Università
della
Svizzera
italiana

Facoltà
di scienze
informatiche

Bachelor Project

X-Ray

An Eclipse Plug-in for Software Visualization

Jacopo Malnati

supervised by
Prof. Dr. Michele Lanza
Dipl. Ing. Mircea Lungu

Abstract

Software is inherently intangible. Systems can be composed of a huge amount of software entities linked together by different kinds of dependencies. Software designers use visualization tools in order to raise the level of abstraction and reduce the amount of information to the one needed. Most of these tools are stand-alone programs, that force the user to switch between different windows and contexts. This context switch represents a problem, being time consuming and forcing the user to download, install and use, tools or systems external to his favorite code editor. Developing an open-source software visualization plug-in for the Eclipse framework represents a significant step towards bringing the visualization tools in the forward engineering process.

Acknowledgments

First of all, I would like to thank Michele Lanza, one of my preferred and most appreciated professors, as far as the advisor of my bachelor project. Three years ago he started to guide me through the complexity and beauty of software systems, sharing some of his precious knowledge. Also, many thanks to Mircea Lungu, my co-advisor. Thanks for being around and spending so much of your time sitting next to me or providing constructive feedbacks.

Finally I would like to thank my girl-friend Giulia, for being so understanding during the long hours while I was coding and writing.

Jacopo Malnati

July 2, 2007

Contents

Chapter 1

Introduction

Programming, and object-oriented programming in particular, is about defining a vocabulary that will help to express a complex problem in a much simpler way. Object-oriented design provides a good way to express systems in a simpler way, but it raises different questions. While designing and developing an object-oriented system we have to decide how the entities will interact with each other, how and if the vocabulary will change or be extended, etc. [LM06].

Software is intangible and its design is about creating abstractions. Therefore is more difficult to asses if the outcome is the desired one. Moreover its design and implementation implies human communication that might become complex as soon as a group of people tries to describe and understand the same abstract concepts. In addition, software systems are not immutable artifacts. Software evolves and changes, that implies that its design is revised and improved continuously. While evolving, systems become more and more complex and by simply reading their source code it becomes impossible to analyze and understand them.

Software visualization is an important aid in software engineering and reverse engineering. Graphically visualizing abstract concepts under different points of view, software visualization provides a way to reduce the complexity of the underlying system, making it easier to be understood and analyzed.

So far, almost all software visualization systems that have been implemented are stand-alone tools, thus we thought about creating a software visualization plug-in for the Eclipse framework (www.eclipse.org). 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 created different views based on different abstractions provided by a set of metrics computed on the project's source code. Although being just a prototype, our X-Ray plug-in can already be used to analyze small and medium sized project, offering the user the possibility to understand and browse any system without being forced to use stand-alone tools or reading source code.

1.1 Goal of the Work

The goal of the work is twofold:

- *Experiment with Eclipse plug-in development.* Eclipse is made up by a small kernel and hundreds, thousands of plug-ins that, together, build its development environment. Being able to contribute 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 framework.
- *Build an eclipse plugin for software visualization.* Once we experimented with the plug-in development, we built on top of our tests the X-Ray plug-in. It provides different views and abstraction to better understand any system.

1.2 Roadmap

In the next chapters we will go through scientific visualizations and software visualization concepts. We will then describe the Eclipse framework and its structure, how to create a plug-in and how to provide its functionalities to Eclipse. Once we described the basis of software visualization and the Eclipse structure, we will introduce our plug-in: X-Ray. There will be a discussion about metrics used by the plug-in and its polymetric views. To better understand his functionalities we will validate it on itself and, then, draw some conclusion about the reached goals and the future work. Finally, there will be a section about design decisions and interesting implementation issues that we faced during the development of X-Ray.

Chapter 2

Related Work

2.1 Information Visualization

A visualization is something that can be seen and observed. Any visualization can be useful to model and represent an abstract entity, making evident attributes or any meaningful property of the visualized object.

Most scientists state that the human brain has two different processing systems which are placed on both hemispheres; they believe that our brain has a visual processing system ("right brain") and a verbal one ("left brain"). The human communication, therefore, makes extensive use of the "left brain" (while the "right brain" becomes less used from our childhood).

Even if our visual processing system tends to be less used than the one responsible for verbal communication, temporal and sequential analysis, the "right brain" still has an important role while we have to face non verbal, synthetic, non temporal and parallel analysis. There is a common proverb that summarizes this concept: *a picture is worth a thousand words*. This piece of popular wisdom reflects the ability of our brain (the right hemisphere) to process parallel information in short time intervals.

A nice example are weather forecasts. With a quick look, you can clearly identify the borders of your country and symbols associated with different kinds of weather conditions (the sun, clouds, etc.). It will take a short time before you will have a complete overview and knowledge of the whole forecast, moreover if the forecast contains some little animations, you will be able to understand the weather's predicted trend. The same knowledge will take a longer time if you have to read (or listen to) the same information in a textual way.

2.1.1 Scientific Visualization

While handling huge amount of data or working on invisible forces, the possibility to graphically represents them and highlight useful attributes becomes an indispensable tool for scientific research.

Data and event visualization is not only a way of communicating results, but also an excellent tool for discovering trends and property "hidden" in the amount of data or simply invisible to human eyes. Today there is a wide set of visualizations used in scientific research and they vary from bi-dimensional representation to n-dimensional ones, using different techniques depending on the context.

The following are two examples of visualizations. They represent the past and the present of visualization, from bi-dimensional, black and white, hand-crafted visualization to three-dimensional, interactive, computer aided models.

Example. Figure 2.1 is the map by Charles Joseph Minard (1781-1870), published in 1861, shows the advance of Napoleon's Grande Arme into Russia in 1812. It maps the thickness of the line as the strength of the army, with numbers indicating strength at critical points. It's important to notice that even if it's just a black and white representation, draw by hand on a sheet of paper, it clearly shows the strength of the army during the advance and retreat of Napoleon as far as temperatures and some geographical locations.

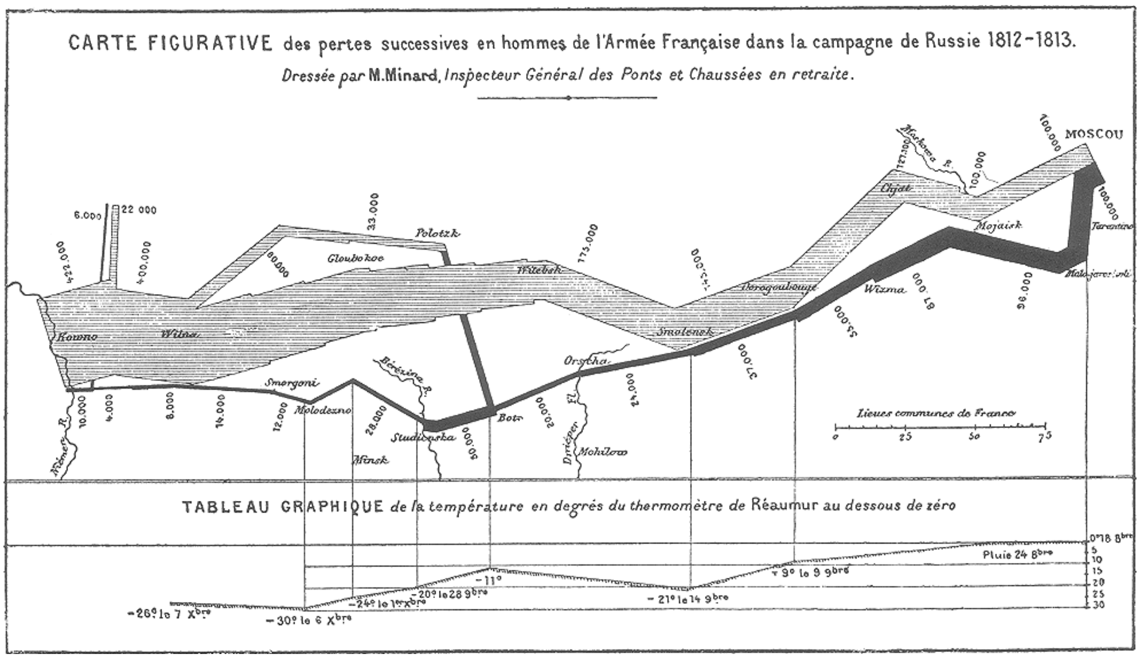

CARTE FIGURATIVE des pertes successives en hommes de l'Armée Française dans la campagne de Russie 1812-1813.
Dressée par M. Minard, Inspecteur Général des Ponts et Chaussées en retraite.
MOSCOU
Kowno
Wilna
Smolensk
Polotzk
Witebsk
Orscha
Mojaisk
Wizma
Smorgoni
Molodezno
Minsk
Studienska
Bobr
Mohilow
Lieues communes de France
TABLEAU GRAPHIQUE de la température en degrés du thermomètre de Réaumur au dessous de zéro
Pluie 24 8bre
-26° le 7 Xbre
-30° le 6 Xbre
-21° le 14 9bre
-9° le 9 9bre
-11°
-20° le 28 9bre
-24° le 1er Xbre

Figure 2.1: Minard: advance of Napoleon's Grande Arme into Russia in 1812

Due to the improvements in the computer science field and especially in its graphics branch, scientists can now interpret potentially huge quantities of laboratory or simulation data or the results from sensors out in the field with complex multi-dimensional, interactive visualization (even reproduced on different types of devices).

Example. The next figure (Figure 2.2) is a three-dimensional representation of a complex molecule. It's a single image captured during an interactive, real-time visualization of the molecule and its iteration with external entities. A sapient use of colors, light and border, can be very helpful to detect more clearly atom intersections and the overall structure of the whole molecule.

This kind of simulations lets scientist to reproduce and study phenomena that would be impossible to study without scientific visualization (and simulation). The possibility to easily navigate in a three-dimensional space, zooming in and out, interacting with the molecule, gives the scientist a new, powerful tool.

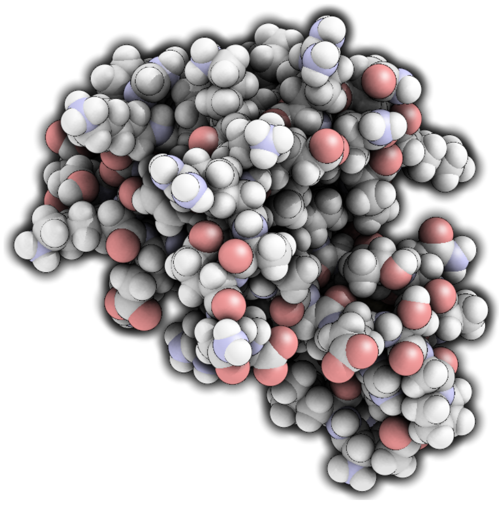


Figure 2.2: A 3-dimensional representation of a complex molecule

2.1.2 Visualization of Abstract Concepts

Presenting abstract, invisible, information artifacts is difficult because it is not about representing physical objects (that already have a graphical representation in our minds). Abstract visualization is presenting a logical construction rather than a physical one. One of the main questions in this kind of visualization is how to map abstract entities and relationships to a graphical, meaningful representation. Moreover the complexity lies also in the context in which the visualization will be used, in fact there are more than one feasible and meaningful ways of modeling the same thing, but one could be more appropriate to a context (designing, understanding, etc.) than another.

Coming back to the weather forecast example, let us discuss how the forecast is mapped to a graphical representation. Every television channel has different layouts and graphics for their forecasts, but it is really easy to find a pattern among them. Every whether forecast basically models the country as a topographic map and weather conditions as little symbols representing, say, the sun, the moon, clouds, lightnings etc.. Given that they have to represents physical objects that we all well know from our every-day life experience, it is easy to represent them as commonly accepted symbols (i.e., the sun is a yellow disk with multiple lines coming out of his circumference, simulating ray-lights. Even if this is not what we actually see looking at the sun, this is one of the common symbolic representation that our culture gave to the sun). It is therefore easier to model something that already has a widely accepted symbol than something that is abstract.

Looking closer to the forecast, it might be noticed that even if it is absolutely possible that in different topological positions there are different set of clouds, our solar system has just one sun, therefore the multiple sun symbols positioned on different locations cannot make sense in a realistic representation; even a simple weather forecast is using abstraction and rules outside our every-day life experience in order to give more, parallel informations.

2.2 Software Visualization

Programming language researchers tried to design programming language structured in order to make it easier to use and understand them, but they have been limited by the structure of the western alphabet and the ASCII code. Software visualization has the same goal of improving usability and understandability, but it doesn't suffer from the same limitations, given that it can use different, parallel and graphical way of modeling and representing the same (and new) concepts. Software visualization is concerned with static or animated multi-dimensional representation of software entities. It uses graphics and animations in order to show data, code, control flows, classes and dependencies among them (algorithms animation has been separated by all of the previous visualizations). It exploits the human visual perception system in order to provide what's lacking in a flat, sequential, text-based representation. The programmer and designer will have therefore a new visualization other than lines of code displayed in a textual editor.

There are multiple dimensions along which visualizations can be classified:

- Scope: Visualizing a single component vs. whole system
- What is visualized: Animate an algorithm vs. model the static structure of a system
- Goal: Understanding a system (reverse engineering) or quality assurance (hunting defects)

Especially in object-oriented software systems this kind of visualization is useful to model element-specific properties (i.e., the content of a class) or system-specific properties (i.e., dependencies among classes, hierarchies, packages composition etc.). The possibility to browse a huge system and quickly spot defects highlighted by the visualization or having the possibility to understand the "shape" of the system to quickly discover dependencies and structures is a key feature in terms of understandability and maintenance.

But, how can the source code and its semantic meaning be mapped to a graphical representation?

Software is made up by lines of code, encoded as 0s and 1s laying on a storing device. There is no symbolic representation, there nothing in our daily experience close to the software reality. As previously stated, when a physical object has to be modeled, usually a well known symbol or ideogram is associated to that object but when, like in the case of software, the thing to model is intangible and abstract, without any physical form, the graphical representation tends to be more complex to design.

2.3 Software Visualization Tools

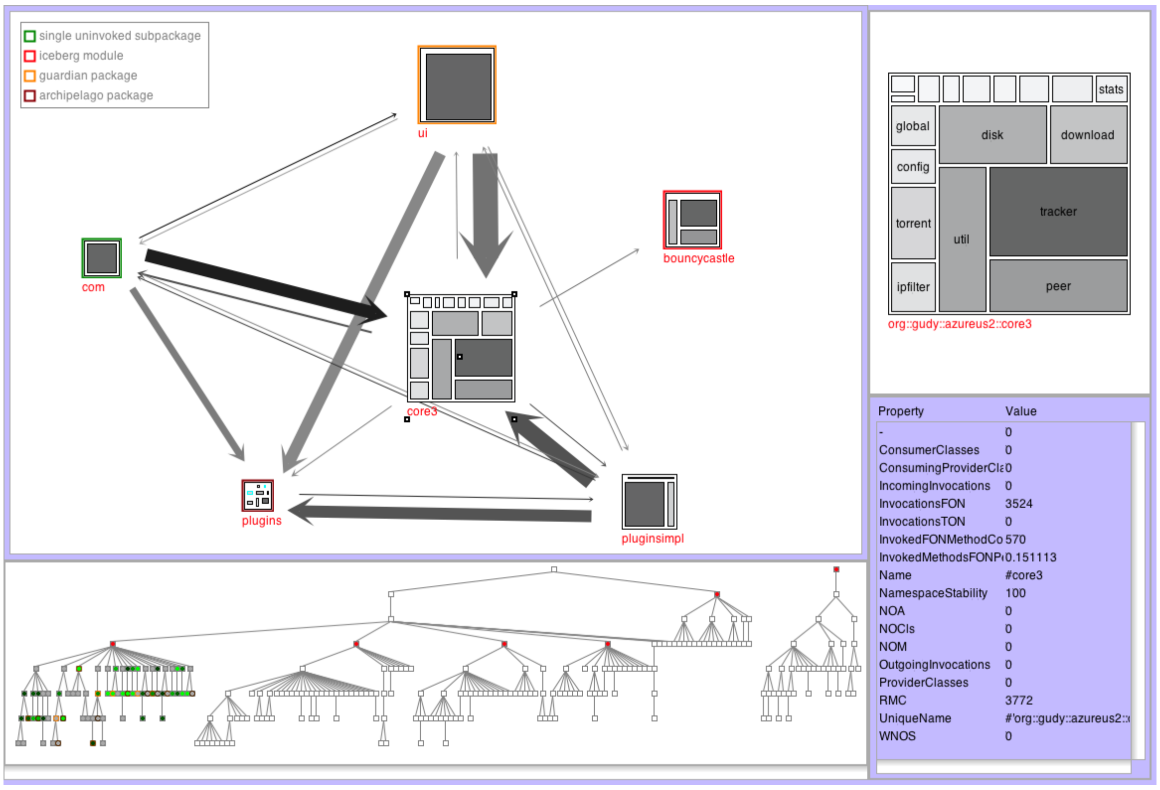
A lot of different tools have been implemented so far, but just an handful of them are not academic experiments without any practical application.

Three softwares that were source of inspiration for the X-Ray plug-in were: **Softwarenaut** [LL06], **CodeCrawler** [Lan03], and **Creole** [LMSW03].

2.3.1 Softwarenaut

Softwarenaut (Figure 2.3) is a software by Mircea Lungu targeted at supporting software understanding and reverse engineering. The tool makes heavy use of visualization and interactivity and provides multiple perspectives on a software system. It is language independent (as far as a model of the program is provided) and provides multiple simultaneous perspective on a system, showing *system complexity views*, *package and class dependencies* as far as providing interactive refinements.

The X-Ray plugin, actually, was born as simplified porting of this system to the Eclipse framework.

