Master's Thesis

J3DVN - A Generic Framework for 3D Software Visualization

carried out at the

Information Systems Institute
Distributed Systems Group
Vienna University of Technology

under the guidance of
Univ.Prof. Dipl.-Ing. Dr. techn. Harald Gall
and
Dipl.-Ing. Dr. techn. Jacek Ratzinger
as the contributing advisor responsible

by

Florian Breier
Barichgasse 6/8, 1030 Vienna
Matr.Nr. 9526715

Vienna, 26. March 2008

Acknowledgements

I wish to thank Prof. Harald Gall for giving me the opportunity to carry out this thesis under his guidance.

It gives me immense pleasure to place on record my sense of deep indebtedness to my supervisor Dr. Jacek Ratzinger. He acted as a lighthouse to steer my project in the right direction.

I thank Michael Fischer for giving me initial input for this project.

Last but not least I want to thank my family for their enduring support.

Zusammenfassung

Die Analyse von Software Architekturen und Software Evolution erzeugt Daten. Visualisierung macht solche Daten verstndlicher. Die Verwendung von drei Dimensionen in solchen Visualisierungen erhöht die mögliche Anzahl gleichzeitig darstellbarer Metriken. Existierende Tools für eine solche Aufgabe sind für gewöhnlich auf einzelne Probleme spezialisiert, lassen sich dann aber nicht verwenden, wenn eine andere Aufgabe damit gelöst werden soll. Wir stellen ein Modell vor, das die Erzeugung dreidimensionaler Datenvisualisierung erleichtert. Das Modell ist erweiterbar und stellt somit eine generische Lösung für viele verschiedene Probleme dar. Es ist insofern flexibel, als dass Repräsentationen der Daten in Echtzeit geändert werden können, womit es auf der Stelle möglich wird, verschiedene Repräsentationen/Aspekte derselben Datenbasis zu erkennen. Wir haben ein Visualisierungs-Framework implementiert, welches unsere Bedürfnisse erfüllt, und wir haben es verwendet, um unser Modell zu validieren, indem wir sinnvolle Kombinationen von Metriken visualisiert haben.

Abstract

The analysis of software architecture and evolution generates data. Visualization has the task of making such data more comprehensible. The use of three dimensions in such visualizations increases the possible number of concurrently shown metrics. Common tools to do so are usually specialized for single problems but lack the ability to conform when a different task has to be fulfilled. We propose a model, which facilitates the creation of three dimensional data visualization. The model is extendable to make it a generic solution for many different problems. It is flexible in a way that data representation can be changed on-the-fly, making it possible to immediately see different representations/aspects of the same database. We implement a visualization framework that fulfills our needs and we use it to validate our model by visualizing useful combinations of metrics.

Contents

1 Introduction **1**
1.1 Problem Domain 1
1.2 Contributions 2
1.3 Organization of this thesis 2

2 Software Visualization in a Nutshell **4**
2.1 Steps in visualizing software 4
2.2 Perception 5
2.3 Graph Drawing 6
2.4 Visualization of Software Architectures 7
2.4.1 UML 8
2.4.2 Software Metrics 8
2.5 Visualization of Dynamic Program Execution 8
2.5.1 Algorithm Animation 8
2.5.2 Visual Debugging 9

3 Related work **10**
3.1 Rigi 10
3.2 SHriMP 11
3.3 SeeSoft 11
3.4 SeeSys 12
3.5 G^{SEE} 12
3.6 The Visual Code Navigator 13
3.7 sv3D 13
3.8 White Coats 14
3.9 EvoLens 14
3.10 Mondrian 15

4 Approach **17**
4.1 MVC Architecture 17
4.2 Addons 17
4.3 Eclipse 18
4.4 Java 3D 20

5 Model and Framework 22
5.1 Addons . . . 22
5.1.1 Data Addons . . . 22
5.1.2 Visual Addons . . . 22
5.1.3 Conversion Addons . . . 23
5.1.4 Input Addons . . . 23
5.1.5 Layout Addons . . . 23
5.2 Connectors . . . 23
5.3 The Data Model . . . 24
5.4 Architecture . . . 25
5.5 Implementation . . . 27
5.5.1 General Structure . . . 27
5.5.2 Project j3dvn . . . 27
5.5.3 Project j3dvneclipse . . . 36
5.5.4 Project addoncollection . . . 37

6 Case Studies 39
6.1 Foundation . . . 39
6.1.1 Versioning System . . . 39
6.1.2 Evolution Metrics . . . 41
6.2 Methodology . . . 42
6.2.1 Data Extraction into Database . . . 43
6.2.2 Computation of Logical Coupling . . . 43
6.2.3 Identifying Refactorings and Bugs . . . 44
6.2.4 Time periods for Analysis . . . 45
6.2.5 Filtering of Data . . . 46
6.2.6 Visualization Approach . . . 47
6.2.7 Implementation . . . 47
6.3 ArgoUML . . . 48
6.4 Azureus . . . 50
6.5 Results . . . 57
6.5.1 Comparison of related work . . . 59

7 Conclusion 64

A Schema of input file 66

B How to contribute to the framework 74

Bibliography 82

List of Figures

4.1 Overview of the Eclipse architecture 19
4.2 Example of extensions and extension points of Eclipse plug-ins . . . 19
4.3 Example of a scene graph in Java 3D 20

5.1 Example model of a file system 24
5.2 Framework architecture 26
5.3 Structure and dependency of implemented projects 28
5.4 Relationship between addon manager, connectors, and addons . . . 31
5.5 Activity diagram of the `createPartControl()` method of `Graph3DEditor` 35

6.1 Time periods for analysis 43
6.2 ArgoUML, min bugs=4, min coupling=6, period = 07.2003 - 12.2003 48
6.3 ArgoUML, min bugs=4, min coupling=6, period = 01.2004 - 06.2004 49
6.4 ArgoUML, min bugs=4, min coupling=6, period = 07.2004 - 12.2004 50
6.5 ArgoUML, min bugs=3, min coupling=4, period = 07.2004 - 09.2004 51
6.6 ArgoUML, min bugs=3, min coupling=4, period = 10.2004 - 12.2004 52
6.7 ArgoUML, min bugs=4, min coupling=6, period = 01.2005 - 06.2005 53
6.8 ArgoUML, min bugs=4, min coupling=6, period = 07.2005 - 12.2005 54
6.9 Azureus, min bugs=5, min coupling=5, period = 07.2003 - 12.2003 55
6.10 Azureus, min bugs=5, min coupling=5, period = 01.2004 - 06.2004 56
6.11 Azureus, min bugs=5, min coupling=5, period = 07.2004 - 12.2004 57
6.12 Azureus, min bugs=5, min coupling=5, period = 01.2005 - 06.2005 58
6.13 Azureus, min bugs=5, min coupling=5, period = 07.2005 - 12.2005 59
6.14 Azureus, min bugs=5, min coupling=5, period = 01.2006 - 06.2006 60
6.15 Azureus, min bugs=5, min coupling=5, period = 07.2006 - 12.2006 61

List of Tables

5.1 Pre-implemented properties of the internal `VisualConnector` implementation . . . 30
5.2 Parameter types and return types of the `createConnector()` method of the `AddonManager` class . . . 31
5.3 Parameters of the `propertyChanged()` method of the `PropertyChangeListener` interface . . . 33
5.4 Properties of the `LinLog` addon . . . 36
5.5 Properties of the visual addons from the addoncollection project . . 37

6.1 Mapping from metrics to visual attributes . . . 44
6.2 Properties of the `Evolizer` addon . . . 45
6.3 Properties of the `EvoClass` addon . . . 46
6.4 Property of the `Coupling` addon . . . 47
6.5 Comparison with related work . . . 63

B.1 Properties of the visual addons from the addoncollection project . . 74

Chapter 1

Introduction

The big part of a software project does not end with the release of the first version of the software, but it only starts with it. Bug fixes are applied and new features are added afterwards. To improve maintainability, code will be refactored. In this process, the internal behavior of a software system is changed while its external behavior remains the same. The bigger a software system becomes the more difficult it is to be understood - obviously. Many attempts to visualize the architecture of software systems have been successfully. Very common are UML class diagrams [Obj08], for example.
Software evolution is a continuous change from a lesser, simpler, or worse state to a higher or better state of a software system [Art88]. It happens from day one of a software project until the end of the software's lifetime. Analyzing this evolution is necessary to identify problematic spots in the design of a software system. When speaking of "analysis" you only have to go a small step further to reach the term "visualization" - it's common knowledge that "a picture is worth a thousand words."

1.1 Problem Domain

As computers constantly become more powerful, approaches of 3D visualizations become more practical. Adding a third dimension to visualization makes it possible to visualize one additional aspect of the underlying data. If coordinates are not used to exhibit specific aspects of data, 3D can still be a benefit. A 3D picture might just be better understood than a 2D one.
Empirical studies that compare the effectiveness of two- and three-dimensional visualizations are mixed.
Wiss and Carr [WC99] gave a negative evaluation. They state that the main problems are lack of overview and lack of custom navigation.
Balzer et al. [BNDL04] made the experience with 3D layouts of large graphs that orientation is sometimes intricate and individual objects are often occluded and therefore barely recognizable.

