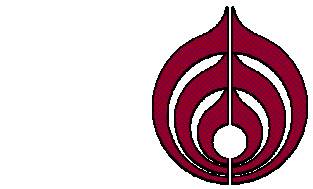

Växjö University

THE VIZZANALYZER HANDBOOK

Thomas Panas
Rüdiger Lincke
Welf Löwe

October 2005

TECHNICAL REPORT

Abstract

THE VIZZANALYZER FRAMEWORK is a composition system to support the rapid construction of reverse engineering tools. In addition, it allows, when instantiated, to measure quality aspects of software systems. We refer to our instantiation as the `VizzAnalyzer` (application).

This handbook contains information about the VIZZANALYZER FRAMEWORK, it's instantiation, usage and extensibility. For this, we explain the VIZZANALYZER FRAMEWORK itself, as well as the two sub-frameworks, the ANALYZER FRAMEWORK and the VIZZ3D FRAMEWORK. In addition, we provide examples of its usage.

Table of Contents

Abstract i

The `VizzAnalyzer` **1**

1 The `VizzAnalyzer` **3**
1.1 The `VizzAnalyzer` 3
1.2 Installation 5
1.2.1 Prerequisites 6
1.2.2 Obtaining the Software 7
1.2.3 Extracting the Installation Archive 7
1.2.4 Adapting the Startup Scripts 8
1.2.5 Specific Installation Steps for UNIX/Linux 8
1.2.6 Running the `VizzAnalyzer` 9
1.2.7 Setting up Eclipse 9

Frameworks 11

2 The VIZZANALYZER FRAMEWORK **13**
2.1 The Architecture 13
2.2 Usage 16
2.3 Design 17

3 The ANALYZER FRAMEWORK **21**
3.1 Usage 22
3.2 Design 22
3.3 Example Analysis Plug-In 23
3.4 Analyses 24

3.4.1 Metrics . . . 24
3.4.2 Focusing algorithms . . . 26
3.4.3 High-level analysis . . . 28

4 The VIZZ3D FRAMEWORK 31
4.1 `Vizz3D` . . . 32
4.2 Usage . . . 34
4.3 Design . . . 35
4.4 VisGraph . . . 39
4.5 Example Layout Plug-In . . . 40
4.6 Layout Algorithms . . . 41

Examples 46

5 Example Usage of the `VizzAnalyzer` 47
5.1 The Front-End . . . 48
5.2 The Analysis . . . 52
5.3 The Model Mapping . . . 55
5.4 The Visualization . . . 57

6 First Contact Analysis - GRAIL 59
6.1 Summary . . . 60
6.2 Global Statistics . . . 61
6.3 Analysis of Architecture and Structure . . . 62
6.3.1 System Architecture . . . 62
6.3.2 Evaluation of the Architecture . . . 63
6.3.3 Inheritance Structure . . . 63
6.4 Design Metrics . . . 66
6.5 Complexity Metrics . . . 70
6.6 Conclusions . . . 71

7 First Contact Analysis - `VizzAnalyzer` 73
7.1 Summary . . . 73
7.2 Global Statistics . . . 74
7.3 Analysis of Architecture and Structure . . . 75
7.3.1 System Architecture . . . 75
7.3.2 Evaluation of the Architecture . . . 76
7.3.3 Intended vs. actual Architecture . . . 78

7.4 Design Metrics 82
7.5 Complexity Metrics 85
7.6 Conclusions 87

Bibliography **89**

Part I

The VizzAnalyzer

Chapter 1

The VizzAnalyzer

The VizzAnalyzer is the instantiation of our reusable framework: the VIZZANALYZER FRAMEWORK. The VIZZANALYZER FRAMEWORK is described in Section 2.

1.1 The VizzAnalyzer

The VizzAnalyzer is our instantiation of the VIZZANALYZER FRAMEWORK. Currently, the VizzAnalyzer consists of the following reverse engineering components (each representing a separate tool):

- FRONT-ENDS

    1. **Recoder** [LNAH01] is a Java framework for source code meta programming aiming at delivering an infrastructure for Java analysis and transformation tools. Recoder is used for source code information extraction. We have extended this API with our own data extraction client.

2. **Kenyon** [BKZ05] is a data extraction, preprocessing, and storage backend designed to facilitate software evolution research.

- ANALYSES

1. **CrocoPat** [BL03] manipulates relations of any arity. Its query and manipulation language is based on first-order predicate calculus. CrocoPat is used for program analysis.
2. ANALYZER FRAMEWORK is our own analysis framework component. It allows to perform focusing on program information, i.e. aggregation, filtering and merge of information. Moreover, the `Analyzer` allows to perform advanced analysis, such as architecture recovery.

- VISUALIZATIONS

1. **yEd** [yed04] is a Java graph editor that can be used to generate drawings and apply automatic layouts to all kinds of diagrams and networks. It is used for information visualization in 2D.
2. **Excel** is used for statistic analyses on metrics and their visualizations.
3. **WilmaScope** [wil04] is a Java3D application which creates real time 3d animations of dynamic graph structures. It is used for program visualization.
4. VIZZ3D FRAMEWORK, our own 3D visualization framework, allowing the illustration of program information. Various layout algorithms can be added at run-time allowing visual complexity reductions. Bindings specify the mapping of metric results with visual properties (such as height, width, type, color etc.) in order to emphasize certain aspects of an analyzed software system.

The `VizzAnalyzer` is a stand-alone tool for analyzing and visualizing structure and internal quality of large software systems. It extracts information from a system's source code, performs further analyses and applies quality metrics. Finally it visualizes the results, e.g. with `Vizz3D`. Further programming languages, analyses/metrics and visualization tools can be integrated using the VIZZANALYZER FRAMEWORK. The `VizzAnalyzer` is easily adaptable to the quality needs of a certain company or project and can be harmoniously integrated in existing development processes.

Screenshot 1.1 shows the control panel of the `VizzAnalyzer`. In the "explorer" view on the left, the `VizzAnalyzer` displays the currently available analysis results. Details of a selected result are displayed in the table views on the right. The status area at the bottom displays generated console output.

For more information about the VizzAnalyzer, please refer to `www.arisa.se`.

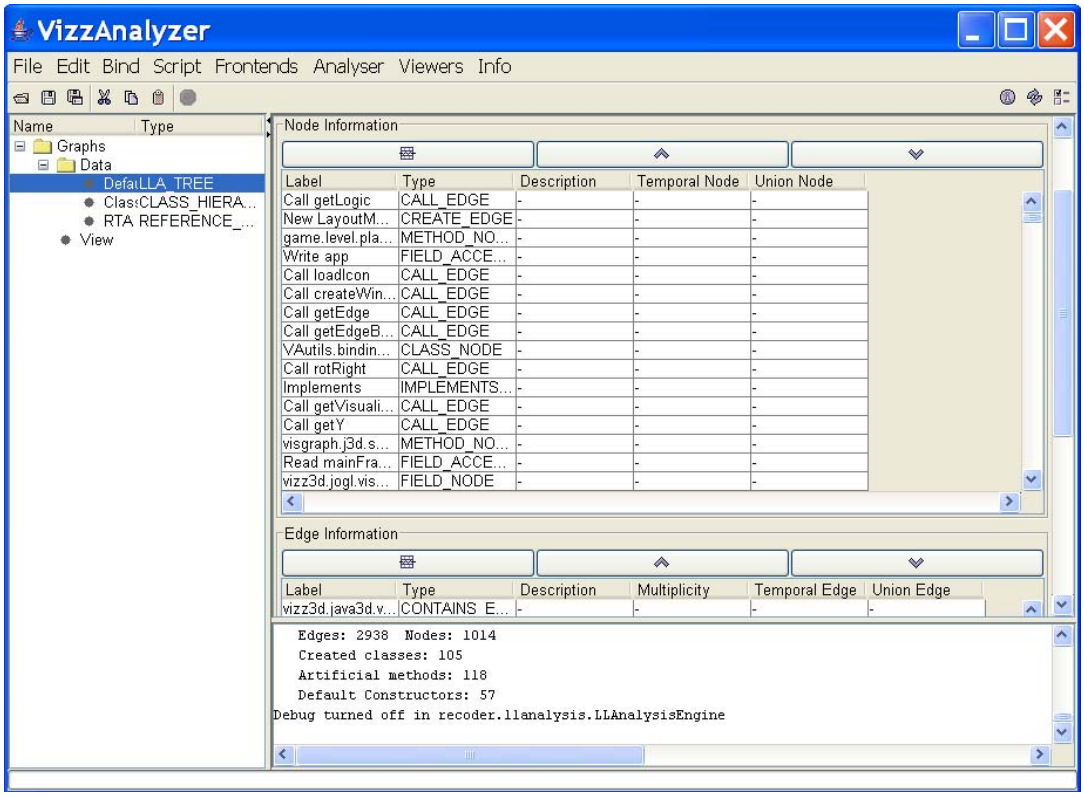

Figure 1.1: The `VizzAnalyzer`

1.2 Installation

This installation is described for `VizzAnalyzer 1.0b3`, released 1st October 2005. For other versions please view the according readme.txt for installation instructions.

The `VizzAnalyzer` is currently distributed as a user and developer version. Both versions contain the `VizzAnalyzer`, an instantiation of the ANALYZER FRAMEWORK (cf., Section 3) and an instantiation of the VIZZ3D FRAMEWORK (cf., Section 4), as well as several plug-ins. The `Vizz3D`, our instantiation of the VIZZ3D FRAMEWORK supports

both Opengl (JOGL) and Java3D[1] as graphical APIs. The developer version contains further analyses (of Analyzer) and layout algorithms (of Vizz3D) as source code. In this way, new analyses and layouts can easily be extended.

1.2.1 Prerequisites

Hardware
The VizzAnalyzer should run on every hardware where a JVM is available. Additionally for Vizz3D it is highly recommended to have a 3D graphics accelerator supporting OpenGL available. For efficiency a minimum of 512MB of RAM is recommended. More memory might be required depending on the size of the software system being analyzed. Depending on the complexity of the visualization a better system might be required (recommended: Pentium 4, 2.6GHz, 1024MB RAM, 3D accelerator 128MB (e.g. GeForce 4)).

Software

- *Java 2 Standard Edition Runtime Environment 1.4.2 or higher.* Tested with JRE 1.4.2 and 1.5.0-b04,
- *Java bindings for OpenGL.* JOGL is an implementation of the Java bindings for the OpenGL API. It is designed to provide hardware-supported 3D graphics to applications written in Java. Tested with version 1.0-b10 and 1.1.1[2]. JOGL is necessary for Vizz3D OpenGL version[3]. It is available at: https://jogl.dev.java.net/.
- *Java3D 1.3.1* Java3D is necessary for the Vizz3D Java3D version.

The .jar files from the JOGL and Java3D installation need to be in the class path, if you want to run the particular version of Vizz3D. If you install them according to their installation instructions, they will be automatically available in the Java class path.

Installation overview
The installation consists out of the following steps:

1. Obtaining the installation archive

[1] The Java3D version of Vizz3D is currently included for compatibility reasons, but will soon be deprecated.

[2] Note that version 1.1-b10 or higher is required, since the thread implementation changed in this version and our implementation relies on the new threading model

[3] Note that in this version Java3D is still included as an alternative to OpenGL. However, since Java3D can not keep up with OpenGL regarding performance, we decided not to support the Java3D version in the future

2. Extracting the installation archive to the installation directory
3. Adapting the startup scripts
4. UNIX/Linux specific installation steps (setting file attributes)
5. Running the `VizzAnalyzer` or `Vizz3D`
6. Setting up Eclipse (developer version)

1.2.2 Obtaining the Software

To obtain a copy of `VizzAnalyzer` please contact us. We will tell you, where you can download the `VizzAnalyzer`. Currently the `VizzAnalyzer` is distributed as a archive in the ZIP format containing the installation files.

1.2.3 Extracting the Installation Archive

Extract the downloaded archive into a directory of your choice, e.g. *C:\VA*. Please avoid spaces in the path to avoid additional operation system dependent problems. The content of this directory should look for the user version as follows:

- **[DIR] binVA** is the directory containing executables of the `VizzAnalyzer` and `Vizz3D`.
- **[DIR] components** contains external and in-house jar-files necessary for the execution.
- **[DIR] examples** contains example graph files(.gml) and example projects for parsing.
- **[FILE] readme.txt** contains last minute information.
- **[FILE] runVA** the Linux/Unix shell script for the `VizzAnalyzer`
- **[FILE] runVizz3D** the Linux/Unix shell script for `Vizz3D`
- **[FILE] runVA.bat** the Windows batch file for the `VizzAnalyzer`
- **[FILE] runVA_auto.bat** utility batch file for Windows to run the `VizzAnalyzer`
- **[FILE] runVizz3D.bat** the Windows batch file for `Vizz3D`

The development version contains some additional files and folders being part of an Eclipse IDE Java project:

- **[DIR] bin** contains binaries that have been compiled with the included sources (empty, but part of the Eclipse project).

- **[DIR] src** is the source directory for the development version. The files inside can be easily modified and extended with a IDE, as e.g. Eclipse.
- **[FILE] .classpath**
- **[FILE] .project** Those two files specify a generic Eclipse project, which can just be added to your Eclipse environment. All necessary components and directories are configured automatically.
- **[FILE] VizzAnalyzer.launch**. Eclipse launch configuration to run VizzAnalyzer from within Eclipse.
- **[FILE] Vizz3D (JOGL).launch**. The Eclipse launch configuration to run Vizz3D (JOGL version) from within Eclipse.
- **[FILE] Vizz3D (Java3D).launch**. This is the Eclipse launch configuration to run Vizz3D (Java3D version) from within Eclipse.

1.2.4 Adapting the Startup Scripts

After extracting the VizzAnalyzer to the installation folder it is necessary to adapt the startup scripts.

Windows
If Java is in the path no modifications in the start scripts should be required. Otherwise you need to modify the call of the *java.exe* in the *runVA_auto.bat* and *runVizz3D.bat*. You might want to adapt the settings for the Java heap size according to your needs and resources. Modify -Xms500m and -Xmx800m to specify the minimal and maximal Java VM heap size.
Note: *runVA_auto.bat* is optimized for use in Windows XP and builds the class path automatically by parsing the components folder. It might be necessary to modify it in other versions, like Windows 2000, so that the class path is available.

UNIX
The va_home variable needs to be set according to the location where the VizzAnalyzer is extracted to. Edit the files: *runVA* and *runVizz3D*. Specify in line 3 *va_home* your installation folder. Replace */home/rle/va* with the absolute path of your installation folder. Also adapt the Java VM heap parameters as needed.

1.2.5 Specific Installation Steps for UNIX/Linux

You need to make the start scripts executable by setting their executable attribute using:

```
chmod +x runVA
chmod +x runVizz3D
```

Additionally you need to make the CrocoPat plug-in executable. Replace va_home with your installation path.

```
cd va_home/binVA/plugIns/analysisPlugIns/crocopat/bin
```

Linux

```
chmod +x crocopat-2.1.1_linux
```

UNIX

```
chmod +x crocopat-2.1.1_solaris
```

1.2.6 Running the `VizzAnalyzer`

To run `VizzAnalyzer` or `Vizz3D` use the start scripts.

Windows

- Double click on runVA.bat for `VizzAnalyzer`. This file calls the runVA_auto.bat making sure, that late variable expansion is enabled.
- Double click on runVizz3D.bat for `Vizz3D`.
- Optionally you can also run the scripts form command line.

UNIX/Linux

- Run the runVA start script for `VizzAnalyzer`.
- Run the runVizz3D start script for `Vizz3D`.

1.2.7 Setting up Eclipse

If you obtained the developer version of `VizzAnalyzer` your installation folder contains as described above the source code and an Eclipse project definition. The project definition and launch configuration has been created with Eclipse 3.1.1.

To open the project in Eclipse import it simply into your workspace:

1. Choose *File* → *Import...*.
2. Select *Existing Projects into Workspace*, then click, *Next*.
3. Select the root directory, either enter the path or click *Browse...*, browse to the folder you extracted `VizzAnalyzer` in.

4. Check in the Projects list: *VizzAnalyzer_1_0b*.
5. Click *Finish*.

After importing the project you will find in the list of available run configurations the entries: *VizzAnalyzer, Vizz3D (JOGL), and Vizz3D (Java3D)*.

Part II

Frameworks

Chapter 2

The VIZZANALYZER FRAMEWORK

The VIZZANALYZER FRAMEWORK is a reusable framework for the rapid composition of reverse engineering tools reusing arbitrary reverse engineering components (tools). Therefore, it is extensible for new programming languages, analyses, and visualization tools.

2.1 The Architecture

The framework consists of the *framework-core*, *frozen-spots* and *hot-spots*. The framework-core is responsible for communicating information between the different reverse engineering components connected to the framework. It has the functionality of a controller and information converter. The frozen-spots are in-house and externally developed reusable components supporting the framework-core with main functionalities. For instance, configurations necessary for reverse engineering tool compositions reuse our tiny-xml editor, a tool reused by the framework-core. Hot-spots are technically realized as directories and allow the simple and fast connection of arbitrary reverse engineering components with the framework.

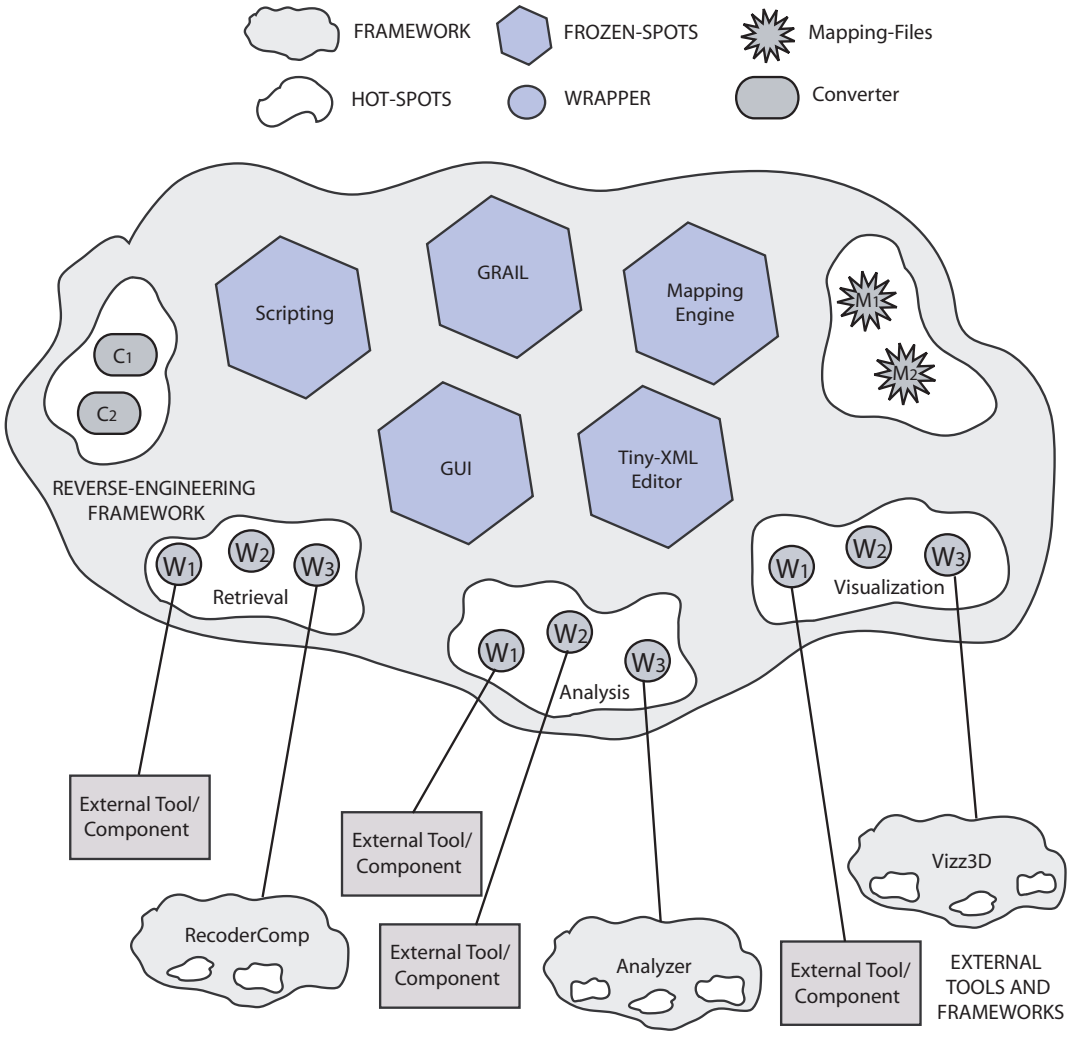

Figure 2.1: The VIZZANALYZER FRAMEWORK's Architecture

Figure 2.1 shows a sketch of our framework. In detail, our framework consists of the following frozen-spots:

- *GUI*. To allow for a fast and easy use of the VIZZANALYZER FRAMEWORK it comes with a predefined GUI, which is filled at run-time with the contents of the hot-spots of the framework. Depending on the components plugged to the framework, the menu in the GUI differs in providing component specific items. The look of the

predefined GUI itself is fixed and can only be changed by a framework developer. However, because of re-usability in mind at the development of the framework, the GUI component can easily be exchanged.

- *GRAIL* is our own graph library. It is used as an internal data representation for the VIZZANALYZER FRAMEWORK. This representation consists of an annotated graph (instantiated from our graph library), where each graph entity (nodes, edges, and the graph itself) has a data object and a set of predicates attached to it. The graph entity predicates are used as a simple dynamic type system [PLL04a]. A tool is applicable to a graph if the graph has predicates satisfying the pre-conditions of the tool. Furthermore, our typing is dynamic in the sense that we do not have a fixed set of predefined predicates. The framework itself knows nothing about the individual predicates, it just provides interfaces to allow the checking of the predicates.
- *Mapping Engine*. Our framework architecture distinguishes the domains of software analysis and information visualization, and their respective program models [LP05]. To support the flexible and rapid compositions of reverse engineering tools, we allow with our mapping engine to map the models of the analysis domain to the information visualization domain. This mapping can be configured and performed at run-time without need for being previously hand-coded, or hard-coded during compile-time.
- *Tiny-XML Editor*. The editor is our stand-alone tool for configuring XML documents. It is mainly used to configure mapping files and other configuration files online.
- *Scripting*. In order to perform a sequence of operations between plugged components repeatedly, we use the Java BeanShell, a free Java source interpreter with object scripting language features. BeanShell executes standard Java statements and expressions directly without first compiling them [Mah05]. This is in particular interesting for making analysis repeatable, and for composing complex analysis steps out of simple ones.

The hot-spots of the framework are:

- *Wrapper*. The VIZZANALYZER FRAMEWORK consists of three variation points: *front-end*, *analysis* and *visualization*. Each variation point (technically realized as a directory) contains plug-in classes (wrappers) that extend predefined interfaces and are mapped to a menu structure in the GUI. These wrappers are read in dynamically on start up to provide the framework with the different reverse engineering components (third party tools). The wrappers just contain delegations to the actual component implementations and are accessible through the menu. In addition, a wrapper checks preconditions that must hold on data moved between components,

see also [PLL04a]. If necessary, wrappers also perform data conversion and other adaptations. For the former, predefined converters are used.

- *Converter*. A converter transforms external exchange formats, such as GML, to our own internal data representation (GRAIL), and vice versa. Converters are developed once and kept for reuse. Our framework contains a collection of such converters and allows for easy implementation of new ones.
- *Mapping-Files*. The configuration of mappings between the models of the software analysis and information visualization domain, is performed within so called mapping-files. Mapping files are reusable XML-documents, which reside at a separate variation point of the framework. They are easy to create and to adapt XML files.

More information on the VIZZANALYZER FRAMEWORK is available from the latest publications [PLL04b, PLL05, LP05, PS05, PLLL05].