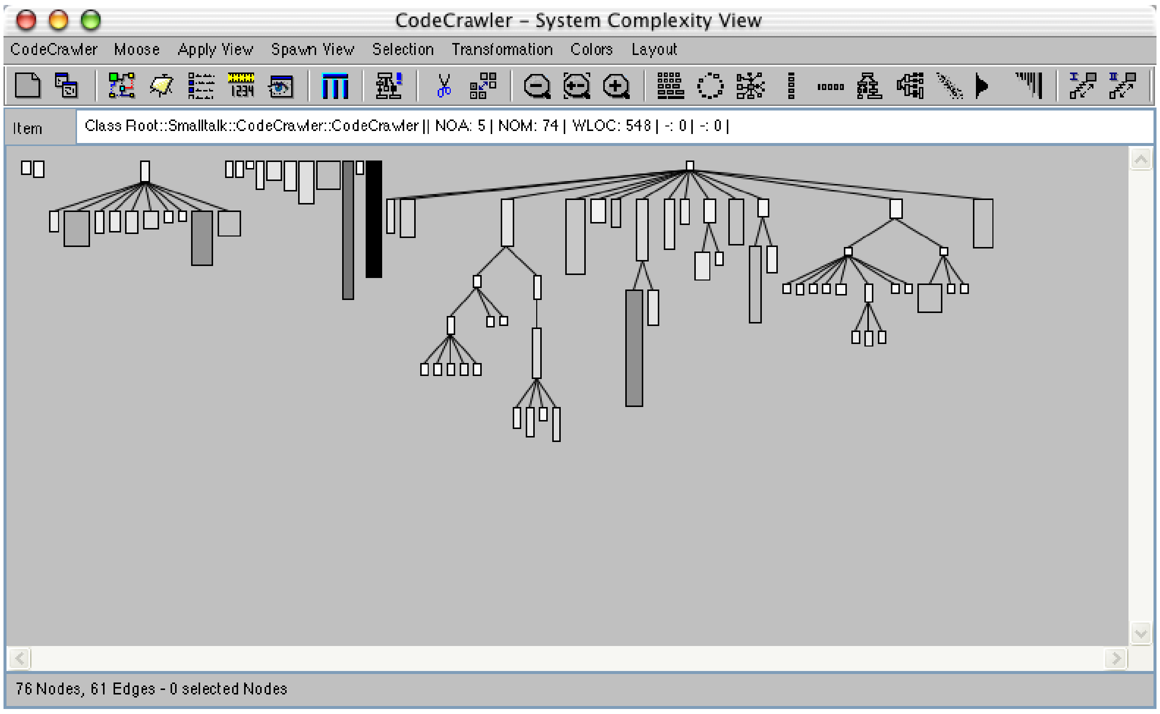


Property	Value
-	0
ConsumerClasses	0
ConsumingProviderCl	0
IncomingInvocations	0
InvocationsFON	3524
InvocationsTON	0
InvokedFONMethodCo	570
InvokedMethodsFONP	0.151113
Name	#core3
NamespaceStability	100
NOA	0
NOCls	0
NOM	0
OutgoingInvocations	0
ProviderClasses	0
RMC	3772
UniqueName	#'org::gudy::azureus2::
WNOS	0

Figure 2.3: Softwarenaut: *System complexity view* and *Packgae Dependency View*

2.3.2 CodeCrawler

CodeCrawler (Figure 2.4) is an information visualization framework created by Michele Lanza. It is language independent and provides a variety of visualization, such as: *system complexity view*, *class blueprint*, *system hotspots*, *evolution matrix* an many others. Moreover it supports customizable views, it has been industrially validated and is mainly used by software industry consultants. It provides *coarse-grained* views in order to visualize very large systems and *fine-grained* views for the visualization of small, internal structures.



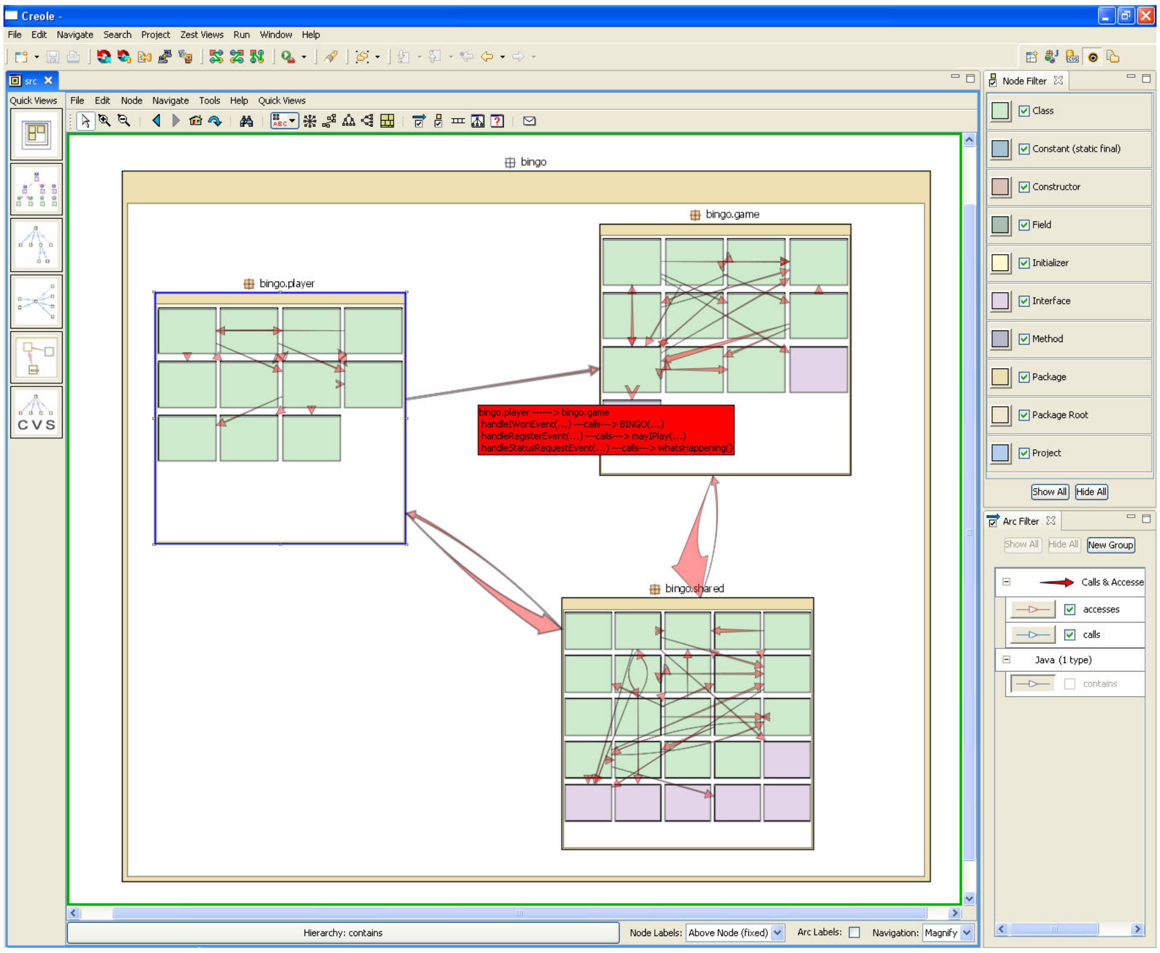
Class Root::Smalltalk::CodeCrawler::CodeCrawler || NOA: 5 | NOM: 74 | WLOC: 548 | -: 0 | -: 0 |

76 Nodes, 61 Edges - 0 selected Nodes

Figure 2.4: CodeCrawler: *System complexity view*

2.3.3 Creole

Creole (Figure 2.5) is the integration of *SHriMP* with Java Development Tools (JDT) included with the Eclipse platform. It gives the possibility to visually explore Java code, allowing to see its structure and links (references, accesses, etc.) between its different entities. *SHriMP* uses a nested graph view to present information that is hierarchically structured. It introduces the concept of nested interchangeable views to allow a user to explore multiple perspectives of information at different levels of abstraction. It provides *system complexity views*, *package and class dependencies* as far as other, multiple views. It also provides perspectives focused on visualizations about versioning systems.



```
bingo.player ------> bingo.game
 handleIWonEvent(...) ---calls---> BINGO(...)
 handleRegisterEvent(...) ---calls---> mayIPlay(...)
 handleStatusRequestEvent(...) ---calls---> whatsHappening()
```

Hierarchy: contains
Node Labels: Above Node (fixed)
Arc Labels:
Navigation: Magnify

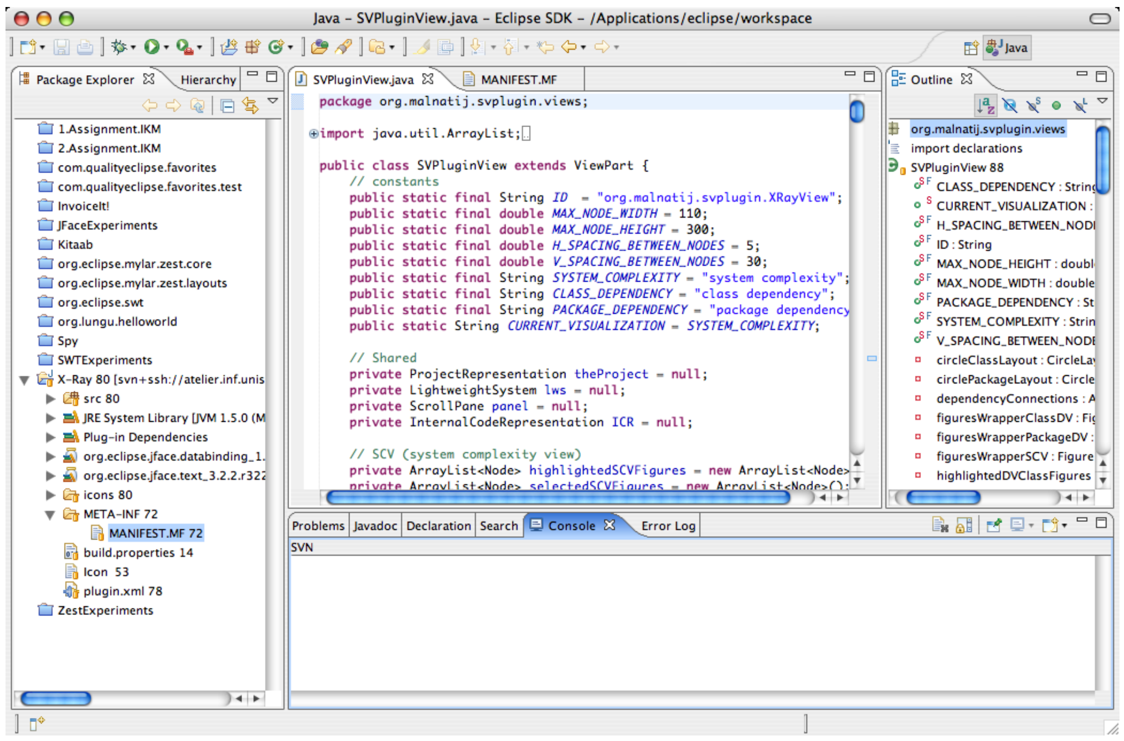
Figure 2.5: Creole: *Package Dependency View*

Chapter 3

Eclipse

Eclipse is an open-source, platform-independent software framework, written primarily in Java. The framework itself has been used to develop an Integrated Development Environment (IDE) and compiler that comes as part of Eclipse.

The next figure (Figure 3.1) shows a screen capture of the main workbench window as it looks with only the standard generic components that are part of the Eclipse Platform:



```
1.Assignment.IKM
2.Assignment.IKM
com.qualityeclipse.favorites
com.qualityeclipse.favorites.test
InvoiceIt!
JFaceExperiments
Kitaab
org.eclipse.mylar.zest.core
org.eclipse.mylar.zest.layouts
org.eclipse.swt
org.lungu.helloworld
Spy
SWTExperiments
X-Ray 80 [svn+ssh://atelier.inf.unis
    src 80
    JRE System Library [JVM 1.5.0 (M
    Plug-in Dependencies
    org.eclipse.jface.databinding_1.
    org.eclipse.jface.text_3.2.2.r322
    icons 80
    META-INF 72
        MANIFEST.MF 72
    build.properties 14
    Icon  53
    plugin.xml 78
ZestExperiments
```

```
package org.malnatij.svplugin.views;

import java.util.ArrayList;

public class SVPluginView extends ViewPart {
    // constants
    public static final String ID  = "org.malnatij.svplugin.XRayView";
    public static final double MAX_NODE_WIDTH = 110;
    public static final double MAX_NODE_HEIGHT = 300;
    public static final double H_SPACING_BETWEEN_NODES = 5;
    public static final double V_SPACING_BETWEEN_NODES = 30;
    public static final String SYSTEM_COMPLEXITY = "system complexity";
    public static final String CLASS_DEPENDENCY = "class dependency";
    public static final String PACKAGE_DEPENDENCY = "package dependency
    public static String CURRENT_VISUALIZATION = SYSTEM_COMPLEXITY;

    // Shared
    private ProjectRepresentation theProject = null;
    private LightweightSystem lws = null;
    private ScrollPane panel = null;
    private InternalCodeRepresentation ICR = null;

    // SCV (system complexity view)
    private ArrayList<Node> highlightedSCVFigures = new ArrayList<Node>
    private ArrayList<Node> selectedSCVFigures = new ArrayList<Node>();
```

```
org.malnatij.svplugin.views
import declarations
SVPluginView 88
    CLASS_DEPENDENCY : Strin
    CURRENT_VISUALIZATION :
    H_SPACING_BETWEEN_NOD
    ID : String
    MAX_NODE_HEIGHT : doubl
    MAX_NODE_WIDTH : double
    PACKAGE_DEPENDENCY : St
    SYSTEM_COMPLEXITY : Strin
    V_SPACING_BETWEEN_NODE
    circleClassLayout : CircleLa
    circlePackageLayout : Circle
    dependencyConnections : A
    figuresWrapperClassDV : Fi
    figuresWrapperPackageDV :
    figuresWrapperSCV : Figure
    highlightedDVClassFigures
```

Figure 3.1: Eclipse Platform UI

The *Package Explorer* (top left) shows the files in the user's workspace; the *Text Editor* (top) shows the content of a file; the *Console View* (bottom right) shows the content of the output stream; the *Outline View* (left) shows a content outline of the file being edited.

3.1 Eclipse Description

Originally developed by IBM, it is now managed by the Eclipse Foundation, a non-profit consortium of software industry vendors. It represents a stable platform, royalty-free and has worldwide redistribution rights. The platform was designed from a clean slate to be extensible and to provide exemplary tools. Eclipse development is based on rules of open source engagements. This includes open, transparent, merit-based, and collaborative development. All individuals can participate and contribute with their plug-in.

A plug-in is a piece of software that interacts with a host application, in this case the Eclipse framework. It adds functionality or it changes the existing ones (provided by other plug-ins or the framework itself). A plug-in can also provide services that can be used by other plug-ins. We say that a plug-in *contributes* to the framework adding or modifying any new or existing service.

A huge community of developers has grown around this framework which, now, is one of the most used world wide.

3.2 Eclipse Infrastructure

Eclipse is not a single monolithic program, but rather a small kernel surrounded by hundreds of plug-ins of which X-Ray is one. Each plugin may rely on services provided by Eclipse's core, by other plug-ins or both. This modular design provides the possibility to share and use code that can be run by different applications and plug-ins.

3.3 Plug-in structure

The behavior and functionality of every plug-in resides in its Java code, while the dependencies and services are declared in the `MANIFEST.MF` (a file describing the runtime aspect of the plug-in, such as identifier, version and dependencies) and `plugin.xml` (an XML formatted file describing extensions and extensions point) files. By dependencies, we intend all the external code (intended as classes, packages or plug-ins) that the current plug-in requires, while as extensions are intended functionality of the current product that can be extended by other plug-ins (using an extension point).

The reason behind the separation between the actual code and the rest resides in the trade off between having the code available and accessible to the code itself and facing the overhead of having to load each Java class. Most of the time a plug-in could (and should) be loaded just on request, without executing its code during Eclipse's startup (this concept is called *lazy-loading*). Loading plug-ins on an as-needed basis reduces both the startup and the memory footprint of the whole framework.

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, using a lot less space than loading all the Java code contained in the plug-in. In some special cases, in which the plug-in must be running all the time and from the start, a plug-in may require to bypass the lazy-loading and get loaded and executed at startup.

Figure 3.2 shows part of the structure of the X-Ray plug-in and the way it references other plug-ins and itself within the `MANIFEST.MF` and `plugin.xml` files.

3.3.1 Plug-in Directory

The whole content of a plug-in resides in a single folder named after a specific name convention: the directory must be a concatenation of the plugin identifier (which should be the plug-in name, even if it's not mandatory), an underscore and the plugin version, in dot-separated form (i.e., `org.malnatij.svplugin_1.0.0`). Moreover, the (possibly compressed) folder must be located in the `plugins` directory of Eclipse, as sibling of all the other plug-in. The decision about how to deploy the plug-in is left to the developer, who can choose among deploying it as a single JAR file (containing the compressed plug-in directory), the directory itself or an hybrid, with links between the different portions of the product.