Ware and Frank [WF94] evaluated that the information in 3D is more easily understood by the users than in 2D. Additionally, the error rate in identifying routes in 3D graphs is much smaller than in 2D graphs [WHF93].
Tavanti and Lind [TL01] recognized that 3D displays support spatial memory better than 2D. Though in their experiments, the 3D display was a 2.5D display, to be precise.
There is no optimal visualization design for every task [WCJ98]. There are always advantages and disadvantages, making a trade-off necessary. That's why a visualization framework makes sense, where different visualizations can easily be realized.
We try to facilitate the building process of a mental model for the understanding of the evolution of software systems. Our focus lies on 3D visualization of software architecture and software evolution. Nevertheless, it is in no way restricted to these two domains.

1.2 Contributions

The following contributions are made by this thesis:

- A data model, which is not dependent on the type of data, thus making it usable for many different problem domains.
- The J3DVN framework, realized as an Eclipse plug-in, which implements this model.
- Additional addons for J3DVN, which extend the framework and virtually build a new tool on their own.
- Evaluation of the model by using the framework plus the supplemented addons to analyze two popular open source software projects.

1.3 Organization of this thesis

The structure of this thesis is as follows:
Chapter 2 gives a brief introduction into the area of software visualization. It explains aspects and basics to take care of when visualizing software. Moreover, it describes some popular approaches to software visualization.
Chapter 3 goes a bit more into detail by giving an overview of the state of the art. Assorted projects are presented, which are either similar in purpose and/or influence our work up to a certain amount.
Chapter 4 initially gives quick descriptions of the Eclipse platform and of Java 3D, two essential technologies for our framework. Later on we explain the addon architecture of our framework.

Chapter 5 goes into detail with the framework. We list the different basic addon types and explain why they are there and what they are doing. Every addon has to be connected to a connector. Why and how this has to be done is explained in more detail in this chapter as well. Afterwards, we show our generic, graph based data model. The chapter concludes with a description of the architecture of our framework. After a short overview of the general structure of the parts of our framework the projects, packages, and classes are characterized and explained.
Chapter 6 validates the model and the framework. Case studies with two open source software projects are made, in which we analyze evolution of the two pieces of software. First of all, though, we announce what kind of data we retrieve for examining software evolution. And second of all we explain what we do with the input data, which we retrieve, and how we use it. We also explain the implementation of the addons, which we created to realize the retrieval and calculation of the necessary evolution information. We finish this chapter with the presentation of the results we could achieve through our case studies.
The thesis ends with Chapter 7, where we summarize our research results and denote points for future work.

Chapter 2

Software Visualization in a Nutshell

Software visualization can be defined as "the visualization of artifacts related to software and its development process" [Die07]. Artifacts need not necessarily be only program code but also requirements and design documents, bug reports, data in memory at a certain time in program execution, etc. We can distinguish between three different aspects of software visualization:

Behavior: Execution of a program, possibly in real time, dynamic state of memory during execution.

Evolution: The process of developing a software program.

Structure: Architecture of a software program, its data structures, a static call graph.

In our work, we will focus on the area of evolution visualization. Nevertheless, other areas are not excluded from our framework. In fact, it all depends on the data gathered. The creation of visualization itself is only one step in the process of visualizing software.
In this chapter, we will explain the general process of visualizing software. We will discuss factors of human perception of visualizations in addition to graph drawing. We look at the aspects *Behavior* and *Structure* visualization. Visualization of *Evolution* is left out in this chapter, since Chapter 3 is totally dedicated to examples of that domain.

2.1 Steps in visualizing software

First of all, data is retrieved. Data can come from various sources, like, for example, a version control system, where all the assets of a software program are stored in every existing version, or it can come from a debugging tool, which observes the state of a program during its execution.

The next step is the analysis of collected data. Usually, there will be too much information to visualize. Therefore, information has to be filtered, classified, or reduced in any other way. Possibly but not necessarily, this step also includes the conversion of existing data into an intermediate data format, which can be used by the visualization tool in the next step.
This next step is basically the "creation of the picture." The previously prepared data is mapped onto a visual model and then rendered to screen. Of course the visualization may not to be a picture only. Our framework, for example, creates a 3D visualization, where users can navigate and interact.
Finally, there is the human perception of the visualization. While we can't really make adjustments on the human eye, we could give aids like, for example, shutter glasses. Over time users will improve in perceiving visualizations, which are new to them. That's why new visualization techniques often might not look promising in the first place but become valuable over time.

2.2 Perception

Any visualization is only as good as the perception is. Perception, in general, involves the use of all our senses. However, information from the real world is mostly perceived visually.
Color is the human perception of light. The hue of a color is related to its dominant wavelength, whereas brightness is related to the intensity or amplitude of the wave. There are two kinds of receptors in the human eye: rods and cones. While rods are much more sensitive at low light levels, during daytime vision only cones are in use, because they are sensitive under normal working light levels. A rod can only measure the intensity of light of the full spectrum, thus with rods alone, we could only see black-and-white. Cones, on the other hand, are sensitive towards light of different wavelengths. There are three types of cones: one for magenta, one for green, and one for yellow-to-red.
Rods and cones are not distributed evenly on the back of the eye (called retina). In the center of the retina there is a small area called fovea, which is densely packed only with cones. That's why we can see sharpest in the center of our field of vision. To be precise, only the central two degrees[1] are seen by the fovea.
Pattern perception is the process of recognizing objects. This happens by deciding, which visual elements belong to each other. Pattern perception happens mostly as 2D processes. Gestalt theory [Kof35] knows a set of Gestalt laws of pattern perception. According to them, humans use the following criteria (among others) to perceive patterns:

- Connectedness
- Proximity

[1]As a rule of thumb we can say that one degree is equivalent to the width of one's thumbnail at arm's length.

- Similarity of color, shape, size, or brightness
- Continuity of curves
- Symmetry of objects
- Closure of areas (p.e. we see a round dotted line as a closed circle)

Motion perception is the process of recognizing, which visual elements perform the same movement in consecutive images. A well-known case of wrong motion perception can be seen in Western movies, where wheels of a coach are supposedly turned in the wrong direction.
Visualization of software can be done as one-, two-, or three-dimensional representations[2]. One-dimensional visualizations can be markers on a scale or a gauge, for example. Very often software is visualized as (2D or 3D) graph based representation, since it tries to visualize elements of a software system and their relation between each other. Visualizations are built from visual elements like points, lines, areas, and volumes. These primitive elements differ in length, width, height, position, orientation, color, transparency, texture, and shape. Besides those properties they can have dynamic properties, which change over time, starting from simple blinking or changing color or position to morphing from one shape into another one.
Metaphors are widely used in the domain of information visualization. A very well-known metaphor is the desktop, which can be found in most modern operating systems.
When drawing graphs the solar system metaphor can be found sometimes. There, nodes of a graph are displayed as planets of a solar system. Most important "planets" can be found in the center, while less important information is drawn further away. (This idea is also used in the fisheye view, see 3.2). The solar system metaphor (for example [GYB04]) can be extended to the galaxy metaphor to visualize information clusters in graphs (see next section), where each cluster is a solar system in the galaxy.
Information visualized as a landscape [BNDL04] is similar to our physical environment. The surface is more or less two-dimensional only, but contains three-dimensional trees, mountains, buildings etc. The city metaphor [WL07] can be used for software visualization. Wettel and Lanza argue, that the advantage of this metaphor is increased comprehensibility and higher ability to navigate through data because of "habitability."

2.3 Graph Drawing

Graphs play an important role in software visualization. Syntax trees, finite state diagrams, and control flow diagrams are all graphs. Purchase et al. [PCJ96] defined various criteria to aesthetically draw graphs:

[2]For simplicity reasons, in this thesis we are not considering text-only representations (e.g. pretty printing or literate programming) as software visualizations.

Crossing minimization: A graph should be drawn with as few crossings as possible. If the graph is planar, it shouldn't contain any crossing edges.

Bend minimization: Edges should have as few bends as possible.

Area minimization: The total area of the graph should be small and its nodes should be distributed evenly.

Length minimization: The total length of all the edges should be as short as possible.

Angle maximization: Angles between edges of a node and angles of bends of an edge should be maximized.

Symmetries: Symmetries in the underlying data should be drawn in the graph.

Clustering: Parts of graphs, which are strongly interconnected, should be separated from other such parts. In such a case, previous criteria like area minimization or length minimization cannot be applied thoroughly. Clustering can happen in larger graphs.

There are many different graph drawing algorithms, which serve different purposes. In an orthogonal graph layout the edges run horizontally and vertically and the edges bends have angles of 90 degrees only. Usually, the goal of orthogonal graphs is to minimize edge crossings and bends. In a force-directed layout the graph is seen as a physical system, for example a planetary system. The nodes are seen as planets with certain gravity, while at the same time they have a repulsion that keeps them away from each other. The idea is basically to minimize the total energy (repulsion and attraction) of the system.