2.2 Usage

The framework's *usage* is divided into three stages:

1. *Extension*. At design-time, the developer may extend the framework with reverse engineering components. Therefore, the developer develops or reuses wrappers and converters, cf. Figure 2.2. Compiled wrappers and converters are collected for reuse. They are, however, deployed separately.

2. *Composition*. The user composes a reverse engineering tool by deploying a selected set of compiled wrappers into the variation points of the compiled application, cf. Figure 2.2. Note that the selection and deployment of wrappers is not only possible at compile-time but even at run-time.

3. *Application run*. At run-time, a user interactively performs program retrieval, analysis and visualization, cf. Figure 2.2. Moreover, wrappers may be exchanged, removed or added at run-time. To describe the model mapping, a user must write or reuse mapping files (XML-files). If required, interactive actions may be recorded, saved and reused at a later program run (scripting).

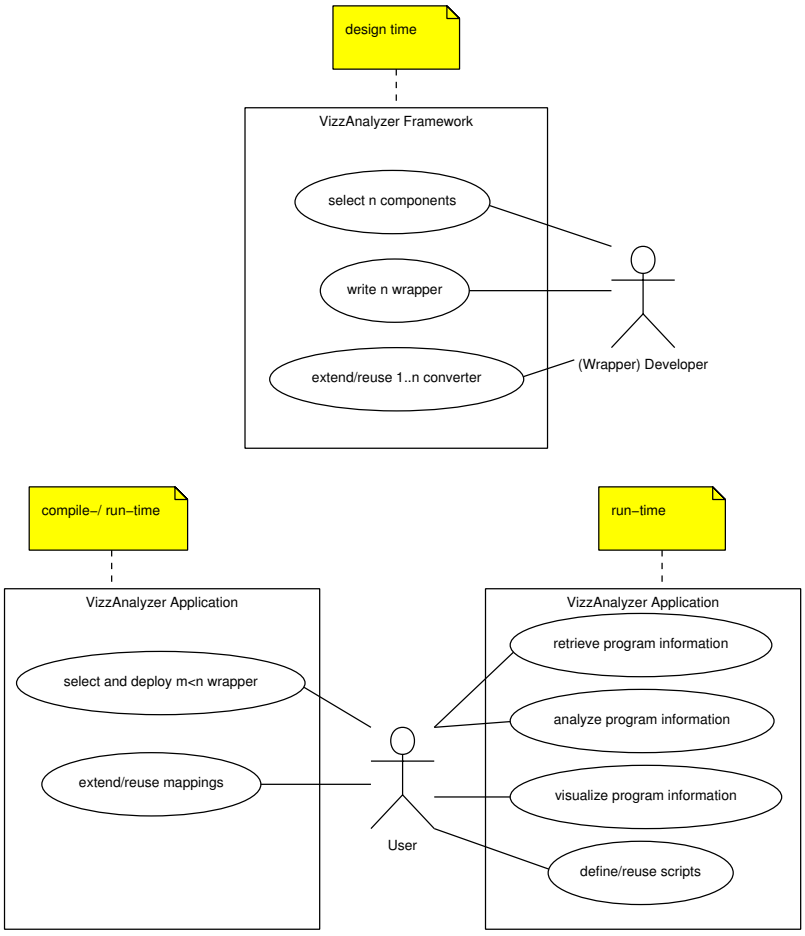

Figure 2.2: Use Case Diagram

2.3 Design

The design of the VIZZANALYZER FRAMEWORK is illustrated in Figure 2.3. It shows our own frozen spots, the Tiny-XML Editor (our stand-alone tool to manage XML configuration), GUI, GRAIL and VAUtils (containing code for the model mapping). The frozen spots are all part of the framework-core (as is the GUI), which is illustrated in its unfold details. The BeanShell component for scripting, also a frozen spot, is not our own product

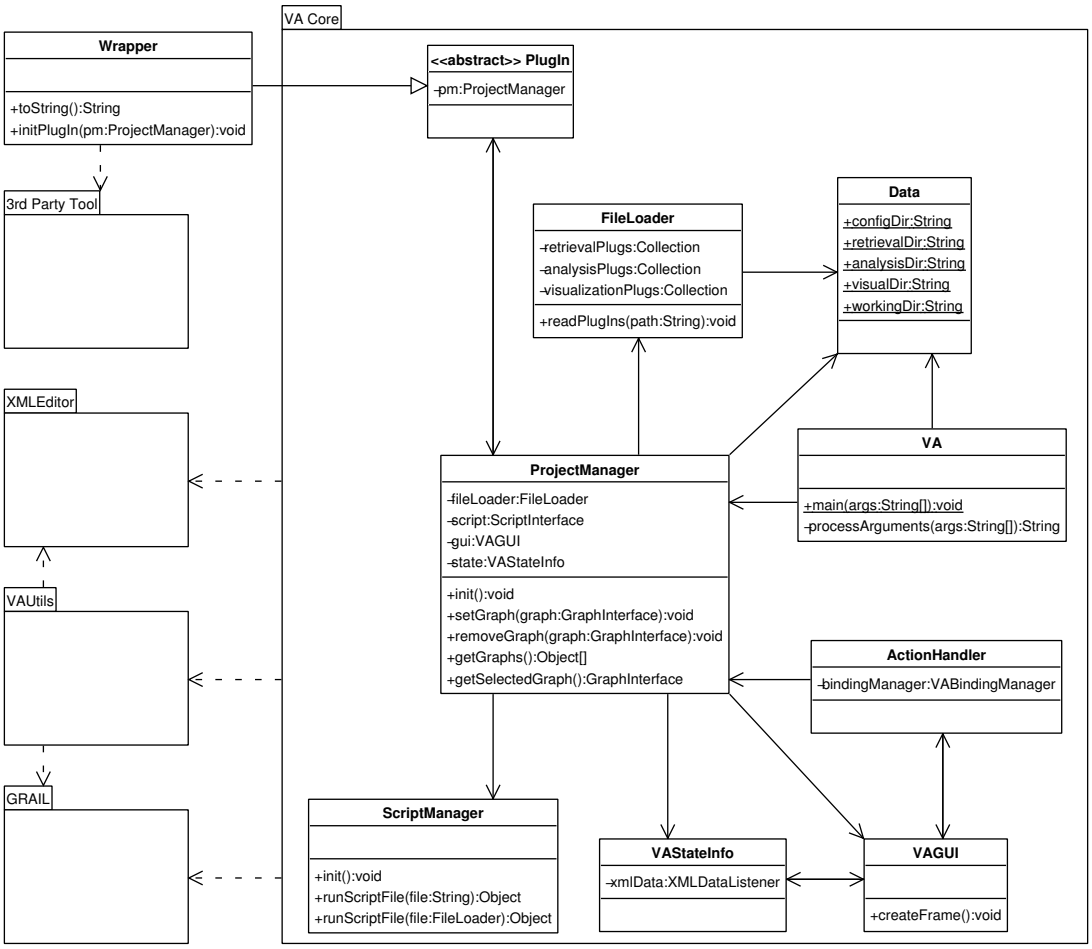

Figure 2.3: VIZZANALYZER FRAMEWORK Design

and therefore not part of the image. The framework-core contains a set of classes, responsible for the composed tool's control-flow. These classes are all implemented in Java. Their interactions are explained next.

At run-time, the *VA* class contains the `static main()` method, used to startup the `VizzAnalyzer` application. It constructs and initializes a *ProjectManager* object, whose `init()` method performs a variety of tasks. First it instantiates a *ScriptManager* object, which is used (with the help of the GUI) to record and replay user interactions. Second, a *VAStateInfo* object is created, containing the reference to the XML-Editor, allowing the online-configuration of XML mapping files. Third, a *VAGUI* object is created, forming

the application's GUI. Next, a *FileLoader* object is constructed and the `readPlugIns()` method called. The FileLoader then investigates the predefined directories, defined as static variables in the *Data* class, for retrieval, analysis and visualization `wrapper` classes. If found, those wrappers are constructed and instantiated with a call to the `initPlugIn()` method. A reference to each wrapper object, is then appended to the GUI, enabling the user to interact with components plugged to the system.

The application is up and running, expecting the user to interact. On GUI interaction, the *ActionHandler* object, previously instantiated by the GUI, is handling the user request, and possible delegating program information and/or reverse engineering tasks to the components at hand. Information exchange between components (wrappers) and the framework (ProjectManager) is handled via the `setGraph()` and `getGraph()` methods. All program information, as discussed before, is internally handled as one or more graphs, which are instances of GRAIL.

In the following, we demonstrate an example wrapper for the yEd visualization tool. The wrapper utilizes our GML file format converter. Semantic conversion or type checking are not used. Semantical mismatch is no issue here and type checking is superfluous, since yEd is capable of handling all kinds of graphs we can currently produce with the VIZZANALYZER FRAMEWORK.

```
1 public class yED extends PlugIn implements PlugInInterface {
2  private static final String YED_PROGRAM_DIR = Data.myViewPlugInDir+"/yEd/bin/";
3  private static final String YED_PROGRAM_FILE = "yed.jar";
4
5  public void initPlugIn(ProjectManager pmP) {
6   this.pm = pmP;
7
8   Action yEdAction = new AbstractAction("start", null) {
9     public void actionPerformed(ActionEvent ae) {
10        run();
11   }};
12   actionList.add(yEdAction);
13  }
14
15  public void run() {
16   GraphInterface g = pm.getSelectedGraph();
17   if (g == null) return;
18   GML gml = new GML();
19   gml.toGML(new File("_test.gml"), g, true);
20     String argv[] = new String[1];
21     argv[0] = "_test.gml";
22     yed.Launcher.main(argv);
23   }
24 }
25
26  public String toString() {
27    return "yED";
28  }
29 }
```

Within the `initPlugIn()` method (line 5-13), a new action, *yEdAction*, for this wrapper is created and appended to the `VizzAnalyzer`'s GUI. On selection of the yEdAction in the GUI, the `run()` method (line 15-24) is executed (just as specified in line 10). In line 16 the currently selected graph in the GUI is chosen and passed on for conversion (GRAIL-to-GML) in line 19. Finally, in line 22 the yEd application is launched with the converted graph as a parameter.

Chapter 3

The ANALYZER FRAMEWORK

The ANALYZER FRAMEWORK is our own component for program analysis. Providing a wrapper it allows for easy integration with the `VizzAnalyzer`. The ANALYZER FRAMEWORK shares some frozen-spots with the VIZZANALYZER FRAMEWORK like GRAIL and the Tiny-XML Editor. Other than that, the ANALYZER FRAMEWORK serves an entirely different purpose and hence embodies a different framework design.

Just like the VIZZANALYZER FRAMEWORK, it is based on a plug-in architecture, where analyses may be deployed at composition-time. At design-time, the framework offers three extensible hot-spots, represented by abstract classes and inherited by user specific *analyses plug-ins* (similar to wrappers). The following three types of analyses plug-ins exist:

- *Metrics*. To measure quality of a software system, the ANALYZER FRAMEWORK contains metrics which are well established in the literature, such as Weighted Method Count, Change Dependency Between Classes, Depth of Inheritance Tree, etc., cf., [PLLL05].
- *Focusing*. Information focusing means the abstraction, compression and fusion of program information [Pan03]. The ANALYZER FRAMEWORK currently contains extensions to filter and aggregate nodes and edges, union and intersection operations to combine graphs, parse filters, etc.

- *HL-Analyses*. High-level(HL-) analyses represent additional examinations on software systems in order to reveal supplementary informations about them. The ANALYZER FRAMEWORK currently provides hl-analysis plug-ins for dead code analysis (static analysis that detects which code is never used), package graph constructor (constructs a package graph out of a call graph), parse analyzer (uses regular expressions in order to perform simple queries at run-time), etc.

3.1 Usage

The framework's *usage* is divided into three stages:

1. *Whitebox extension*. Initially, only the whitebox framework is available. A framework developer may extend the framework at the hot-spot. Three hot-spots exist, i.e. the abstract classes *FocusEngine*, *HLAnalysis* and *MetricsAnalysis*. Each may be inherited, resulting in a new analysis plug-in. The invariant methods are provided, and the variant methods `toString()` and `startAnalysis()` overwritten. The name (description) of the plug-in is provided by the `toString()` method and the plug-in is executed with `startAnalysis()` method. Plug-ins may be collected for blackbox composition.

2. *Blackbox composition*. An application is instantiated, i.e. a user selects a set of plug-ins and deploys them in the plug-in points. The new analysis tool is ready to be used.

3. *Application run*. The ANALYZER FRAMEWORK has no own GUI, it must be used at run-time as a blackbox component, e.g. being plugged to the `VizzAnalyzer`.

3.2 Design

Figure 3.1 depicts the design of the ANALYZER FRAMEWORK. Most of the interfaces are neglected in that image to reduce complexity. The figure shows the following: three wrappers, part of the `VizzAnalyzer`, the main entry class *AnalyzerMain*, the class *FileLoaderAnalysis* for dynamic plug-in loading, and a hierarchy of abstract classes for analysis plug-in extension. Example plug-ins for mergers, aggregators, filters, hl-analyses and metrics are illustrated in the lower part of the image.

For the usage together with the VIZZANALYZER FRAMEWORK, we have developed three wrappers, as indicated in Figure 3.1. One wrapper for focusing, one for hl-analyses and one for metrics. Wrappers are detected and initialized by the `VizzAnalyzer` at application-run. On detection of the first `Analyzer` plug-in and initialization via its `initPlugIn()` method, the wrapper delegates the initialization to the static `init()` method of the AnalyzerMain class. On initialization, a new AnalyzerMain object is created. In order to guarantee that no other wrapper instantiates the same class once more, a boolean variable called `initialized` is used as a guard. In addition, the reference to the constructed object is kept via the public static variable `analyzer`. Finally, at first initialization, the `readAnalyses()` method within the FileLoaderAnalysis class is executed. Corresponding directories for focusing, hl-analyses and metrics are scanned to retrieve available analyses plugins. On success, a reference to each plug-in is passed back to the wrappers, and hence to the `VizzAnalyzer`'s GUI.

Now, the `VizzAnalyzer` is waiting for user interaction. On user selection of an analysis in the GUI, a predefined action in the wrapper is executed, delegating the request to the corresponding `Analyzer` plug-in. Depending on the type of wrapper, different kind of information is delegated, e.g. plug-ins inheriting from the class *Merger* must implement the method `accept()` containing *two graphs* as parameters. All other plug-ins implement the `accept()` method with only *one* input graph as a parameter. The execution of each plug-in is performed through the `startAnalysis()` method. The final result is available via the `getResultGraph()` method, returning the `newGraph` of the abstract class *HLAnalysisAbstract*.

3.3 Example Analysis Plug-In

In the following we show some example source code, of a simple analysis plug-in. The purpose of the class *RenameGraph* is to change the Label of the current GRAIL graph.

```
1 public class RenameGraph extends MetricsAnalysis {
2
3  public void startAnalysis() {
4   String name = "Default";
5   name = JOptionPane.showInputDialog(null,
6    "Please enter the new Graph Name");
7   newGraph.setProperty(GraphProperties.LABEL, name);
8  }
9
10  public String toString() {
11   return "Rename Graph [ANY GRAPH]";
12  }
13 }
```

The `startAnalysis` method contains the entire code necessary for the analysis. Line 7 contains the actual renaming of the graph. `GraphProperties.Label` is part of our GRAIL package. It defines the property label of a graph. Label and type are the only predefined properties for the graph, nodes and edges. All other properties are created at run-time via the dynamic typing system. The `toString()` method in line 10-12 is used to display the name of the plug-in in the `VizzAnalyzer` GUI menu entry.

3.4 Analyses

3.4.1 Metrics

The below described metrics are in version 1.0b3 implemented within the ANALYZER FRAMEWORK. Please refer to FAMOOS [Bea99] for details about their definition and interpretation. In the following we describe our implementation of each metric. All metrics are implemented in the package `hlanalysis.metrics`. With type constants is are the defined values in the class `data.TypeConstants` meant.

- *Change Dependency Between Classes (CDBC)*. This metric is implemented in the class `ChangeDependencyBetweenClasses`. It is a variation of the CDBC metric described in FAMOOS. In our implementation the CDBC of a class CC is not calculated as the number of methods possibly effected by a change in a server class SC, but as the sum of CDBC values for all server classes a client class depends on. The following type properties (defined in class `TypeProperties`) are used for its calculation:

  - CLASS_NODE. Represents a class. The metric calculates for every class CC the number of field and method accesses/calls to other classes SC or interfaces being referenced from the class CC.
  - INTERFACE_NODE. Represents a interface which is referenced by the client class CC.
  - CONSTRUCTOR_NODE. A constructor call from CC to a server class SC.
  - FIELD_NODE. A field in a server class SC accessed by CC.
  - EXTENDS_EDGE. Used for determining if the client class CC is extending a server class SC.
  - IMPLEMENTS_EDGE. Used for determining if the client class CC is implementing a server interface SC.

This metric can currently be applied to a Type Reference Graph. It is stored as graph property `CDBC`.

- *Data Abstraction Coupling (DAC)*. This metric denotes the number of Abstract Data Types (ADTs) a class refers to via an attribute. It counts the fields (`FIELD_NODE`) defined in a class or interface (`CLASS_NODE` or `INTERFACE_NODE`). It is implemented in the class `DataAbstractionCoupling` and accepts graphs of type Attributed Tree containing nodes of the described type. It is stored as graph property `DAC`.
- *Package Data Abstract Coupling (PDAC)* This metric is similar to DAC. It is an extension of the standard DAC described above for classes to package level. It counts for each package the number of outgoing type (ADT) references to classes in packages different to the current one. It is stored as graph property `PDAC`. It is implemented in the class `PackageDataAbstractionCoupling`.
- *Depth of Inheritance Tree (DIT)*. This metric denotes the depth of a class or interface in the inheritance tree. The algorithm takes the current class node as root of a tree, and uses depth first search to find the leafs of the tree, counting the number of nodes on the path (including the root and leaf node). The count of nodes from the longest path is the DIT of the class. It is implemented in `DepthOfInheritanceTree` and type constants are: `CLASS_NODE`, `INTERFACE_NODE`, `EXTENDS_EDGE`, and `IMPLEMENTS_EDGE`. It is stored as graph property `DIT`.
- *Edge Size*. This metrics can be applied to any graph on which the EdgeAggregator has been applied. It relies on the graph property MULTIPLICITY. It calculates the normalized value 0..1 for the number of edges between two nodes of the same type. 1 represents the maximum number of edges counted in the whole graph. It is stored as graph property `edgeSize_rel`. The metric is implemented in the class `EdgeSize`.
- *Number of Children (NOC)*. This metric calculates the NOC for a class C which represents the number of immediate subclasses subordinated to C in the class hierarchy. It counts in a class hierarchy graph the number of direct successors of type `CLASS_NODE` and `INTERFACE_NODE` and stores it in as graph property `NOC`.
- *Size*. Calculates for a graph the following statistics:
  - Lines of Code (LOC)
  - AST size
  - Number of interfaces and classes
- *Tight Class Cohesion (TCC)*. This metric denotes the cohesion within the methods and attributes of a class by dividing the number of actual method pairs in a class through the number of possible connections. It is implemented in the class

TightClassCohesion and stores the metric value for each class in the graph property TCC. It uses the CLASS_NODE and METHOD_NODE type constants.

- *Tight Package Cohesion (TPC).* Implementation of the Tight Package Cohesion metric (TPC) which is similar to the TCC but on package level. The algorithm calculates the proportion of the actual existing interaction between the classes in a package to the number of possible interactions. This metric is implemented in the class TightPackageCohesion and stores the metric value for each class in the graph property TPC. It uses the PACKAGE_NODE and CALL_EDGE type constants.
- *Lack of Documentation (LOD).* This metric calculates the the lack of documentation in a class or interface by dividing the the number of documented methods by the number of total methods in a class. It uses CLASS_NODE and INTERFACE_NODE nodes and the Recoder function for determining the count of documented and undocumented elements. It is implemented in the class Undocumented
- *Weighted Method Count (WMC).* Implementation of the Weighted Method Count (WMC) metric. Summing up the complexity for each method METHOD_NODE of a class CLASS_NODE. The complexity is measured using the McCabe's Cyclomatic Complexity. Counted are:

  – do statements DO_NODE
  – for statements FOR_NODE
  – while statements WHILE_NODE
  – if statements IF_NODE

  The result is stored in the graph property WMC. Additionally the CONTAINS_EDGE is used for the relation between the single elements.

3.4.2 Focusing algorithms

The following focusing algorithms are implemented within the ANALYZER FRAMEWORK. They are all implemented in the package hlanalysis.focusEngines. Focusing algorithms have in general the task to modify the structure and content of graphs by filtering out information or merging existing graphs.

- *Default Graph Merging Algorithm.* The *Default Merger* is a focusing algorithm. It is implemented in the class DefaultMerger. It takes a two graphs and creates a new one by cloning the first graph and adding node and edges of the second graph to the clone. Graph properties are not merged. The new graph label is a concatenation of the two graphs' labels.

- *Edge Aggregator*. The Edge aggregator analysis aggregates all edges of the same type of edge (`GraphProperties.TYPE`) between two nodes to one edge resulting in a multiplicity property `GraphProperties.MULTIPLICITY`. It is implemented in the class `EdgeAggregator`.
- *Edge Filter*. This focusing algorithm filters (removes) certain edge types from the graph. It presents a list of the edge types existing in the graph and removes the ones selected. It is implemented in the class `EdgeFilter`.
- *Node Filter*. Similar to the edge filter are the nodes removed according to the selection. Edges connected to the removed nodes are removed as well. It is implemented in the class `NodeFilter`.
- *Inner Class Filter*. This focusing algorithm filters (removes) inner classes from from a graph containing classes. It is implemented in the class `InnerClassFilter`. It uses the Recoder types for identifying them.
- *Intersection*. This focusing algorithm accepts either two directed or two undirected graphs. Mixing is not allowed. It offers the user to build the set theoretic intersection between the two graphs should be build according to the nodes or the edges. The result is returned as new graph. It is implemented in the class `Intersection`.
- *Symmetric difference*. This focusing algorithm accepts either two directed or two undirected graphs. Mixing is not allowed. It offers the user to build the set theoretic symmetric difference between the two graphs. The result is returned as new graph. It is implemented in the class `SymmetricDifference`.
- *Union* is a focusing analysis. It takes a two graphs and creates a new one. The new graph contains all nodes and edges existing in the first and second graph. Nodes and edges existing in both graphs are represented by just one copy which is colored in a distinguished way. The nodes and edges are compared using their label. The `unionNode` and `unionEdge` properties are used for marking the nodes and edges. They can have the following values:

  - first, if the node or edge belongs exclusively to the first graph.
  - second, if the node or edge belongs exclusively to the second graph.
  - both, if the node or edge belongs both, the the first and second graph.

  It is implemented in the class `Union`.
- *UnionProperties*. The same as Union, but here the properties are united as well, if a node or edge exists in both graphs. It's implemented in the class `UnionProperties`.
- *PackageGraphAggregator*. This focusing analysis iterates over a Graph and aggregates the nodes according to their package structure. It is implemented in the class `PackageGraphAggregator`. It accepts directed or undirected graphs.

- *ParseAggregator*. The class `ParseAggregator` implements a focusing analysis hiding the parser filter handling. The filter is used to mask out classes and packages which should not be included in the analysis. The filter accepts a text string, where the single classes/packages which are to be filtered are separated with a $< code >$ ";" $< /code >$. Until now the filter string is parsed into a simple array.
- *ParseFilter*. The class `ParseFilter` implements a focusing analysis hiding the parser filter handling. The filter is used to mask out classes and packages which should not be included in the analysis. The filter accepts a text string, where the first regular expression defines what will be replaced and the second, what will be used for replacing. Further is the source and target property specified, on which the regular expression will be applied. filtered are separated with a $< code >$ ";" $< /code >$.
- *VersionMerger*. This focusing algorithm takes two graphs, each representing a different version of the same software system, and merges them into one single graph. It is implemented in the class `VersionMerger`.