3.3.2 Plug-in Manifest

Within each manifest there are string representing entries for *name, identifier, version, class and provider* of the current plug-in. The following is an extract of the manifest of the X-Ray plugin:

```
Bundle-Name: X-Ray Plug-in
Bundle-SymbolicName: org.malnatij.xray; singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator: Bundle-Activator: org.malnatij.xray.XRayPluginActivator
Bundle-Vendor: Jacopo Malnati
```

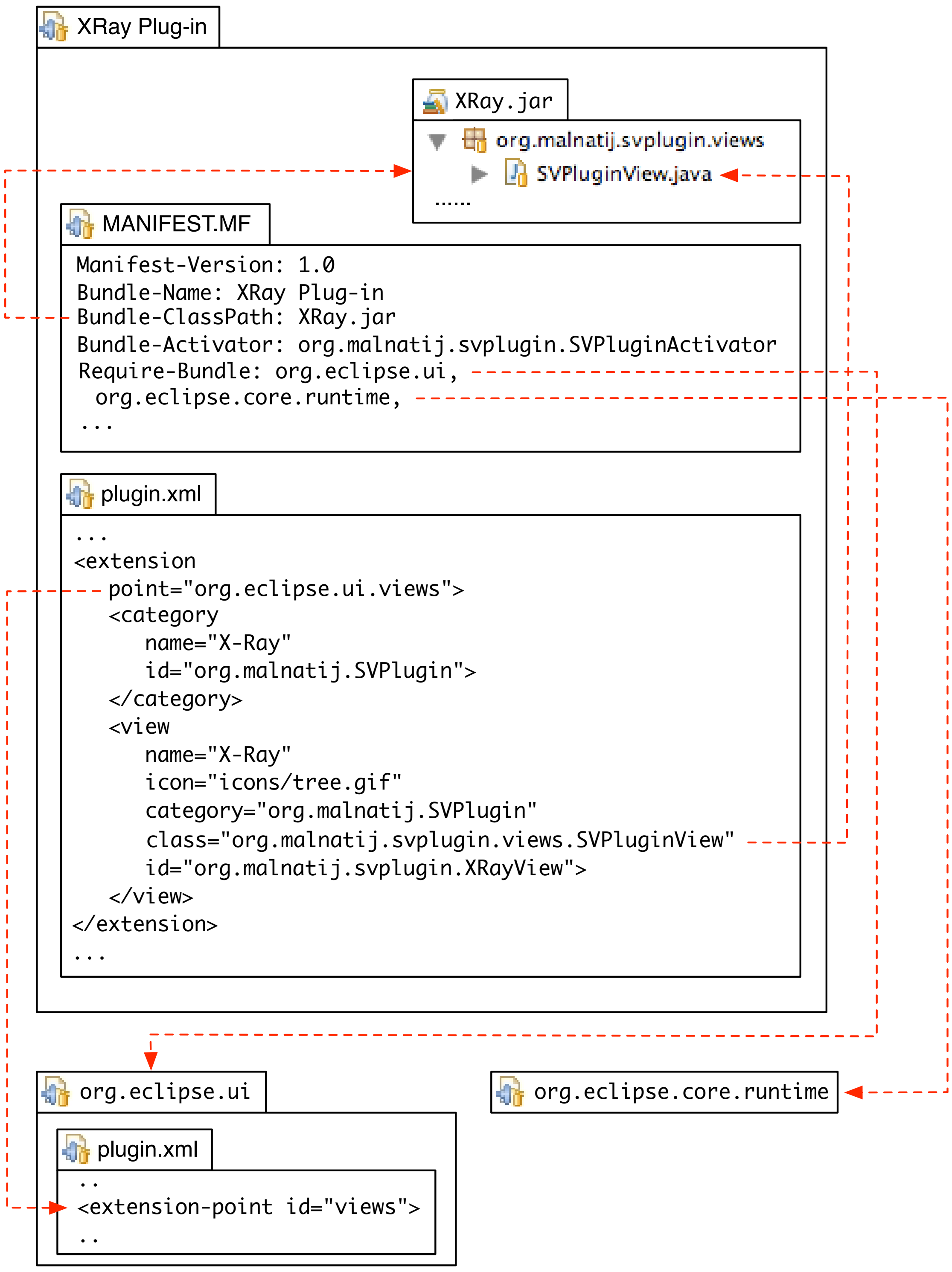
While the `Bundle-Name` represents the name of the plug-in, the `Bundle-SymbolicName` univocally identifies the plugin (usually named after Java package name convention). The `Bundle-Version` states the version of the product and `Bundle-Vendor` holds information about the designer or the producer of the plug-in. The `Bundle-Activator` declares the activator class, that is the class responsible for boot-strapping the plug-in.

Moreover every manifest might also contain declaration such as `Bundle-ClassPath`, `Export-Package`, `Require-Bundl`, responsible for, respectively, the set of libraries which contain the plug-in code, which packages are accessible to other plug-ins and which packages or plug-ins are require by the current one.

3.3.3 Plug-in Class

The plugin class (defined by the `Bundle-Activator`) provides methods for accessing static resources associated with the plug-in and for setting up the plug-in preferences. It's the first class notified after the plug-in loads and the last one when the product is about to be shut down.

As previously stated, Eclipse uses lazy-loading in order to decrease its memory footprint, anyway, if the plug-in requires an early startup, the `start()` method of the class defined by the `Bundle-Activator` will be executed.



```
Manifest-Version: 1.0
Bundle-Name: XRay Plug-in
Bundle-ClassPath: XRay.jar
Bundle-Activator: org.malnatij.svplugin.SVPluginActivator
Require-Bundle: org.eclipse.ui,
  org.eclipse.core.runtime,
...
```

```
...
<extension
   point="org.eclipse.ui.views">
   <category
      name="X-Ray"
      id="org.malnatij.SVPlugin">
   </category>
   <view
      name="X-Ray"
      icon="icons/tree.gif"
      category="org.malnatij.SVPlugin"
      class="org.malnatij.svplugin.views.SVPluginView"
      id="org.malnatij.svplugin.XRayView">
   </view>
</extension>
...
```

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

Figure 3.2: This is an example of how a new extension is declared in the plug-in manifest (with lines highlighting how the plug-in manifest references various plug-in artifacts)

X-Ray does not need any early startup, therefore its `start()` method will be called as soon as the user require the plug-in to show up.

3.3.4 Eclipse Plug-ins widely used by X-Ray

Eclipse itself is composed by different plug-ins that cooperate and represents a source of functionality for other plug-ins. There are hundreds of plug-in distributed with Eclipse, but the ones that were heavily used during the development if X-Ray were:

- **Core:** a general low-level group of non-UI plug-ins providing basic services and interfaces regarding extension processing.
- **SWT:** also known as the Standard Widget Tool, a general, platform independent, UI library providing graphical 2D widgets (Lists, Figures, Labels etc.). All the graphics created by X-Ray has been implemented using SWT components.
- **Workbench Core:** plug-in providing services and interfaces about the core of Eclipse, its organization and internal data structures.
- **JDT Core:** non UI-based Java Development Tooling, handling the infrastructure of the Java IDE, including an incremental Java compiler and parser, a Java Model that provides API for navigating its structure, an indexed based search infrastructure that is used for searching, code assist, type hierarchy computation and so on.

3.4 Eclipse Platform Overview

Now that the architecture of the framework has been discussed as far as some of its main components, let us graphically summarize them in Figure 3.3.

The workspace consists of one or more top-level projects, where each project maps to a corresponding user-specified directory in the file system. The workbench represents the user interface through which the user access every action that the framework provides and it is made up by JFace objects, provided by the JFace plug-in which sits on top of the SWT (Standard Widget Toolkit) library. Moreover the platform is composed by the Core and JDT Core plug-ins, as far as hundreds of others. Within this context, the X-Ray plug-in ”plugs” itself in the platform, becoming part if it, providing functionality as far as other plug-ins that could be added to the framework.

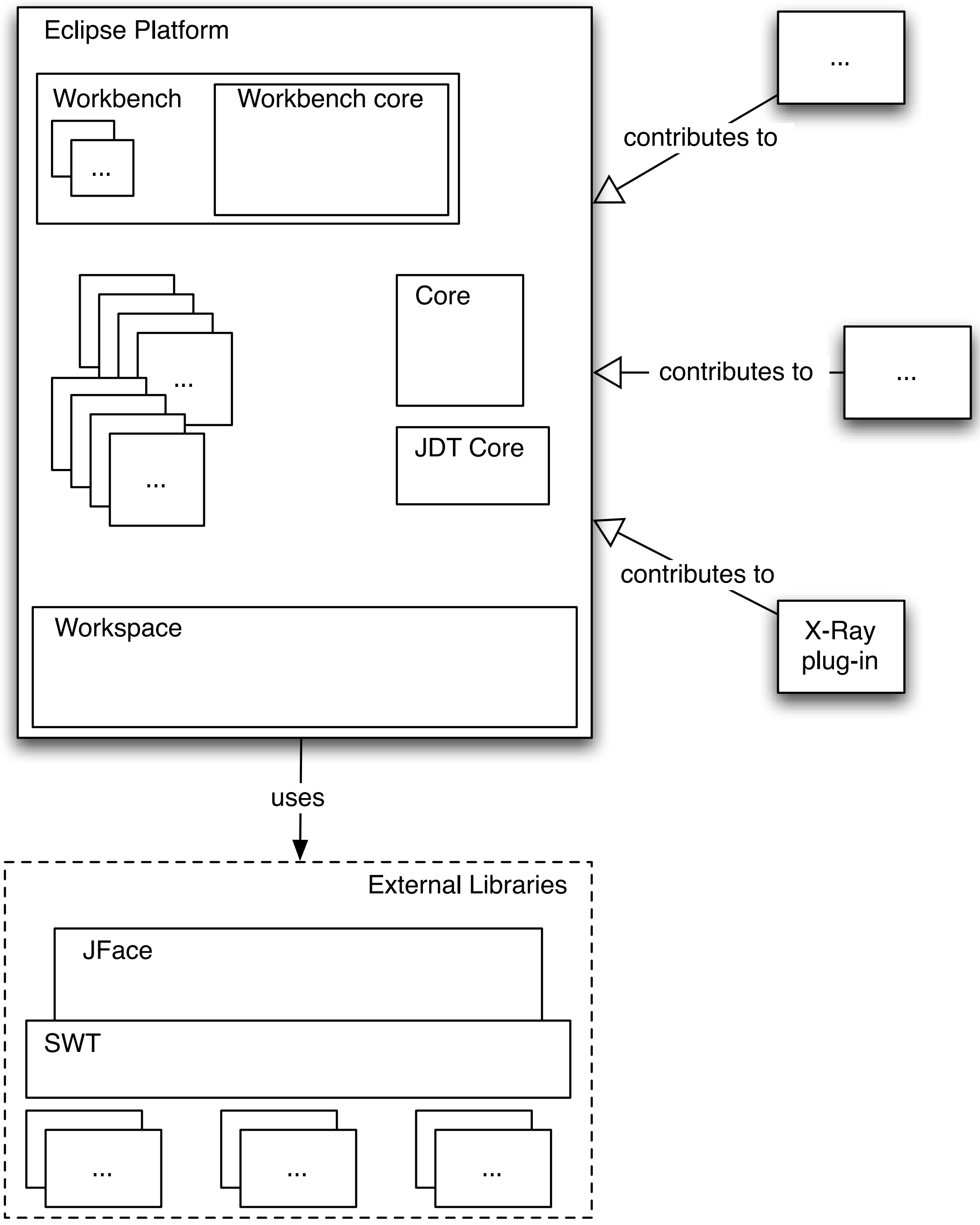


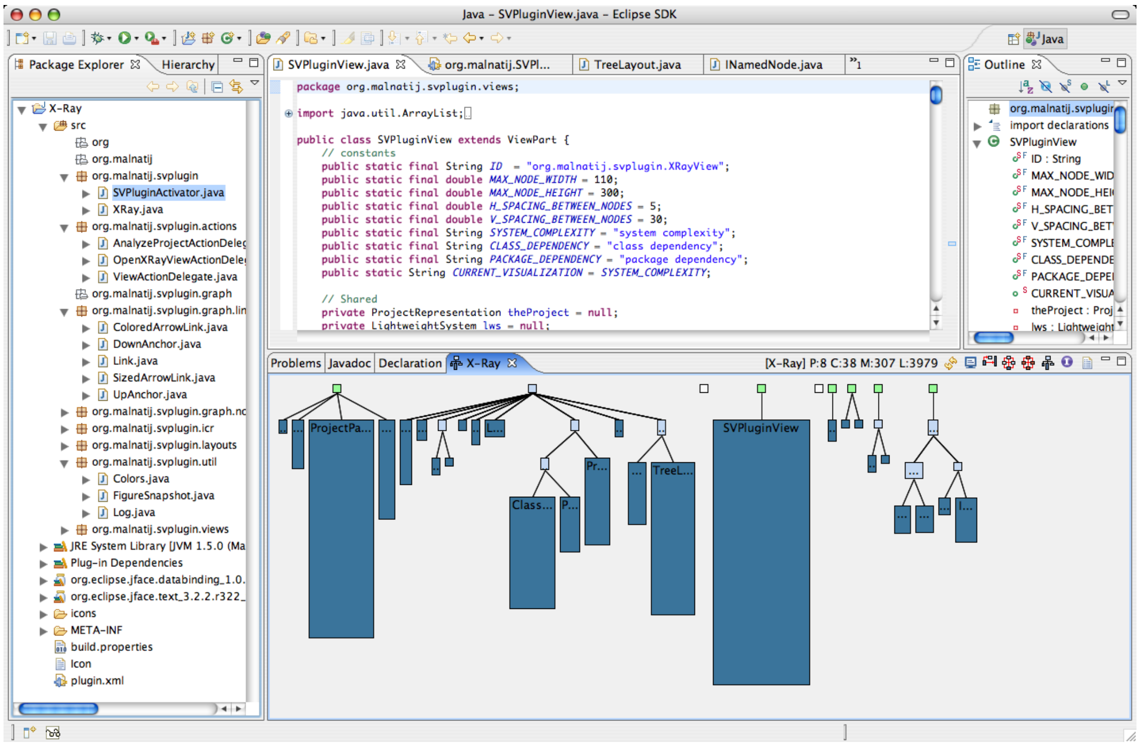
Figure 3.3: The Eclipse Platform and some of its component

Chapter 4

X-Ray

4.1 The Idea

Given that most of the software visualization tools are stand-alone and the Eclipse framework is widely used, we built a plug-in for Java code visualization that gives the possibility to the user to visualize the project on which it is working on without using any external tool, with a simple click. Providing different views and using different metrics, the user can navigate any system, analyzing its shape and spotting defects and useful information.



```
package org.malnatij.svplugin.views;

import java.util.ArrayList;

public class SVPluginView extends ViewPart {
    // constants
    public static final String ID  = "org.malnatij.svplugin.XRayView";
    public static final double MAX_NODE_WIDTH = 110;
    public static final double MAX_NODE_HEIGHT = 300;
    public static final double H_SPACING_BETWEEN_NODES = 5;
    public static final double V_SPACING_BETWEEN_NODES = 30;
    public static final String SYSTEM_COMPLEXITY = "system complexity";
    public static final String CLASS_DEPENDENCY = "class dependency";
    public static final String PACKAGE_DEPENDENCY = "package dependency";
    public static String CURRENT_VISUALIZATION = SYSTEM_COMPLEXITY;

    // Shared
    private ProjectRepresentation theProject = null;
    private LightweightSystem lws = null;
```

Figure 4.1: The *System Complexity View* provided by X-Ray visualizing itself within the Eclipse IDE

4.2 Metrics

4.2.1 Software Metrics

Metrics are an important tool in summarizing large amounts of information.

A metric is a number resulting by the mapping of a particular characteristic of an entity to a numerical value. Metrics are extremely useful because they express numerically something that is not necessarily a number. It scales up and summarizes a particular aspect of an entity, providing the possibility to have a meaningful representation of that aspect in the overall graphical representation.

A visualization that exploits different metrics in order to show a set of entities is said to be *polymetric*.

Software metrics can be divided into two major groups [ML94]:

- *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. Being at higher level of abstraction, they are less prescriptive, but are more important from an overall project perspective.
- *Design metrics:* they deal with size and in some case complexity and quality of software. They look at the quality of the project's 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 product's components.

Using metrics, one possibility is to model entities (such as classes, packages, methods, etc.) that can be abstracted and modeled as boxes, more precisely as rectangles, while dependencies (such as hierarchy, dependencies, etc.) are modeled as edges between entities. This choice, even if it might seem over-simplistic, it is actually an excellent trade-off between simplicity and the amount of information that can be expressed.

Recently, new ways of displaying data have been proposed an implemented, one of them is modeling entities in three-dimensional spaces, moving the abstraction from rectangles to cuboids, displaying dependencies without edges, but using different abstractions; other techniques imply the usage of isotropic textures and spectral models.

The sizes (and positions) of entities and dependencies are computed through different metrics and applied to the final graphical representation.

The following figure (Figure 4.2) is a small example of polymetric view. Let us say that we have to model and represent 3 entities, named A, B and C. We will represent them as rectangles and we want to model the *number of attributes* (NOA), *number of methods* (NOM) and finally the *lines of code* (LOC) for each entity (say, Java classes).

Using these 3 metrics, it is possible to graphically display the 3 entities. A simple way of mapping the metrics is to use the NOA as width and NOM as height of the rectangles. Moreover a color gradient has been associated to the LOC, giving to each

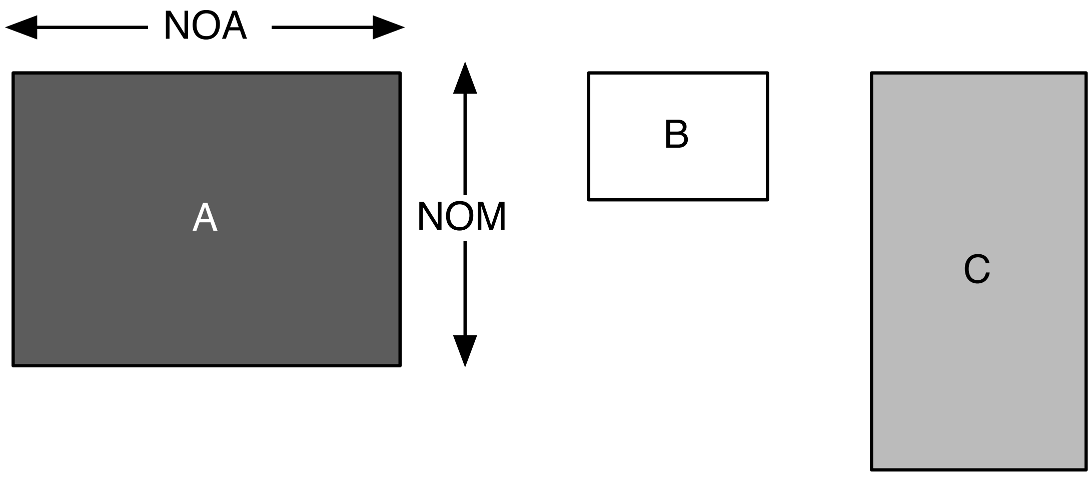


Figure 4.2: A simple polymetric visualization

rectangle a different color (the darker, the more lines of code).