2.4 Visualization of Software Architectures

A software system can be described in many ways and many points of views, and many aspects can be considered, like: gross organization and global control structure, protocols for communication, synchronization, data access, assignment of functionality to design elements, physical distribution, composition of design elements, scaling and performance, and selection among design alternatives [GS94]. Descriptions can be in textual ways. Sometimes this is just not enough for a descent understanding of the architecture. Then, diagrams might help. Dividing a software system into smaller parts helps making it easier to understand the whole, and thus to design and develop such a system.

2.4.1 UML

The Unified Modeling Language [Obj08] is an object-oriented approach to describe software architecture graphically. There are different diagrams for different areas, for example class diagrams, use case diagrams, sequence diagrams, collaboration diagrams, state chart diagrams, activity diagrams, and many more. UML has become a standard for a notation of software architecture. Every UML diagram is a graph built from simple visual primitives, so it is easy for a designer to construct a diagram even by hand. It is two-dimensional only. It can be extended to three dimensions [GRR99]. The advantages of 3D-UML - together with animation even - should be improved comprehension of complex diagrams.

2.4.2 Software Metrics

A software quality metric is defined as *a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality* [IEE98].
Many metrics can be found for assessing the quality of software and its development process, like lines of code, cyclomatic complexity, number of bugs, number of classes and interfaces, cohesion, coupling, number of authors, and number of refactorings. Although, the use of metrics alone will certainly not eliminate the need for human decisions in software assessments, it can be of great help to do so.
Visualizing metrics will usually increase understandability of metrics[3]. Showing metrics (e.g. lines of code) on scales will already help to get a better idea of software assets, compared to only writing the plain numbers in a table. Nevertheless, it only starts to get interesting when combining two or more metrics in one single visualization. In this way, relations between different metrics (for example lines of code and number of bugs) can become obvious.

2.5 Visualization of Dynamic Program Execution

When visualizing execution of a program a continuous data flow or a sequence of snapshots of the system has to be visualized. This can happen in real time or after program execution has been recorded. In either case at least data acquisition has to happen during actual runtime.

2.5.1 Algorithm Animation

Algorithm animation visualizes the behavior of an algorithm. The execution of an algorithm leads to a sequence of states, where each step in the algorithm results in a change from one state to another state[4]. Algorithm animation maps every state to a

[3]Bad visualizations can of course also help in misleading the viewer of the visualization.
[4]Or from one state to the same state again, if we have an infinity loop.

separate image and shows the transitions of the images (i.e. states) as an animation. Algorithm animation is often used as educational aid to help understanding the way an algorithm works. A famous example is the video *Sorting out Sorting* [Bae81], which used animation of program data coupled with an explanatory narrative to teach nine different sorting algorithms.

2.5.2 Visual Debugging

Finding bugs in a program can be a painful task. By using debuggers it can become much easier to understand what happens during program execution at a certain point. Usual debuggers allow - frankly spoken - to execute a program step by step, one line at a time and inspect the value of existing variables. Visual debuggers not only allow to see textual information of the data value of a variable, but also represent information in a graphical way. Data Display Debugger (DDD) [Fre08] is a popular visual debugger that allows interactive graphical data display, where data structures are displayed as graphs. Thus, we can see, how memory consumption of a data structure becomes bigger over time.

Chapter 3

Related work

A great deal of research has been done on Software Evolution Visualization, and particularly, on visualizing single versions of software architectures. Some tools, which are explained here, have their origin in visualizing single versions of software architecture, but can also be used for evolution of software. A wide range of tools is limited to 2D. We have also mentioned such work in this chapter as they often serve as a basis for 3D approaches.

3.1 Rigi

Rigi [MK88] is a rather old, but popular approach. It uses a graph model and abstraction mechanisms to structure and represent the information accumulated during the development process. It creates simple box-and-line diagrams. However, it provides extensive navigation facilities that allow the user to create different views of the systems. In the graph model, a node is a component of the system (e.g. subsystem, interface, variant, revision, specification, data set, or picture), while the arcs represent relations between nodes (e.g. change, compilation, binding, revision, or aggregation)
It also supports a programming language, RCL, to define different problem domains (e.g. different programming languages), which both allows it to be tailored to different scenarios. In the current version, it even provides Shrimp views (see Section 3.2) of nodes.
One goal of Rigi was to give aid in version and release control. This is possible by visualizing revisions of subsystems, classes, and so forth.
One disadvantage of the traditional approach is the multitude of windows. To look further into one node, that node is opened in a new window, where the sub-nodes are displayed in a new graph model. To navigate to a specific class in a package, possibly nested, it will be necessary to open several windows from one package to its sub package and so on. This technique not only clutters the screen, but also lets the user lose context.

3.2 SHriMP

The SHriMP (**S**imple **Hi**e**r**archi**c**al **M**ulti-**P**erspective views) project by Storey and Müller [SM95] is built up on Rigi, but tries to compensate its drawbacks by implementing new concepts. Its key goal is to present large amounts of information on a computer screen. According to [Tuf90] it depends on how the information is visualized rather than how much of the information is visualized. Storey and Müller take that into account and use fisheye views of nested graphs to visualize Software Architectures.
In section 3.1, we can see in detail the problem Rigi faced. As mentioned earlier, the display of only a part of the graph in one window is recognized. Though the whole plain graph is displayed in a window, it is very difficult to see the minute details. To overcome these obstacles, SHriMP uses fisheye views [SB92]. In a fisheye view of a graph, only the area of interest is displayed in a relatively large and detailed manner, while the remainder of the graph is successively smaller and less detailed. In addition to the fisheye views, SHriMP uses nested graphs [Har88], which display sub graphs of a node inside that node. The fisheye view makes it possible to prevent the details (i.e. the sub graphs inside of a node) from getting too small. By default, in SHriMP sub graphs are closed, i.e., they are not visible. By double-clicking on a (non-leaf) node, its sub graph is opened and displayed in the node.
As Rigi's work has been instrumental for SHriMP, both can be used in the same domain. SHriMP is developed extensively. There even exists a plug-in for Eclipse [Com08], which integrates SHriMP into Eclipse. This allowes you to use all SHriMP views for any Java project in Eclipse.

3.3 SeeSoft

Another popular approach for visualizing Software Evolution is SeeSoft [ESS92] where the visualization is line-oriented. Whole files are displayed as columns, while the separate lines of each file are rows of one such column. The width of one row and its indentation conform to the width and indentation of the actual source code line in the file. The color of each line is mapped to a certain statistic. The statistic might be age, programmer, feature, type of line, number of times the line was executed in a recent test, modification request to a version control system, etc. A color bar containing the whole color spectrum is always visible on screen. This permits the activation of only one single statistic value. In this way, it is possible to have a display of only the changes in one single modification request over all the files.
However, it is not possible to display more than 50.000 lines of code[1] at once. It only visualizes the code lines and not any of the other information, such as design or architectural information.

[1]On a 1284x1024 resolution display

3.4 SeeSys

SeeSys [BE95] tries to eliminate the problems of SeeSoft. It uses as much space of the screen as possible. Visualization does not occur by using single rows, but rather, one big area which is then partitioned into separate rectangles. Each rectangle represents one subsystem. The size of such a rectangle is based on some metric, for example, lines of code.
Its functionality addresses project managers, feature engineers, and software developers. According to the authors, SeeSys applied to a system can be used to

- show the sizes of the subsystems and directories, and where the recent development activity has been,
- zoom in on particularly active subsystems,
- discover how much of the development activity involves bug fixes and new functionality,
- identify directories and subsystems with high fix-on-fix rates, and
- locate the historically active subsystems and find subsystems that have shrunk and even disappeared.

Since the total area is always as big as possible (and only limited by the size of the screen), and the size of the rectangles is a relative value (to always fill up the whole area), SeeSys, unlike SeeSoft, is not limited to a certain total size of a system.

3.5 G^{SEE}

Favre developed G^{SEE} [Fav01], a framework to build tools from generic components. He argues that most of the work in software exploration can be qualified as specific approach, where a specific tool is provided for each specific perspective. Especially in the context of large software companies, this approach is weak because only a few software models are covered by specific tools and the cost of building a new specific tool is expensive.
G^{SEE} consists of an object-oriented framework, a set of customizable tools, and a tool builder. The framework approach brings maximum flexibility and extensibility, while making almost no assumption about the way it will be used. A set of customizable tools is delivered with the framework, which illustrates the use of the framework and helps to solve common problems without the need to start programming. The tool builder can generate specific exploration tools by interactively assembling components.
The framework contains source components and visualization components, which one could qualify as Model components and View components, when speaking of an MVC concept. Many visualization components are available, e.g. for drawing

organization charts, tree maps like SeeSys (Section 3.4) uses, or line sequences as in SeeSoft (Section 3.3).