3.4.3 High-level analysis

The following high-level analysis are currently implemented within the ANALYZER FRAMEWORK. They are all implemented in the package `hlanalysis.analyses`

- *Dead code analysis*. This high-level analysis marks all nodes in a graph as unreachable. Then it parses the graph from a given start node (root node) and marks all parts traversed as reachable. The un-traversed parts are dead code assuming the specified entry point is the only entry point in the system.
- *Excel*. Writes the given nodes, edges and their properties to a comma separated text file, which can be read in into MS Excel. It is implemented in the class `Excel`.
- *Initialize numeric properties to null*. This high-level analysis sets all numeric property values to 0. It traverses all nodes and edges in the given graph. It is implemented in the class `FillPropertyZero`.
- *GroupPackage*. Aggregates a graph according to its type "Package". If a graph contains a property Package, then all nodes sharing the same property are aggregated into the same node. A new graph is created.
- *Package Graph Constructor*. This high-level analysis requires a graph of type LLA-Tree. It creates a package graph from it. The graph is returned as new graph. It is implemented in the class `PackageGraphConstructor`.
- *ParseAnalyzer*. Allows the user to define own properties via regular expressions.
- *RelativeProperty*. Calculates the relative values for absolute properties.
- *RenameGraph*. Renames a given graph (sets a new Label).

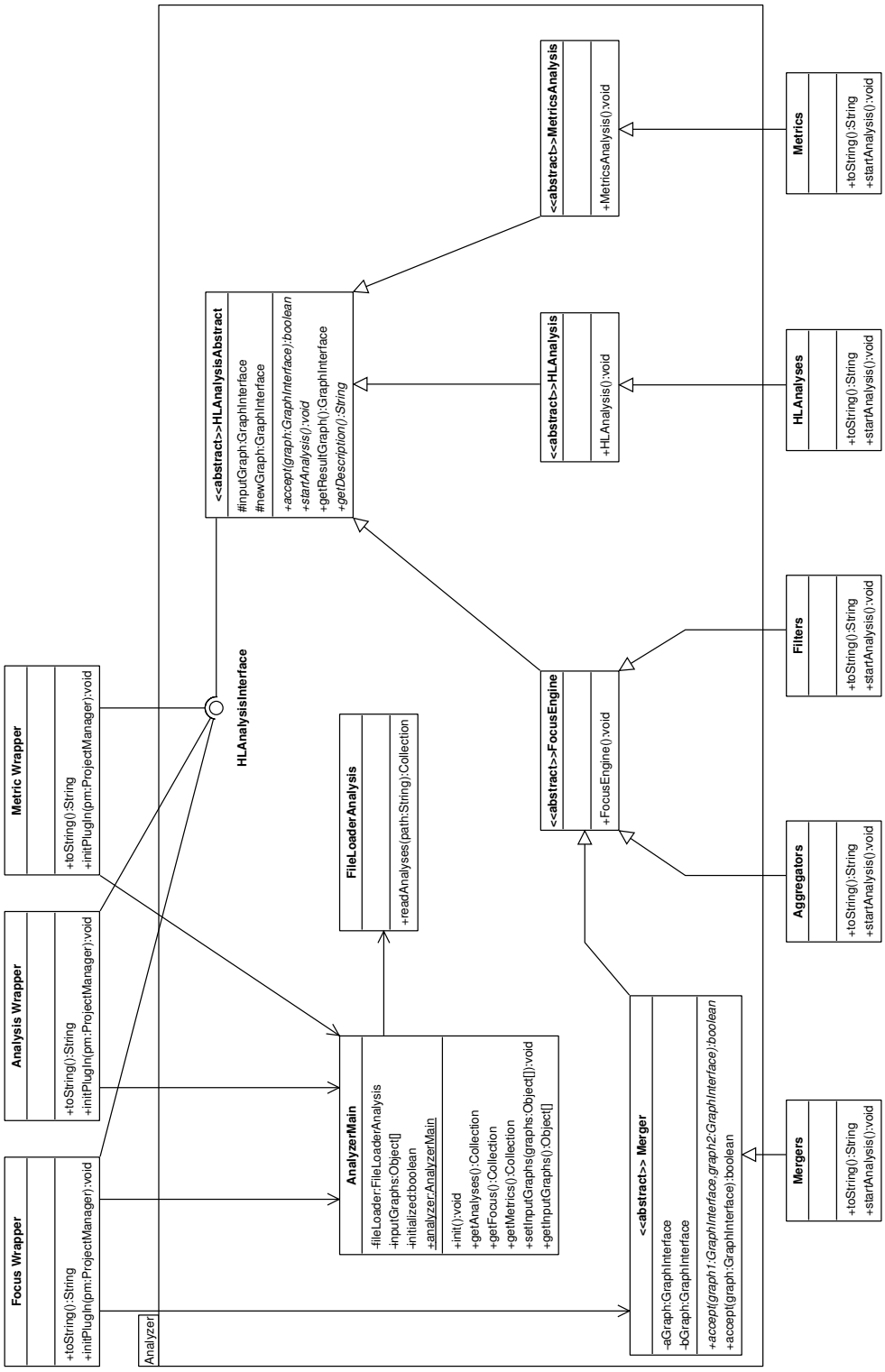

Figure 3.1: ANALYZER FRAMEWORK Design

Chapter 4

The VIZZ3D FRAMEWORK

The VIZZ3D FRAMEWORK is our own 3D visualization framework, allowing the illustration of program information in 3D. The VIZZ3D FRAMEWORK reuses some frozen-spots described earlier: GRAIL, the Tiny-XML Editor and Mapping Engine. Implemented in Java, `Vizz3D` can be launched as a stand-alone *Java3D* [jav05] or *OpenGL* [ope05] application. Two very similar OpenGL implementations exist, they vary on the type of OpenGL-to-Java wrapping, i.e. jogl [jog05] or gl4java [gl405].

The VIZZ3D FRAMEWORK reuses the `VizzAnalyzer`'s plug-in architecture and allows therefore the extension of hot-spots at design-time, and the deployment of these extensions at compilation-time. The extensions of the VIZZ3D FRAMEWORK are *mapping files*, *layout algorithms* and *metaphors*:

- *Mapping Files* define binding functions to *map* view graph property values to scene graph property values. The `Vizz3D` binding functions are the same as defined in the `VizzAnalyzer`. For online-configuration, we use the XML descriptors again (mapping files).
- *Layout* algorithms assign position properties to scene graph nodes. Besides this layout function, a layout class may restrict the set of metaphors it is applicable for. Layout algorithms operate on the internal, visual graph structure of the framework. We refer to this structure as a *scene graph*, instantiated through our own *visgraph* package, described further below.

- *Metaphors* are families of visual objects fitting together. E.g. in a UML Class diagram the according 2D UML representations for classes, interfaces, associations, etc. 3D boxes and spheres would not be allowed, since they are not part of the metaphor. Metaphors contain even files describing the environment of the visualization, i.e. the background, additional visual entities and other environmental factors like fog and light sources. The individual implementations are API dependent (*Java3D* or *OpenGL*). A metaphor class may restrict the set of layouts it allows to be applied.

4.1 Vizz3D

Vizz3D is a 3D information visualization system. It promotes system structure and quality information to a user in a comprehensible way and leverages the understanding of that system. It is, however, also usable in other contexts since it can display virtually any kind of information which can be expressed in values and relations between these values.

Vizz3Dwas originally developed for the VizzAnalyzerbut is now a stand-alone tool. It allows to show different graphs simultaneously and even their changes over time can be displayed. The screenshots below give some examples. The graphs can be defined by the user. Then different layouts, visual metaphors can be selected to display the information. Even own layouts and metaphors can be integrated within the VIZZ3D FRAMEWORK.

The first screenshot 4.1 shows a software visualization of the Vizz3D program code using a city metaphor for capturing the level (quantity) of code documentation. It contains class (boxes) and interface (cylinders) nodes. The relations between these are here not important and therefore not displayed. The height of the houses relates to the amount of code within the class/interface and the applied flame texture indicates a low or high lack of documentation.

The second screenshot 4.2 shows interactions of classes (solid spheres) belonging to packages (transparent spheres enclose classes of the same package).

The third screenshot 4.3 illustrates the evolution of a system with its classes (boxes) and interfaces (spheres) over seven versions. Each version has its own level (y-coordinate) and colors. Same classes and interfaces in the different versions have same x- and z-coordinate. The size of items corresponds to the code size.

The forth screenshot 4.4 illustrates some quality aspects classes of a system. It shows four standard metrics that are maintainability indicators of a system: Lack of Documentation (LOD), Weighted Method Count (WMC), Change Dependency Between Classes (CDBC), and Lines of Code (LOC).

WMC is mapped to the x-coordinate of the boxes, LOC to the y-, CDBC to the z-

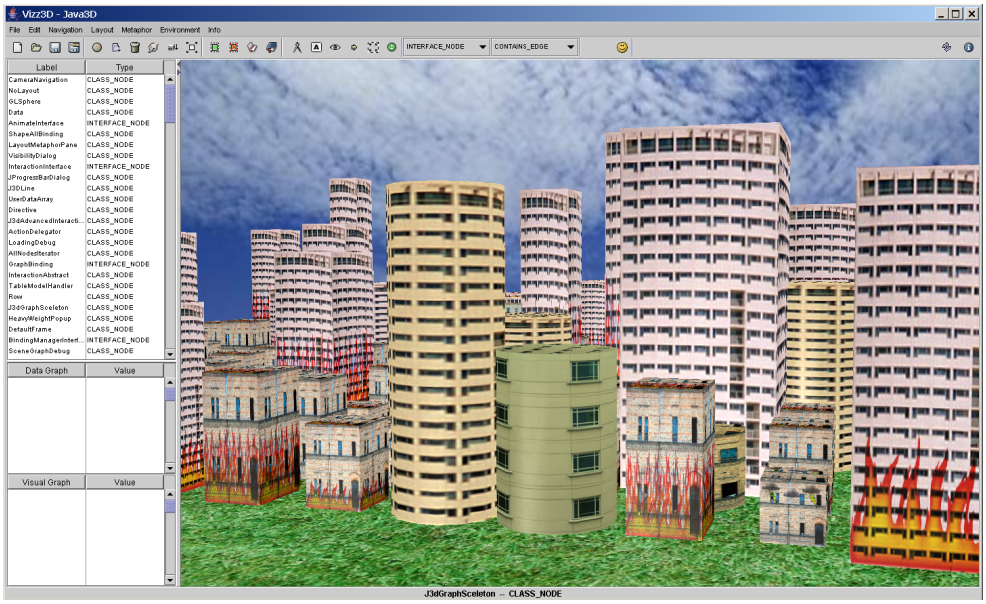

Figure 4.1: Class Hierarchy visualization with Vizz3D

coordinate. LOD is mapped to a color enoding small valuers to green, medium to yellow, and large to red.

The screenshot indicates outliers according to the quality model used, which indicate maintainability problems in system. To the upper right, classes have many lines of code and a high complexity. To the front, classes have a high CDBC, which means if one of those classes is going to be changed, many other classes will need to be changed as well. The ”healthy” classes are expected to be green and positioned close to the origin of the coordinate system.

The visualization of a system’s quality can also be connected to the evolution of that system. The fifth screenshot 4.5 is a snapshot of an animation showing in a series of visualizations the changes of LOC and of WMC between consecutive versions. The height of boxes shows the change of LOC. A boxes in the positive/negative y-direction indicates an increase/decrease of LOC. The color changes between red and green, indicating that a class became more complex (red) or less complex (green). The default is gray color indicating no changes. Same classes remain at the same position throughout the versions, so that their change in size and complexity can easily be perceived.

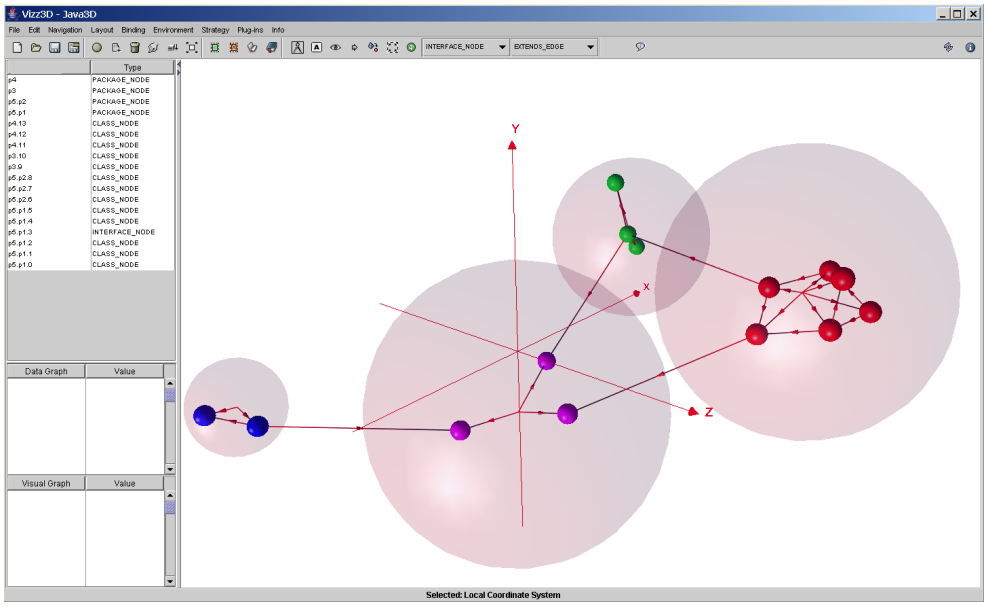

Figure 4.2: Package Graph with `Vizz3D`

4.2 Usage

The framework's *usage* is divided into three stages:

1. *Whitebox extension.* A framework developer may extend the framework at the only hot-spot, the abstract class *LayoutEngine*. Inherited classes are referred to as *layouts*. The invariant methods are provided, and the variant methods `init()`, `retrieve()` and `doLayout()` overwritten. The initialization of a specific layout and the generic layout GUI are part of the `init()` method, the update of parameters between layout and layout GUI is implemented in the `retrieve()` method and the `doLayout()` method encodes the actual layout algorithm. Layouts may be collected for blackbox composition.

2. *Blackbox composition.* An application is instantiated, i.e. a user selects a set of layouts and deploys them in the plug-in points. The visualization tool is ready to be used.

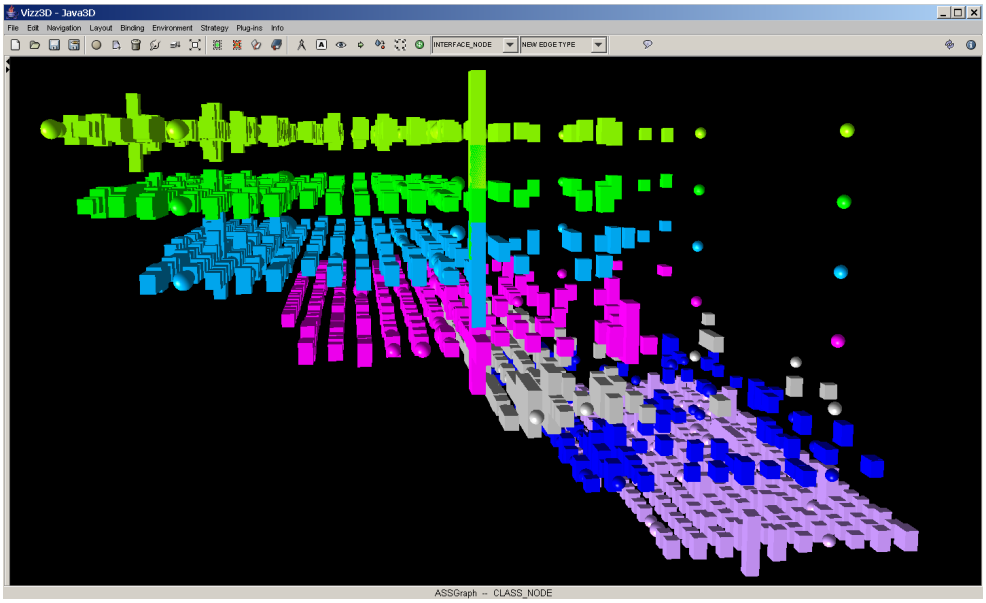

Figure 4.3: Evolution of a System with `Vizz3D`

3. *Application run*. At run-time, the user may execute and even deploy new layout algorithms, while interactively illustrating program information in 3D. In addition, mappings between view graph and scene graph [LP05] *may* be applied, in order to change the properties of visualizations, interactively, flexibly and at run-time. Here, mapping specifications (XML-files) contain also information about the shapes of visual properties. The individual implementations are API dependent (*Java3D* or *OpenGL*). Families of visual objects fitting together are referred to as *Metaphors*.

Metaphors and possible interactions, such as rotation, selection, aggregation, translation, zooming, etc. are further documented in [Mar05], [Ahl05] and [HQ05].

4.3 Design

Originally, the VIZZ3D FRAMEWORK was developed with Java and Java3D. Over time, performance issues arose and the VIZZ3D FRAMEWORK was extend with gl4Java, a simple wrapper of OpenGL for Java. The while GL4Java was discontinued, a new OpenGL

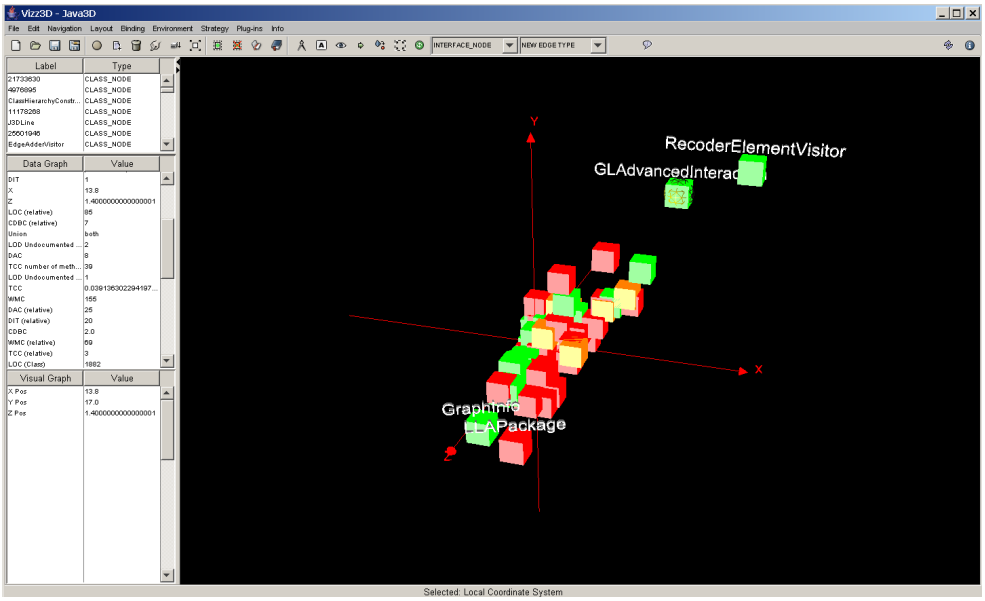

Figure 4.4: Software Quality with `Vizz3D`

binding for Java arose in the OpenGL community: JOGL. Java OpenGL bindings is supported by the Java Gaming Group at Sun. During extension, the antiquated technologies of the VIZZ3D FRAMEWORK remained. The architecture of the current version is illustrated in Figure 4.6. It shows that the VIZZ3D FRAMEWORK consists of three sub-packages, *VizzJ3D*, the Java3D variant, *VizzGL*, the gl4Java variant, and *VizzJOGL*, the jogl variant. The executable start-up classes within the sub-packages inherit from the abstract class *PlugIn*.

In the following, to simplify the comprehension of the VIZZ3D FRAMEWORK's design, the focus is merely towards its jogl variant, cf. Figure 4.7. The executable start-up class is *VizzJOGL*, containing the static `main()` method, and inheriting invariant functionality from the abstract class *PlugIn*. Interfaces are, to keep the design and description simple, removed from that figure. VizzJOGL coordinates most of the work, such as the dynamic loading of layouts (via the *FileLoader* class), setup of the GUI (via the *MainFrame* class), creation of a canvas (via the *JOGLGraphicsHandler* class), and creation, deletion and layout of a scene graph (via the *JOGLMLSceneGraph* class). MainFrame, on the other hand, keeps a reference to the abstract class *LayoutEngine*, allowing for user extensible layout algorithms. User interaction is handled by the class *JOGLAdvancedInteraction*.

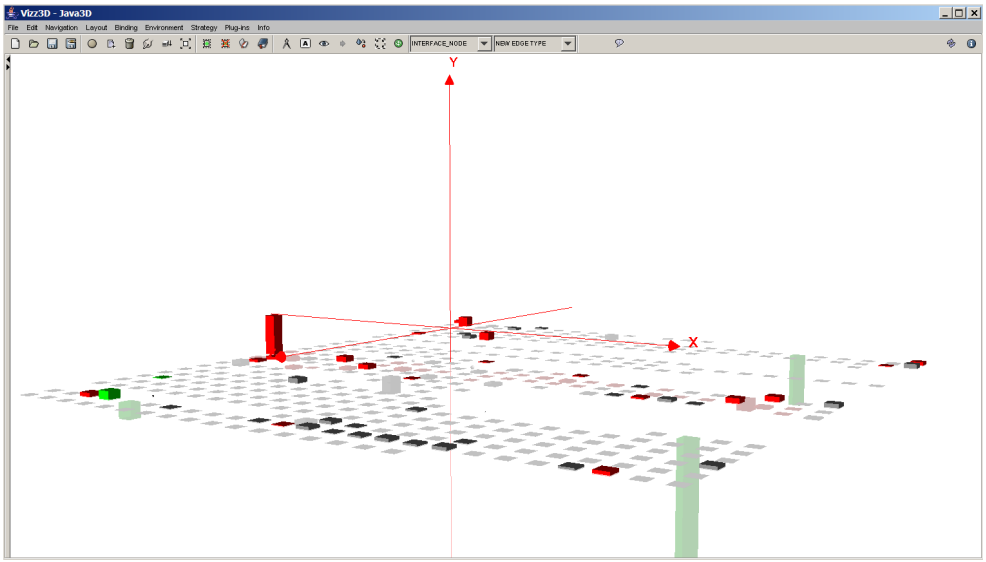

Figure 4.5: Evolution of Software Quality with `Vizz3D`

`Vizz3D` may, at run-time, be used either a stand-alone tool, or a visualization component for the VIZZANALYZER FRAMEWORK. `Vizz3D` accepts as input only graphs of type GRAIL (view graphs). However, to allow other formats, converters from the `VizzAnalyzer` are being reused. For visualization purposes, GRAIL graphs are internally converted to visualization specific graphs (scene graphs) of type VisGraph. The VisGraph is similar to GRAIL, with the difference that it constructs graphs containing visual objects, defined in a specific visual implementation language, such as Java3D or OpenGL.

On construction, the VizzJOGL object calls the inherited method `init()`, initializing a FileLoader and a MainFrame object. The FileLoader then scans predefined directories for layouts and mapping files via the methods `readLayouts()` and `readMappings()`. The MainFrame retrieves these collections, appends them to its visible GUI, and in addition, constructs a LayoutEngine and LayoutDialog object. The LayoutDialog is generic and reused with all layouts available for `Vizz3D`.