Now, even without knowing exactly the numerical value for each metric, it is easy and quick to assert which entity has more lines of code, less attributes or more methods. Without even knowing the actual values, it is easy to compare the entities and draw meaningful conclusions.

In this simple example the position of every rectangle did not model any characteristic of the underlying entity, while, as it will be shown later, the position of entities could be used to model relationships among them. Anyway, three specific characteristic of the rectangles has been mapped by three different metrics: NOA, NOM, LOC. This was just one of the possible ways of mapping these metrics to the rectangles. Before even mapping them, there is the need for defining which and how many metric to use; this really depends on the kind and aim of the visualization. Different characteristic of the entity might be more or less important depending on the point of view of the observer, or his current perspective (software visualization tools can provide different perspectives, using different metrics in each of them). Moreover, just a handful of variables can be simultaneously mapped to metrics in a single graphical image without overlapping and creating confusion, providing too many information in the same location. With any metric we use, we also must know if it is too high or low, too much or too little. Basically, a reference point is needed, in order to give a useful meant to the metric. There are different way of applying and defining thresholds: thresholds based on statistical information and thresholds based on generally accepted semantics. Then, it must be decided how to map the metrics on the graphical representation. Using one metric for width, or height, color, border or texture etc., might completely affect the understandability of the visualization.

There are no precise algorithms in order to solve these problems, and any decision must be taken accurately. Anyway there are some commonly accepted semantics like the ones described in the next figure (Figure 4.3).

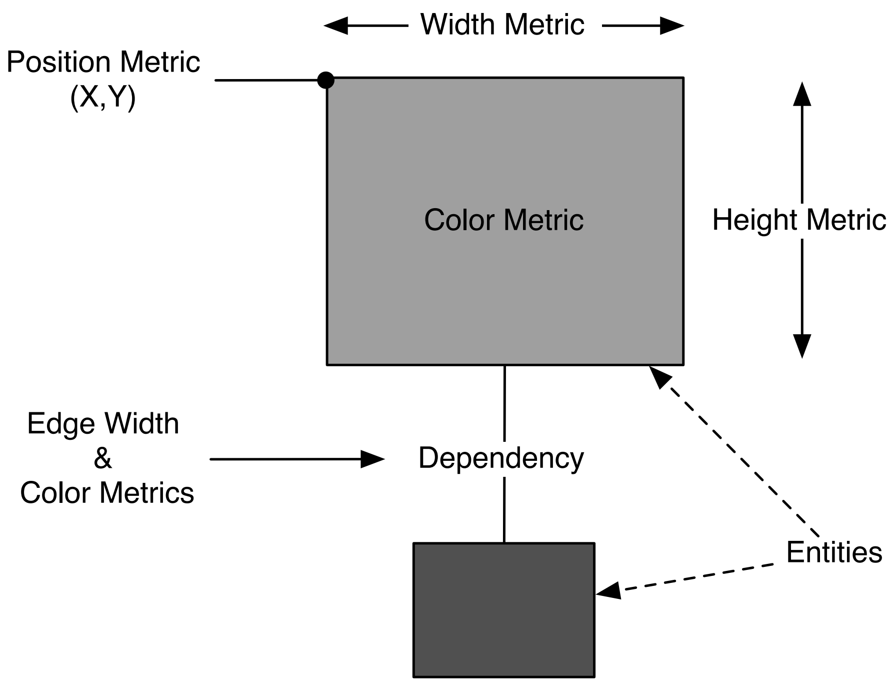


Figure 4.3: Common metrics semantic in a polymetric view

Rectangles could be used to model entities and edges between rectangles could represent relationship between the underlying entities (the software system is therefore modeled as a directed or undirected graph). Moreover the size of any node represents the combination of its width and height, where the wider and higher the node, the bigger the numerical value of the metric mapped on it. The node color is usually a gradient between 2 colors, in most cases black and white. The darker is the node, the higher is the measurement mapped to the gradient. The node position (X and Y values in a fixed coordinate system) can represent 2 more metric, or can be used by a layout (say, a tree layout or a circle layout) in order to position the node according to some constrain (i.e., hierarchy dependency). Edges width and color can be mapped to 2 more metrics, i.e., method invocation and weight of this invocations between the two entities connected by the edge (moreover an edge can also have different styles (solid, dashed, dotted, etc.), and decoration (arrows, dots, etc.) mapping other metrics).

Example. In Figure 4.4 it is represented a *System Complexity View* (which is a special type of polymetric view) of the Eclipse plug-in *X-Ray*. The metric used are the number of methods for the width, the lines of code for the height, colors for the Java class type (blue = concrete class, light blue = abstract class, white = interface, green = classes external to the project of unknown type). Edges represent inheritance relationship between classes (classes in higher positions are parents of classes in lower positions). Using this simple visualization it is easy to spot the large classes in terms of functionality (derived by the amount of lines of code and methods).

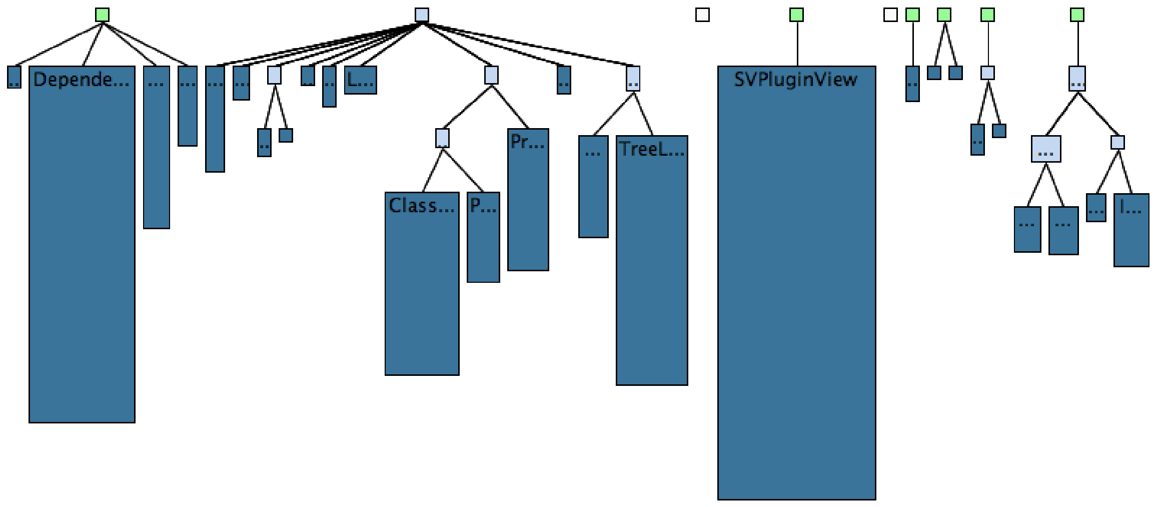


Figure 4.4: A polymetric view. The metric used are the following: width metric = number of methods, height metric = lines of code, color metric = Java class type (concrete, abstract, interface, external to the project), edges represent hierarchy dependency

4.2.2 X-Ray Metrics

The plug-in uses different metrics in its views, modeling classes and packages entities as nodes (rectangles) according to the view in which they are represented. It models NOM, LOC, Class Type, Hierarchy, Dependency and their weights.

Metrics in the *System Complexity View*

In the *System Complexity View* the **position** of every node reflects the position of the class represented by that entity in the inheritance hierarchy of the underlying project. Nodes are positioned according to a tree layout which represents the class hierarchy.

Nodes are visualization of Java classes and they have different **colors**: white, light blue, blue and green. Every color is mapped to one kind of Java class, respectively they visualize interfaces, abstract classes and concrete classes. Green nodes represent classes external to the project, their type is unknown (retrieving it would add a significant parsing overhead to the whole parsing process) and they represent roots external to the project. While analyzing a system it is useful to know the kind of the represented entities; it might be therefore possible to spot design flaws (i.e., misplaced abstract keyword)

The black **border color** represents a stand-alone class, while an orange border implies that the visualized entity is an inner class (a class defined entirely within another class). Knowing if a class is stand-alone or it is defined within another class is crucial while spotting design disharmonies; inner classes should be smaller than stand-alone ones.

The **width** of the node reflects the number of methods of the visualized class, while the **height** reflects the lines of code in the underlying entity. Methods represent units of functionality inside a Java class, it is therefore interesting to quantify their number for every entity and the number of lines of code reflects the number of instructions that a Java class might execute, giving information about the weight of the functionalities provided by the entity. In the case of a node representing an external class, both width and height have a default size, being unknown the number of methods and lines of code for that entity.

Edges represent hierarchy between nodes, they reflect inheritance relationships between classes. There is no color or width metric associated with hierarchy edges given that X-Ray does not associate any other property to them other than inheritance.

The following table (Table 4.1) summarizes the metrics applied on nodes in the *System Complexity View*.

Metric	*Description*
Position	Computed following a tree layout (top-bottom oriented)
Color	Green (external), white (interface), light blue (abstract) or blue (concrete)
Width	Number of Methods
Height	Lines of Code
Border	It cab be black for a stand-alone class or orange for an inner class.
Edges	Inheritance between classes

Table 4.1: Node metrics in the *System Complexity View*

Example. Figure 4.5 reflects graphically the metrics used in the *System Complexity View*. There are 5 entities, and three edges. One node is green, therefore it is a class external to the project, it is a root and parent of an internal abstract class (the light blue one). Moreover there is an interface (the white node) and two concrete sibling classes (the blue nodes, children of the abstract class) one of which is an inner class (the one with the orange border). The external node has default width and height, while number of methods and lines of code are used to compute the bounds of internal nodes.

Metrics in the *Class & Packages Dependency View*

In the *Class & Packages Dependency View* nodes represent classes or packages, depending on the granularity of the visualization which can visualize class dependencies or package dependencies.

- *Class Dependencies.* Nodes are visualization of Java classes and they have different **colors**: white, light blue and blue. Every color is mapped to one kind of Java class, like in the *System Complexity View* (except for external classes that are not visualized, therefore there are not green nodes).
- *Package Dependencies.* Nodes are visualization of Java packages and they are brown (the default package color used by the majority of Java IDEs)

Edges, for both *classes and packages*, represent dependencies between Java entities. They are arrows, each of them with a certain **weight and color**, highlighting how

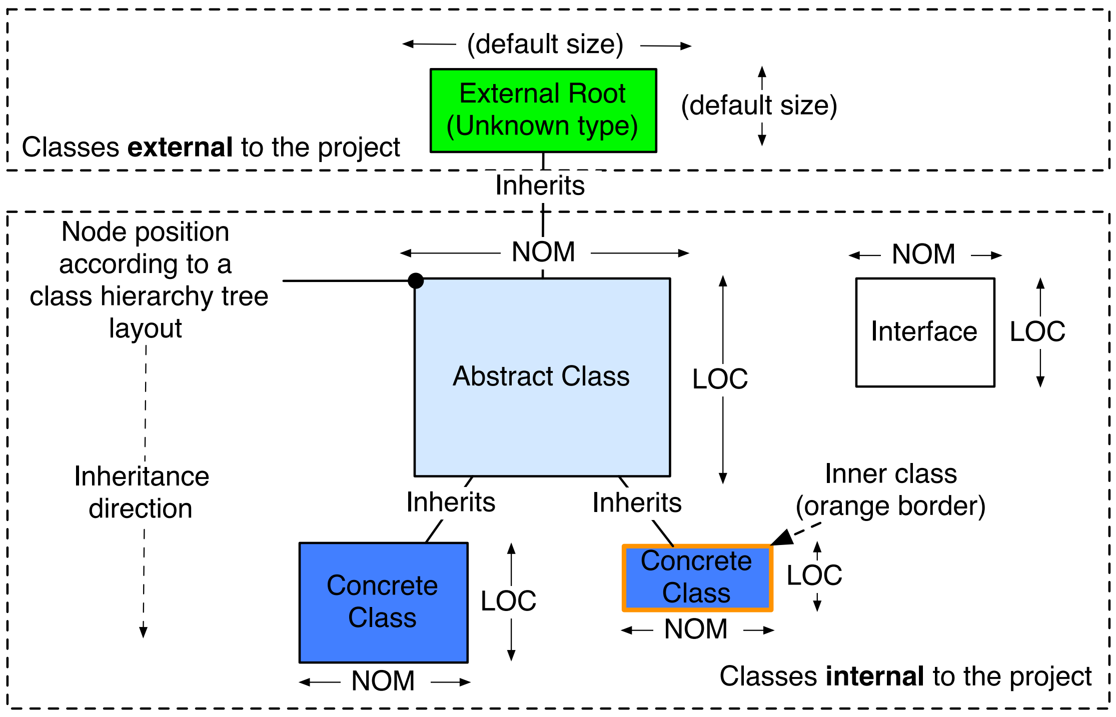

(default size)
External Root (Unknown type)
(default size)
Classes **external** to the project
Inherits
Node position according to a class hierarchy tree layout
NOM
NOM
Abstract Class
LOC
Interface
LOC
Inheritance direction
Inherits
Inherits
Inner class (orange border)
Concrete Class
LOC
Concrete Class
LOC
NOM
NOM
Classes **internal** to the project

Figure 4.5: *System Complexity View* node metrics in a nutshell

strength is the dependency between the two connected entities. Regardless of the kind of entity, an arrow means that the entity from which the arrow is coming out has a dependency to the entity the arrow is coming in (basically the first entity uses some code implemented in the second one). The color of the edge and the size of the edge (intended as the width of the arrow) represent the same metric: the weight of the dependency .The more external code used (and by external we intend code implemented in another entity) the stronger the dependency will be. There are six levels of dependence, as shown in Table 4.2.

Level	*Nr.ofexternalcalls*	*Color*	*Width*
Very Light	1 - 2	Light Pink	Default, thin size (2 pixels)
Light	3 - 4	Pink	Increased by a factor of 2 (4 pixels)
Moderate	5 - 9	Orange	Increased by a factor of 3 (6 pixels)
Severe	10 - 19	Light Brown	Increased by a factor of 4 (8 pixels)
Heavy	20 - 49	Dark Brown	Increased by a factor of 4 (8 pixels)
Very Heavy	more than 50	Black	Increased by a factor of 5 (10 pixels)

Table 4.2: Edges (arrows) metric in the *Class & Packages Dependency View*

In the latest version of X-Ray we decided to include dependencies (on demand) even in the *System Complexity View*, therefore Table 4.2 also refers to metrics applied to edges (arrows) in the hierarchy tree drawn by the *System Complexity View*.

The following table (Table 4.3) summarizes the metrics applied on nodes in the *Class & Packages Dependency View*.

Metric	*Description*
Color	package: brown, class: white (interface), light blue (abstract), blue (concrete).
Edges	Dependency between entities (colors and widths at Table 4.2).

Table 4.3: Node metrics in the *Class & Packages Dependency View*

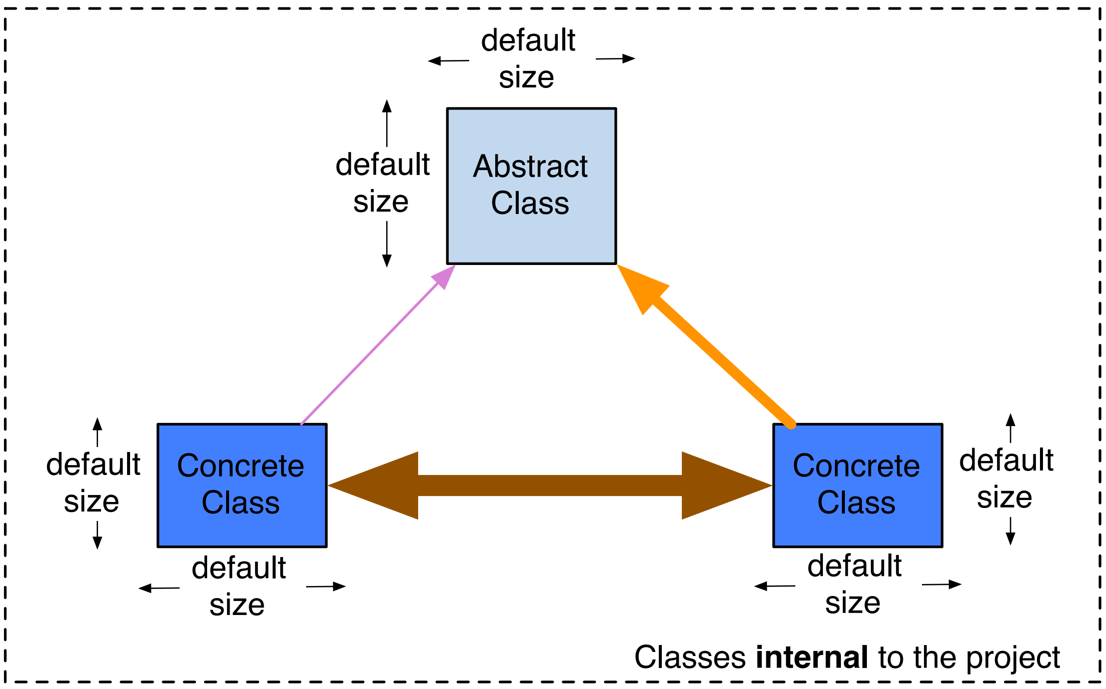


Figure 4.6: *Class & Package Dependency View* node metrics in a nutshell

Example. Figure 4.6 graphically reflects the metrics used in the *Class & Package Dependency View*. This example shows a *Class Dependency View*, the represented entities are three Java classes. All of them have the same size and their color reflects their class type (one abstract class and two concrete classes). There are three arrows between them, highlighting dependencies of different strength, according to Table 4.2.

4.3 Views

The same entity or the same system can be represented in different (potentially infinite) ways. Depending on the point of view under which the user wants to see the system, different abstractions and metrics should be used in order to reduce the complexity of the displayed entity and let the user analyze it.