3.6 The Visual Code Navigator

The Visual Code Navigator [LNVT05] is a set of tools that enables the exploration of large software projects from different perspectives, or views:

- *The syntactic view* shows the constructs in the source code. For every construct, a cushion is displayed, which outlines the construct's text in the source code. In this way, a source file is displayed as a bunch of different colored areas, where each one is a single construct block.
- *The symbol view* shows all the symbols that are created by a compiler, i.e. the symbols which a linker would see after compilation. Since the Visual Code Navigator is C/C++-centric, displayed symbols are global scope objects, like function signatures, class and namespace method and data members, templates, enumerations, typedefs, and global variables. These symbols are displayed as cushions in a tree map.
- *The evolution view* displays the change in source code files in a project's lifetime. Like the syntactic view, it uses a file's layout, i.e., only one single file is displayed. It uses the concept of bi-level code display, which gives a view of both the contents of a code fragment and its evolution in time. In this 2D pixel-filling display, the x axis maps the version number and the y axis maps the line number. Thus, a file is displayed over the whole area, where the very left column is the first version of the file and the very right column is the latest version.

3.7 sv3D

sv3D [MFM03] extends the Seesoft metaphor (Section 3.3) to 3D space. Since it extends this metaphor, it can be used for any problem, for which Seesoft can be used.
Seesoft displays files as columns, lines of code in the files as lines in the columns, positions of these lines of code as the position of the line, and a selectable attribute as color of the lines. Instead of painting one column per file, sv3D uses a container, i.e. a bordered plane (simply a rectangle) in 3D space, which is not even visible. Instead of lines of a column, poly cylinders in a container are used. The position (on x- and y-axis of the container) of the line in the file is mapped to the position of the poly cylinder in the container. Like the 2D model that uses color of a line to display a certain attribute, color can also be used here. There are, however, still three more attributes that can be mapped to other visualization elements, namely

to height, depth and shape of a poly cylinder. So, by extending the visualization to 3D space, additional information can be displayed simultaneously.

3.8 White Coats

White Coats [ML05] visualizes CVS [BF03] repositories in 3D using VRML [Int97]. It provides access to versioning information by using a web interface. As a result it is easily accessible on very many platforms. Besides 3D (VRML) data, additional textual information is displayed. The visualization happens through polymetric views [LD03] in 3D.
Visual entities are blocks of specified height, width, length, and position (x-, y-, and z-coordinate) in 3D space. Additionally, texture and transparency of the visual entities can be used. Up to eight metrics could be mapped to one single entity.
Some interesting ideas are realized, which help in understanding and navigating the visualization:

- *Reference Cube*: A wireframe cube that surrounds all the entities is always displayed. This gives the user an idea of the orientation.
- *Horizon*: As another navigation aid, a ”sky” and a ”background” are always shown to let the viewer know about the direction they are looking at.
- *Normalizing metrics*: Metrics could have arbitrary high values, which would make visualizations unreadable. Due to that and the fact that the entities are displayed within the reference cube, all the metrics are normalized to values between 0 and 100.
- *Interactivity*: Rotating, zooming, and panning are possible. This kind of interaction is not novel in any kind, but rather expected. In addition to that, a set of pre-defined viewpoints is given, which helps users navigate to a certain point with only a single click.

3.9 EvoLens

Ratzinger, Fischer, and Gall generated EvoLens [RFG05], a system for visualization and navigation of software information extracted from versioning systems. It is especially tailored for Java programs stored in CVS [BF03]. Only CVS information is extracted, but not the source code itself. That’s why a class is the smallest unit EvoLens handles. Nested classes are not supported.
EvoLens visualizes coupling between software modules (i.e. packages and classes, when ”speaking in Java language”). The following information is extracted:

- author of each change to a file
- date and time of each change

- files included in a change event
- package structure of the source code

Nested graphs are used to visualize the structure of the source code. Packages and their sub-packages are displayed as rectangles, Classes, the smallest units in the model, are represented as ellipses. Not more than two levels are shown at any time. In other words, from a package, the user can see its sub-packages, but not its sub-sub-packages. In this way, the hierarchical structure of the software is nested. The edges of the graph itself represent the change coupling between two classes. The stronger the coupling, the thicker are the edges. Color indicates the evolution metric of a class. If a class has a low growth value for the selected time window, it will be drawn in a light color. If its growth value is high, it is drawn in dark color. By using nesting and regular graph elements (nodes and edges) at the same time, a multi-dimensional visualization (namely, structure and evolution) is realized.
The larger a software system is, the more important it is to hide irrelevant information. That's why the concept of a *focal point* is used. A user selects the focal point (a certain package). This is now the center of consideration, the starting point of the analysis. Only those relationships following from this focal point to other packages and classes are represented. This drastically reduces the amount of information that has to be visualized. The user can navigate through the structure by changing the focal point.
Another technique that is used to reduce the visualized information is structure folding. Instead of the coupling between classes, the coupling between packages may be of interest. The edges are painted between the packages. External packages (i.e. packages which are outside the focal point) are also displayed in that case. If the user is interested in a certain package, it can be unfolded, i.e., the classes in that package can be displayed separately again.
A way to avoid "edge clutter" is to set a coupling threshold. Loose (but still existing) coupling between classes happens soon. To paint a "correct" graph, many narrow edges would have to be painted between classes. However, this information is not very interesting, and so it is better to filter out those narrow edges completely. How much of this sort of information should be filtered out depends on the user and the particular software structure. That's why a threshold can be adjusted to individually set the lower bound of visualized coupling.
Navigation in time lets the user see the graph within a specific period of time. Start date and end date can be set and only the changes between those two dates will be displayed.

3.10 Mondrian

Mondrian [MGL06] is a visualization framework for two-dimensional visualization of data. The idea behind it is to move the visualization tool to the data and not the other way around.

The authors try to solve two main problems in adopting software visualization tools. Firstly, tools that are designed for a specific purpose cannot support the user when they need a slightly (or sometimes even drastically) different visualization on the data related to the task at hand. Secondly, preparing and converting data to the format, which the visualization tool is able to process, can be difficult. They solve the first problem by creating a flexible framework instead of a single visualization tool, and the second problem by bringing the visualization to the data and not the other way round.
This is achieved by using Figure classes for each entity. Each Figure class is connected to exactly one Object (which can be any sort of data). Moreover, each Figure is connected to one Shape, which acts as a translator between the data (Object) and the visualization (Figure). The Shape contains no state but only the rules that dictate how the Figure should be displayed.

Chapter 4

Approach

In this chapter, we want to give an overview of our approach to our model and the according framework J3DVN. We describe how it integrates into the Eclipse platform. Moreover, we give a quick introduction into Java 3D, for we rely on this API when creating our 3D visualization.

4.1 MVC Architecture

The model was designed with an MVC architecture [Ree73] in mind. That's why it consists of three packages: model, view, and control. One advantage of using an MVC architecture is that the framework could be implemented for different visualization platforms easily. Another major important factor of this architecture is, that the data model is separated from the other parts. This makes it possible to create a generic framework that can be used with many different data types of different problem domains. Nevertheless, the idea of separating functionality is extended even further by the plug-in concept, which we describe in the next section.

4.2 Addons

Our framework uses a plug-in concept. To contribute to J3DVN, we have to write new addons. Well-defined properties make it possible to assign or read certain values of an addon.
There are basically five different types of addons:

- Data Addons
- Visual Addons
- Conversion Addons
- Input Addons
- Layout Addons

We will explain those basic addon types later.
The different basic addon types[1] are defined as different interfaces. There, functions are defined, which are commonly needed for every addon of a certain type. On the other hand, custom features, which are unique for implemented addons, cannot be known in advance by the framework. This problem is solved by the realization of so called *Properties*, which are an important aspect of an addon. A property is a public function of an addon, which has an annotation that marks it as a property function and has a return value of a property interface type.
Examples of how to create own addons and their properties are shown in Appendix B.

4.3 Eclipse

The architecture of Eclipse is based completely on plug-ins. A tiny core exists, which is responsible to load all the plug-ins, which make up the functionality. Figure 4.1 (taken from [GB03]) shows the basic architecture of Eclipse. The parts in this figure are:

- *Runtime*: This defines the plug-in infrastructure. It is responsible for discovering and loading other plug-ins.
- *Workspace*: A Workspace manages projects. A project consists of files and folders that map onto the underlying file system. Under a Model-View-Controller paradigm the Workspace is the Model.
- *Standard Widget Toolkit (SWT)*: The SWT is a widget toolkit that provides a standard set of graphical widgets.
- *JFace*: A UI toolkit built on top of SWT providing an abstraction layer. While SWT contains basic UI controls, JFace provides helper classes that simplify the implementation of common UI tasks.
- *Workbench*: The workbench defines the Eclipse UI paradigm. Its typical elements are Editors, Views, and Perspectives. The workbench naturally uses JFace and SWT.