So far, a GUI with a menu is displayed, containing available layouts and mapping files. Next, a JOGLGraphicsHandler object is created, through the method `run()` of VizzJOGL, to create and apply a canvas to the GUI, cf. `createCanvas()`. In sequence, the method `createSceneGraph()` is executed, delegating the call to the method `updateMetaphor`

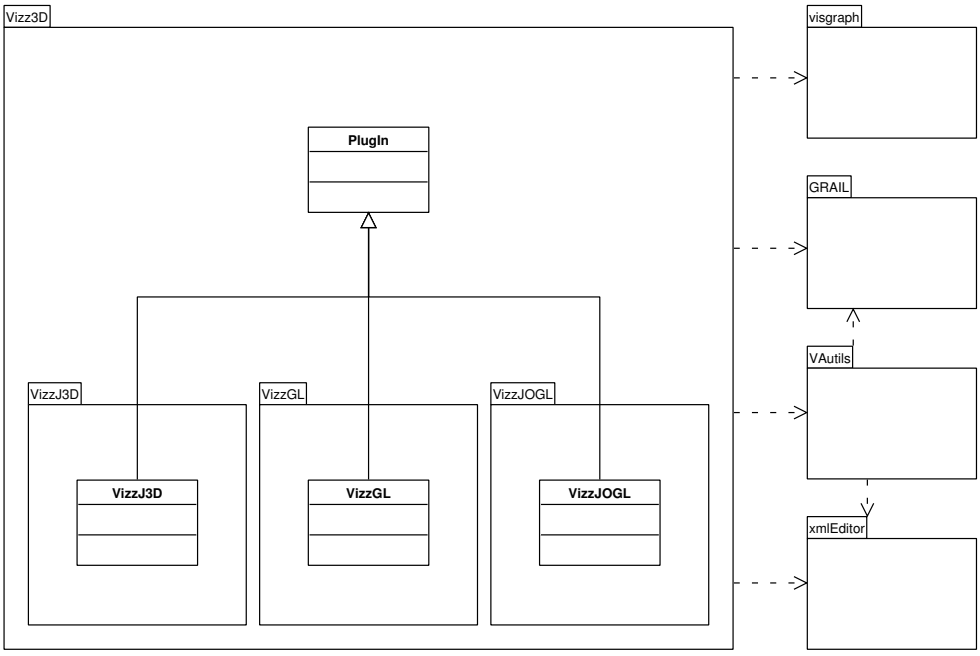

Figure 4.6: Architecture of the VIZZ3D FRAMEWORK

of the newly instantiated JOGLMLSceneGraph object. JOGLMLSceneGraph, in turn, constructs a JOGLVizzMetaphor object, which is responsible to encode the shape, color, size, etc. of nodes and edges to be drawn. The encoding can be flexible changed via mapping files. On initial startup, a default encoding is applied.

In the following, `updateMetaphor` invokes its private method `buildSceneGraph()`, which iterates through all available nodes and edges of the view graph to create the scene graph via calls to the `createSceneNode()`, `addSceneNode()`, `createSceneEdge()` and `addSceneEdge()` methods. Note that these methods are public, since they are accessed also through the user via the GUI.

Within the `createSceneNode()` and `createSceneEdge()` methods, the type of the node and edge, respectively, is determined. Based on the distinct type, a distinct geometry for the node or edge is chosen, created and returned with the help of the JOGLVizzMetaphor object. In Java3D and OpenGL a set of predefined geometries are available, such as spheres, cubes, cylinders, etc. However, `Vizz3D` allows also for the extension of user defined geometries, or the usage of a *mesh loader*. The mesh loader allows

to load predefined geometries from external tools (e.g. AutoCAD). Finally, JOGLMLSceneGraph constructs an instance of the JOGLAdvancedInteraction class, handling canvas specific user interaction, including the rotation and translation of nodes, the graph itself, mouse interactions, filtering or aggregation, selection, etc. For further information about `Vizz3D`'s interactive possibilities, we refer to [Mar05].

Finally, on selection of a layout in the GUI, the `init()` method of that layout is called first, providing the LayoutDialog with necessary layout parameters. Second the `doLayout()` method is executed, performing the layout of the presented graph. On change of the layout parameters via the LayoutDialog, first the `retrieve()` method is invoked to return the changed parameters to the layout, and second, the layout is invoked once again.

4.4 VisGraph

The *VisGraph* is our own API, used by the VIZZ3D FRAMEWORK to build scene graphs. This API initially supported functionality to build Java3D specific scene graphs, and was later extended to support also OpenGL specific scene graphs. Figure 4.8 gives an overview of the architecture.

From a developers perspective, it is much simpler to access only the unifying interfaces of the VisGraph, i.e. `VisualGraphInterface`, `VisualNodeInterface` and `VisualEdgeInterface`. A developer needs not to know whether the scene graph is build with Java3D's or OpenGL's technology. Common functionality is enclosed within abstract classes. Only during the creation of the graph, and its nodes and edges, the concrete classes, such as `GLGraph`, `GLNode` and `GLEdge` are being directly accessed.

To increase comprehensibility the detailed design of the visgraph package is depicted in Figure 4.9 for the OpenGL variant only. Scene graphs, also referred to as visual graphs, allow only few operations, such as the adding or removing of nodes and edges to or from the graph, respectively. In addition, node and edge iterators are provided. Nodes and edges are stored within Java HashMaps allowing to correlate the scene node (edge) with the view node (edge) and vice versa. This helps to separate data from its representation.

The design of the visgraph package is kept for performance reasons very simple. Its purpose is simply to provide a scene graph and allow some interaction. Therefore, the mentioned HashMaps are used, in order to access advanced graph specific operations within GRAIL and not the VisGraph. For instance, if the user selects a scene node and is interested in all its properties, such as height, width, color or even maybe some metric information, such as lines of code, then, the HashMap for all nodes is accessed, in search for

the corresponding GRAIL (view graph) node, to retrieve its properties. Vice versa, if scene graph nodes have been deleted, and the scene graph is being returned for instance to the `VizzAnalyzer`, an updated view graph must be returned. Therefore, there is another HashMap holding the inverse relation.

The classes *GLNode* and *GLEdge* provide functionality for visual nodes and edges. A visual node has a shape, expressed by its geometry. All different geometries use the common interface *GLMetaphorType* (not illustrated). On construction, visual nodes are assigned a shape, a height, depth, width and radius. A radius is necessary for shapes such as spheres or cones. The color, or texture and transparency of a node is encoded in its appearance, encoded in the class *GLAppearance* (not illustrated). GLAppearance and GL-MetaphorType are part of VisGraph's utility package. This package exceeds, however, the contents of this thesis and is hence not further discussed. All implementations of Figure 4.9 are based on OpenGL. Common functionality, such as assigning the position of a node, or collapsing and expanding nodes is part of the abstract class *VisualNodeAbstract*.

Visual edges, similar to the nodes, have a geometry and an appearance, i.e. a color, texture, pattern or transparency. In addition, edges might carry arrows as an indication of their direction (for directed edges). Otherwise the direction might be encoded in the color or texture.

4.5 Example Layout Plug-In

In the following we demonstrate some example source code of a user specific layout, to show its simple development. The purpose of the class *Randomizer* below, is to lay out all visual nodes randomly within a certain range.

```
1 public class Randomizer extends LayoutEngine implements LayoutInterface {
2   private int animations = 15;
3   private int random = 0;
4
5   public Randomizer() {super("Randomizer", "Randomizing the nodes.");}
6
7   public void retrieve() {
8     getArguments("Randomizer");
9     random = Integer.valueOf(getArgument("Randomizer","Factor")).intValue();
10    animations = Integer.valueOf(getArgument("Randomizer","Animation")).intValue();
11  }
12
13  public void init(VisualGraphInterface graph, AnimateInterface ani) {
14   anim = ani;
15   currentGraph = graph;
16   if (!graphsInit.contains(graph)) {
17     random = graph.nodesSize() * 2;
```

```
18     int max = random * 2;
19     setArgument("Randomizer", "Factor", random, 0, max, true);
20     setArgument("Randomizer", "Animation", animations, 0, 100, true);
21     setArgument("Randomizer", "Description", "Randomizer Algorithm. 25Jul2004.");
22     graphsInit.add(graph);
23    } else
24     retrieve();
25     showDialog("Randomizer", this);
26   }
27
28   public boolean doLayout() {
29    double X = 0.0; double Y = 0.0; double Z = 0.0;
30    VisualNodeIterator ni = graph.sceneGraphNodes();
31     while (ni.hasNext()) {
32      VisualNodeInterface node = ni.next();
33      if (!(node.isVisible()))
34        continue;
35      X = (Math.random() * random);
36      Y = (Math.random() * random);
37      Z = (Math.random() * random);
38      node.setFinalPosition(X - (random / 2), Y - (random / 2), Z - (random / 2));
39    }
40    anim.animate(animations);
41    return true;
42   }
43 }
```

Most of the methods and their functionality was explained in Section 4.3. Unexplained was the purpose of the *AnimatorInterface* used in this example. This interface is used, to support cognitive understanding, and hence to allow a slow animation of graph layouts, a user may utilize the animation functionality provided in order to smoothen the layout of nodes. In this way, nodes are slowly moved from their current position to their final position, defined by the `setFinalPosition()` method in line 38. In addition, one can check whether a node is visible or not, cf. line 33, to increase the performance of the layout algorithm. Finally, `setArgument()` and `getArgument()` methods are used to communicate layout parameters between the algorithm and the generic layout dialog. The dialog allows for quick modification of layout parameters during run-time, allowing users to interactively adjust their layouts to the best.

4.6 Layout Algorithms

The following layout algorithms are currently implemented within the VIZZ3D FRAMEWORK:

- *CCVisu.* 3D clustering layout algorithm by Dirk Beyer (Ecole Polytechnique Fdrale de Lausanne) and Andreas Noak (Cottbus University).

- *Circularizer.* A simple 2D layout algorithm aligning nodes in a circle.
- *EvolveInOne.* Visualize a serious of graphs in one image (CVS).
- *EvolveSlide.* Slide through a serious of graphs (CVS).
- *Gem3D.* 3D spring embedder based on the algorithm by Arne Frick (former Karlsruhe University).
- *Gem3DArray.* Algorithm based on Gem3D. However, it is used to visualize huge data amounts recovered from e.g. CVS.
- *Level.* 2D hierarchical layout algorithm.
- *Leveller.* 3D hierarchical layout algorithm.
- *Package Checker.* Aligns nodes according to their package structure (metric).
- *Package Sorting.* Sorts nodes according to some property (metrics).
- *Randomizer.* Random layout of nodes.

Vizz3D allows a user to easily write new layout algorithms. For this, only one new class must be implemented, reusing given functionality. Metaphors can also easily be changed within reusable XML-mapping files.

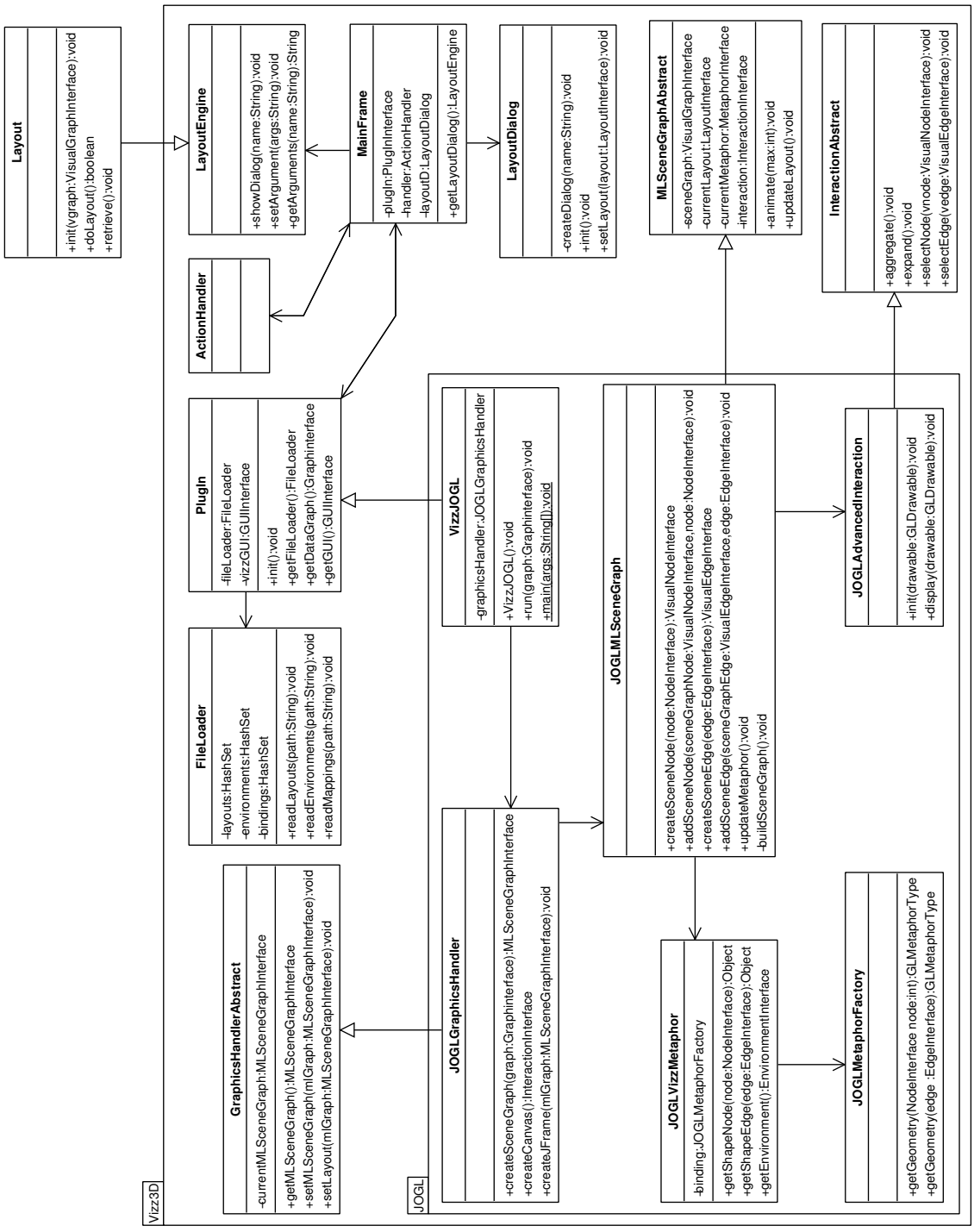

Figure 4.7: Design of the VIZZ3D FRAMEWORK

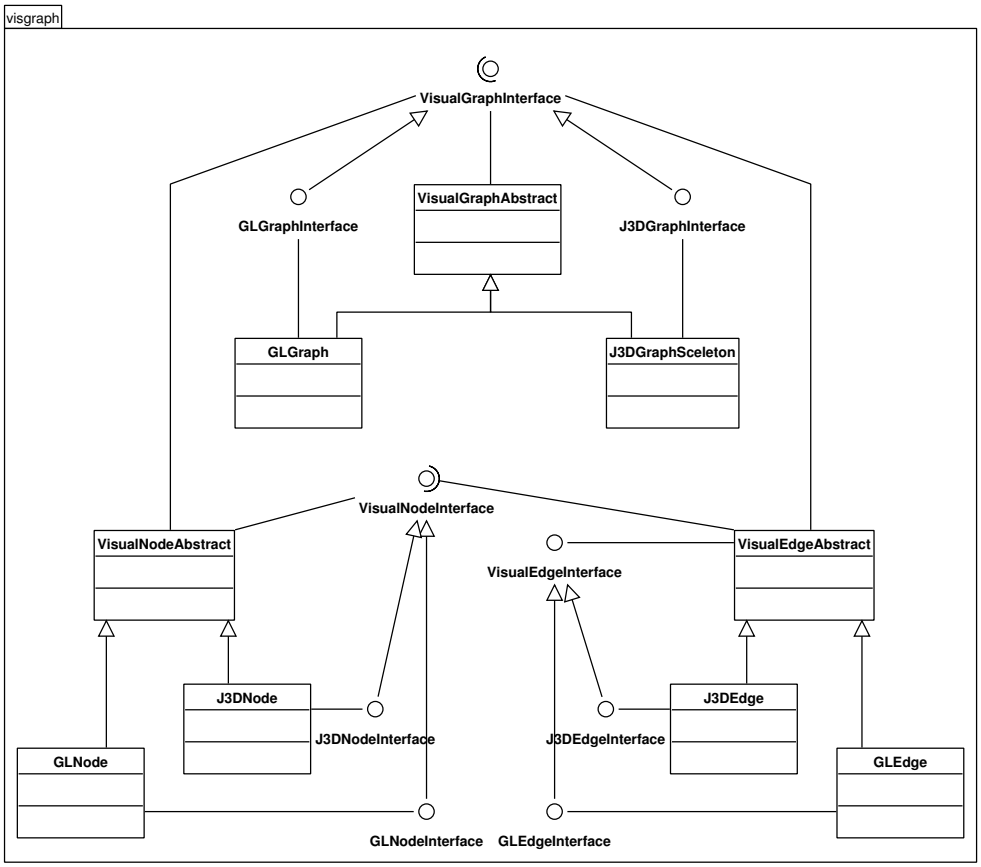

Figure 4.8: Architecture of VisGraph

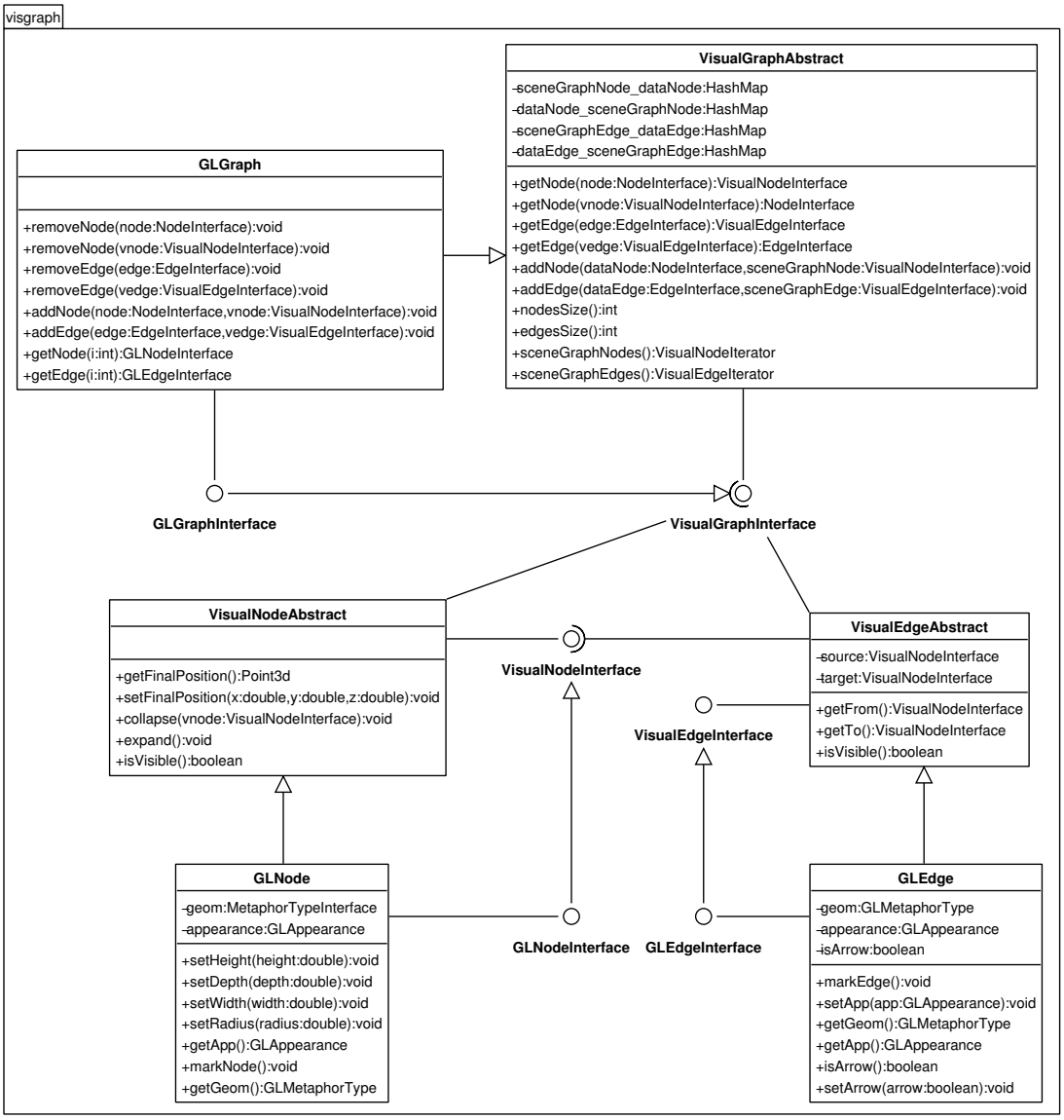

Figure 4.9: Design of VisGraph

Part III

Examples

Chapter 5

Example Usage of the `VizzAnalyzer`

Within this section, we utilize the `VizzAnalyzer` application to perform some example reverse engineering tasks. For this, the following components have been composed: RecoderComp, `Analyzer` , yEd and `Vizz3D`. RecoderComp is used to extract some Java source code, `Analyzer` to perform some package analysis (classes are grouped to packages according to their structural deployment) and finally, yEd and `Vizz3D` to visualize the results.

Figure 5.1 shows the initial GUI of the `VizzAnalyzer` on startup. In the GUI's menu, one can choose between the different front-ends, analyses and visualizations available at run-time.

Our reverse engineering task for this example is to show the hierarchy structure of a software called WilmaScope [wil04]. In addition, visual entities representing classes that are part of the same package, shall possess the same color. In Java, classes within the same directory are part of the same package. However, due to the natural manifoldness of directories and sub-directories, a significant number of colors becomes necessary. The problem inherent is that humans may not be able to distinguish the vast variety of colors. Therefore, our aim is to restrict the coloring of packages (including sub-packages) to a user selection.

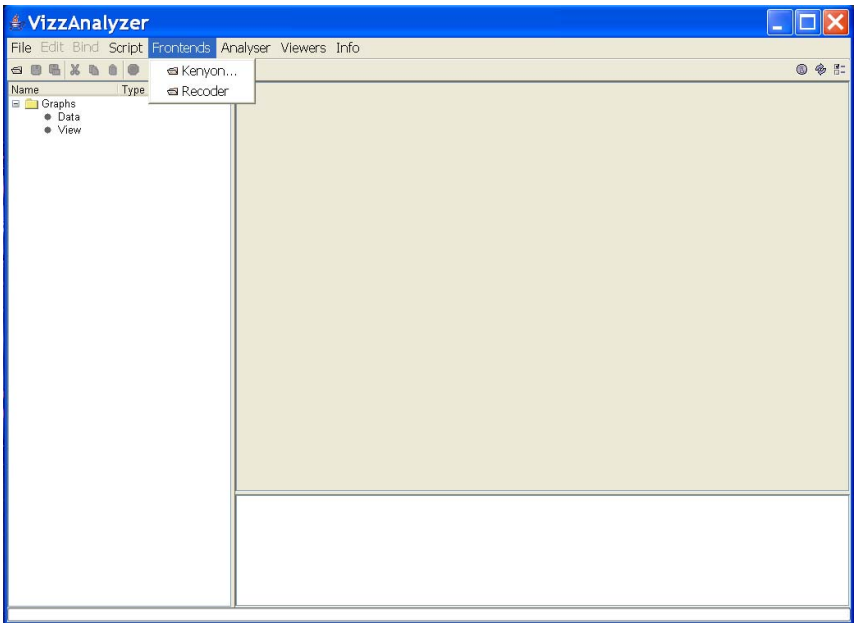

Figure 5.1: VizzAnalyzer GUI

5.1 The Front-End

As a first step, we select and execute a front-end. Figure 5.1 shows two available front-ends: Kenyon, to access CVS and SVN repositories, and RecoderComp, to parse Java source code. We select RecoderComp.