4.3.1 X-Ray view

In order to provide different abstractions of the same system, X-Ray contributes to the Eclipse framework its own workbench view which gives the user the possibility (being a general purpose container) to display and browse different polymetric views: *System Complexity, Class Dependencies & Package Dependencies.*

Description

Any Eclipse *workbench view* (a panel in which to draw and present any kind of data to the user) 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.

Views are contained in a *workbench page* and because there are potentially hundreds of views in the workbench, they are organized into categories. The **Window - Show View** dialog present a list of views organized by category so that a user can more easily find a desired view.

By default, when a plug-in contributes to the Eclipse framework adding a new view, the view is positioned in the bottom right part of the workbench. Anyway, the workbench itself is made up by different, dynamic, views that can be manually resized or repositioned by the user at any time.

Initialization

The X-Ray workbench view appears by clicking on the **Window - Show View** dialog, **Others..**, then **X-Ray** and finally on the **X-Ray icon**. The empty X-Ray view will therefore appear in the default position declared by the framework.

This is the default procedure to open a given view, but it is not really useful for our plug-in. X-Ray is intended to visualize a specific project selected by the user, therefore we contributed, as shown in Figure 4.7, to the project's context menu an action (**Analyze Current Project**) that will activate the plug-in and analyze the currently selected project; adding this specific action 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. Thus the user is ”somehow” forced to select which project the plug-in will visualize.

Interaction

The X-Ray workbench view provides *general information* about the project and different *actions* to the user. Some actions are available for every polymetric view, while others are specific to certain visualizations.

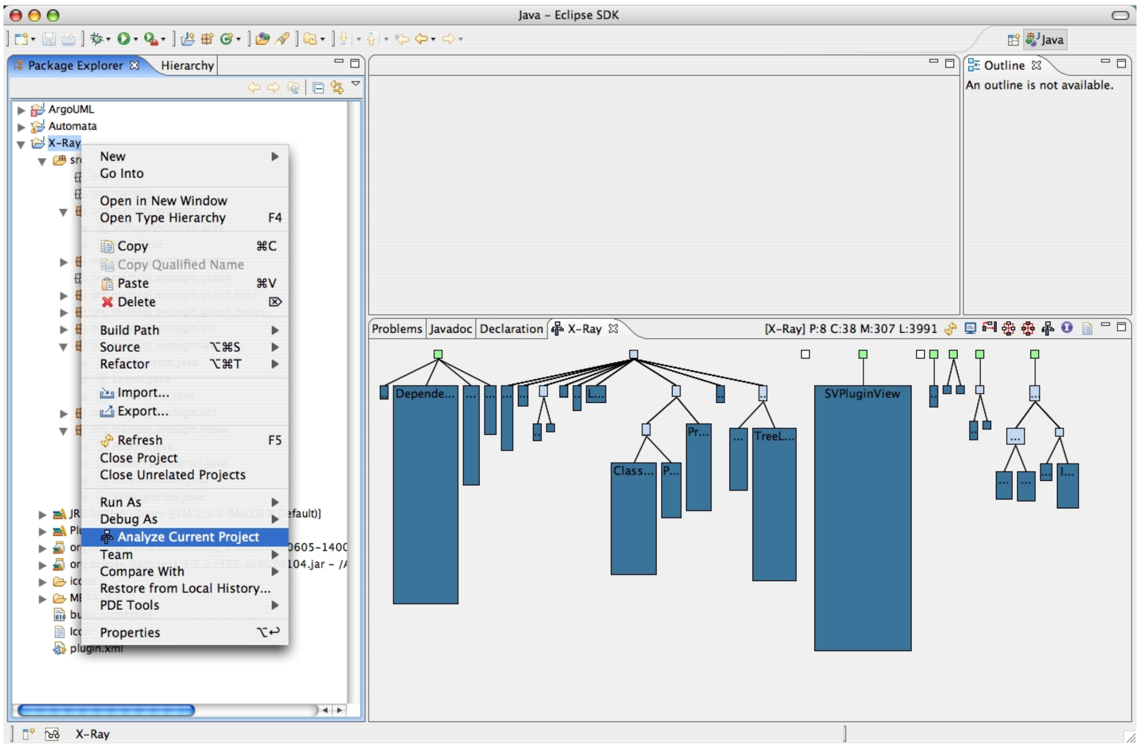


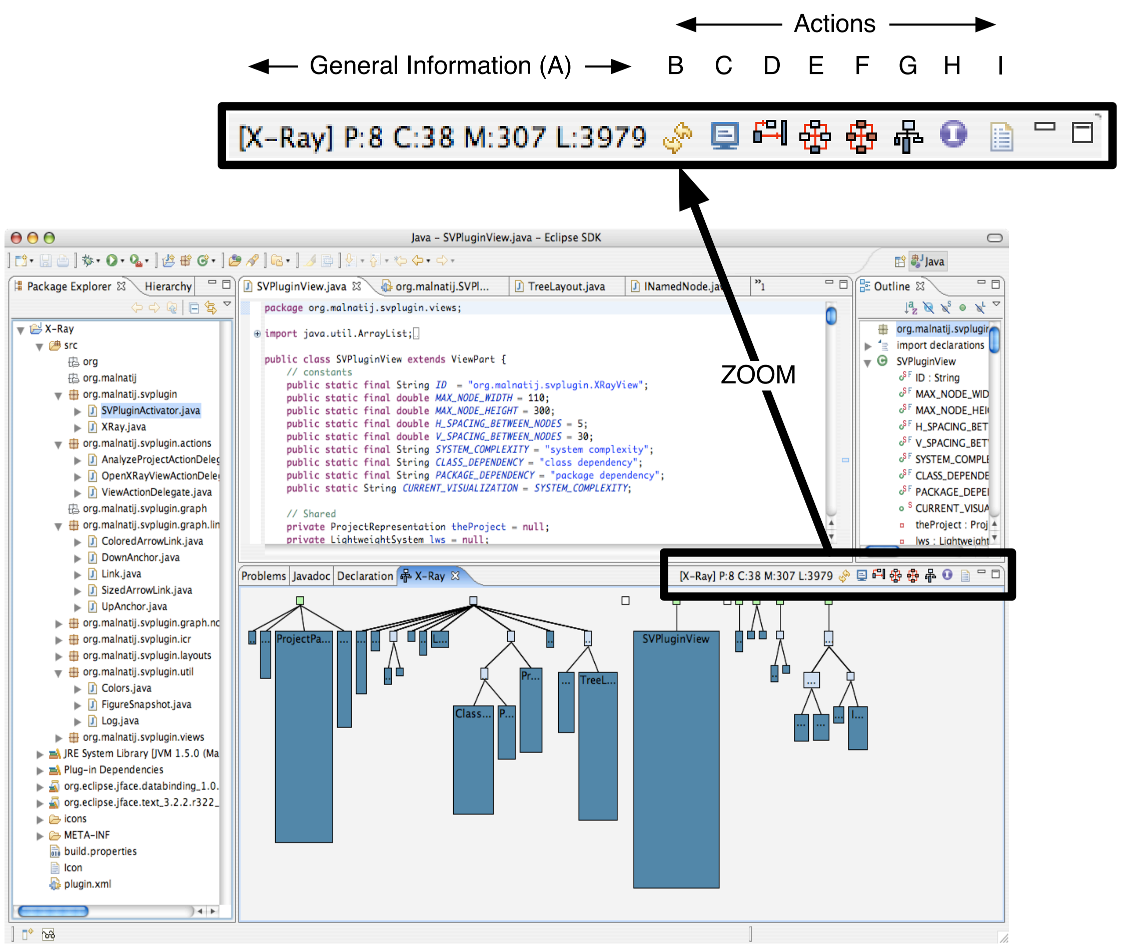
Figure 4.7: The plug-in adds an action that will be visible after right-clicking on a project in the *Package Explorer* (In this example, the X-Ray project itself). Clicking on **Analyze Current Project** action, X-Ray will be activated, and it will create and show its view containing the *System Complexity View* (which is clearly visible in the bottom right part of figure).

The user will be able, clicking on a class entity, to select it; a red border indicates that the node is selected and every action that requires a selection will be applied to the currently selected node. The plug-in can even highlight nodes (with a green border), while they are useful for a particular view or action.

Actions and general information (Figure 4.8) become visible as soon as the plug-in analyzes a project, they are text and little icons on the top left part of the X-Ray's view. General information gives a short description of the analyzed project in terms of number of packages, classes, methods and lines of code (i.e., `[X-Ray] P:8 C:38 M:307 L:3979` is the general information of the plug-in itself). The idea is to give a general overview of the project, providing the number of entities (and metrics) that X-Ray is visualizing.

The followings are the available actions:

- *(B) Refresh.* Refreshes the current visualization. The plug-in will parse and analyze again the whole project, therefore it might be an intensive task.
- *(C) Screenshot.* Takes a screenshot of the current visualization and it saves it as a `jpeg` image on the user's Desktop.
- *(D) Show Dependencies.* The plug-in will overlay, to the current visualization, edges



```
package org.malnatij.svplugin.views;

import java.util.ArrayList;

public class SVPluginView extends ViewPart {
    // constants
    public static final String ID  = "org.malnatij.svplugin.XRayView";
    public static final double MAX_NODE_WIDTH = 110;
    public static final double MAX_NODE_HEIGHT = 300;
    public static final double H_SPACING_BETWEEN_NODES = 5;
    public static final double V_SPACING_BETWEEN_NODES = 30;
    public static final String SYSTEM_COMPLEXITY = "system complexity";
    public static final String CLASS_DEPENDENCY = "class dependency";
    public static final String PACKAGE_DEPENDENCY = "package dependency";
    public static String CURRENT_VISUALIZATION = SYSTEM_COMPLEXITY;

    // Shared
    private ProjectRepresentation theProject = null;
    private LightweightSystem lws = null;
```

Figure 4.8: (A) = general information, (B) = refresh, (C) = screenshot, (D) = show dependencies, (E) show *Class Dependency View*, (F) = show *Package Dependency View*, (G) = show *System Complexity View*, (H) show implemented interfaces / interface implementers, (I) open selected file

(arrows) between the selected class node and its dependencies (following the convention declared in Figure 4.2).

- *(E) Show Class Dependency View.* Switches the current view to the *Class Dependency View* (Section 4.3.3).

- *(F) Show Package Dependency View.* Switches the current view to the *Package Dependency View* (Section 4.3.3).

- *(G) Show System Complexity View.* Switches the current view to the *System Complexity View* (Section 4.3.2).

- *(H) Show implemented interfaces / interface implementers.* The plug-in will overlay, to the current visualization, edges (solid lines) between interfaces and implementers. If the currently selected node represent an interface, X-Ray will create links between it and all the nodes representing classes which implement the interface in the underlying system. if the currently selected node represents a class, it will link that class to any interface that the class is implementing.

- *(I) Open selected file.* The plug-in will open the `.java` file corresponding to the class represented by the selected node inside the source code editor provided by the Eclipse framework. This action will also trigger the show dependencies action.

Actions may or may not be available, depending on the currently polymetric view. Table 4.4 summarizes the availability of every action for every view.

Action	*System Complexity*	*Class Dependency*	*Package Dependency*
B	x	x	x
C	x	x	x
D	x	-	-
E	x	-	x
F	x	x	-
G	-	x	x
H	x	-	-
I	x	-	-

Table 4.4: Actions and their availability within the three polymetric views provided by X-Ray. "x" means available, "-" means not available

As we previously said, the X-Ray workbench view is just a container for different polymetric views. Once the project has been parsed by the plug-in, the user can switch between different views and analyze the selected system. Every view provides a different abstraction and point of view to better understand the design and dependencies of the current project.

4.3.2 System Complexity View

The *System Complexity View* is the default polymetric view that appears to the user right after he/she selected a project as target for X-Ray.

Description

Nodes are rectangles that model classes of the underlying system while edges represent inheritance between classes. Nodes are arranged in a vertical (top-down) tree that highlights inheritance hierarchy. Both nodes and edges are displayed following the metrics explained in Section 4.2.2.

Figure 4.9 is the *System Complexity View* of the X-Ray project, drawn inside the workbench view created by the plug-in. The workbench view provides general information about the project and all the available actions for the currently shown polymetric view.

Goal

The *System Complexity View* is particularly efficient while spotting disharmonies in the design and implementation of a system. It is easy to find and identify big nodes (compared to the others) or anomalies in the shape of the project (provided by the inheritance

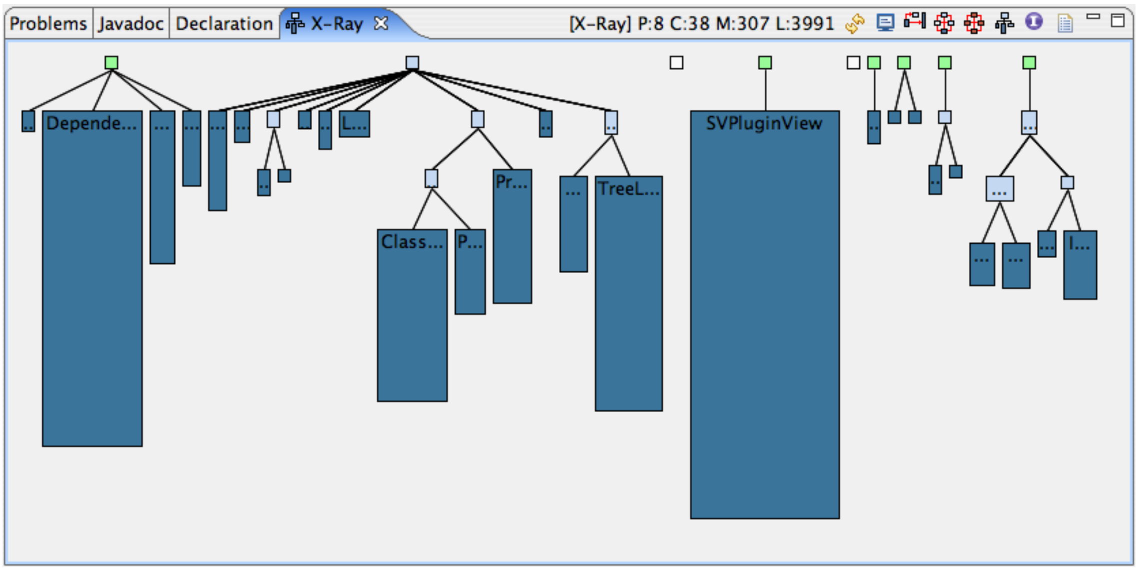


Figure 4.9: X-Ray's workbench view containing the *System Complexity View* for the selected project.

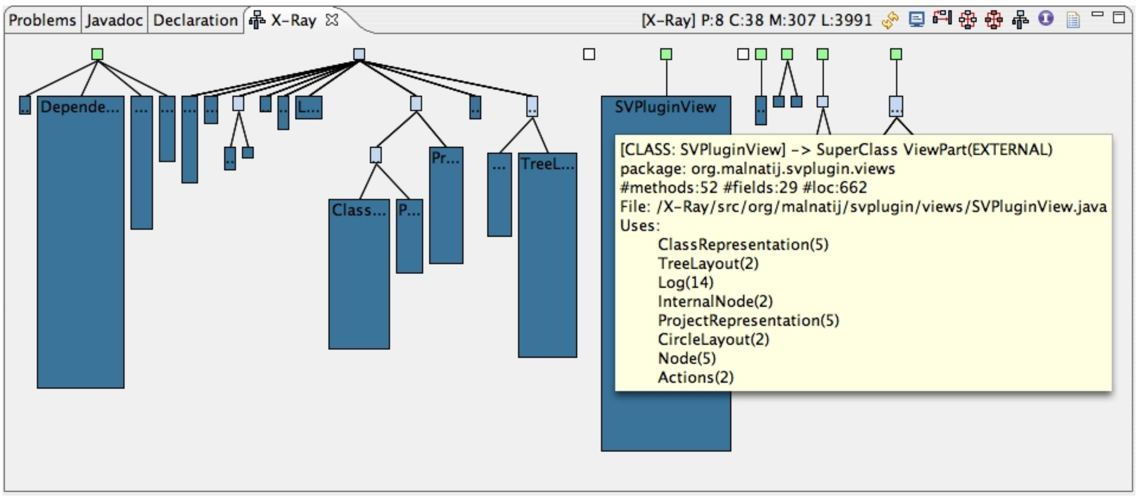
tree). The user is therefore able, with a single picture, to analyze and understand complex systems in terms of methods, lines of code and inheritance hierarchies without the need of reading source code. The *System Complexity View* highlight symptoms of identity disharmonies that can be noticed by considering design elements *in isoloation* [LM06].

Example of Usage

Right-clicking on a project in the *Package Explorer* and selecting **Analyze Current Project**, X-Ray will be initialized and the *System Complexity View* will appear in the workbench view created by the plug-in. Depending on the resolution of the monitor, the size of the view and the complexity of the analyzed system, the project will be completely visualized within the workbench view boundaries or it will require horizontal or vertical scrolling. Every node (except green ones, representing classes external to the project) can be selected and inspected.

Example. Let us say that we want to analyze the X-Ray project. Once the project has been selected as target for the plug-in, the polymetric view appears as shown in Figure 4.9. The most immediate information provided by the plug-in is the shape of the system. The hierarchy tree offers information about inheritance and levels of inheritance, as far as identifies sibling entities. Looking at the polymetric view it is easy to see which are the major inheritance subtrees. Moreover, visualizing classes external to the system, X-Ray shows inheritance hierarchy that would have been otherwise hidden.

Looking at the system, node, by node, one thing immediately stands out: a big node in the right-hand part of the view. The name of the class (*SVPluginView*) represented by this node is clearly visible, given that the width of the node is large enough to accommodate it without being truncated. If we want to have more information about the node and investigate why it is so big, we just have to move the mouse pointer on the node body and wait for the *tooltip* to show up. The tooltip (Figure 4.10) gives information about the name of the class, its superclass (if any) the package in which it is contained, the implemented interfaces (if any), the number of methods, fields and lines of code, the path of the Java file containing the source code and which other classes are used by the current one (and how strong is the dependency, between parenthesis).