Adding new plug-ins to Eclipse is usually done by extending existing plug-ins. Plug-ins have extension points, onto which new plug-ins can be hooked to extend those plug-ins again.
Figure 4.2 shows how different plug-ins are connected to each other. You can see three plug-ins, which have extensions (left hand sided) and extension points(right hand sided). The *at.ac.tuwien.j3dvn.j3dvneclipse* plug-in extends the plug-in

[1]As a convention we use the term *basic addon type* for the different addon interfaces listed above plus the "abstract" *Addon* interface, while an *addon type* can also be an implemented class of one of the basic addon types.

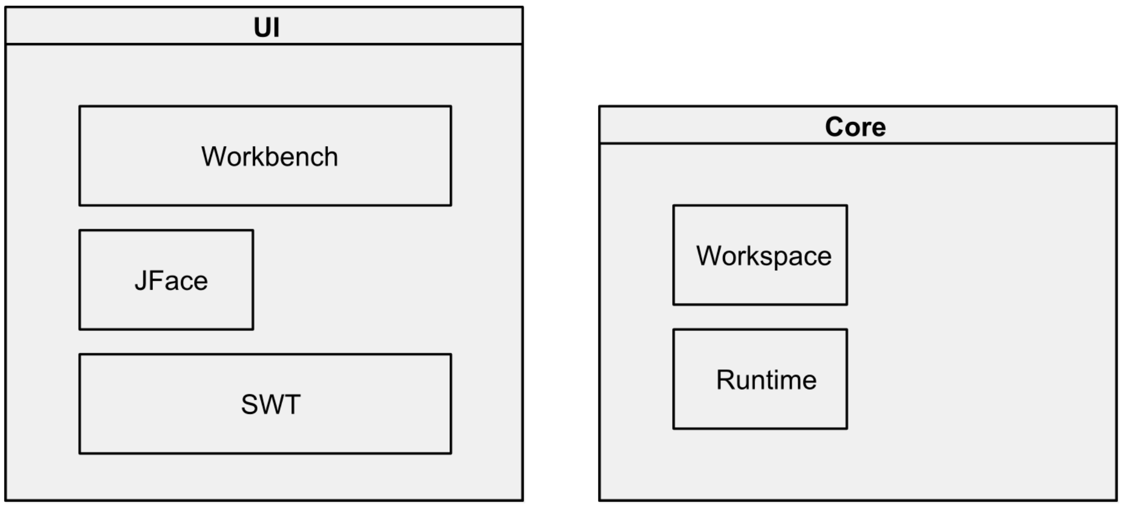

Figure 4.1: Overview of the Eclipse architecture

org.eclipse.ui.views on the extension point *view* with the *Mapping* extension, so that a new view is added to the Eclipse Workbench. The same plug-in also provides extension points on its own, namely one extension point for each basic addon type of our framework (see next section). The plug-in *at.ac.tuwien.evolizerinput* extends to the *inputs* extension point with the extension *input1*. Moreover, the extensions *class* and *coupling* of the *evolizerinput* plug-in are both extended to the extension point *data*. Additionally, *evolizerinput* extends *org.eclipse.ui* with *Evolizer 3D Graph* to *editors*, and with *Layout* to *editorActions*.

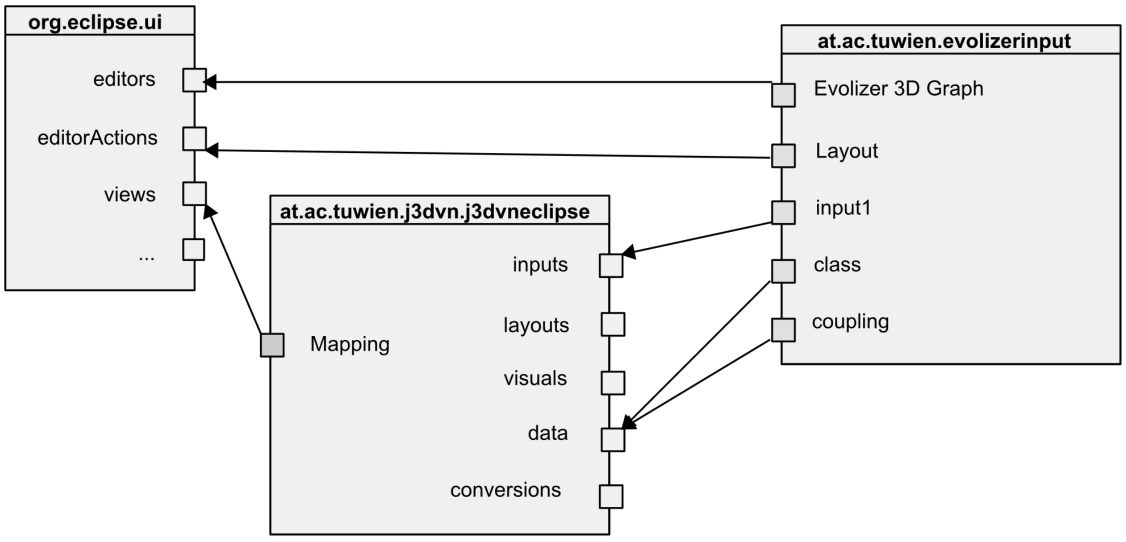

Figure 4.2: Example of extensions and extension points of Eclipse plug-ins

4.4 Java 3D

Java 3D [Sun08] is a scene graph based API for rendering 3D graphics using Java. It relies on OpenGL (or on the Windows platform optionally DirectX) to perform native rendering. All the other programming logic, such as scene description or interaction, happens in Java only.