The dialog appearing after selection is illustrated in Figure 5.2. Note that the dialog is part of the wrapper. Within this dialog, the user is requested to select a project file (.vap file), specifying the project to be parsed. In our case, the project file stores information about the location of WilmaScope and, in addition, necessary libraries (.jar files) for a successful parsing. The corresponding project file is listed below:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE project SYSTEM "projects.dtd">
3 <project>
4 <projectname>Wilma</projectname>
5 <projectspecification>
6   <entrypoints />
7   <paths>
8     <sourcepath>$VA_HOME\examples\wilmaexample</sourcepath>
```

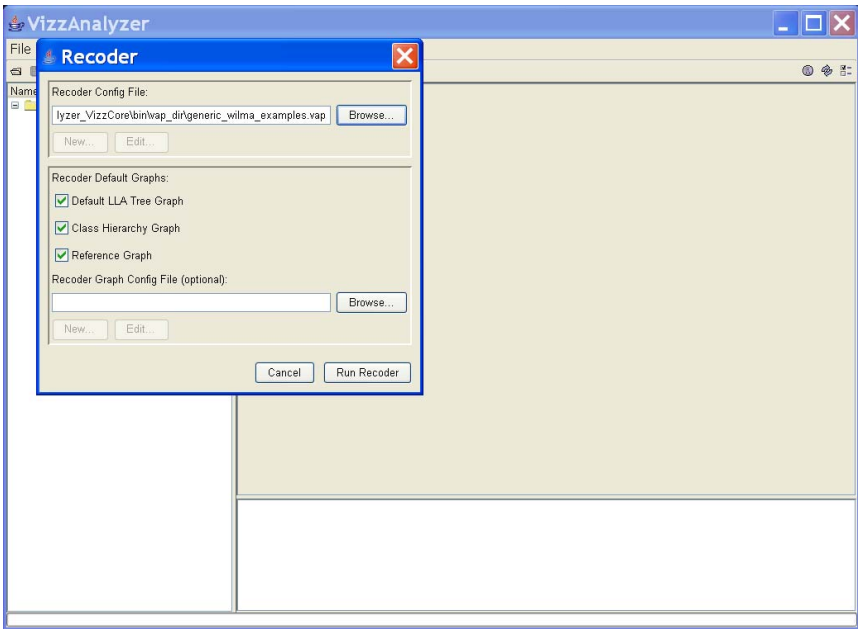

Figure 5.2: Recoder Dialog

```
9     <binarypath>$JAVA_HOME\lib\rt.jar</binarypath>
10    <binarypath>$VA_HOME\examples\wilmacomponents\xalan.jar</binarypath>
11    <binarypath>$VA_HOME\examples\wilmacomponents\crimson.jar</binarypath>
12    <binarypath>$VA_HOME\examples\wilmacomponents\jaxp.jar</binarypath>
13    <binarypath>$VA_HOME\examples\wilmacomponents\ant.jar</binarypath>
14    <binarypath>$VA_HOME\components\j3dcore.jar</binarypath>
15    <binarypath>$VA_HOME\components\j3dutils.jar</binarypath>
16    <binarypath>$VA_HOME\components\vecmath.jar</binarypath>
17   </paths>
18  </projectspecification>
19 </project>
```

In order to parse another project, the following changes are necessary: the adaption of the `sourcepath` in line 8 and the specification of dependent components (.jar files between line 9 and 16). Finally, `entry points` for the parser may be defined in line 6. This means that not all source code is parsed, but instead only source code reachable from the entry points. In this way, dead code is ignored.

In addition to the specification of a project file, the RecoderComp dialog allows to specify the types of resulting graphs. Various types of graphs are specified by default, cf. Figure 5.2, such as the class hierarchy graph. Other types of graphs, may be user defined

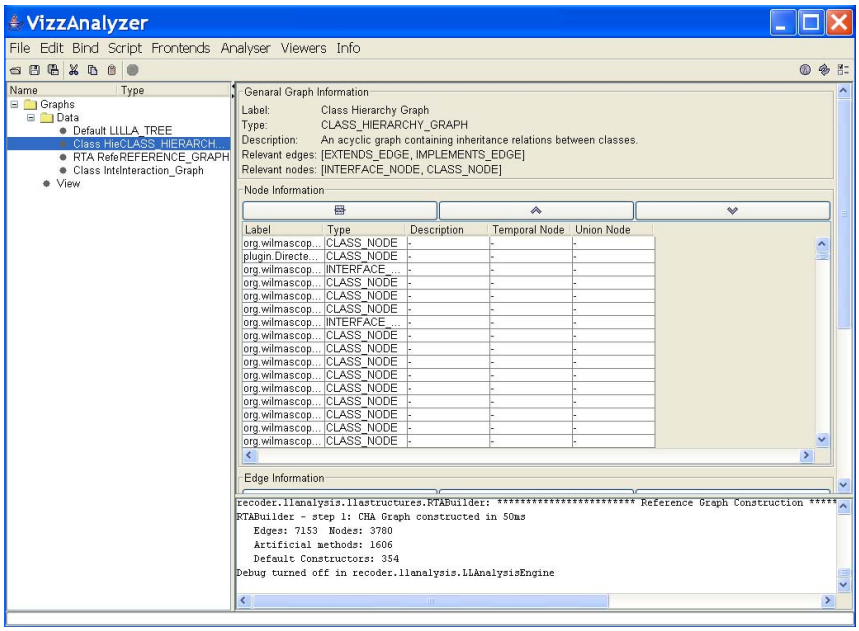

Figure 5.3: Code Retrieval Result

via configuration files (xml files), and deployed in the *graph_spec_dir* directory. The following example specification defines a graph of type *Class Interaction* with nodes of type *Class* and *Interface*, as wells as edges of the type *MethodReference*, *ConstructorReference*, *FieldReference*, *Extends* and *Implements*.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE graphs SYSTEM "graphs.dtd">
3 <graphs>
4   <graph>
5     <name>Class Interaction Graph</name>
6     <type>Interaction Graph</type>
7     <description>Register declared interactions
8                  between classes and interfaces</description>
9     <edges>
10      <edge>MethodReference</edge>
11      <edge>ConstructorReference</edge>
12      <edge>FieldReference</edge>
13      <edge>Extends</edge>
14      <edge>Implements</edge>
15    </edges>
16    <nodes>
17      <node>ClassDeclaration</node>
18      <node>InterfaceDeclaration</node>
19    </nodes>
```

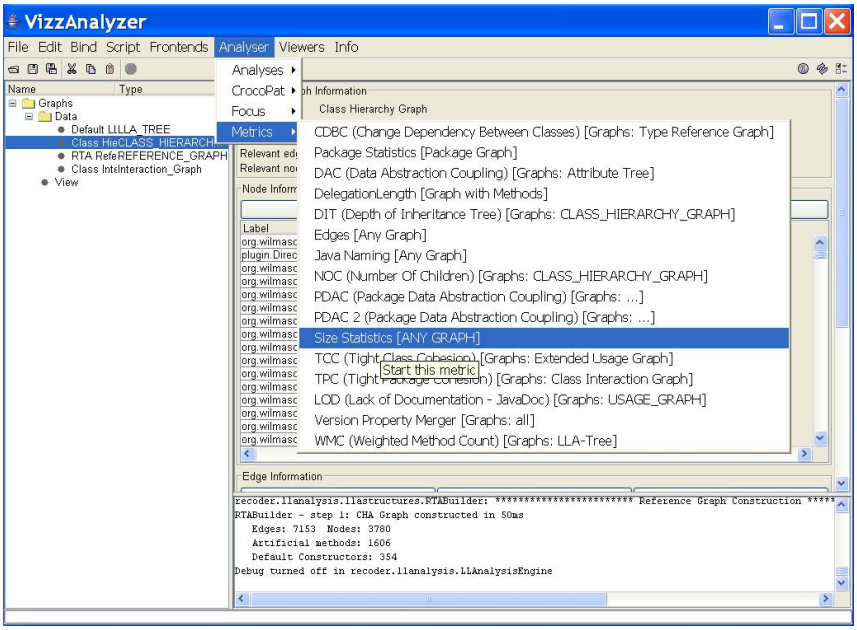

Figure 5.4: Analyser plugin

```
20  </graph>
21 </graphs>
```

The machinery behind the RecoderComp parser is build as an event caster and listener. While RecoderComp is parsing the WilmaScope program, events for each identified node are thrown. On the listener side, several listeners are waiting for input; one listener for each graph specified. On reception of listener specific nodes and edges, a listener builds and extends the corresponding graph.

On successful execution of our front-end, resulting graphs are displayed under the tab *DATA* in the GUI, cf. Figure 5.3. Moreover, the name, type and other properties of the graph are shown in the neighbor tab *General Graph Information*. Right below, nodes are listed with their properties, where each column represents one property. Edges and their properties follow right after the nodes, however, due to space limitations, edges are not visible in Figure 5.3. Finally, a status panel is attached to the lower part of the GUI.

So far, we have resulted in a hierarchy graph. The parsing is complete. In the following, we apply the Analayzer in order to retrieve additional information about the class hierarchy graph.

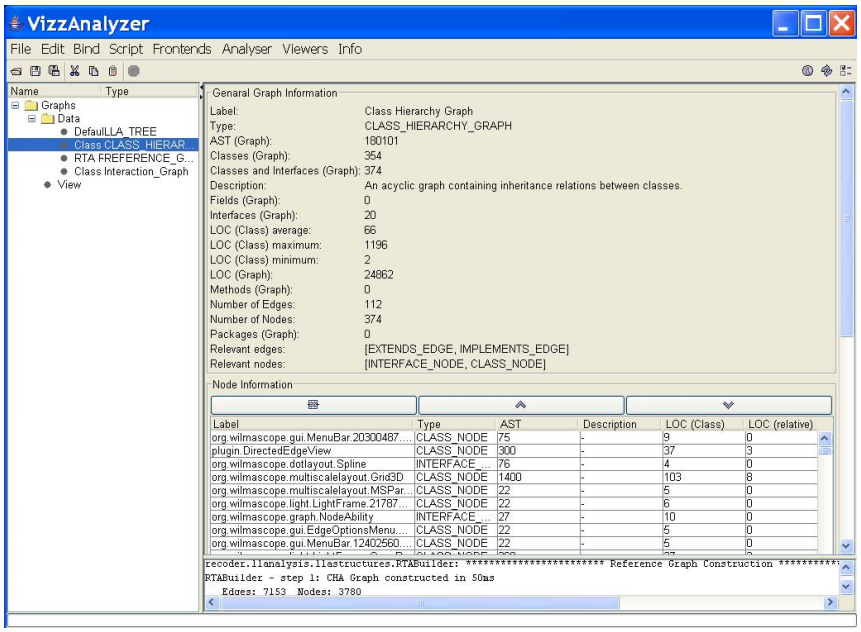

Figure 5.5: Metrics results

5.2 The Analysis

For some statistical information, we select *Size Statistics* under the `Analyzer` *Metrics* panel in the GUI, as shown in Figure 5.4. The results of this analysis are shown in Figure 5.5. Apparently, the WilmaScope program example consists of 180101 abstract syntax tree (AST) nodes, 374 Classes and Interfaces, and 24862 Lines of Code (LOC). Moreover, new columns (properties) for the nodes were created, i.e. a property for AST, LOC (per Class) and LOC (relative). LOC (relative) is the amount of lines of code normalized to a scale between 0 and 100. The normalization simplifies the model mapping, because the minimum and maximum values are known.

Having in mind that the current hierarchy graph is to be visualized, we would like to simplify its structure as much as possible to support comprehension. Therefore, we filter for this problem meaningless program information out, such as inner classes, cf. Figure 5.6. This figure shows a screenshot of the *Parse Filter* analysis, which is part of the `Analyzer`. By specifying a property (here Label), a regular expression is applied on all nodes, filtering those out that match the expression. In Figure 5.6, we use the regular expression *.*\d;* in

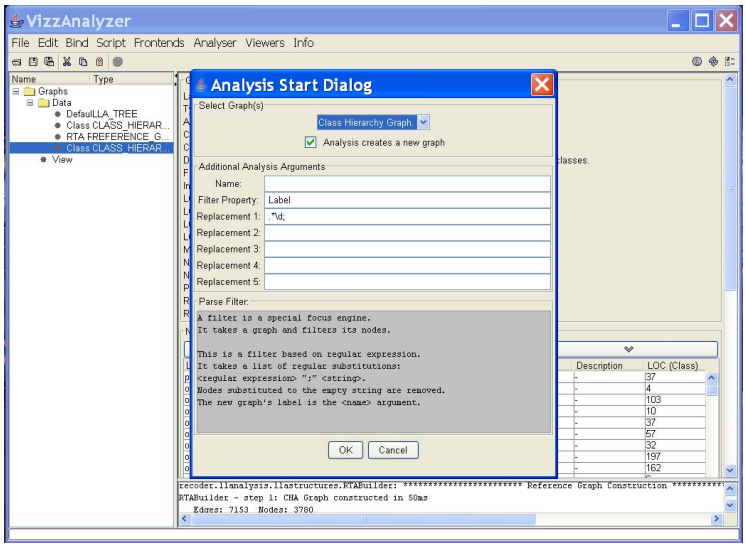

Figure 5.6: Parse Filter

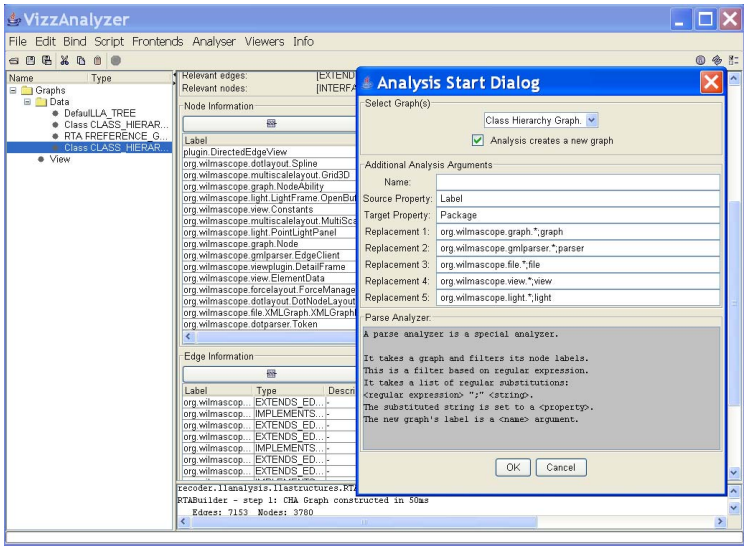

Figure 5.7: Parse Analyzer

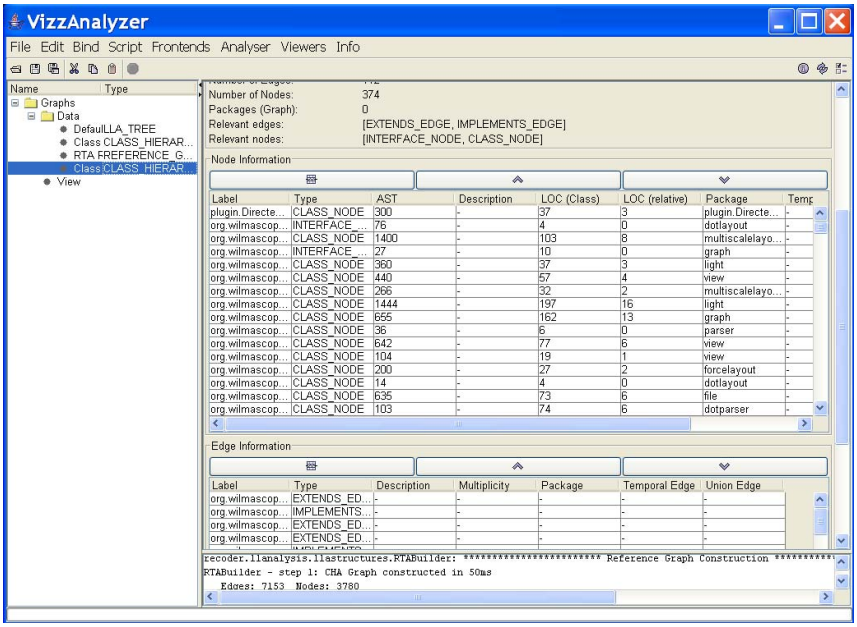

Figure 5.8: Analysis Results

order to define all digits followed after a dot. When a node with a label containing that pattern is found, the node is removed. For instance a node *org.wilmascope.graph.1232343*, indicating an inner class, will be removed.

To tackle the actual problem, namely to color classes containing to the same package equally, we need to create a new property for each node, specifying the name of the package it belongs to. This property (of the model graph) is later mapped to the property color (of the view graph). To create a new property *Package*, we select the *Package Analyzer*, cf. Figure 5.7.

The analysis takes two parameters, the source property (here Label) and target property (here we choose Package). Next, a regular expression is applied to select nodes that should have the same package name, cf. Figure 5.7. The figure shows for instance that all nodes starting with the node label*org.wilmascope.gmlparser.** are assigned to the new package property *gmlparser*. Figure 5.8 shows the results. You may notice that the GUI of the Package Analyzer, allows only for the entry of five replacements. Nevertheless, the analysis may be applied multiple times (with the Package property as source and target property after the first run).

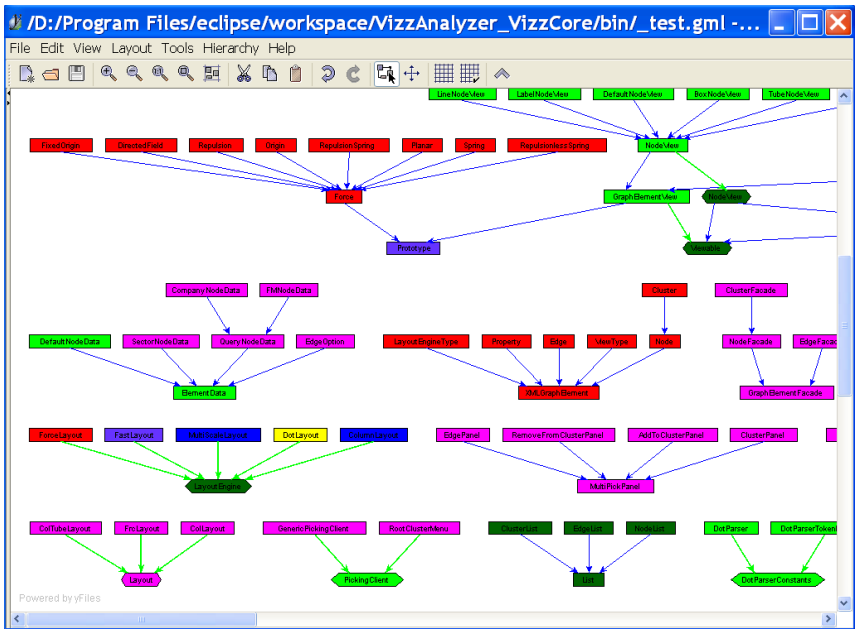

Figure 5.9: yEd Screenshot

5.3 The Model Mapping

The model mapping specifies the mapping between our model graph (created hierarchy graph) and a view graph (to be created). The mapping is specified within a XML-file. The GUI allows with the option *bind*, the creation, loading and modification of such a file. Our example mapping file to map the *Package* property of each node to the *Color* property, looks as follows:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE Bindings PUBLIC "www.msi.vxu.se/tiny" "VAbinding.dtd">
3 <Bindings>
4  <!-- Binding of node view properties -->
5  <NodeBindings>
6    <Binding>
7      <Source>Label</Source>
8      <Target>Label</Target>
9      <Function>
10        <StringFilterBinding RegExp = ".*\." Replacement = ""/>
11      </Function>
12   </Binding>
13    <Binding>
```

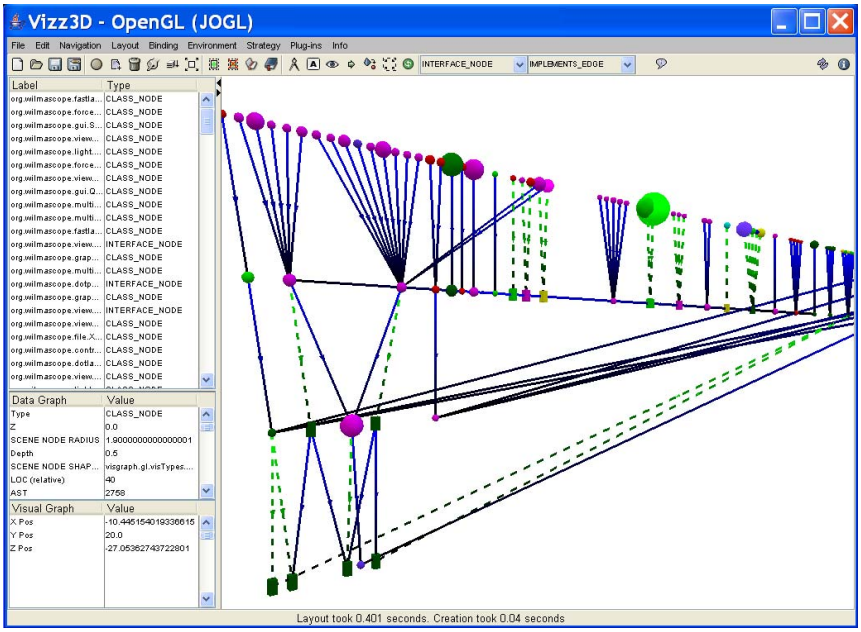

Figure 5.10: Vizz3D Screenshot