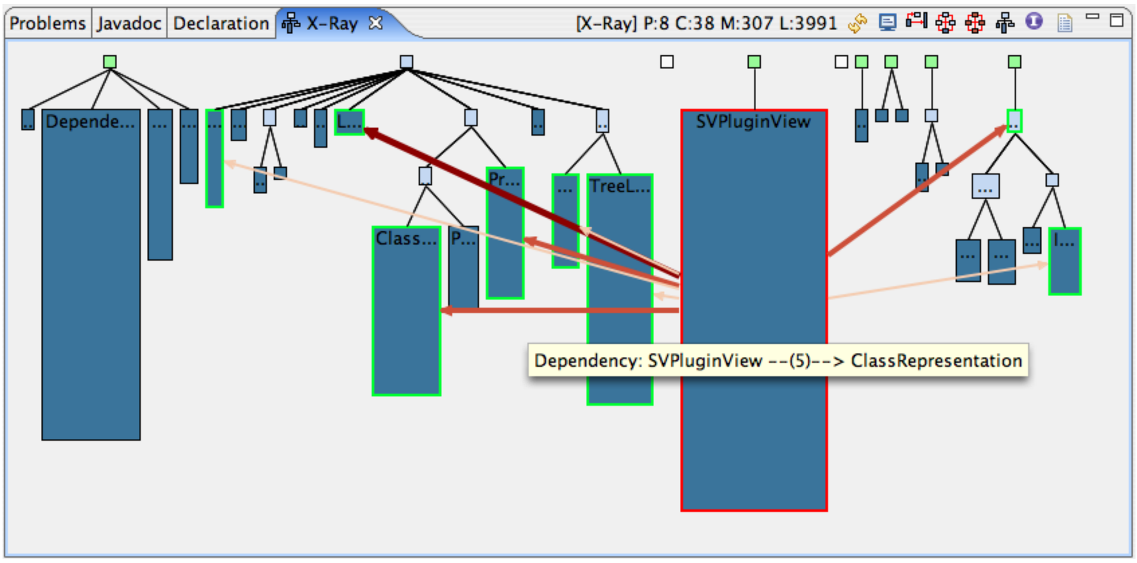


```
[CLASS: SVPluginView] -> SuperClass ViewPart(EXTERNAL)
package: org.malnatij.svplugin.views
#methods:52 #fields:29 #loc:662
File: /X-Ray/src/org/malnatij/svplugin/views/SVPluginView.java
Uses:
        ClassRepresentation(5)
        TreeLayout(2)
        Log(14)
        InternalNode(2)
        ProjectRepresentation(5)
        CircleLayout(2)
        Node(5)
        Actions(2)
```

Figure 4.10: Tooltip of the *SVPluginView* node. The node represents the *SVPluginView* class which is a subclass of the external class *ViewPart*. The *SVPluginView* class is contained in the *org.malnatij.svplugin.views* package and is composed by 52 methods, 29 fields and 662 lines of code. The source code of the class is contained in the file */X-Ray/src/org/malnatij/svplugin/views/SVPluginView.java* and it uses 8 other classes; 5 times the classes *ProjectRepresentation* and *ClassRepresentation*, 14 times the class *Log* etc.

The tooltip gives a textual description of the graphical representation of the project. Moreover, it lists all the dependencies of the class; this information is missing, by default, in the graphics of the current polymetric view (given that it is fully covered by the *Class Dependency View*), but it can be visualized and shown to the user by selecting the node and clicking on the *(D) Show Dependencies* action, as shown in Figure 4.11. Dependency arrows use metrics described in Section 4.2.2 and have a tooltip explaining the strength of the represented dependency. By clicking on a blank area the *SVPluginView* node will be deselected and all the dependency links will be removed.

If we still want to investigate the *SVPluginView* class, we can reach the lowest level of analysis: the source code. Selecting the node and double-clicking it, the source code of the class represented by the selected node will be opened in the *Text Editor* provided by the Eclipse framework. The same result can be obtained by selecting the node and clicking on the *(I) Open selected file* action. The *Text Editor* is made up by different tabs, one for every Java file currently opened. While the user is switching between one file and another, the plug-in will automatically react centering the *System Complexity View* in the node corresponding to the class the user is browsing, visualizing its dependencies. This feature is particularly useful: while reading or writing code, the user will be able to see the position of the current class in the inheritance hierarchy and its



Dependency: SVPluginView --(5)--> ClassRepresentation

Figure 4.11: The *System Complexity View* after selecting the *SVPluginView* node and clicking on the *(D) Show Dependencies* action. Eight dependency arrows are created and visualized. They have different sizes and colors, representing different levels of strength. A tooltip describes the nature (origin class, strength, target class) of the dependency. Every node representing a target class is highlighted (with a light-green border).

dependencies.

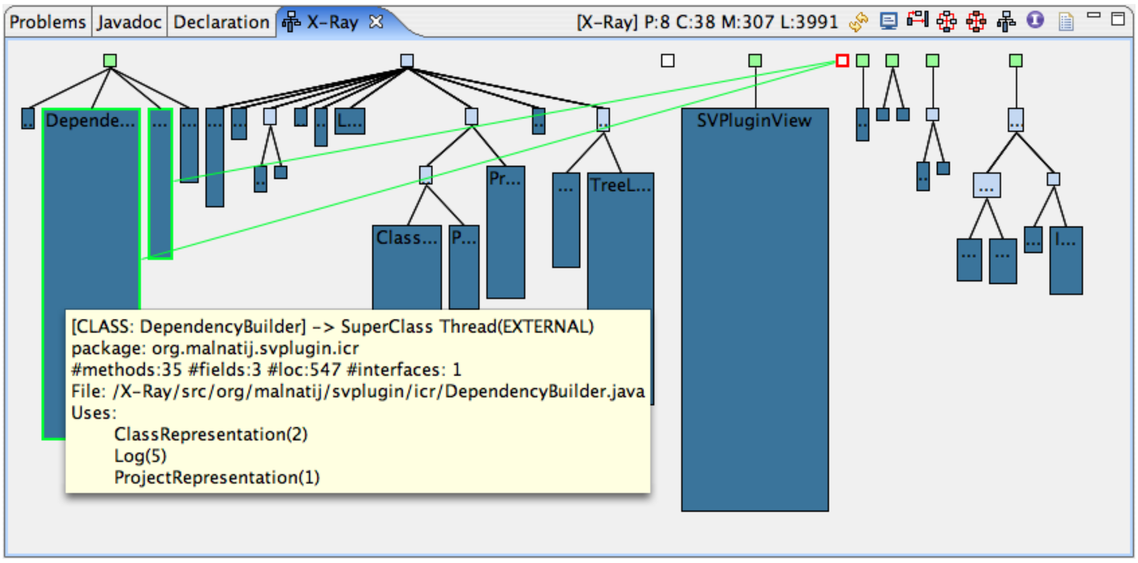
Now that we have a better understanding of the *SVPluginView* class, of its dependencies and inheritance, we can shift our attention to something that is still obscure.

The shape of the system highlights two isolated, white, nodes. They represent interfaces declared in the underlying project, potentially implemented by some of the classes represented by other nodes. If we select the interface node and we click on the *(H) Show implemented interfaces / interface implementers* action, X-Ray will create and visualize a green edge between the interface and all the classes that are implementing it, as shown in Figure 4.12. On the other hand, selecting a node that is implementing one or more interfaces and clicking on the *(H) Show implemented interfaces / interface implementers* action, the plug-in will create and visualize a green edge between the class and every interface it is implementing.

Discussion

The *System Complexity View* provided by X-Ray achieves its goal of being useful while spotting design disharmonies and it is an important aid in order to understand and explore an unknown or complex project. Anyway, the implementation of this view is still a prototype, therefore there are some of its aspects that might be improved. One of the most important is *scalability*.

Scalability is the ability of a system (in this case, the polymetric view) to handle grow-



[CLASS: DependencyBuilder] -> SuperClass Thread(EXTERNAL)
package: org.malnatij.svplugin.icr
#methods:35 #fields:3 #loc:547 #interfaces: 1
File: /X-Ray/src/org/malnatij/svplugin/icr/DependencyBuilder.java
Uses:
ClassRepresentation(2)
Log(5)
ProjectRepresentation(1)

Figure 4.12: Selecting the *IProjectAnalyzer* interface and clicking on the *(H) Show implemented interfaces / interface implementers* action, we discover that 2 classes are implementing that interface: *DependencyBuilder* and *ModelExtractor*

ing amounts of work in a graceful manner. Software systems can be huge, composed of millions lines of code and thousands classes. Visualizing 10 entities is really different than visualizing 1000 of them. Software visualization was born to provide abstraction and to give the possibility to focus on one or few aspects of a system, without getting lost in millions of details. No matter which abstraction are we using, if there are too many entities and information shown at a time, the user will not be able to exploit the provided abstraction. Therefore zooming and filters must be implemented and offered to the user while analyzing big and complex software systems.

The *System Complexity View*, so far, does not implement any filtering or zooming mechanism, this implies that it is more difficult to browse and understand a big project (that will require a lot of scrolling) than a small one.

4.3.3 Class and Package Dependency View

From the *System Complexity View*, clicking on the *(E) Show Class Dependency View* action or *(F) Show Package Dependency View* action, the plug-in will create and show, respectively, the *Class Dependency View* or *Package Dependency View*.

Description

Arranged in a bi-dimensional circle, packages or classes are linked together by dependency links, each of them with a certain weight, highlighting how strength is the dependency between entities. Nodes and edges are displayed following the metrics explained in Section 4.2.2. We say that a class has a dependency when the class uses some code implemented by another class.

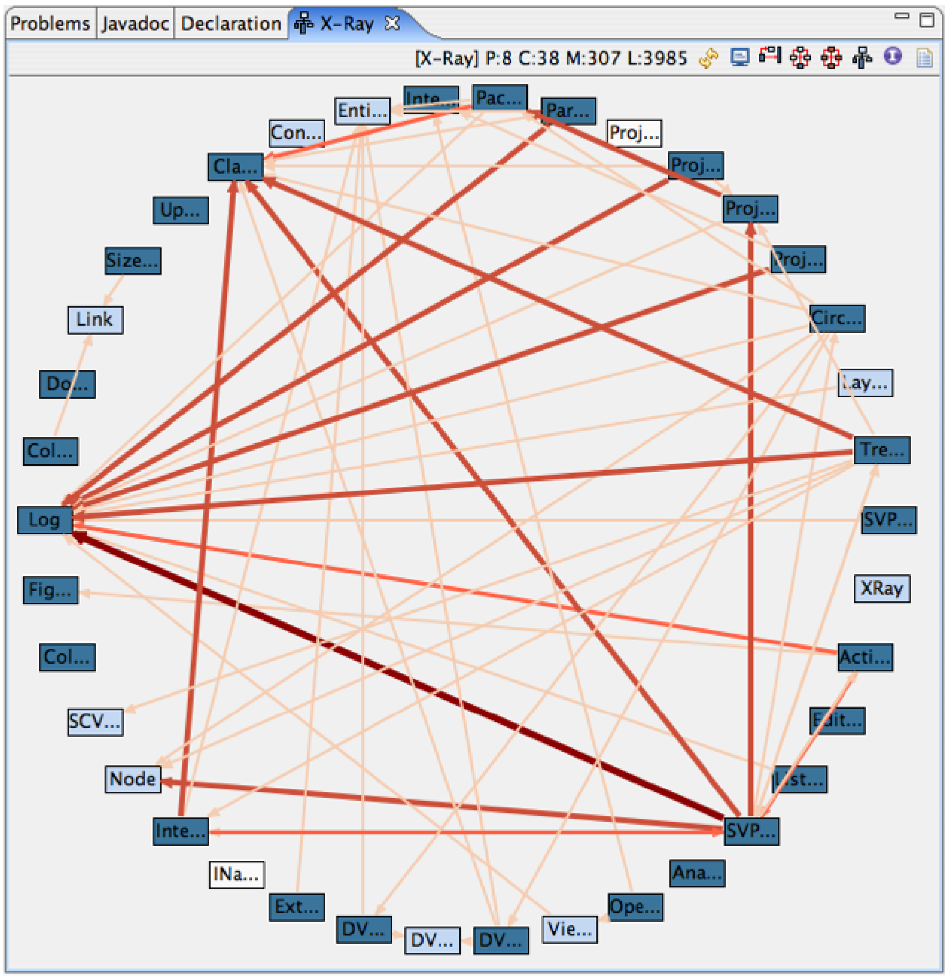


Figure 4.13: A *Class Dependency View*. Classes are arranged in a bi-dimensional circle and are connected by dependency arrows. Tooltips describe classes and dependencies. The figure shows the *Class Dependency View* for the X-Ray project.

Goal

This visualization highlights design defects, providing information about coupling and cohesion (making it easier, for example, to decide where and when to use a design patter such as *façade*). *Class and Package Dependency Views* highlight disharmonies that affect several entities at once in terms of the way they collaborate to perform a specific functionality [LM06].

Example of Usage

In order to provide an example, let us analyze the *Class and Package Dependency Views* for the X-Ray project.

Example. Once we initialized the plug-in visualizing the *System Complexity View*, we can click on the icons responsible for creating and visualizing the two dependency views. Clicking on *(E) Show Class Dependency View*, the *System Complexity View* will be replaced by a view containing something similar to Figure 4.13. We suddenly notice that classes external to the project are not visualized (external libraries might add a lot of noise to the representation) and that all the entities have the same dimensions. In terms of metrics, this view is poorer than the *System Complexity View* because it focuses exclusively on dependencies.

Weak dependencies are visualized as light and small arrows while strong dependencies are bigger and darker so that it is easy to spot which classes are the most used and which others use code and attributes defined by a many other classes. In other words, this view provides information about collaboration disharmonies.

The view evinces that the most used classes (in terms of incoming dependencies) are: *Log*, *Node*, and *ClassRepresentation*. Excessive incoming dependencies are undesirable because the more one is used by the others, the more responsible and immutable one becomes, given that any little change in the class will potentially affect all the entities that are using it. The view also evinces that classes with more outgoing dependencies are *SVPluginView*, *TreeLayout* and *CircleLayout*. Excessive outgoing dependencies are undesirable because the more one uses the others, the more it is dependent and vulnerable. These "interesting" outgoing and incoming dependencies for the X-Ray project are highlighted in Figure 4.14.

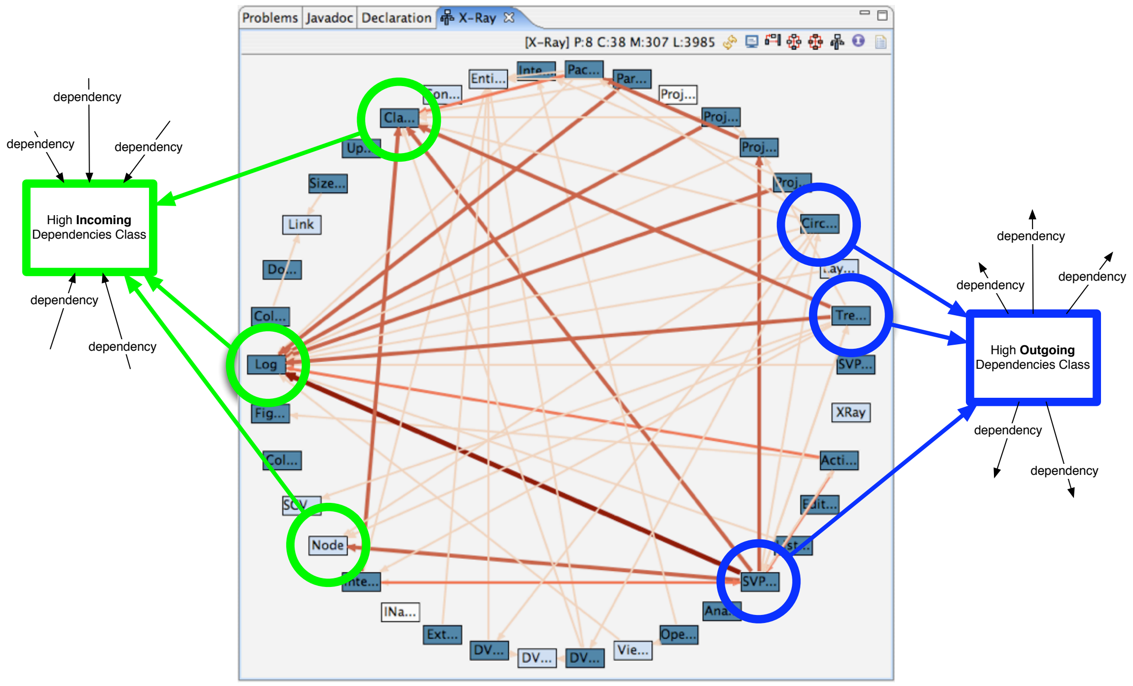
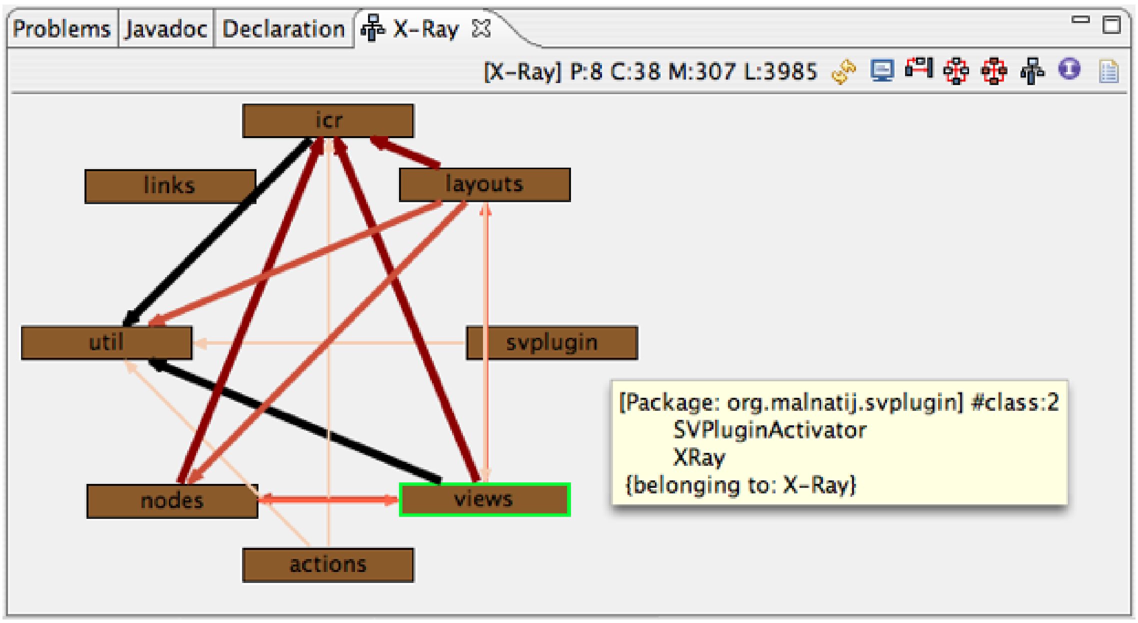


Figure 4.14: Green circles highlight classes with high incoming dependencies (*Log*, *Node*, and *ClassRepresentation*), while blue circles highlight classes with high outgoing dependencies (*SVPluginView*, *TreeLayout* and *CircleLayout*)

While browsing source code using the *Text Editor* provided by Eclipse, the node corresponding to the class that we are currently editing or reading will be centered in the polymetric view and highlighted.