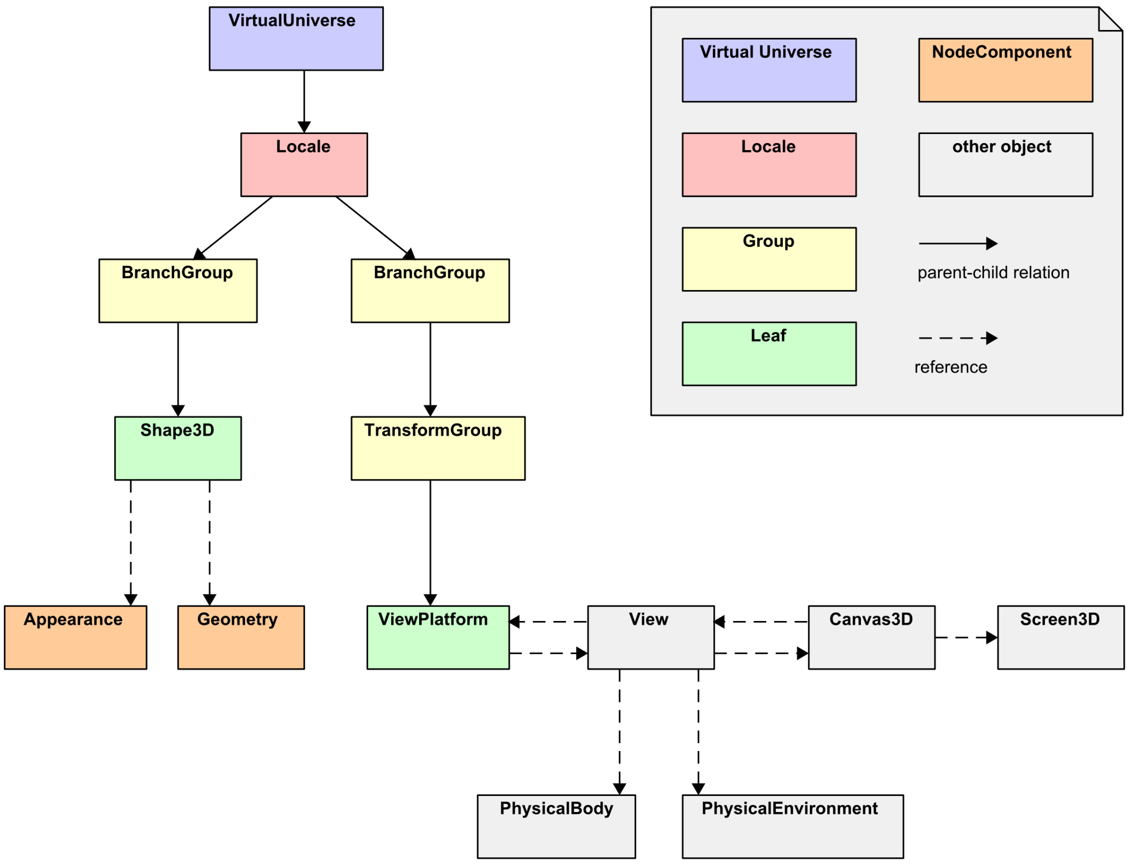

Figure 4.3: Example of a scene graph in Java 3D

Figure 4.3 shows an example of such a scene graph. All of Java 3D takes place in a `VirtualUniverse` class, which defines the highest level of object aggregation. One universe can contain none, one, or more `Locale` objects. A `Locale` is positioned in the universe with high precision, while other objects are positioned in the `Locale` with lower computer-friendlier precision. When continuing the universe metaphor, we can speak of a `Locale` as a galaxy. Other objects - primarily of type `Node` and its descendants - build the scene graph by creating parent-child relations between those objects forming a tree-structure. `BranchGroup` nodes can be attached to and detached from other nodes in the scene graph. All the visual 3D objects are attached to one "root" `BranchGroup` node, which in turn is attached to a `Locale`. A `TransformGroup` is used to make transformations to underlying children. A scene graph usually needs many `TransformGroup` nodes to get all the objects into their right position, angle, and size. The `ViewPlatform` node controls the position, orientation and scale of the viewer. The `View` contains all parameters needed in rendering a 3D scene from one viewpoint. A view can be used by many `Canvas3D` objects. It exists outside of the scene graph, but attaches to a `ViewPlatform` node in the scene graph. It also contains a reference to a `PhysicalBody` and a `PhysicalEnvironment`. These are both classes, which address physical world model parameters. `Canvas3D`, an extension of the AWT `Canvas` class, serves as the display window image. All the 3D objects are painted on it. It uses a `Screen3D` object to find out about the physical screen characteristics. A `Shape3D` is a leaf node in the scene graph and specifies all (visible) geometric objects. Such an object has an `Appearance` and a `Geometry` object, which define the object's rendering state and shape, respectively. Each visual addon is a sub graph of the whole scene graph. The root element of a visual addon is a `Group`. When looking at the scene graph example in Figure 4.3, every visual addon would be attached to the left `BranchGroup` node of the graph. Or more precisely: The root of every visual addon, which is a `Group` or a descendant of it, is a direct child of the left `BranchGroup` in the scene graph example.

Chapter 5

Model and Framework

This chapter explains the design of our model and the framework J3DVN, which implements this model.

5.1 Addons

Just like Eclipse itself does, our framework also uses plug-ins. We don't use the term plug-in for our needs, though, because now we are not speaking of Eclipse plug-ins anymore. Instead we use the term *Addon*. We explain the different basic addon types in the following sections.

5.1.1 Data Addons

A data addon describes a piece of data of a data model. We have to distinguish between two different types of data addons: *Entities* and *Relations*. The first ones describe entities of the data model, while the latter characterize the relation between two entities of a data model. A (useful) data addon needs to have properties, which contain the data. Contributing new data addons is different from contributing other addons as the `DataAddon` interface is never implemented directly. Instead, its direct descendants `Entity` or `Relation` are implemented.

5.1.2 Visual Addons

A visual addon is a visual representation of a data addon. A visual addon must be aware of Java 3D, which means it must provide a `Group`, which may contain additional 3D objects. Furthermore, it is supposed to have well-defined properties, which allow changing the representation - for example the color - of the visual objects.
A visual addon represents a data addon. It also has a reference to it. However, data addons don't know anything about visual addons.
A visual addon can be as simple or as complicated as the creator of the addon wants it to be. It could be only a primitive object like a cube, a sphere, or a cylinder.

Or it could be a 3D model created in a 3D graphics application, like 3DSmax, and imported into Java3D.

5.1.3 Conversion Addons

A conversion addon is used to convert from one data type to another one. It basically has a `convert()` method, which does the job. Conversion addons are used for the mapping of input data to its visual representation. A mapping of a data value to a visual property can only be done, if there exists an appropriate conversion addon with the correct input and output data types. For example, in a file system, every single file is visualized through a cube in 3D space, and the size of a file is mapped to the color of a cube. The size is an `Integer` property, while the color is a `Color` property. To realize the mapping, we need an `IntegerToColor` conversion addon.
Unlike data and visual addons, conversion addons are rather "static" - usually it only needs one single instance of each conversion addon.

5.1.4 Input Addons

An input addon can open a resource of a specific type, read its data, and construct a model from that data. The resource can, for example, be a file of a specific format, or a database connection.
Input addons are different from the previous addons as for one model there exists only one single input addon. It is responsible for creating data addons and populating them into the data model.

5.1.5 Layout Addons

To layout the visualized graph, a layout addon is needed. This type of addon takes nodes of a graph (i.e. visual addons) as input data and calculates their new position, based on the internal layout algorithm. This calculation runs in a separate thread. Only after the calculation has finished completely all the nodes in the graph are set to their newly calculated positions.
A layout addon is a very important part of the system, because by using different layout algorithms different visualizations can be realized.

5.2 Connectors

The reason of using connectors for the addons is to keep as much logic as possible out of addons, to make development of addons simpler. Connectors provide methods, which make access to addons easier. For example, access to properties of addons is simplified by using a connector method, which invokes the correct method of the addon.

Another approach, instead of using connectors, would have been to make abstract addon classes, from which new addons have to inherit. But this would also have been the major drawback of the approach as every addon had to inherit from another super-addon. This constraint would have complicated development of addons, because they couldn't have been descendants of completely other classes (which may be necessary for some reason). Since a new addon must only implement a certain interface, flexibility is higher.
A connector always contains exactly one addon. This addon is then connected to the connector. Speaking in design pattern terms [GHJV95], a connector is an *Adapter*, where the addon and the *AddonManager* (see Section 5.4) are subsystem classes of the connector. For each of the different basic types of addons exists one connector type.
Different connectors have different functions: An `EntityConnector` has functions to access the entity's relations, an `InputConnector` has a method to access the data source, and so on. That's why we couldn't simply design connectors with generics based on their addons in the form

```
public interface Connector<T extends Addon>
```

And that's why, unfortunately, our architectural model looks a bit more complicated, because for every basic addon type there exists a separate connector type.

5.3 The Data Model

Our data model is completely graph based. It contains entities and relations (see Section 5.1.1), where entities are the nodes and relations the edges of the graph.

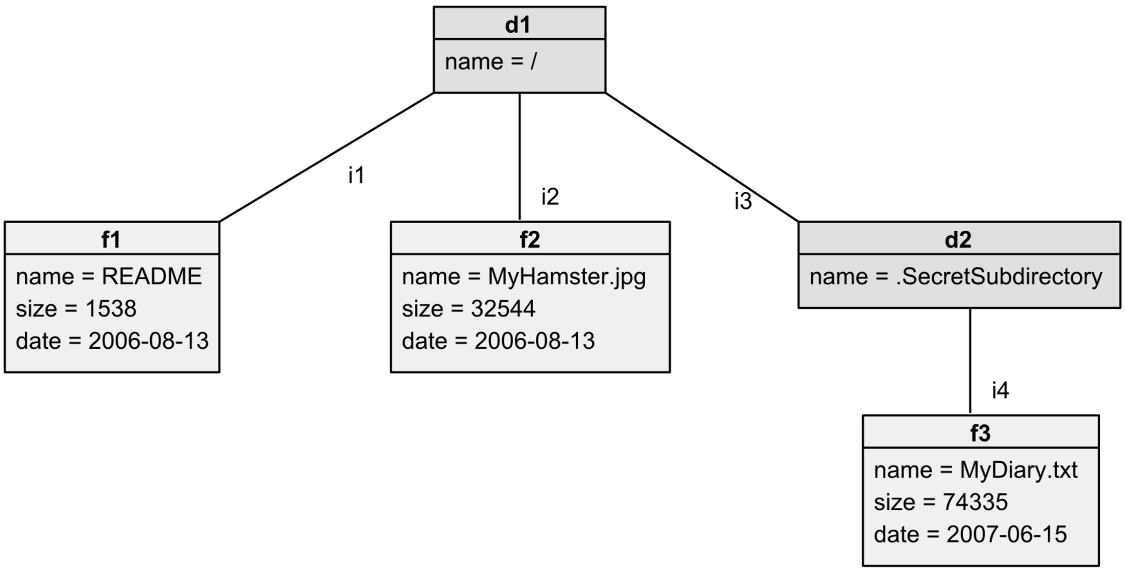

Figure 5.1: Example model of a file system

For example, we could want to get a model of a file system. We could write an entity *Directory*, which has the property *name* of type *String*, and another entity *File* with the properties *name* of type *String*, *size* of type *Integer*, and *date* of type *Date*. Additionally, we could write a relation *IsIn*, which describes, if a file is inside of a directory.
Now, a data model of a file system could look like the one in Figure 5.1. A textual representation of the same can be seen in Listing 5.1. The data model contains `d1`, `d2`, `f1`, `f2`, `f3`, `i1`, `i2`, `i3`, and `i4`.
Of course a model does not necessarily have to be a tree (like in this example), but can be any kind of graph. Moreover, unlike in the example, the model can not only contain nodes but also relations of different types. This way we can get graphs with different kinds of edges, or even two graphs, which have the same set of nodes but different edges. This allows the framework to be used for different applications. Another example can realize a class hierarchy and, additionally, describe coupling between classes as another metric. The visualization of inheritance and coupling can happen at the same time by using two different visualization addons.

```
d1: Directory(name="/")
f1: File(name="README", size=1538, date=#2006-08-13#)
f2: File(name="MyHamster.jpg", size=32544,
      date=#2006-08-13#)
d2: Directory(name=".SecretSubdirectory")
f3: File(name="MyDiary", size=74335, date=#2006-09-04#)
i1: IsIn(d1, f1)
i2: IsIn(d1, f2)
i3: IsIn(d1, d2)
i4: IsIn(d2, f3)
```