```
14      <Source>Package</Source>
15      <Target>Node Color</Target>
16      <Function>
17        <DefaultColorBinding/>
18      </Function>
19    </Binding>
20    <Binding>
21      <Source>Type</Source>
22      <Target>Node Shape</Target>
23      <Function>
24        <IndexBinding/>
25      </Function>
26    </Binding>
27 </NodeBindings>
```

This code snippet shows the mapping of the nodes only. First the Label of the source nodes (model graph) is mapped to the label of the target nodes (view graph), cf. lines 6-12. For this, a *StringFilterBinding* function is used. This function, as specifies in line 10, binds only the appendix of each nodes label, i.e. filters all leading package information from each label. Lines 13-19, in turn, specify the mapping between the package and the color property. Here, a *DefaultColorBinding* function is applied, mapping to every new entity in the source property a new color. Additionally, between the lines 20 and 26, the

type of a node, i.e. the String ”Class” or ”Interface”, is mapped to a shape (Integer) with a *IndexBinding* function. Refer to [LP05] for more information about binding functions. The model mapping of the edges underlies the same principle and is not further discussed.

Finally, the model mapping results in a new view graph. The view graph can be illustrated with any visualization tool available in the `VizzAnalyzer` 's GUI.

5.4 The Visualization

Finally, we use yEd to visualize our hierarchy graph (view graph), containing already information about color and shape of nodes and edges. Missing is to this point only the assignment of appropriate x,y and z-coordinates (which are all zero by default). Therefore, a hierarchical layout algorithm within yEd is chosen.

The resulting hierarchy structure of the WilmaScope source code is depicted in Figure 5.9. The color indicates that classes belong to different physical packages. Classes within one and the same hierarchy tree but with a different color, i.e. belonging to a different package, may indicate bad design decisions and hence should be investigated further.

The same hierarchy graph is illustrated once more in Figure 5.10, utilizing our `Vizz3D`. The main differences are the metaphor (circles and boxes) and the layout algorithm used.

Chapter 6

First Contact Analysis - GRAIL

First Contact Analysis is an assessment method for software systems. It is based on a number of automated analyses. They give a graphical overview of the systems:

- Architecture and structure,
- Design, and
- Complexity.

Such an overview is a valuable documentation of the system allowing to teams members a better communication on system matters. New team members experience a lower learning curve when trying to comprehend an unknown system.

Since all analyses are objective and their results clearly separated from our conclusion, both project managers and developers might experience insights on actual system properties and their divergence from expected and intended system properties. In addition to the system overview, First Contact Analysis pinpoints parts of the system that might be critical in further maintenance of the system. We recommend reviewing and refactoring specifically these system parts if necessary.

First Contact Analysis is based a tool called VizzAnalyzer. It implements state-of-the-art structural program analyses, checkers for best practice designs (design patterns), and metrics calculations. Results are presented on various levels of abstraction with software

visualizations and statistics charts. The VizzAnalyzer supports both the identification of weak points or design flaws in software systems and a better understanding of the analyzed system. An appropriate architecture and good design cannot be formalized and correctly measured. However, research in the field of software architecture and design propose a number of heuristics. All heuristics applied are confirmed in numerous practical field studies.

Heuristics embody knowledge about good design, which has been proven to be valid in numerous industrial cases studies. However, finally, only system architects, designers, and developers knowing the system are able to properly interpret the analysis results and identify false alarms. Hence, First Contact Analysis is only an initial step that needs to be followed-up by discussions among the people responsible for the system. The software visualizations and statistics charts that the First Contact Analysis provides have proved an excellent basis therefore.

6.1 Summary

We performed the First Contact Analysis on the system Grail.

1. Grail is a small-medium size system. The average of 9 classes per top-level package (subsystem) is rather small. The package hierarchy is flat, just contains grail and all 11 subsystem-packages. The averages of 8 methods and 92 code lines per class, respectively, are reasonable values and do not uncover deviations from state-of-the-art design rules.

2. We observe that the declared packages do not contain natural components. If this were the intension major restructuring reducing the interactions over package boundaries would be recommended.

3. The inheritance structure does not show any violations to standard designing guidelines. It should be checked if all of them are planned being extended in the future or if some could be removed as unnecessary abstraction. There are no MatrixBasedUndirectedGraph, MatrixBasedUndirectedNode, and MatrixBasedUndirectedEdge classes, as one would expect when looking at the naming and inheritances of related classes.

4. The classe grail.properties.GraphProperties shows in general a high external coupling and low internal cohesion at the same time. This class also shows in general a high

change possibility and lack of documentation at the same time. This is a critical class in the system. It should be better documented and, maybe refactored.

5. The class grail.matrixbased.MatrixBasedDirectedGraph has a high change possibility but appears to be well documented. Because of its high change possibility, it should be documented better.

6. The following classes are both complex and large:

   a. grail.matrixbased.MatrixBasedDirectedGraph
   b. grail.converters.GML
   c. grail.interfaces.AbstractGraph
   d. grail.DefaultTreeView

These classes might be considered splitting. Special attention should be paid to the class grail.matrixbased.MatrixBasedDirectedGraph. This class was recommended to re-engineer because of its high change possibility.

6.2 Global Statistics

In most cases, it is a good idea to measure some basic properties of the software system we are dealing with. Knowledge of some basic numbers indicating the size of the system may help to put the more specific measurements in later sections of this report into a better context.

Top level packages (subsystem)	11
Packages	12
Classes	85
Interfaces	15
Methods	785
Public Attributes	77
Lines of code	9.275

Table 6.1: Global Statistics for GRAIL

Table 6.1 summarizes these basic figures for the GRAIL system as of December 2, 2004.

Interpretation: GRAIL is a small-medium size system. The average of 9 classes per top-level package (subsystem) is rather small. The package hierarchy is flat, just contains grail and all 11 subsystem packages. The averages of 8 methods and 92 code lines per class, respectively, are reasonable values and do not uncover deviations from state-of-the-art design rules.

6.3 Analysis of Architecture and Structure

We will analyze both architectural properties and properties of the inheritance structure.

6.3.1 System Architecture

The architecture of a system consists of components and their interactions. Components are interacting sets of classes and smaller components, i.e. the notion of a component is recursive. Interactions include method calls and field accesses.

An architecture is considered good if its components have a high internal cohesion and low external coupling. The cohesion of a component is defined as the degree interactions among contained classes and components. The coupling of a set of components is defined as the degree interactions among them. The idea is that components do a lot of the work internally and only rarely communicate with other components. This allows, e.g., maintaining, reusing or understanding individual components regardless of other components around. Our analyses will therefore check for high internal cohesion and low external coupling in the components.

On architectural level, different styles have been proven worthwhile for different kinds of systems. For instance, a Tree-Tier-Architecture helps to separate presentation related components, from business logic and data storage. System architects and designers always have an architectural style in mind when designing their system. However, in development and maintenance the style might get deteriorated due to time pressure or misunderstandings. Our analyses check the conformance of the existing architecture in the system with the intended architectural style to uncover deviations.

6.3.2 Evaluation of the Architecture

Figure 6.1 shows the architecture of the system. It compares ”natural” components and declared packages. Natural components are sets of classes with high cohesion among another and low coupling to the rest of the system. Declared packages of the systems are classes/interfaces of the same top-level package. Note, that packages might be used for structuring of the system that is orthogonal to the component structure, e.g. bringing together classes solving similar problems.

Interpretation: We observe that the declared packages do not contain natural components. This is ok in library systems. However, if it were the intension to declared packages along with components major restructuring reducing the interactions over package boundaries would be recommended.

6.3.3 Inheritance Structure

The inheritance structure of a system is defined by implements and extends relations of classes and interfaces. Such a structure is expected to follow a few general design rules.

Chains in the inheritance structure, i.e. substructures where super-classes/-interfaces only have one single sub-class/-interface and are themselves the only super-classes/-interfaces are considered a sign of unnecessary abstraction. In libraries and frameworks however, further subclasses might come form the application using them. Hence, chains do not necessarily indicate a design flaw.

Inheritance structures that contain direct and indirect, i.e. transitive, relations between two classes are to avoid since the transitive relation does not contributed to the system semantics.

Inheritance structures should not cross too many package boundaries. Moreover, they should neither be too deep nor too wide. These are rather fuzzy recommendation. Even though, one should observe the average in a system and have a closer look at escapes.

Figure 6.2 depicts the inheritance structure of Grail. Nodes represent classes/interfaces; edges represent extends/inherits relations. Again, the color scheme encodes the top-level packages of classes and interfaces. It only depicts classes/interfaces that are in an inheritance relation.

In the major inheritance structure, there is only on chain from the interface TreeInterface to the class DefaultTreeView.. We do not observe any two classes with both direct and indirect inheritance relations.

There are at most three packages involved in any of the inheritance hierarchies; the maximum inheritance depth is 4, maximum width is 7. Only the latter could be considered an escape from the average.

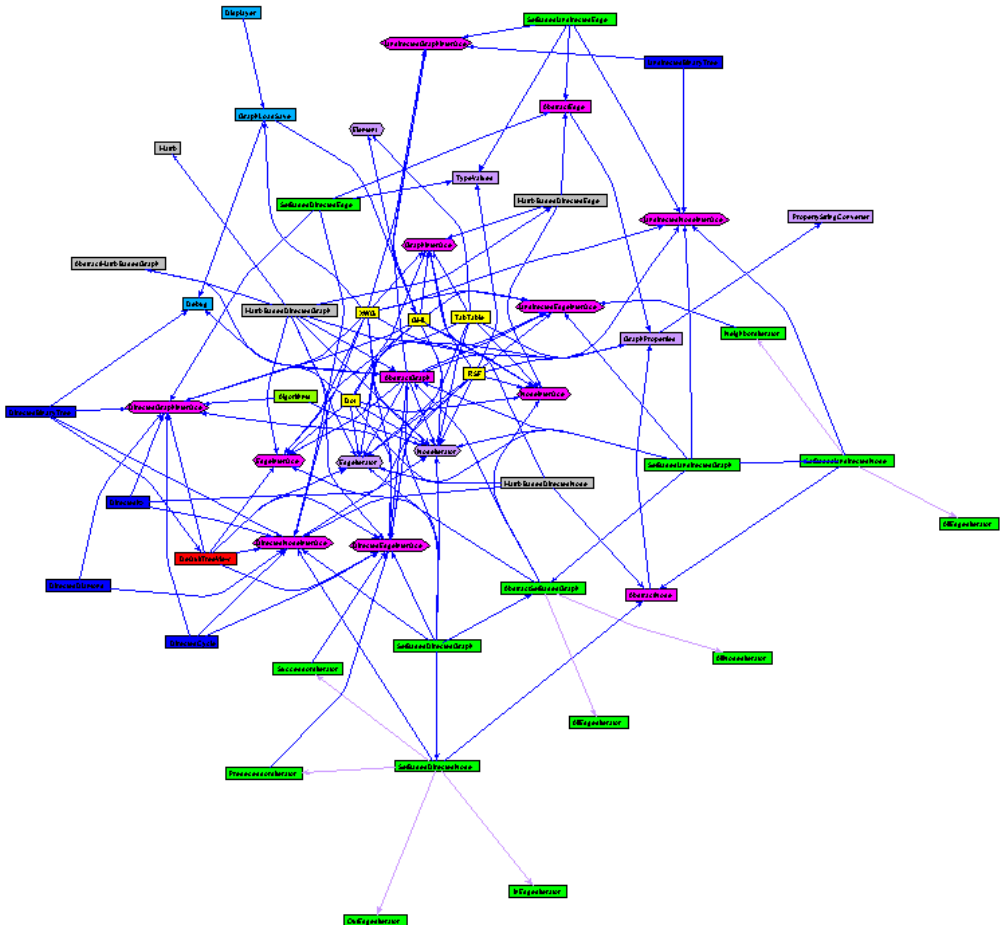

Figure 6.1: The class interaction graph shows classes and interface nodes and their interaction, i.e. calls and field accesses. The colors indicate declared packages of the systems where classes/interfaces of the same top-level package are depicted with the same color. This shows the architecture of Grail. No ”natural” components become obvious, i.e. sets of classes with high cohesion among each other and low coupling to the rest of the system.

Figure 6.3 depicts the part of the inheritance structure of Grail that is related to Set-Based* and MatrixBased* classes. The name schema of these classes and their dependency induce an intended similarity of these two sets of classes. The respective inheritance

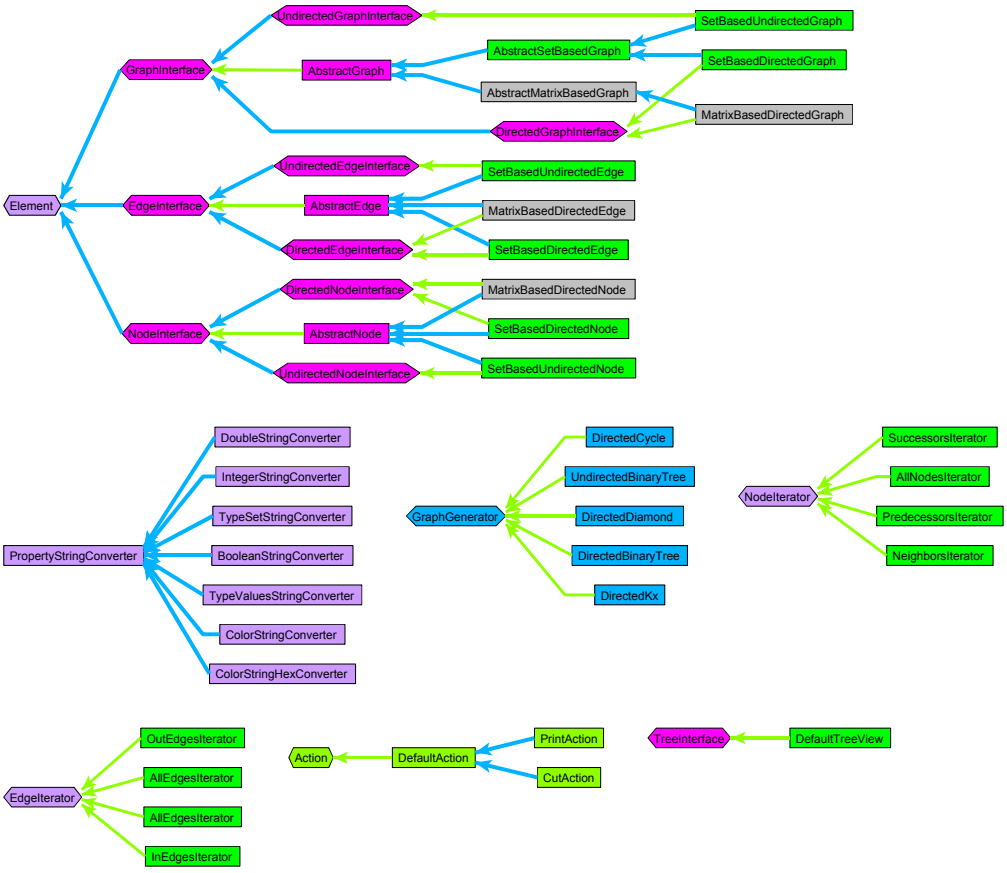

Figure 6.2: The inheritance structure of Grail. Nodes are classes (boxes) and interfaces (diamonds). The color scheme of nodes encodes the top-level packages of classes and interfaces. The color scheme of edges distinguishes implements (green) and extends (blue) relations.

structures are diverging.

Interpretation: The inheritance structure does not show any violations to standard designing guidelines. It should be checked if all of them are planned being extended in the future or if some could be removed as unnecessary abstraction. There are no MatrixBasedUndirectedGraph, MatrixBasedUndirectedNode, and MatrixBasedUndirectedEdge classes, as one would expect when looking at the naming and inheritances of related classes.

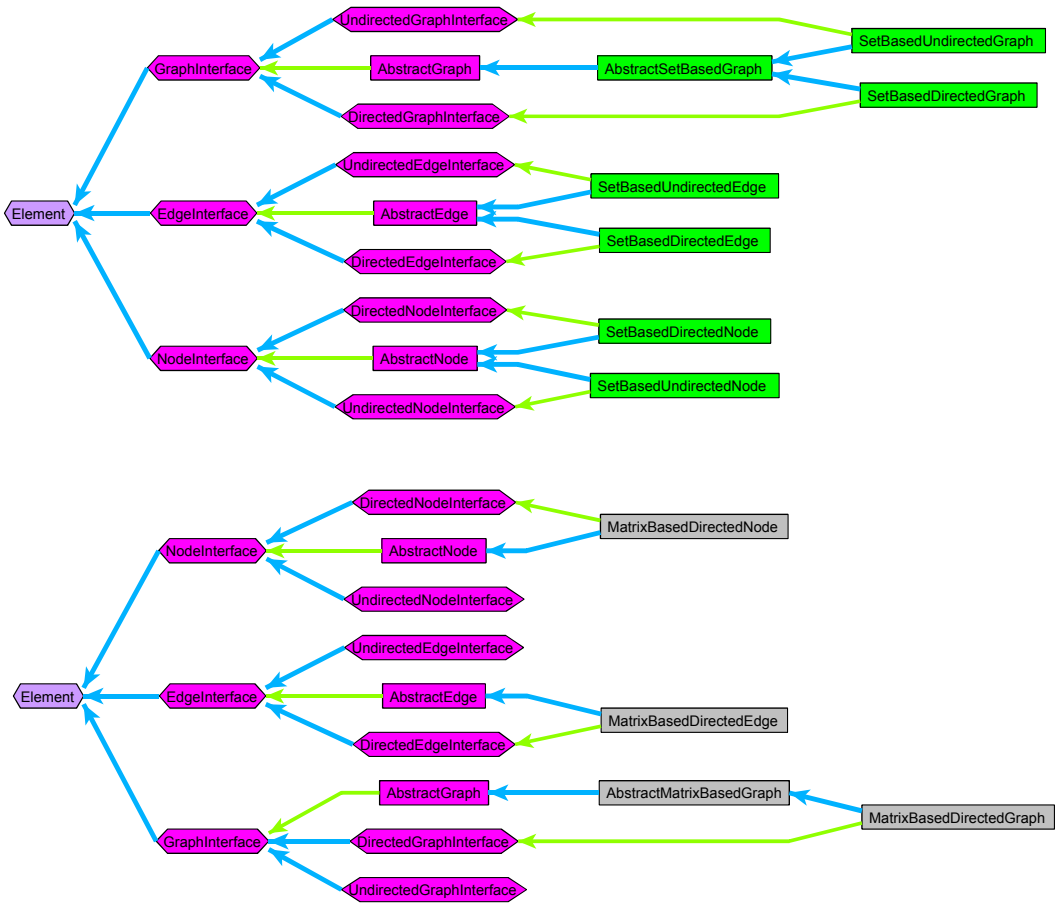

Figure 6.3: Figure 3. Copies a part of inheritance structure of Grail. Depicted are inheritance structures related to SetBased* (top, green) and MatrixBased* (bottom, gray) classes.

6.4 Design Metrics

In the same way cohesion and classes indicate good design of components, these measures are applicable to assess the design of individual classes. Here, we expect high cohesion of methods and fields within a class and a low coupling between classes.

Tight Class Cohesion (TCC) is the relative number of directly connected methods in a class. TCC indicates the degree of connectivity between visible methods in a class. Given

the number n of local methods (excluding inherited methods), TCC is defined as $TCC = \frac{ndp}{np}$ with $np = \frac{n \times (n-1)}{2}$ the possible pairs of these methods and ndp the number of method pairs actually calling another.

The TCC for a class is 0 if np = 0. The resulting values range from 0.0 to 1.0 on a rational scale; we scale them between 0 and 100. Higher values indicate better cohesion of the classes. Low values indicate that a class could be split.

Data Abstraction Coupling (DAC) represents the number of abstract data types (ADTs) defined in a class. Counted are the fields defined in a class referencing a user defined type, not a primitive, language or library defined type. Inherited fields are not counted. The DAC is calculated for each class and interface. The values are integer values ranging from 0, indicating no other ADT is referenced, to a maximum number on an absolute scale; we scaled them between 0 and 100.