The X-Ray plug-in is a small project, made up by 48 classes, therefore the current view is useful and understandable. Bigger projects might have a huge set of classes which implies an intricate and complex dependency view. Browsing the resulting view will require scrolling and resizing, making the user experience difficult and useless, given that the user will not be able to exploit the provided abstraction (handling too much information in a limited amount of space). In order to face this problem (which in some sense is a scalability problem) the X-Ray plugin provides the *Package Dependency View* which exploits the same abstraction offered by the *Class Dependency View*, but it applies that concept to packages instead of classes. Packages are containers of classes, therefore, modeling packages, we are raising the abstraction by one layer, visualizing less entities.

Even if it is not strictly needed by the project (given its limited complexity), let us say that we want to see and analyze its *Package Dependency View*. Clicking on the *(F) Show Package Dependency View* action, a figure similar to Figure 4.15 will appear.



[Package: org.malnatij.svplugin] #class:2
SVPluginActivator
XRay
{belonging to: X-Ray}

Figure 4.15: The *Package Dependency View* of the X-Ray plug-in and the tooltip for the *org.malnatij.svplugin* package. The node corresponding to package *org.malnatij.svplugin.views* is highlighted because one of the classes contained in the package is currently visualized in the *Text Editor*.

As we clearly see, there are less nodes compared to the previous view, and each node represents a package of the X-Ray plug-in. Given that Java packages follow the dot-separated naming convention, their names can be extremely long. To improve the usability, just the latest token of the package name is shown as node label. This way the most significant portion of the name is immediately visible, while the full, longer name is visible by reading the tooltip associated with each node. In this example we wanted to further investigate the *svplugin* node, positioning the mouse cursor over it and waiting for the tooltip to appear.

Discussion

As the *System Complexity View*, the *Class and Package Dependency Views* suffer from scalability. The entities are arranged in circles, therefore big project might imply a lot of space, in order to be visualized. So far, the views have limited functionalities and they do not provide any zooming, scaling or filtering mechanism. These limitations moderately affect the usability of the plug-in, therefore will have high priorities in the future work. The implementation of the *Package Dependency View* is one, first, step toward providing different abstractions and layers of abstraction, in order to offer views which focus on one or few properties, without visualizing too many entities or too much information in the same context. The *Package Dependency View* has been implemented as soon as we noticed that the *Class Dependency View* was completely useless with big projects. The user wants to browse the system without too many clicks and scrolling, having the possibility to understand a clear visualization of the whole system without being forced to analyze it piece by piece.

Figure 4.16 shows how intricate and complex might be a *Class Dependency View* of a system of 1700 classes. There are thousands of dependency arrows, classes are spread through a huge circumference; even resizing the view to the size of the whole Eclipse workbench, just a small portion of the view will be visible, too limited to be useful.

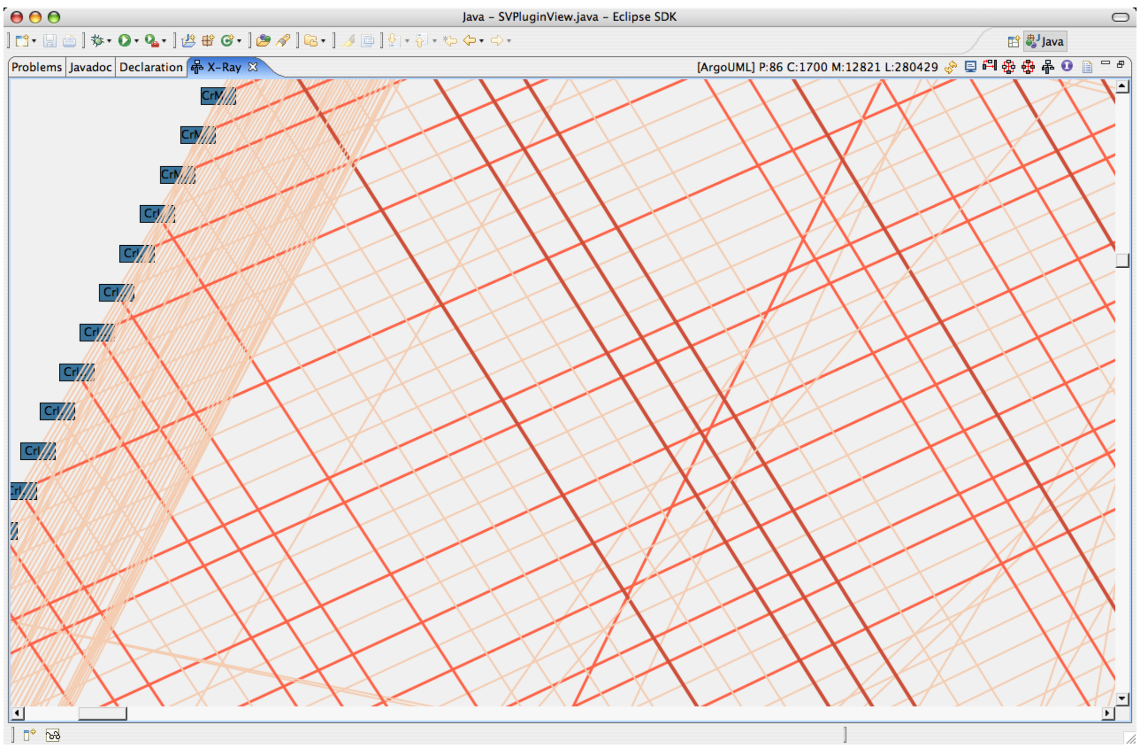

Java – SVPluginView.java – Eclipse SDK
Java
Problems
Javadoc
Declaration
X-Ray
[ArgoUML] P:86 C:1700 M:12821 L:280429

Figure 4.16: A portion of the *Class Dependency View* of the project Argo UML, composed by 1700 classes. In this particular case, visualizing big projects without being able, so far, to filter and zoom, this polymetric view results being useless.

4.4 The Plugin

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

The plug-in can be downloaded at http://atelier.inf.unisi.ch/∼malnatij/xray/, under the BSD license. Moreover, this document can be found within the X-Ray documentation on the website.

Once the plug-in has been downloaded, it must be placed in the `/plugins` folder of Eclipse, as sibling of all the other plug-ins. The next time that the framework will be executed, the X-Ray plug-in will be available to the user.

Chapter 5

Validation

5.1 Case Study

The previous chapter gave an overview of the plug-in. We described its structure, polymetric views and functionalities using as case study the plug-in itself. In the whole chapter we analyzed the X-Ray project and provided different visualization screenshots without drawing any conclusion about the design of the project highlighted by the three polymetric views.

In the next section, using the plug-in, we will try to draw meaningful conclusions about a project, exploiting different abstractions and visualizations. We will analyze the X-Ray project.

5.1.1 X-Ray

Analyzing the *System Complexity View* of the plug-in (Figure 4.10), we already noticed a ”suspicious” node: *SVPluginView* (52 methods and 670 lines of code). The size of the node is a symptom of identity disharmony: operations and classes should have a harmonious size i.e., they should avoid both size extremities (*proportion rule*, [LM06]).

If we further investigate this class analyzing its dependencies (Figure 4.11) we clearly notice that this class interacts with a lot of other classes. *SVPluginView* interacts with the class *Actions*, *Log*, *ClassRepresentation*, *ProjectRepresentation*, both the *TreeLayout* and *CircleLayout* classes as far as multiple subclasses of the *Node* class.

By its name and the comments inside its source code, it becomes clear that this class is responsible for creating and maintaining the workbench view used by our software visualization plug-in. It is therefore legal a collaboration with the *Actions* and *Log* classes as far as the the *TreeLayout* and *CircleLayout* classes (responsible, respectively, for the layout exploited by the *System Complexity View* and the *Class & Package Dependency Views*). All the other dependencies are suspicious, and require further investigations. Double-clicking on the node corresponding to the class on which we are focusing, we will be able to browse its source code and understand the details of its collaborations.

The suspicious class is one of the oldest class of the X-Ray system, its design has been revised and modified different times. Originally, it was developed in order to han-

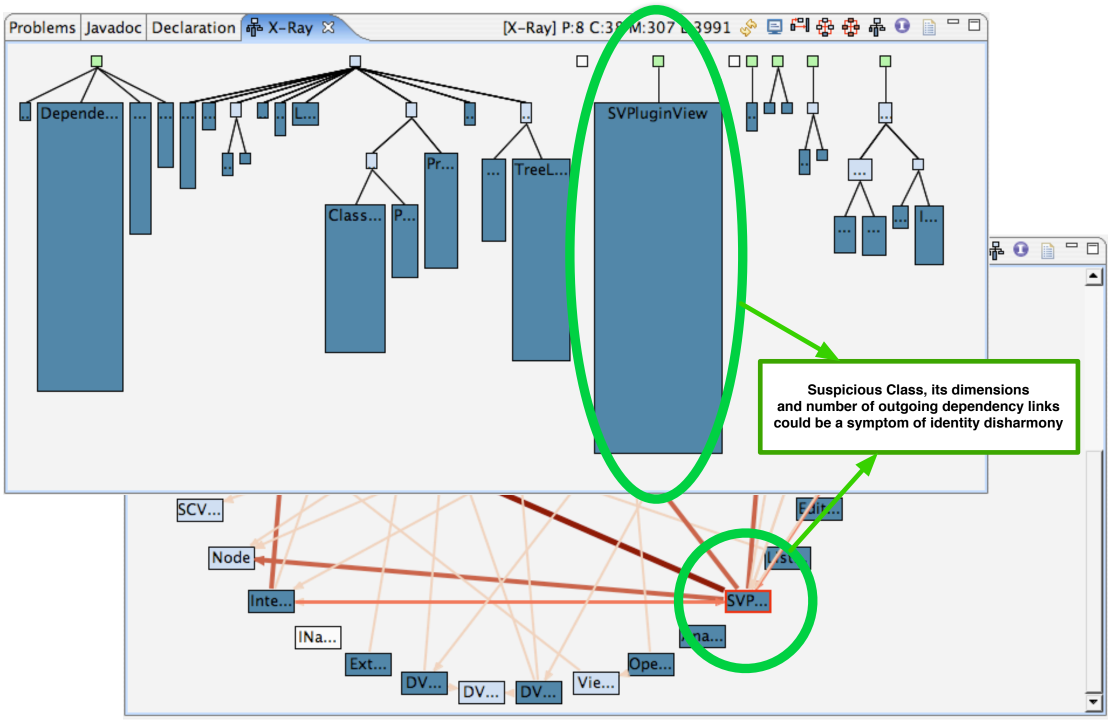
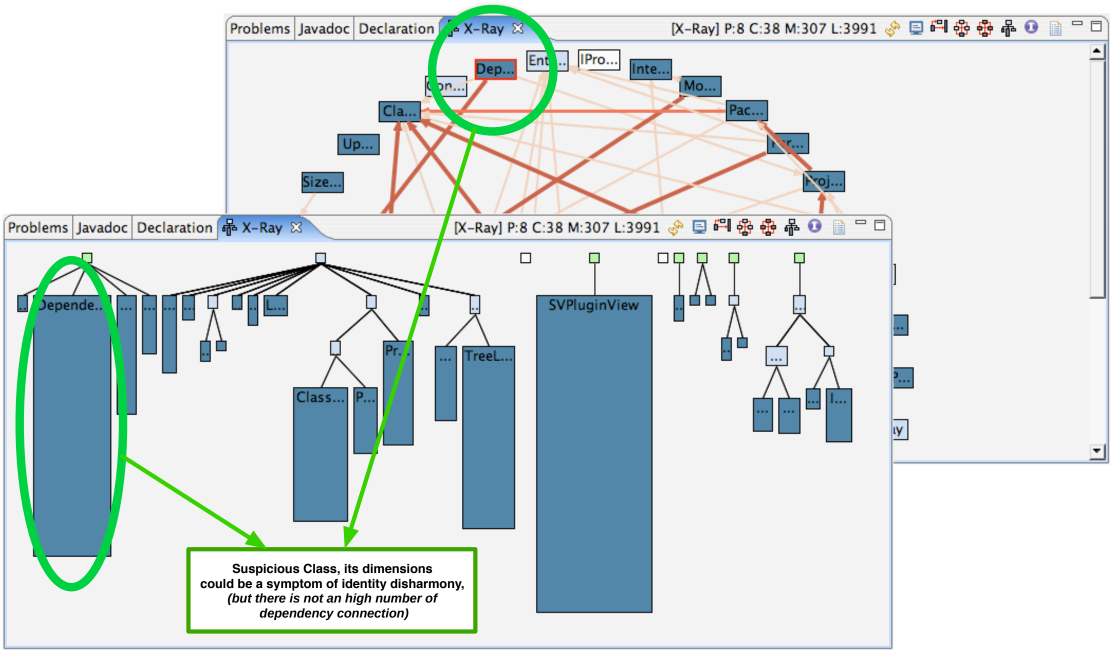


Figure 5.1: The *System Complexity View* (top) and the *Class Dependency View* (bottom) highlighting the suspicious dimension and number of outgoing dependency links for the *SVPluginView* class.

dle just a single view (the *System Complexity View*) and then, with the necessity of meeting new requirements, has been extended without any refactoring. New functionalities, implemented on top of a previous design, made this class as big and complex as X-Ray visualizes it, violating also the *presentation rule* ([LM06]): each class should present its identity by a set of services, which have one single responsibility and which provide a unique behavior.

Our analysis highlighted therefore that the *System Complexity View* can, potentially, become a *God Class*. A *God Class* is a class that 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 in the reusability and understandability of that part of the system [LM06] while in a good object-oriented design the intelligence and functionalities should be uniformly distributed among the top-level classes [Rie96]. Thus, 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.

Now that the the behavior and design of the *SVPluginView* class is clear, we can analyze the other class that catches out attention: *DependencyBuilder*, associated with a big node on the left-hand side of the *System Complexity View*. This class could seem to be a potential candidate for another *God Class* but further investigations will show that this is not the case (Figure 5.2). Analyzing the dependencies of the *DependencyBuilder* class, we notice that there are just few dependency connections, therefore this class doesn't centralize the intelligence of the system, but rather it contains something similar to a *Brain Method*, centralizing the functionalities of the class (in the same way as the *God Class* centralizes the functionality of the system). A *Brain Method* often starts as a "normal" method but then more and more functionality is added to it until it gets out of control, becoming hard to maintain [LM06].



Suspicious Class, its dimensions could be a symptom of identity disharmony, *(but there is not an high number of dependency connection)*

Figure 5.2: The *Class Dependency View* (top) and the *System Complexity View* (bottom) highlighting the suspicious dimension of the *DependencyBuilder* class but also its limited amount of dependency connections.

If we spend a little time browsing the source code of the class, we notice that the `processStatement` method is composed of a lot of switch statements, which is a symptom of a bad object-oriented design that ignores polymorphism. The *DependencyBuilder* class, therefore, can become a *Brain Class* if in the future releases of the plug-in, as scheduled, we will add even more functionality to it. To prevent this situation, before adding any functionality, we will refactor the `processStatement` method in such a way to cure its disharmonies.

Two other classes that can catch our attention for their high number of incoming dependency links are the *Log* and *ClassRepresentation*, as previously stated in Figure 4.14. Further investigating the two entities, we notice that the *Log* class just contains static methods that provide basic and trivial logging functionalities. Its implementation is static and frozen, therefore, even being used by a lot of other classes in the system, it does not have to worry about its responsibilities. For the *ClassRepresentation* class, the scenario is different. Its size is over the system's average size and is a crucial entity of the project. Being used by a lot of other classes, its implementation must take into consideration the issues generated by changing any of its implementation details.

Actually, browsing its source code, we can notice that the methods used by external entities are a sort of interface that provides a layer of abstraction between the methods and the actual implementation of their functionality. Anyway, if more and more entities will be dependent on this class, in the next releases of the plug-in, we should decide how to limit the responsibilities of this class that might affect other entities.

Chapter 6

Conclusions

6.1 Reached Goals

So far, the plug-in provides 3 different views:

- *System Complexity View*
- *Class Dependency View*
- *Package Dependency View*

They are stable prototypes with limited functionalities but mature enough to be useful while analyzing small and medium sized projects. With the *System Complexity View*, the user can spot identity disharmonies and identify design issues only by looking at the shape of the visualized system. With the *Class and Package Dependency Views* the user can spot collaboration disharmonies and identify incoming and outgoing dependencies. Exploiting the abstractions provided by all the three polymetric views, the user can analyze any system at class and package level, deepening his knowledge browsing dependencies and hierarchies.

6.2 Future Work

In the future releases of the system, we will perform some refactoring, fix some known bugs and add more features. We will also perform a *user study* which will show us how people use the tool, which are the useful parts and which are the ones that must be improved. Moreover, we will provide a set of meaningful tests.