Listing 5.1: Textual representation of an example model of a file system

5.4 Architecture

An overall (simplified) view of the architecture can be seen in Figure 5.2. Please note that we left out connectors from this diagram. The *Addon Manager* is the central instance, which keeps track of all the different addon types by managing a list of *Descriptor* objects, which are basically $\langle Class, Name\rangle$ tuples of addons. The Descriptor is also capable of instantiating a new addon.
As already explained in Section 5.1, there are five different basic addon types, which are used in different areas of the system: *DataAddon*, *VisualAddon*, *ConversionAddon*, *InputAddon*, and *LayoutAddon*.
A *Model* is basically a container for *DataAddons*. The *InputAddon* is responsible to add the data addons to the model.
The duty of the *DataVisualMapping* is to translate data addons to visual addons, as it contains all the rules for visualizing the elements of the model. Each data addon

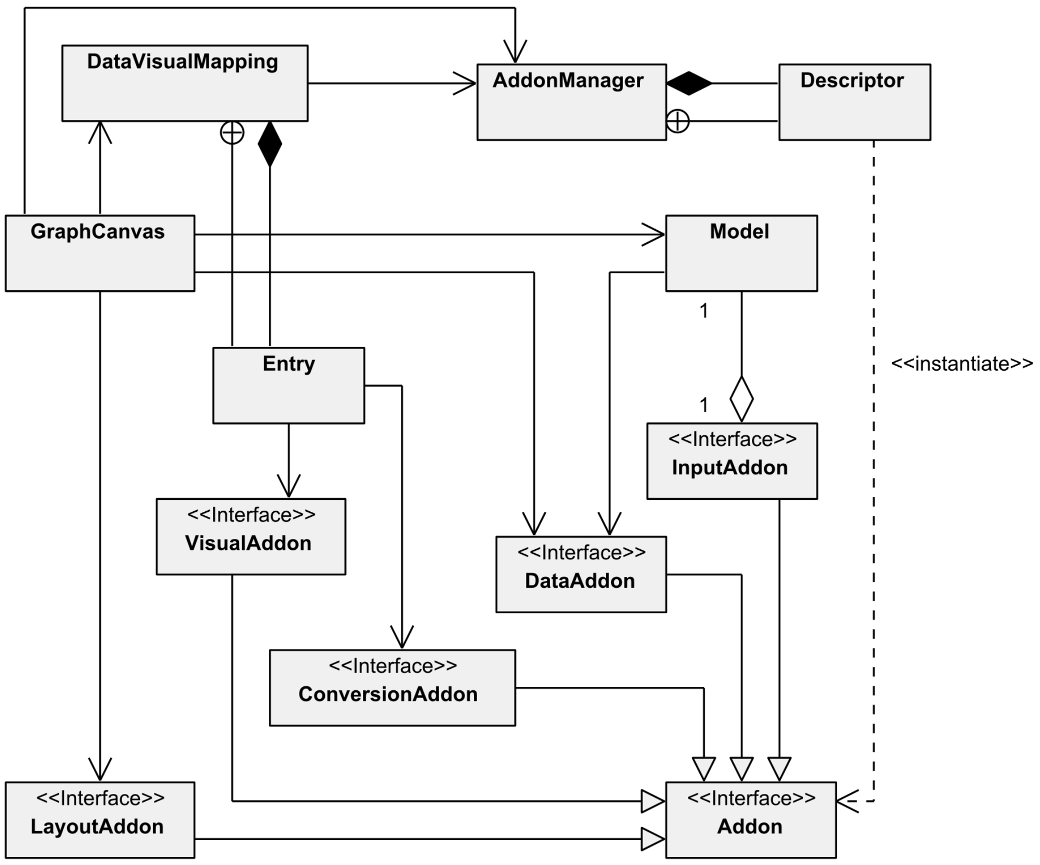

Figure 5.2: Framework architecture

type can be mapped to zero or one visual addon type; arbitrary many different data addon types can be mapped to the same visual addon types. Each property of a data addon type can be mapped to zero or more properties of the visual addon type to which the data addon type is mapped. A property of a visual addon type can be mapped to zero or one property of its data addon type. The mapping between properties needs, as an additional information, to know about the conversion addon to use. Only then a conversion from the data addon property's data type to the visual addon property's data type can be done. Each mapping from one data addon to a visual addon (including their property mappings) is stored in one *Entry*. The *DataVisualMapping* contains a list of Entries and the logic to access them easily.
Speaking about our file system example from the previous section, we can think of visualizing files as *Box*es, and directories as *Sphere*s. Next, the size of a file is mapped to the *width*, *height*, and *length* of its box. We assume that our fictional visual addon *Box* has those three properties, and we know that the data addon *File* has a property *size*. Moreover we could map the directory's name to the sphere's color. Here we assume the existence of a property *color* in the visual addon *Sphere* and a conversion addon that converts a string to a color by discretely selecting different colors from a color table for every distinct string.

The *GraphCanvas* is the part of the framework, where the visualization takes place. It contains a `Canvas3D` object (see Section 4.4) and a virtual universe, including the root of the scene graph, a default view, default background, and default light sources. All those settings can be accessed and changed to suit custom needs. It is aware of the mappings and the data model. Using this information it creates the visual objects on the canvas. It can also call the layout addon's `calculate()` method, which lays out the graph.
The concepts of a graph canvas and visual addons are taken from Kerren et al. [KBK04]. In their work they focused on visualizing data from a pattern recognition algorithm. The differences in the problem domains and the similarities in the technical realization demonstrate the genericness of our framework.

5.5 Implementation

This section explains the implementation of the framework. We try to explain the significant parts to ease the understanding of the whole program constructs. A complete documentation of the implementation is given in the Javadoc documentation.

5.5.1 General Structure

The whole work is developed in Java with Eclipse [Ecl08]. That's why it is divided into several Eclipse projects. The main functionality lies in *j3dvn*. The integration into Eclipse is done with an Eclipse plug-in. This plug-in was developed in the *j3dvneclipse* project. The complete framework is implemented in those two projects, *j3dnv* and *j3dvneclipse*. Preexisting addons are provided to the framework, but are not part of it. A bunch of addons exist in the *addoncollection* project. Figure 5.3 shows how the different technologies are interconnected.

5.5.2 Project j3dvn

The framework consists of three packages: model, view, and control.

The control package

The package `at.ac.tuwien.j3dvn.control` contains abstract addons and connectors, plus the super-classes (or -interfaces) for input addons. Together with additional utility classes, this makes up the infrastructure to work with addons.

Addon This is the super-interface of all addons. Only one of the direct sub-interfaces (`ConversionAddon`, `DataAddon`, `InputAddon`, `LayoutAddon`, or `VisualAddon`) are meant to be implemented, but not `Addon` itself.

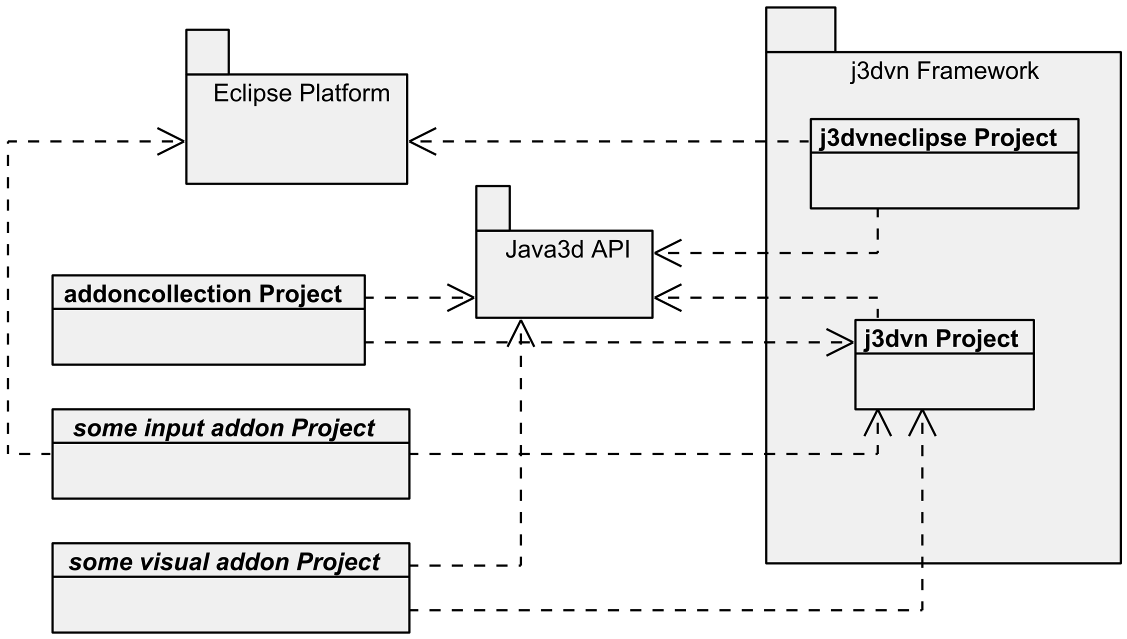

Figure 5.3: Structure and dependency of implemented projects

Connector This interface is the parent for all other connectors. Addons are accessed through connectors. The task of a connector is to keep as much programming logic as possible away from an addon, so the development of addons becomes as easy as possible.
Connectors themselves are again interfaces and not classes. This is to avoid possible cyclic dependencies of classes.
Connector interfaces are not meant to be implemented. Instead, there are internal classes, which implement this interface, and sub-interfaces of `Connector`, respectively. These classes are created and returned by the `AddonManager` class.
This interface has, among others, a method `getAddon()` that returns the addon, which is connected to this connector. Every connector interface, which extends `Connector` overrides this method as the return types differ. The method of the extended interface will return a type, which extends `Addon`. For example the `getAddon()` method of the `InputConnector` interface returns an `InputAddon`, while the `LayoutConnector` interface's `getAddon()` method returns a `LayoutAddon`.