The higher the DAC is, the more complex is the coupling of a class with other classes. It is recommended to keep DAC low, or merge some classes, otherwise.

Figure 6.4 depicts the values of TCC and DAC for classes and interfaces of Grail. Each class/interface is a point in the diagram; its x-position indicates the class'/interface's TCC value, its y-position indicates the class'/interface's DAC value. Both values are linearly scaled between 0-100.

Since, high cohesion and low coupling are desired, classes in the top-left corner are to review.

Change Dependency Between Classes (CDBC) determines the potential amount of follow-up work to be done in a client class when a server class is modified. It also indicates the strength of a coupling. The goal is to keep changes in the system local by reducing the system coupling. Lower values indicate lower coupling and, thus, a better stability in the system.

The CDBC value is defined between a client and a server class as the number of methods, which need to be (potentially) changed in the client if a server changes. The CDBC value is between 0 and the count of methods in the class. We compute the average CDBC value of each (client) class over all (server) classes it is directly connected with. The scale is rational.

Lack of Documentation (LOD) measures the amount of undocumented declarations per class (counted are the class declaration itself and the method declarations, but not field declarations). Only "JavaDoc" style documentation is taken into account. Documentation within methods is ignored. Only the syntax of the comments is parsed, not the semantics.

The LOD value is calculated for each class or interface as the number of undocumented declarations (local not inherited methods). A LOD of 0 indicates that all possible entities are documented; a higher value indicates the lack of documentation.

Figure 6.5 depicts the values of CDBC and LOD for classes and interfaces of the

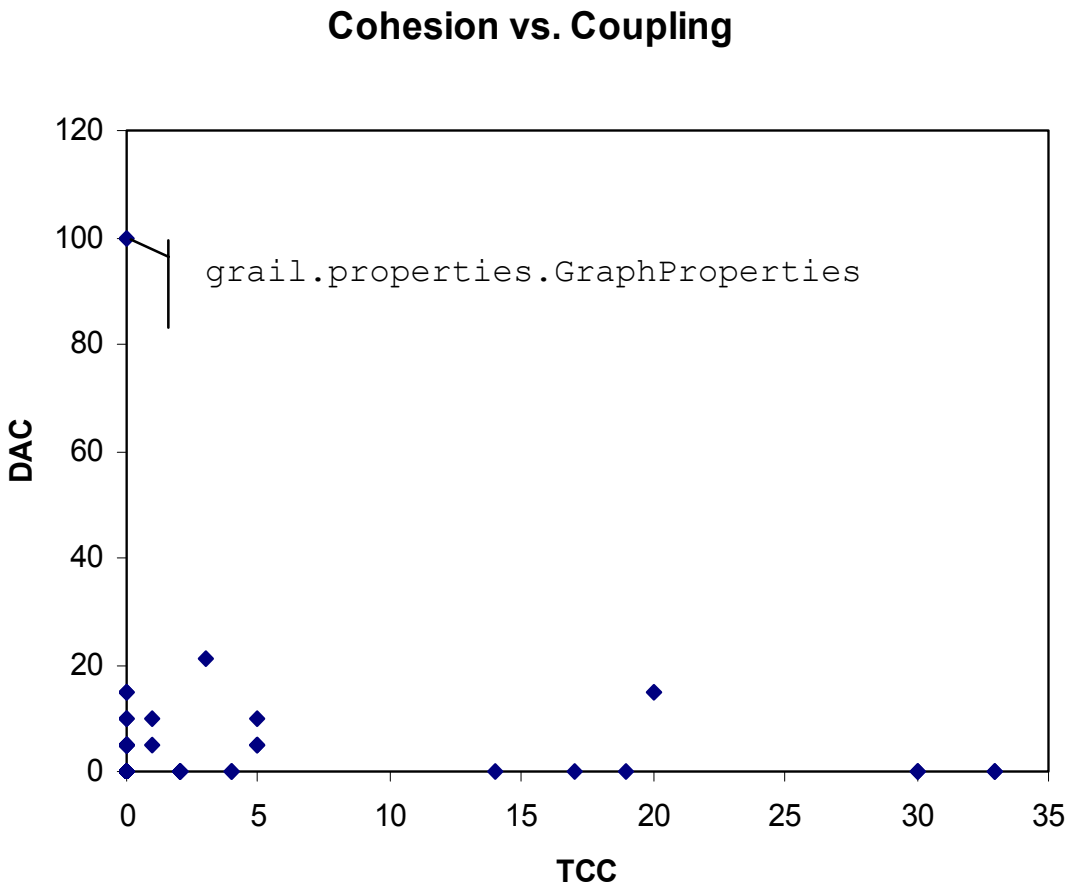

Figure 6.4: Relative intra-class-cohesion (TCC) and relative inter-class-coupling (DAC).

VizzAnalyzer. Each class/interface is a point in the diagram; its x-position indicates the class'/interface's CDBC value, its y-position indicates the class'/interface's LOD value. Both values are linearly scaled between 0-100, where the highest absolute values measured in each dimension are mapped to 100.

Since, low possibility of changing a class/interface and low lack of documentation are desired, class/interface in the top-right corner are to review. These classes are likely to be changed and are poorly documented at the same time. Class/interface in the bottom-right corner might be critical, too. These classes are likely to be changed, too, but at least well documented.

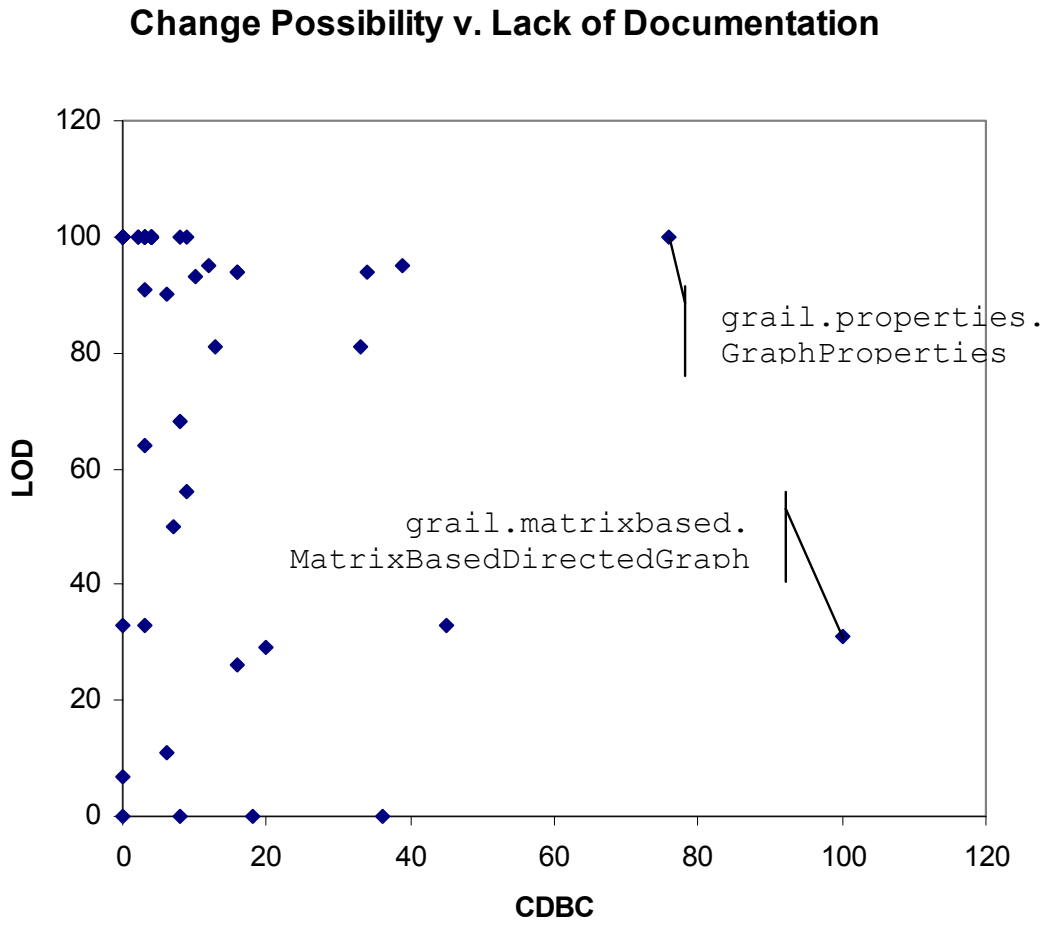

Figure 6.5: Relative possibility of changes of a class (CDBC) and relative lack of documentation of classes and methods (LOD).

Interpretation: The classe grail.properties.GraphProperties shows in general a high external coupling and low internal cohesion at the same time. This class also shows in general a high change possibility and lack of documentation at the same time. This is a critical class in the system. It should be better documented and, maybe refactored.

The class grail.matrixbased.MatrixBasedDirectedGraph has a high change possibility but appears to be well documented. Because of its high change possibility, it should be documented better.

6.5 Complexity Metrics

The complexity of a class is often considered interesting since it points at classes that are hard to understand and to maintain.

Weighted Method Count (WMC) computes the complexity of the methods of a class and is also a measure for the complexity of a class. It gives a good idea about how much effort is required to develop and maintain a class. The methods are weighted according to McCabe's Cyclomatic Complexity Metric. It counts the possible execution branches in a method for the branching statements: if, for, while, do. It is assumed that each branch has the same complexity/weight. We scaled the values between 0 and 100.

Lines of Code (LOC) count the lines of code in a class and interface, respectively. It's an absolute metric that we scaled between 0 and 100. Classes that are both highly complex and large are critical when it comes to understanding and maintaining a system. It is especially alerting if they are recommended to be restructured because of structural and/or design issues (see previous sections).

Figure 6.6 depicts the values of LOC and WMC for classes and interfaces of the VizzAnalyzer. Each class/interface is a point in the diagram; its x-position indicates the class'/interface's LOC value, its y-position indicates the class'/interface's WMC value. Both values are linearly scaled between 0-100, where the highest absolute values measured in each dimension are mapped to 100.

Since, high complexity of class/interface and large classes are critical, class/interface in the top-right corner are to review. These classes are hard to be changed.

Interpretation: The following classes are both complex and large:

1. grail.matrixbased.MatrixBasedDirectedGraph
2. grail.converters.GML
3. grail.interfaces.AbstractGraph
4. grail.DefaultTreeView

These classes might be considered splitting. Special attention should be paid to the class grail.matrixbased.MatrixBasedDirectedGraph. This class was recommended to re-engineer because of its high change possibility (CDBC).

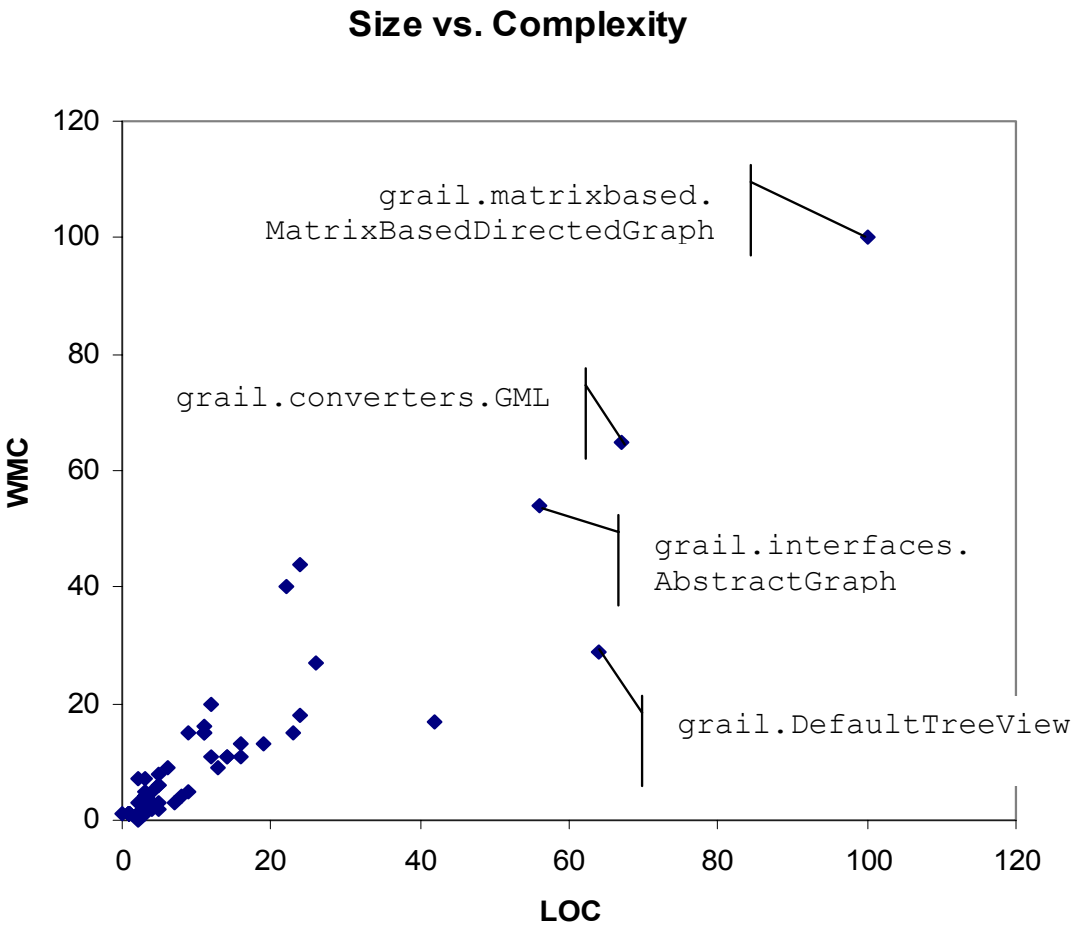

Figure 6.6: Relative size in lines of code (LOC) and relative complexity in weighted method complexity (WMC).

6.6 Conclusions

This report documented the results of the First Contact Assessment of Grail. Our assessment included and architecture analysis in general, a comparison between intended and actual architecture, an analysis of the inheritance structure, analyses of coupling and cohesion on class level, and analyses of the understandability and maintainability of the software.

We separated analyses from interpretation. All analyses results are factual. However,

we need to emphasize that their interpretation should be taken with care. It can only point to suspicious spots of the system. Developers and designers experienced with the system should crosscheck each interpretation. None of the results revealed a severe problem. We conclude that the current design and implementation of the software provides a solid foundation for future development cycles.

It is recommended to integrate a quality assessment in this future development. The group would be like to give support in that venture with both tools and expert knowledge.

Chapter 7

First Contact Analysis - `VizzAnalyzer`

This chapter describes the first contact analysis with the `VizzAnalyzer` on the VIZZANALYZER FRAMEWORK.

7.1 Summary

We performed the First Contact Analysis on the system VizzAnalyzer.

1. The VizzAnalyzer is a medium size system. The averages of 59 classes per subsystem and 8 methods and 90 code lines per class, respectively, are reasonable values and do not uncover deviations from state-of-the-art design rules.

2. Moving the packages hlanalyses and analyses into a common package analysis and the packages layout, vizz3d, and visgraph into a common package visualization, respectively, would lead to a top-level packaging reflecting the top-level architecture of the system. The only class in package data should be moved into the recoder package. The direct interaction of classes in the recoder and the analyses packages breaks the intended centralized architectural style. It is highly recommended to refactor it.

3. The inheritance structure does not show any severe violations to standard designing guidelines. However, there are quite view inheritance chains of length two. It should be checked if all of them are planned being extended in the future or if some could be removed as unnecessary abstraction.

4. The following classes show in general a high external coupling and low internal cohesion at the same time:

   a. vizz3d.java3d.visengine.interactions.J3dAdvancedInteraction
   b. vizz3d.common.gui.MainFrameInitiator
   c. recoder.llanalysis.basetree.recoder.RecoderElementVisitor
   d. recoder.llanalysis.LLAInfo

5. The following classes show in general a high change possibility and lack of documentation at the same time:

   a. recoder.llanalysis.llaelements.LLAPackage
   b. analyzer.hlanalysis.HLAnalysis
   c. recoder.llanalysis.basetree.baseelements.BTType
   d. recoder.llanalysis.llaelements.LLAMethod

   The class vizz3d.opengl.visengine.util.GLAppearance has a high change possibility but appears to be well documented. These classes in the packages recoder. llanalysis and vizz3d might be considered worth being reengineered.

6. The following classes are both complex and large:

   a. recoder.llanalysis.basetree.recoder.RecoderElementVisitor
   b. vizz3d.opengl.visengine.Interactions.GLAdvancedInteraction

   These classes might be considered splitting. Special attention should be paid to the class recoder.llanalysis.basetree.recoder.RecoderElementVisitor. This class was recommended to re-engineer because of its high external coupling and low internal cohesion.

7.2 Global Statistics

In most cases, it is a good idea to measure some basic properties of the software system we are dealing with. Knowledge of some basic numbers indicating the size of the system

may help to put the more specific measurements in later sections of this report into a better context. Table 7.1 summarizes these basic figures for the VizzAnalyzer system as of November 13, 2004.

Top level packages (subsystem)	9
Packages	83
Classes	472
Interfaces	61
Methods	4.478
Public Attributes	327
Lines of code	47.737

Table 7.1: Global Statistics for the VIZZANALYZER FRAMEWORK

Interpretation: The VizzAnalyzer is a medium size system. The averages of 59 classes per subsystem and 8 methods and 90 code lines per class, respectively, are reasonable values and do not uncover deviations from state-of-the-art design rules.

7.3 Analysis of Architecture and Structure

We will analyze both architectural properties and properties of the inheritance structure.

7.3.1 System Architecture

The architecture of a system consists of components and their interactions. Components are interacting sets of classes and smaller components, i.e. the notion of a component is recursive. Interactions include method calls and field accesses.

An architecture is considered good if its components have a high internal cohesion and low external coupling. The cohesion of a component is defined as the degree interactions among contained classes and components. The coupling of a set of components is defined as the degree interactions among them. The idea is that components do a lot of the work internally and only rarely communicate with other components. This allows, e.g., maintaining, reusing or understanding individual components regardless of other components

around. Our analyses will therefore check for high internal cohesion and low external coupling in the components.

On architectural level, different styles have been proven worthwhile for different kinds of systems. For instance, a Tree-Tier-Architecture helps to separate presentation related components, from business logic and data storage. System architects and designers always have an architectural style in mind when designing their system. However, in development and maintenance the style might get deteriorated due to time pressure or misunderstandings. Our analyses check the conformance of the existing architecture in the system with the intended architectural style to uncover deviations.

7.3.2 Evaluation of the Architecture

Figure 7.1 shows the architecture of the system. It compares ”natural” components and declared packages. Natural components are sets of classes with high cohesion among each other and low coupling to the rest of the system. Declared packages of the systems are classes/interfaces of the same top-level package. Note, that packages might be used for structuring of the system that is orthogonal to the component structure, e.g. bringing together classes solving similar problems. We observe:

- The ”natural” components and declared components are not contradicting. The declared components are finer grained than the natural components.
- The declared component ”data” contains only one single class.

Further analyses split the ”green” components depicted in the top ellipsoid of Figure 7.1 from the others depicted in the two lower ellipsoids of Figure 7.1.

7.2 shows the declared components, i.e. packages, of one of the ”natural” components of the VizzAnalyzer. For the declared packages there is no distinctively high cohesion and low coupling observable.

Figure 7.3 shows the declared components, i.e. packages, of the other two of the ”natural” components of the VizzAnalyzer. For the declared packages plugIn there is no distinctively high cohesion and low coupling observable. The packages hlanalyses and analyses are highly interwoven. Together, however, they show high cohesion and low coupling. So do the packages core and recoder.

Interpretation: The packages core and recoder contain ”natural” components. A merger of the packages hlanalyses and analyses would also be such a ”natural” components, likewise a merger of packages layout, vizz3d and visgraph. One might consider moving these packages in into two analysis and visualization, respectively. This would lead to a more balanced top-level packaging representing the top-level architecture of the system. This would increase the understandability of the system.

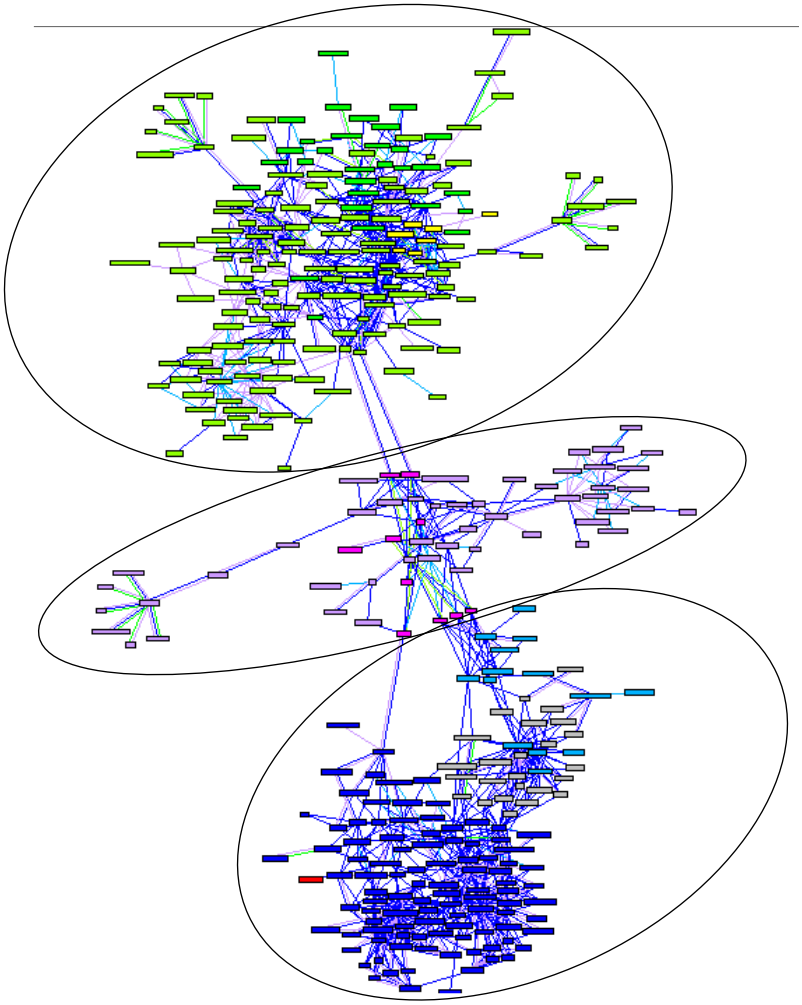

Figure 7.1: The class interaction graph shows classes and interface nodes and their interaction, i.e. calls and field accesses. It uncovers the architecture of the VizzAnalyzer. Ellipsoids characterize ”natural” components, i.e. sets of classes with high cohesion among each other and low coupling to the rest of the system. The colors indicate declared packages of the systems where classes/interfaces of the same top-level package are depicted with the same color.

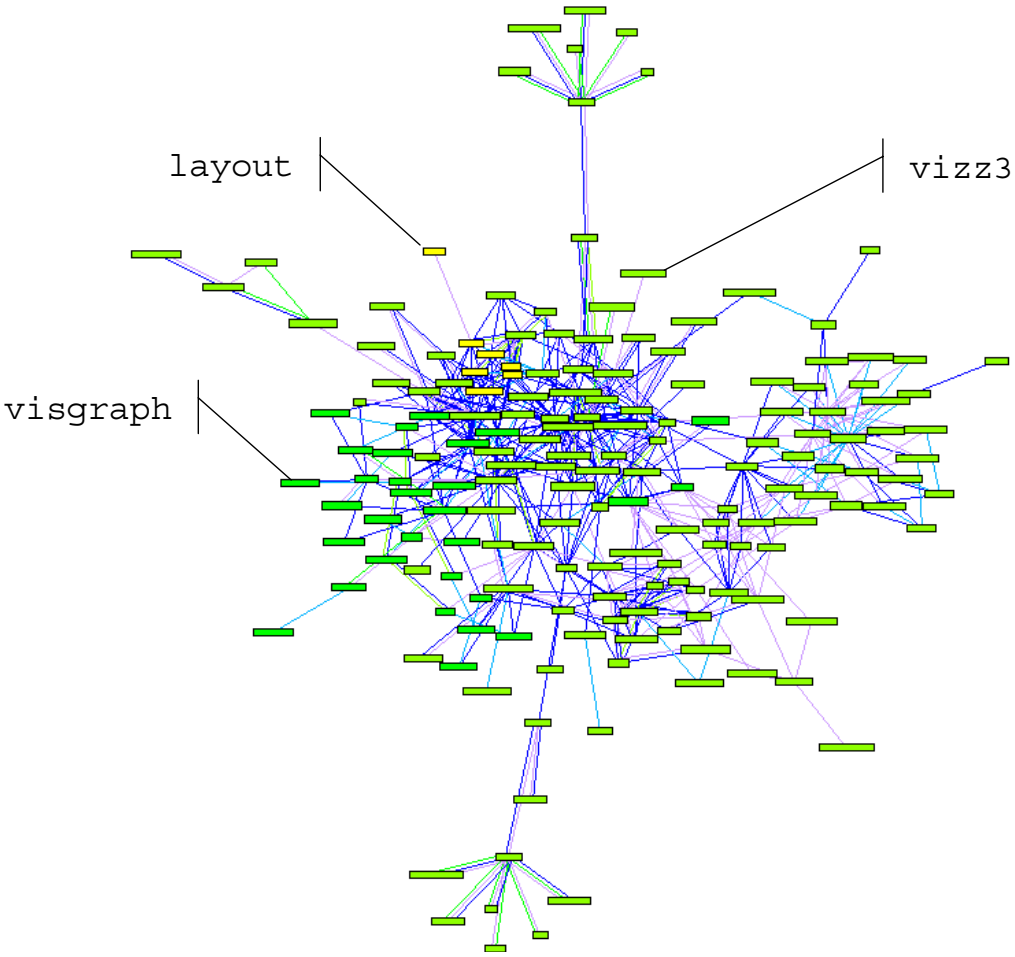

Figure 7.2: The class interaction graph of the top level packages layout (yellow), visgraph (dark green), and vizz3d (light green).

The only class in package data should be moved into the recoder package.

Package plugIn does not define a component. Again there are other structuring principles that are expressed using the package construct and, hence, this is not a design flaw.

7.3.3 Intended vs. actual Architecture