6.2.1 Features

- *More polymetric views.* As future work, we are planning to implement at least one more view: the *System Hotspot* [LM06]. In this view nodes represent classes, while their size represent the number of methods they define. Color represents the class type. This simple view helps to identify large and small classes and scales up to very large systems. The nodes are sorted according to the number of methods which make the identification of behavioral outlier easy. If the implementation of this view will be successfully, we might take into consideration the idea of implementing any other view that will be requested by our beta testers. Moreover, all the views implemented so far are displayed in a single workbench view, therefore the user can analyze them one at a time. In the next releases, X-Ray will create a dedicated workbench view for every polymetric visualization, to offer a complementary representation of the system without forcing the user to manually click and switch view.

- *Scalability.* So far, X-Ray suffers from scalability especially in the *Class and Package Dependency Views*. In order to solve this problem we will provide different options and actions. The user will be able to visualize the whole system or just a subset of it, he will moreover be able to zoom in and out his current view, showing or hiding details. Filters and semantic zoom will be implemented.
- *Options*. The user will be able to change and set the color and metrics used by every entity, as far as their default sizes. It will be possible to dynamically adjust parameters while analyzing a view, in order to show, hide or spot out different entities or relationships.
- *Persistency.* As future work, we will implement the possibility to move nodes from their default position to new, user-defined, locations. Moreover this changes in the layout will be preserved through executions, being stored.
- *Incremental data collection.* This feature regards both the time spent in the initialization of the plug-in and the time spent while refreshing the visualized system. One of the most intensive, time and space consuming task of the system is the one that collects information about dependencies and fills the plug-in's data structures. It is a particular heavy task because it has to create different parse trees and resolve the binding for all the entities. So far, this task is executed by one thread during the initialization, while in the future it will be exclusively executed on demand and incrementally (just handling entities for which it was required). This will result in an improved usability, decreasing the user's waiting time.

6.2.2 Known bugs

So far, just two known bugs have been noticed.

Dependencies

While building the dependency data structures for the system, not all dependencies are correctly handled. The actual version of the plug-in (1.0) doesn't handle dependencies declared outside *expression statements*.

Screenshot

While creating a screenshot of the currently selected polymetric visualization, if the visualization is bigger than a certain size (that still have to be quantified), a *memory out of bounds* exception might occur. This issue, so far, can be solved by increasing the virtual memory associated to the Java Virtual Machine.

We know where these two functionalities are missing or buggy in our code and where to look for documentation, but the limited amount of time left before the upcoming release did not permit to fix them.

Appendix A

Implementation

In the following sections we will describe some of the most important and interesting design decisions about the X-Ray plug-in.

A.1 Architecture of X-Ray

A.1.1 UML Diagrams

The inheritance tree can be analyze executing the plug-in and visualizing the *System Complexity View*, therefore it will be a useless duplication to provide a UML class diagram containing this redundant information. Anyway, the most important part of the plug-in is its core, the *internal code representation*. To better understand its components, let us include a UML class diagram modeling the classes that are involved.

Figure A.1 shows how every Java entity is modeled by an `EntityRepresentation`. Moreover, packages and classes are contained within another entity (respectively, a project and a package), therefore we created the `ContainedEntityRepresentation` class. The `InternalCodeRepresentation` class containes a `ProjectRepresentation` that is made up by zero or more `PackageRepresentations` that are composed by zero or more `ClassRepresentation`.

A.2 Internal Code Representation

One of the most important part of the plug-in is the *model extractor*, responsible for creating an internal code representation which reflects the underlying source code. The *model extractor* is composed by 2 phases: the first is divided into two subtasks, the *hierarchy builder* and *metric computer*, while the second is the *dependency builder*.

A.2.1 Model Extractor

This is the first task performed by the plug-in immediately after being initialized. It is responsible for analyzing the source code of the project that the user wants to visualize. The class that performs this analysis is `ModelExtractor.java`. All the tasks executed by the plug-in are, by default, executed by the Eclipse *User Interface thread*,

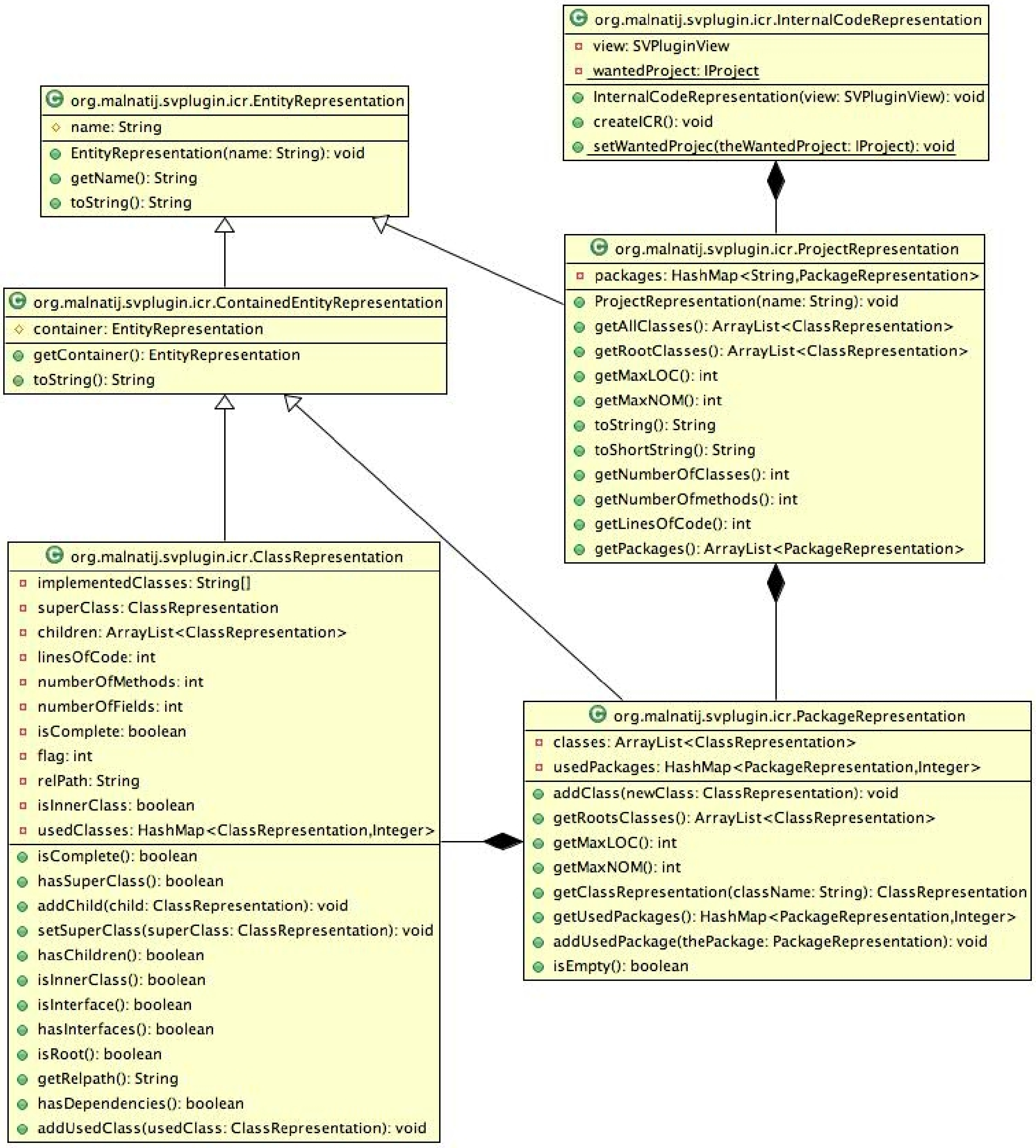

org.malnatij.svplugin.icr.InternalCodeRepresentation
view: SVPluginView
wantedProject: IProject
InternalCodeRepresentation(view: SVPluginView): void
createICR(): void
setWantedProjec(theWantedProject: IProject): void
org.malnatij.svplugin.icr.EntityRepresentation
name: String
EntityRepresentation(name: String): void
getName(): String
toString(): String
org.malnatij.svplugin.icr.ProjectRepresentation
packages: HashMap<String,PackageRepresentation>
ProjectRepresentation(name: String): void
getAllClasses(): ArrayList<ClassRepresentation>
getRootClasses(): ArrayList<ClassRepresentation>
getMaxLOC(): int
getMaxNOM(): int
toString(): String
toShortString(): String
getNumberOfClasses(): int
getNumberOfmethods(): int
getLinesOfCode(): int
getPackages(): ArrayList<PackageRepresentation>
org.malnatij.svplugin.icr.ContainedEntityRepresentation
container: EntityRepresentation
getContainer(): EntityRepresentation
toString(): String
org.malnatij.svplugin.icr.ClassRepresentation
implementedClasses: String[]
superClass: ClassRepresentation
children: ArrayList<ClassRepresentation>
linesOfCode: int
numberOfMethods: int
numberOfFields: int
isComplete: boolean
flag: int
relPath: String
isInnerClass: boolean
usedClasses: HashMap<ClassRepresentation,Integer>
isComplete(): boolean
hasSuperClass(): boolean
addChild(child: ClassRepresentation): void
setSuperClass(superClass: ClassRepresentation): void
hasChildren(): boolean
isInnerClass(): boolean
isInterface(): boolean
hasInterfaces(): boolean
isRoot(): boolean
getRelpath(): String
hasDependencies(): boolean
addUsedClass(usedClass: ClassRepresentation): void
org.malnatij.svplugin.icr.PackageRepresentation
classes: ArrayList<ClassRepresentation>
usedPackages: HashMap<PackageRepresentation,Integer>
addClass(newClass: ClassRepresentation): void
getRootsClasses(): ArrayList<ClassRepresentation>
getMaxLOC(): int
getMaxNOM(): int
getClassRepresentation(className: String): ClassRepresentation
getUsedPackages(): HashMap<PackageRepresentation,Integer>
addUsedPackage(thePackage: PackageRepresentation): void
isEmpty(): boolean

Figure A.1: UML class diagram for the classes involved in the *internal code representation*

therefore the *ModelExtractor*, its subtasks and the *Dependency Builder* (implemented in the `DependencyBuildre.java` file) have been created as separated threads, in order to minimize the impact of their intensive tasks on the overall Eclipse user interface (which, otherwise, would remain frozen during their computations).

Hierarchy Builder

The *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. The X-Ray's *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.

Once the *Hierarchy Builder* performed its task, the system, previously seen as a set of files and directories, slowly takes shape.

Metric Computer

The *Metric Computer* is responsible for collecting information about the metrics used by the *System Complexity View*. It retrieves the number of methods, attributes (even if, so far, they are not visualized in any view other than in the class tooltip), number of lines of code, etc..

As soon as the *Metric Computer* thread finishes its task, the *Model Extractor* has filled the *internal code representation* with all the information needed by the *System Complexity View* to create and show its graphical representation in the workbench (Figure A.2).

A.2.2 Dependency Builder

The *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.

The task performed by this operation is so time and space consuming that we decided to separate it by the *Metric Computer* (originally, *Dependency Builder* was a phase of the *Metric Computer*). Creating a new component for this task, we had the possibility to execute it on demand, as soon as the user visualizes the *Class or Package Dependency Views*.

Later on, adding the possibility to visualize dependency links *even* in the *System Complexity View*, we had to execute the *Dependency Builder* at initialization time, coming back to the original design (but still maintaining the different threads). That's the reason we decided to provide, in the next releases of the plug-in, an *Incremental Dependency Builder* that will execute on a limited set of entities on request, improving the usability of X-Ray.

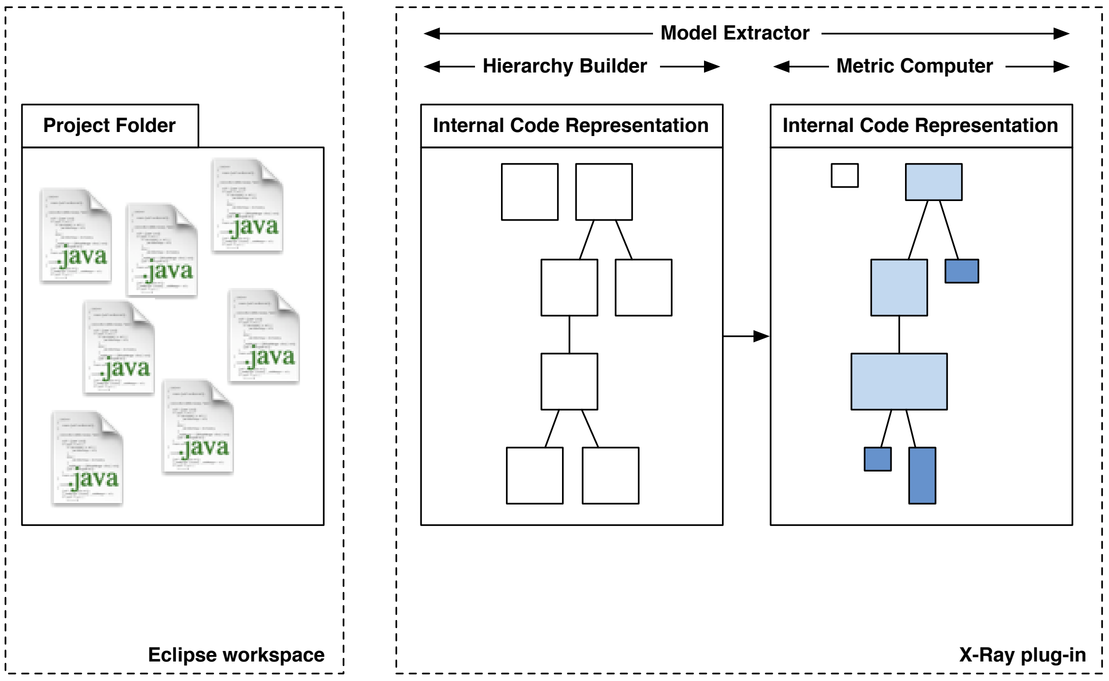


Figure A.2: The source code that composes the project lays in the project's folder, inside the Eclipse workspace. This folder is made up by Java files, binaries, preferences and subfolders. Once the plug-in executes the *Hierarchy Builder* phase, its *ICR* is filled by nodes representing Java entities and their inheritance hierarchy. After the completion of the *Metric Computer* phase, the *ICR* contains information about all the computed metrics, reflected in the appearance of every node. At this point, the system could be able to visualize the *System Complexity View* with limited functionalities (without the dependency-related actions).

As soon as the *Dependency Builder* collected all the data about the metrics used by the *Class and Package Dependency Views*, the *internal code representation* contains all the information needed by X-Ray to create and visualize all its views.

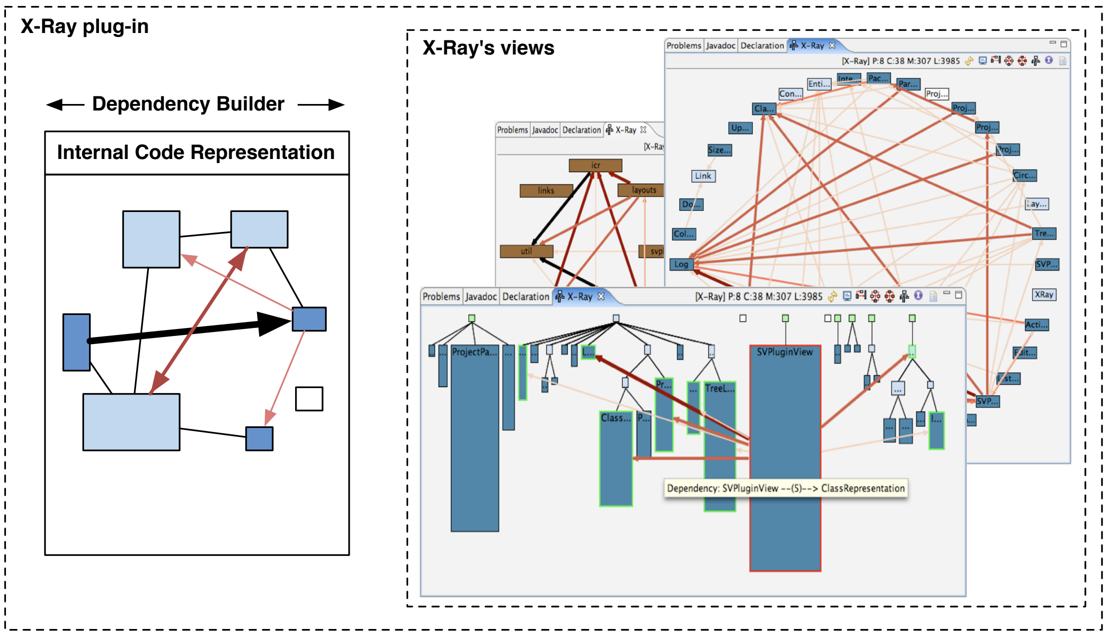

X-Ray plug-in
Dependency Builder
Internal Code Representation
X-Ray's views
Dependency: SVPluginView --(5)--> ClassRepresentation

Figure A.3: Once the *Dependency Builder* phase has been accomplished, the *ICR* contains also information about dependencies between entities. The plug-in is therefore able to display all of his polymetric views to the user.

List of Figures

List of Tables

Bibliography

[Lan03] Michele Lanza. Codecrawler – a lightweight software visualization tool. In *Proceedings of VisSoft 2003 (2nd International Workshop on Visualizing Software for Understanding and Analysis)*, pages 51–52. IEEE CS Press, 2003.

[LL06] Mircea Lungu and Michele Lanza. Softwarenaut: Exploring hierarchical system decompositions. In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*, pages 351–354, Los Alamitos CA, 2006. IEEE Computer Society Press.

[LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[LMSW03] Rob Lintern, Jeff Michaud, Margaret-Anne Storey, and Xiaomin Wu. Plugging-in visualization: experiences integrating a visualization tool with eclipse. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 47–ff, New York, NY, USA, 2003. ACM Press.

[ML94] J. Kidd M. Lorenz. Object oriented software metric. 1994.

[Rie96] Arthur Riel. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA, 1996.