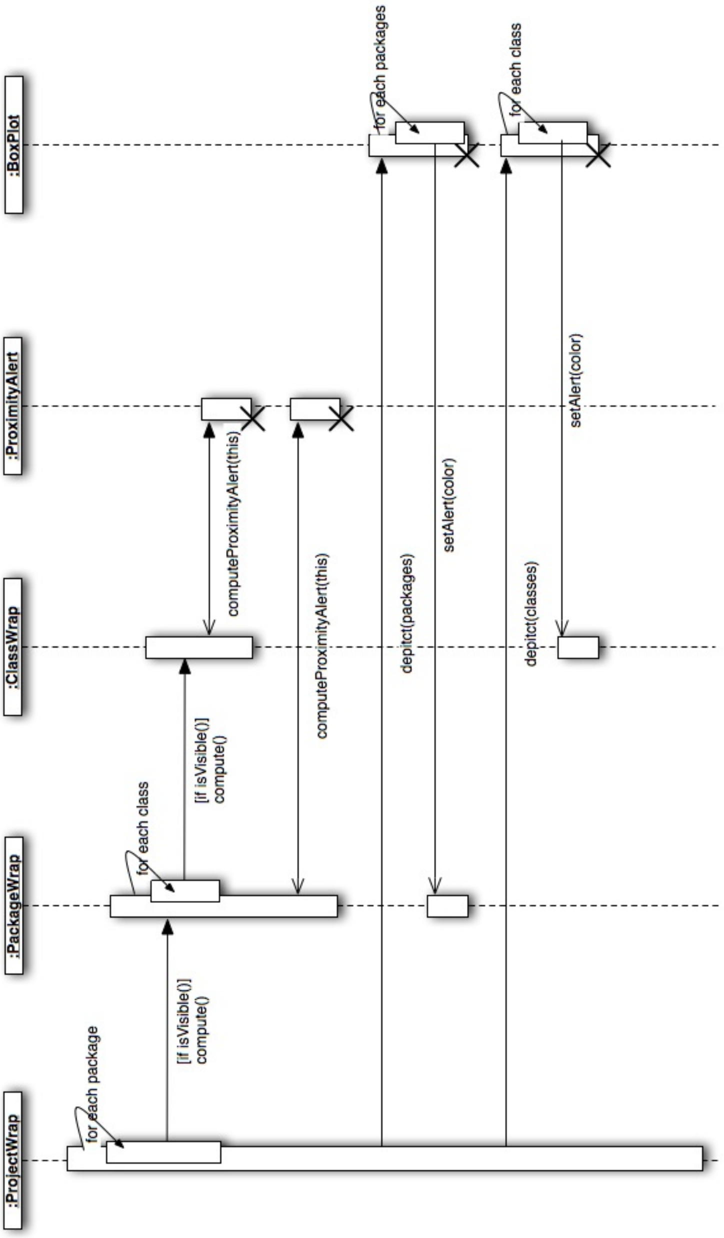


Figure A.2: **Sequence Diagram** of the Proximity Alert computation of the analysis.

List of Figures

List of Tables

Bibliography

[BDW06] Gary Pollice Brett D.McLaughilin and Dave West. *Head First Object-Oriented Analysis & Desing.* O'Reilly Media, Inc., 1st edition, 2006.

[CR06] Eric Clayberg and Dan Rubel. *Eclipse - Building Commercial-Quality Plug-ins.* Addison Wesley, 2nd edition, 2006.

[KJM89] Lieberherr Karl J and Ian M.Holland. Assuring good style for object-oriented programs. *IEEE Software*, 1989.

[LD02] Michele Lanza and Stephane Ducasse. Beyond language independent object-oriented metrics: Model independent metrics. *QAOOSE*, 2002.

[LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice - Using Softwaare Metrics to Characterize, Evaluate and Improve the Design of Object-Oriented Systems.* Springer, 2006.

[Mal07] Jacopo Malnati. X-ray, an eclipse plug-in for software visualization. Master's thesis, University of Lugano, 2007.