A discussion with the designers of the VizzAnalyzer uncovered the actually intended architectural style: it is a centralized architecture. The central component is located in package core. All other components should only interact with the core component via classes in the

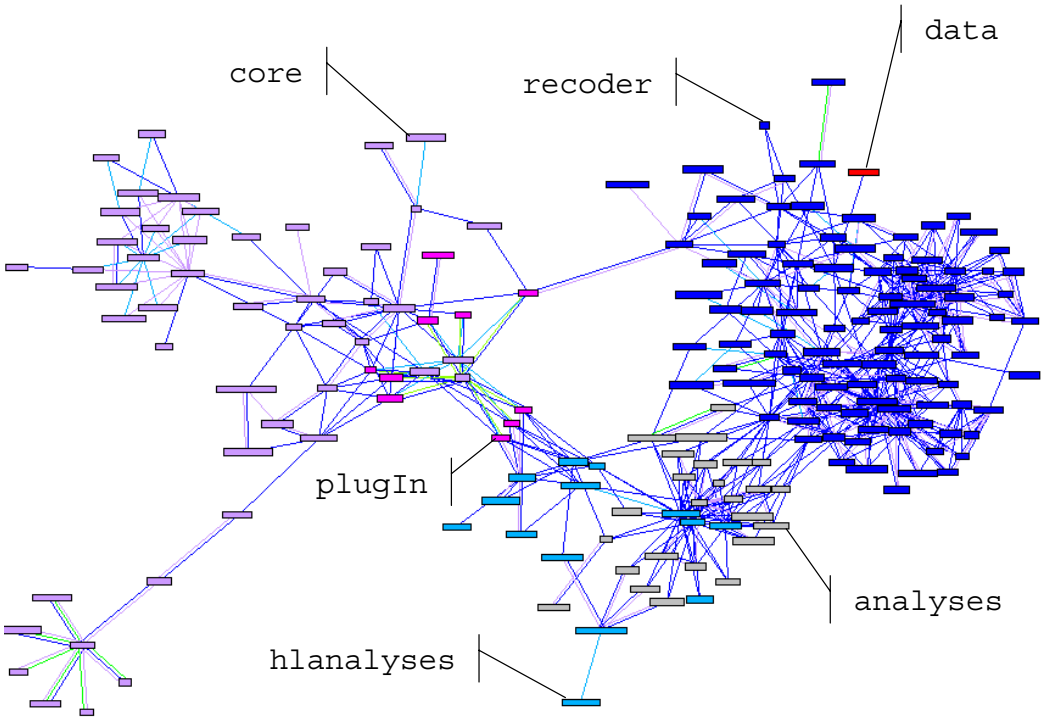

Figure 7.3: The class interaction graph of the top level packages core (lila), plugIn (purple), hlanalyses (light blue), analyzer (grey), and vizz3d (dark blue).

plugIn package (which, by the way, showed that this package is not intended to contain a component).

Figure 7.3 uncovers a direct interaction of classes in the recoder and the analyses packages.

Interpretation: The direct interaction of classes in the recoder and the analyses packages breaks the intended architectural style. It is therefore a severe design flaw; it is highly recommended to refactor it.

Inheritance Structure The inheritance structure of a system is defined by implements and extends relations of classes and interfaces. Such a structure is expected to follow a few general design rules.

Chains in the inheritance structure, i.e. substructures where super-classes/-interfaces only have one single sub-class/-interface and are themselves the only super-classes/-interfaces

are considered a sign of unnecessary abstraction.

Inheritance structures that contain direct and indirect, i.e. transitive, relations between two classes are to avoid since the transitive relation does not contributed to the system semantics.

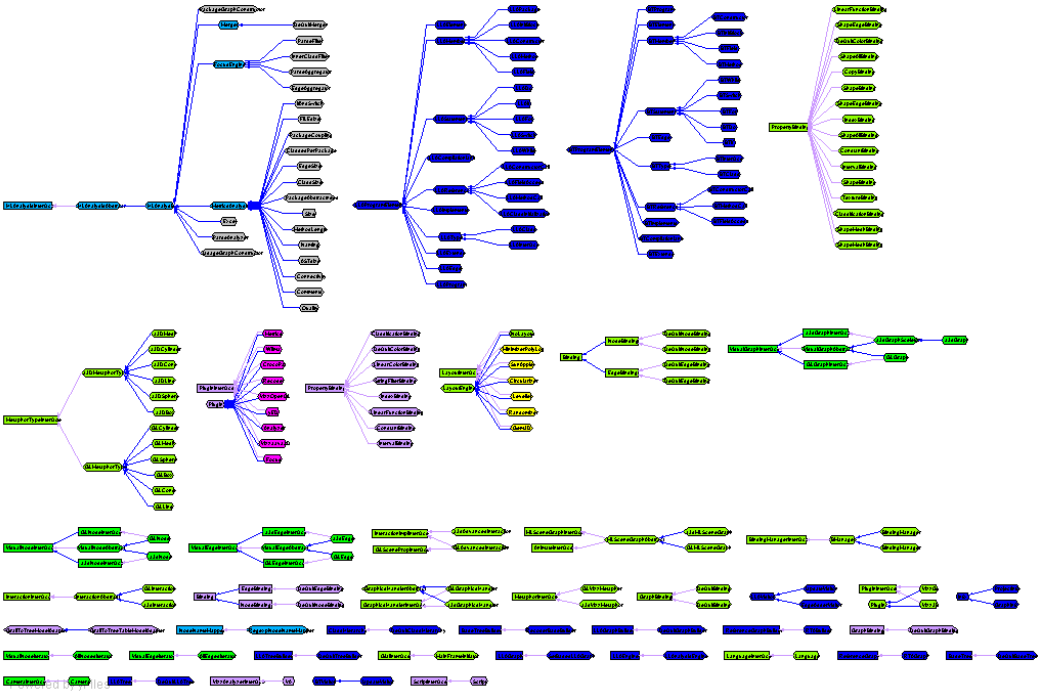

Figure 7.4: The inheritance structure of the VizzAalyzer. The color scheme encodes the top-level packages of classes and interfaces.

Inheritance structures should not cross too many package boundaries. Moreover, they should neither be too deep nor too wide. These are rather fuzzy recommendation. Even though, one should observe the average in a system and have a closer look at escapes.

Figure 7.4 depicts the inheritance structure of the VizzAalyzer. Nodes represent classes/interfaces; edges represent extends/inherits relations. Again, the color scheme encodes the top-level packages of classes and interfaces. It only depicts classes/interfaces that are in an inheritance relation.

In the major inheritance structure, there is only on chain from the interface HLAnalysisInterface to the abstract class HLAnalysisAbstract. On the other hand there are quite a

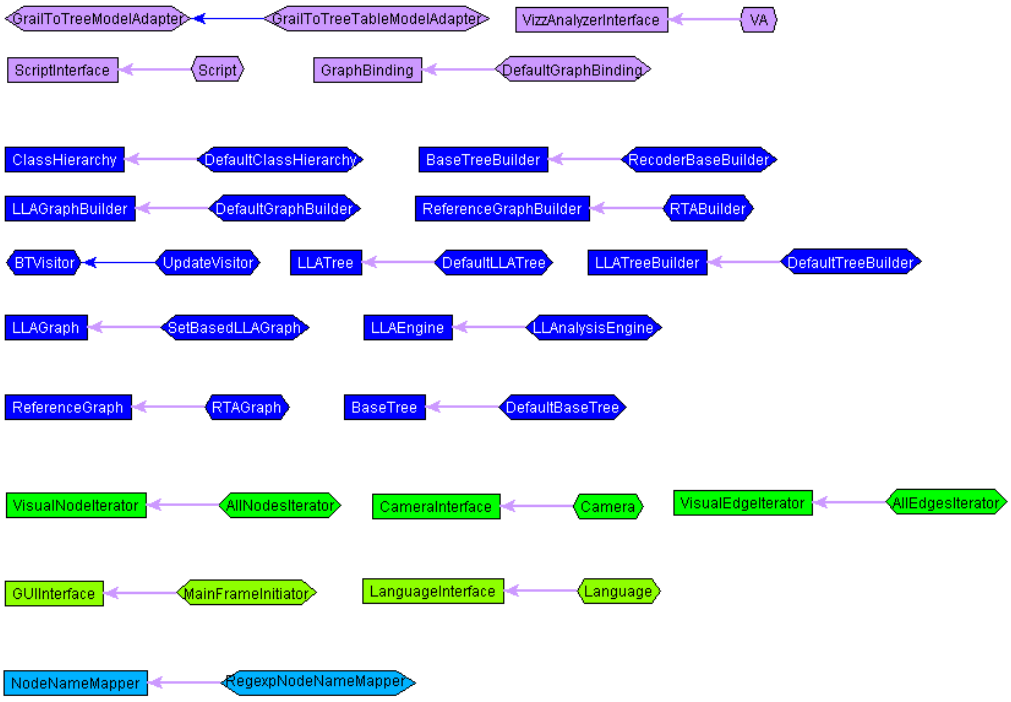

Figure 7.5: Reordering of the classes and zoom into the inheritance structure of the VizzAalyzer.

few interfaces with just one implementation. These are depicted in the lower part of Figure 7.4. Figure 7.5 reorders this part according to the packages the structures are contained in and zooms into this part of the structure. From top to bottom, colors encode the packages core, recoder, visgraph, vizz3d, and hlanalyses.

We do not observe any two classes with both direct and indirect inheritance relations.

There are at most two packages involved in any of the inheritance hierarchies; the maximum inheritance depth is 4, maximum width is 16. Only the latter could be considered an escape from the average.

Interpretation: The inheritance structure does not show any severe violations to standard designing guidelines. There are quite view inheritance chains of length two. It should be checked if all of them are planned being extended in the future or if some could be removed as unnecessary abstraction. There are 16 classes implementing the interface PropertyBinding. It is the root of a rather flat hierarchy. One might consider to further structure this hierarchy.

7.4 Design Metrics

In the same way cohesion and classes indicate good design of components, these measures are applicable to assess the design of individual classes. Here, we expect high cohesion of methods and fields within a class and a low coupling between classes.

Tight Class Cohesion (TCC) is the relative number of directly connected methods in a class. TCC indicates the degree of connectivity between visible methods in a class. Given the number n of local methods (excluding inherited methods), TCC is defined as $TCC = \frac{ndp}{np}$ with $np = \frac{n \times (n-1)}{2}$ the possible pairs of these methods and ndp the number of method pairs actually calling another.

The TCC for a class is 0 if np = 0. The resulting values range from 0.0 to 1.0 on a rational scale; we scale them between 0 and 100. Higher values indicate better cohesion of the classes. Low values indicate that a class could be split.

Data Abstraction Coupling (DAC) represents the number of abstract data types (ADTs) defined in a class. Counted are the fields defined in a class referencing a user defined type, not a primitive, language or library defined type. Inherited fields are not counted. The DAC is calculated for each class and interface. The values are integer values ranging from 0, indicating no other ADT is referenced, to a maximum number on an absolute scale; we scaled them between 0 and 100.

The higher the DAC is, the more complex is the coupling of a class with other classes. It is recommended to keep DAC low, or merge some classes, otherwise.

Figure 7.6 depicts the values of TCC and DAC for classes and interfaces of the VizzAnalyzer. Each class/interface is a point in the diagram; its x-position indicates the class'/interface's TCC value, its y-position indicates the class'/interface's DAC value. Both values are linearly scaled between 0-100, where the highest absolute values measured in each dimension are mapped to 100.

Since, high cohesion and low coupling are desired, classes in the top-left corner are to review.

Change Dependency Between Classes (CDBC) determines the potential amount of follow-up work to be done in a client class when a server class is modified. It also indicates the strength of a coupling. The goal is to keep changes in the system local by reducing the system coupling. Lower values indicate lower coupling and, thus, a better stability in the system.

The CDBC value is defined between a client and a server class as the number of methods, which need to be (potentially) changed in the client if a server changes.

The CDBC value is between 0 and the count of methods in the class. We compute the average CDBC value of each (client) class over all (server) classes it is directly connected

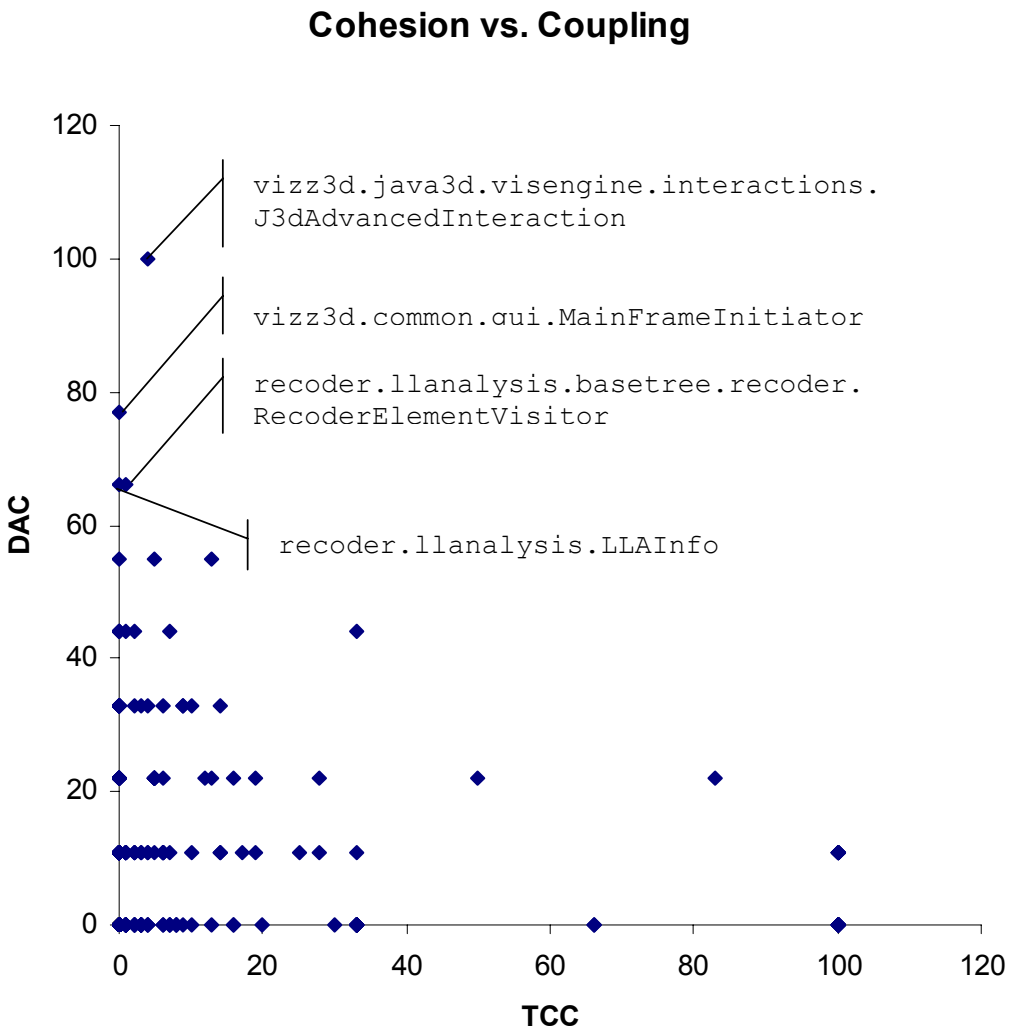

Figure 7.6: Relative intra-class-cohesion (TCC) and relative inter-class-coupling (DAC).

with. The scale is rational.

Lack of Documentation (LOD) measures the amount of undocumented declarations per class (counted are the class declaration itself and the method declarations, but not field declarations). Only ”JavaDoc” style documentation is taken into account. Documentation within methods is ignored. Only the syntax of the comments is parsed, not the semantics.

The LOD value is calculated for each class or interface as the number of undocumented

declarations (local not inherited methods). A LOD of 0 indicates that all possible entities are documented; a higher value indicates the lack of documentation.

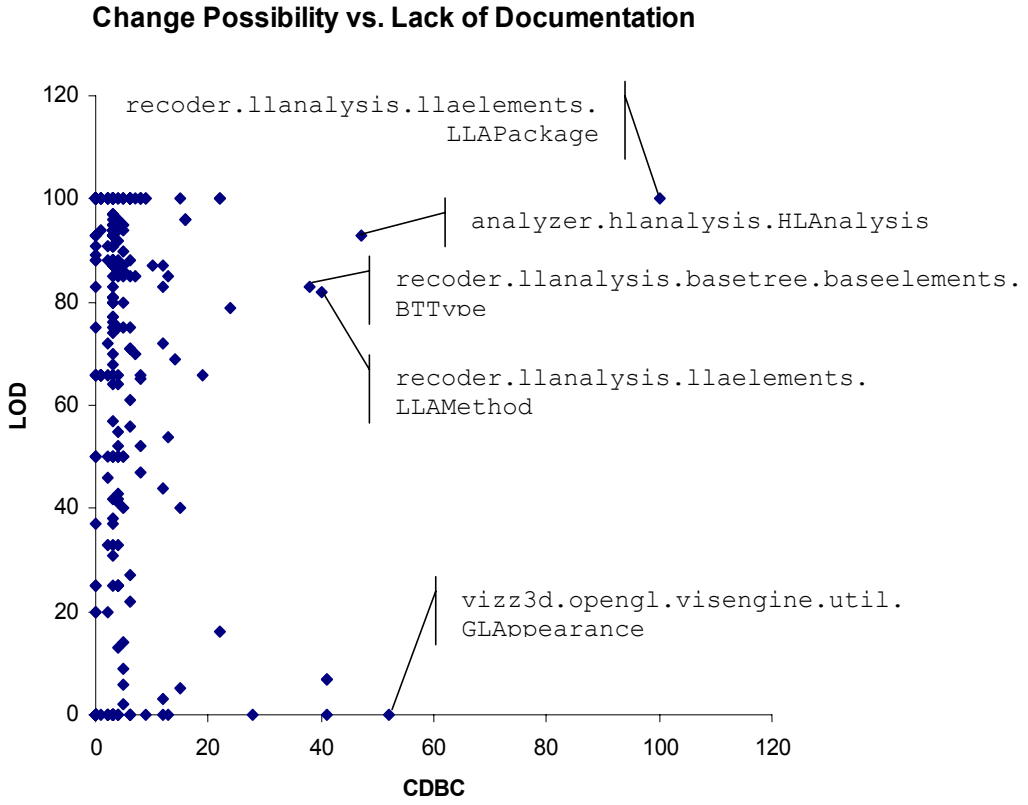

Figure 7.7: Relative possibility of changes of a class (CDBC) and relative lack of documentation of classes and methods (LOD).

Chart 7.7 depicts the values of CDBC and LOD for classes and interfaces of the VizzAnalyzer. Each class/interface is a point in the diagram; its x-position indicates the class'/interface's CDBC value, its y-position indicates the class'/interface's LOD value. Both values are linearly scaled between 0-100, where the highest absolute values measured in each dimension are mapped to 100.

Since, low possibility of changing a class/interface and low lack of documentation are desired, class/interface in the top-right corner are to review. These classes are likely to be changed and are poorly documented at the same time. Class/interface in the bottom-right

corner might be critical, too. These classes are likely to be changed, too, but at least well documented.

Interpretation: The following classes show in general a high external coupling and low internal cohesion at the same time:

1. vizz3d.java3d.visengine.interactions.J3dAdvancedInteraction
2. vizz3d.common.gui.MainFrameInitiator
3. recoder.llanalysis.basetree.recoder.RecoderElementVisitor
4. recoder.llanalysis.LLAInfo

The following classes show in general a high change possibility and lack of documentation at the same time:

1. recoder.llanalysis.llaelements.LLAPackage
2. analyzer.hlanalysis.HLAnalysis
3. recoder.llanalysis.basetree.baseelements.BTType
4. recoder.llanalysis.llaelements.LLAMethod

The class vizz3d.opengl.visengine.util.GLAppearance has a high change possibility but appears to be well documented. These classes in the packages recoder.llanalysis and vizz3d might be considered worth being reengineered.

7.5 Complexity Metrics

The complexity of a class is often considered interesting since it points at classes that are hard to understand and to maintain.

Weighted Method Count (WMC) computes the complexity of the methods of a class and is also a measure for the complexity of a class. It gives a good idea about how much effort is required to develop and maintain a class. The methods are weighted according to McCabe's Cyclomatic Complexity Metric. It counts the possible execution branches in a method for the branching statements: if, for, while, do. It is assumed that each branch has the same complexity/weight. We scaled the values between 0 and 100.

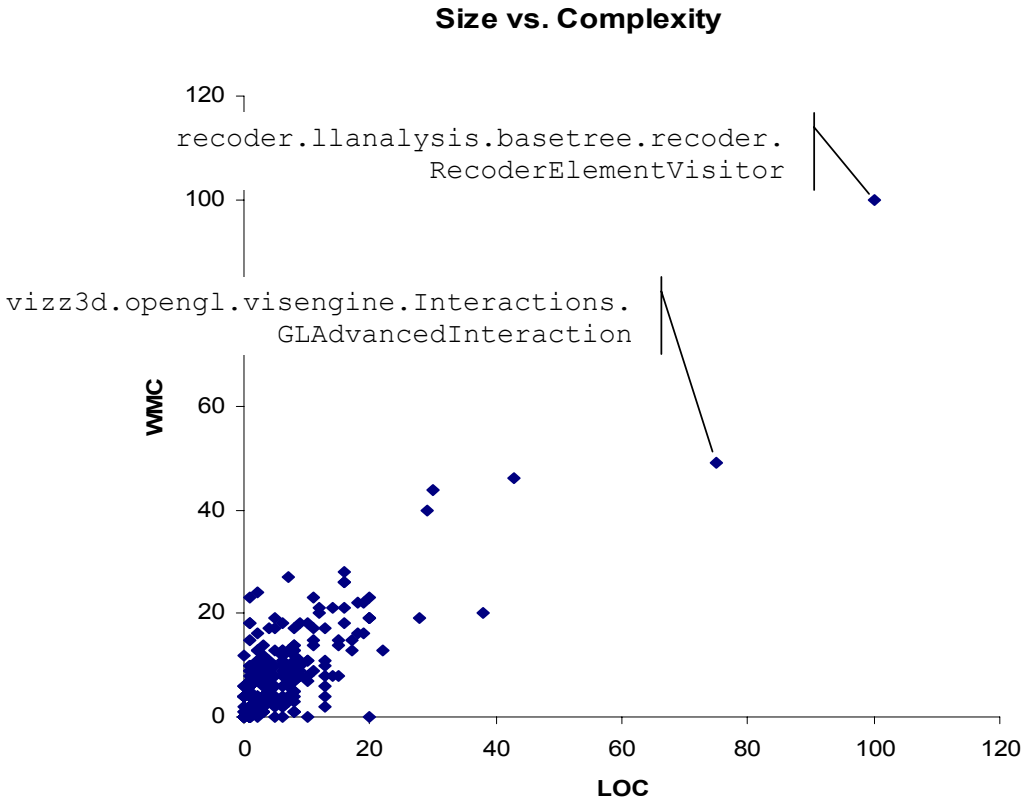

Figure 7.8: Relative size in lines of code (LOC) and relative complexity in weighted method complexity (WMC).

Lines of Code (LOC) count the lines of code in a class and interface, respectively. It's an absolute metric that we scaled between 0 and 100. Classes that are both highly complex and large are critical when it comes to understanding and maintaining a system. It is especially alerting if they are recommended to be restructured because of structural and/or design issues (see previous sections).

Chart 7.8 depicts the values of LOC and WMC for classes and interfaces of the VizzAnalyzer. Each class/interface is a point in the diagram; its x-position indicates the class'/interface's LOC value, its y-position indicates the class'/interface's WMC value. Both values are linearly scaled between 0-100, where the highest absolute values measured in each dimension are mapped to 100. Since, high complexity of class/interface and large classes are critical, class/interface in the top-right corner are to review. These classes are hard to be changed.

Interpretation: The following classes are both complex and large:

1. recoder.llanalysis.basetree.recoder.RecoderElementVisitor
2. vizz3d.opengl.visengine.Interactions.GLAdvancedInteraction

These classes might be considered splitting. Special attention should be paid to the class recoder.llanalysis.basetree.recoder.RecoderElementVisitor. This class was recommended to re-engineer because of its high external coupling (DAC) and low internal cohesion (TCC).

7.6 Conclusions

This report documented the results of the First Contact Assessment of VIZZANALYZER FRAMEWORK. Our assessment included and architecture analysis in general, a comparison between intended and actual architecture, an analysis of the inheritance structure, analyses of coupling and cohesion on class level, and analyses of the understandability and maintainability of the software.

We separated analyses from interpretation. All analyses results are factual. However, we need to emphasize that their interpretation should be taken with care. It can only point to suspicious spots of the system. Developers and designers experienced with the system should crosscheck each interpretation. None of the results revealed a severe problem. We conclude that the current design and implementation of the software provides a solid foundation for future development cycles.

It is recommended to integrate a quality assessment in this future development. The group would be like to give support in that venture with both tools and expert knowledge.

Bibliography

[Ahl05] H. Ahlgren. Graph Visualization with OpenGL. Master's thesis, Department of Computer Science, Växjö University, 2005.

[Bea99] H. Bär and et. al. The FAMOOS Object-Oriented Reengineering Handbook. Technical report, Forschungszentrum Informatik, Karlsruhe, Software Composition Group, Uni. of Berne, ESPRIT Project 21975, 1999.

[BKZ05] J. Bevan, S. Kim, and L. Zou. Kenyon: A common software stratigraphy system. `http://www.soe.ucsc.edu/research/labs/grase/kenyon/`, 2005.

[BL03] D. Beyer and C. Lewerentz. CrocoPat: Efficient Pattern Analysis in Object-Oriented Programs. In *11th International Workshop on Reverse Engineering,Portland,USA*, May 2003.

[gl405] gl4java home page. `http://gl4java.sourceforge.net/`, 2005.

[HQ05] M. Hjalmarson and K.-M. Quach. The VizzAnalyzer Metaphor Editor. Master's thesis, Department of Computer Science, Växjö University, 2005.

[jav05] Java3d home page. `http://java3d.dev.java.net/`, 2005.

[jog05] jogl home page. `http://jogl.dev.java.net/`, 2005.

[LNAH01] A. Ludwig, R. Neumann, U. Aßmann, and D. Heuzeroth. Recoder homepage. `http://recoder.sf.net`, 2001.

[LP05] W. Löwe and T. Panas. Rapid Construction of Software Comprehension Tools. *IJSEKE*, August 2005.

[Mah05] S. Maheshwaran. Java - scripting. `http://www.thejavahub.com/Articles/article2.html`, 2005.

[Mar05] R. C. Martin. Java3D Interaction: The interactive VizzAnalyzer. Master's thesis, Department of Computer Science, Växjö University, 2005.

[ope05] OpenGL at SGI. `http://www.sgi.com/products/software/opengl/`, 2005.

[Pan03] T. Panas. *Towards a Generic Framework for Reverse Engineering*. Licentiate thesis, Växjö University, Sweden, November 2003.

[PLL04a] T. Panas, J. Lundberg, and W. Löwe. Reuse in Reverse Engineering. In *Int. Workshop on Program Comprehension, Bari, Italy*, June 2004.

[PLL04b] T. Panas, J. Lundberg, and W. Löwe. Reuse in Reverse Engineering. In *12th International Workshop on Reverse Engineering,Bari,Italy*, June 2004.

[PLL05] T. Panas, R. Lincke, and W. Löwe. Online-Configuration of Software Visualizations with Vizz3D. In *Proc. of ACM Software Visualization Conference, St. Louis, USA*, 2005.

[PLLL05] T. Panas, R. Lincke, J. Lundberg, and W. Löwe. A Qualitative Evaluation of a Software Development and Re-Engineering Project. In *IEEE/NASA SEW-29*, April 2005.

[PS05] T. Panas and S. Staron. Evaluation of a Framework for Reverse Engineering Tool Construction. In *ICSM 2005, Budapest, Hungary*, September 2005.

[wil04] Wilmascope 3D. `http://www.wilmascope.org/`, 2004.

[yed04] yWorks. `http://www.yworks.com/en/products_yed_about.htm/`, 2004.