ConversionAddon This interface provides a way to make a conversion of values from one type to another type. This can be used, for example, to map a numerical value to a color value. One other example is show in Listing 5.2.

```
ConversionAddon<Integer, String> numberEvaluator =
  new ConversionAddon<Integer, String>() {
  public String convert(Integer value) {
    if (value < 0) return "That's negative";
    else if (value < 10) return "Quite a small number";
    else if (value < 100) return "A pretty ok number";
    else if (value < 1000) return "A rather big number";
    else if (value < 10000) return "It's huge";
    else return "Way too big!";
  }
  public String getName() {
    return "Simple Number Evaluator";
  }
}
```

Listing 5.2: Example of a conversion addon

When looking at the declaration of this interface, you can see that there is a generic input type `I`, which must be a `Comparable`, and a generic output type `O`. Its `convert()` method does the mapping (i.e. conversion) from input data to output data.

InputAddon To create a new input addon, you have to implement this interface. Basically there exists the `open()` method, which opens a file and reads in (the first couple of or all) data.

ConversionConnector This interface is a connector for a conversion addon. There is no other characteristic of this interface.

DataConnector This is the parent interface for `EntityConnector` and `RelationConnector`.

EntityConnector This sort of connector is used to connect to entities.

RelationConnector This is a connector for relations. It contains methods to access the two entities of its relation.

InputConnector A connector for input addons. Just like `InputAddon` it has the method `open()`. Additionally it contains a method `getModel()`, which returns the `Model` that is associated to the connector's input addon.

LayoutConnector A connector for layout addons. The layout addon is responsible for calculating the layout of the visualization. The connector calls the `calculate()` method of the addon in a separate thread, so the layout calculation (which can take quite some time) can run in the background. When creating a layout addon, you don't need to be bothered with thread programming.

VisualConnector A connector for a visual addon. The internal implementation is marked as **`private`** and is thus hidden from outside. Instances of this private implemented class are created by the `AddonManager` whenever a new visual connector is created either as a visual entity or as a visual relation. This implementation provides "pseudo" properties, which can be applied to any visual addon. So they don't have to be implemented when creating an addon. These properties are listed in Table 5.1.

Name	**Type**	**Description**
height	`Double`	Height of the visual element.
length	`Double`	Length of the visual element.
title	`String`	A text, which will appear next to the visual element. The text will always face the viewer's position, regardless of any rotation.
width	`Double`	Width of the visual element.

Table 5.1: Pre-implemented properties of the internal `VisualConnector` implementation

IProperty This interface is a wrapper interface for accessing data. This is done through its `setValue()` and `getValue()` methods. An example of how to use and implement this interface can be seen in Listing 5.3 on page 32.

AddonManager The `AddonManager` is a class, which has all the necessary information to work with the available addons. Addon classes get registered in this class (via the `registerAddonClass()` method). The relationship between the addon manager, connectors, and addons is described in Figure 5.4.
Implementations of the connector interfaces keep a reference to the addon manager to access information about addons. The addon manager is implemented as a singleton. To access this central instance, the static method `getInstance()` has to be called.
The overloaded method `createConnector()` creates a connector for an addon as a wrapper and returns that connector. It takes one parameter. Depending on that parameter it has different return types. Table 5.2 shows the return types depending on the parameter types of this method. Please note that the table has no entry for a `VisualAddon` parameter type. The reason for this is that a visual addon can either be a visual entity or a visual relation. So, to create a visual entity connector, you cannot call the method `createConnector()`. Instead, you have to call the separate method `createVisualEntityConnector()`. In the same way you have to call `createVisualRelationConnector()` to create a visual relation connector.

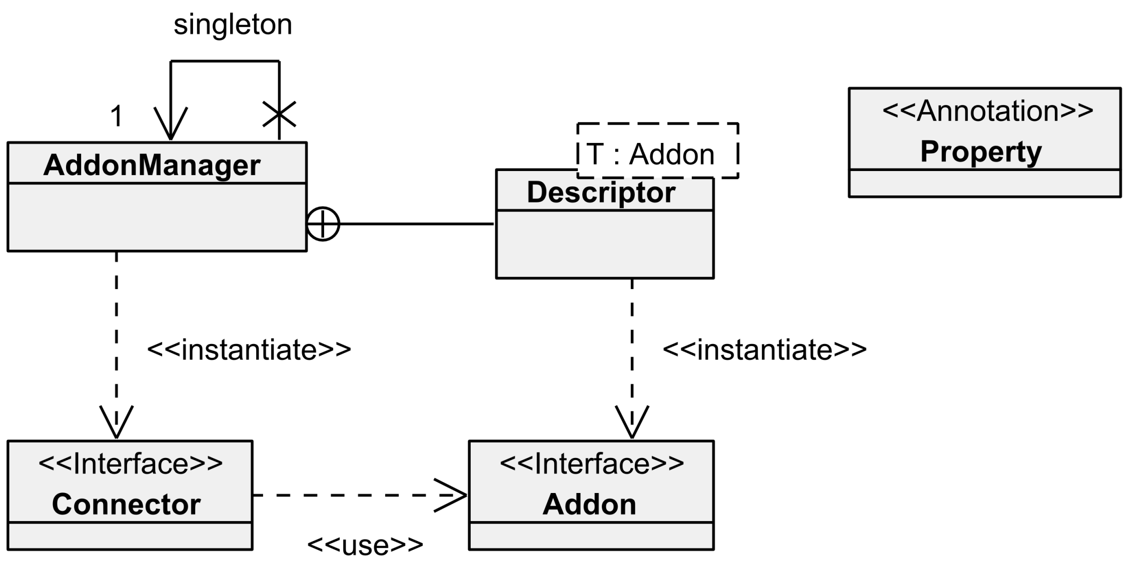

Figure 5.4: Relationship between addon manager, connectors, and addons

Parameter Type	**Return Type**
`ConversionAddon`	`ConversionConnector`
`Entity`	`EntityConnector`
`InputAddon`	`InputConnector`
`LayoutAddon`	`LayoutConnector`
`Relation`	`RelationConnector`

Table 5.2: Parameter types and return types of the `createConnector()` method of the `AddonManager` class

AddonManager.Descriptor This nested class is used to store information of addons and their names. It is also used to instantiate addons. There are three important methods: `getName()` returns the name of the addon as a string, `getAddonClass()` returns the class of the addon (so the return type is `Class<T>`), and `newInstance()`, which instantiates a new addon (the return type of the method is `T`).

DataVisualMapping This class provides a conversion between data addon types and visual addon types and between their properties.
The conversion between visual addon type and data addon type is a 1:n-relation, i.e. 1 visual addon type can be mapped to n data addon types. The conversion between properties is a 1:n-relation in the other direction: 1 property of a data addon type can be mapped to n properties of a visual addon type. For example, when trying to visualize a class hierarchy of an application, there could be two data addon types, *Package* and *Class*, and two visual addon types, *Cube* and *Pyramid*.

The mapping could look like this: *Packages* are visualized by *Cubes*, while *Classes* are visualized by *Pyramids*. It is also possible to visualize both, *Package* and *Class*, by *Cubes*. However, it is not possible to visualize some *Packages* by *Cubes* and other *Packages* by *Pyramids*. Suppose the *size* and *method count* of a *Class* are properties of that data addon, and *width* and *transparency* are properties of the visual addon *Cube*. There can be a mapping from *size* of *Class* to *width* of *Cube*. Additionally, there can be a mapping from *size* of *Class* to *transparency* of *Cube*. However, there cannot be a mapping from *size* of *Class* to *width* of *Cube*, while at the same time there is a mapping from *method count* of *Class* to *width* of *Cube*.

DataVisualMapping.Entry This class stores one mapping between a data addon and a visual addon. It is aware of its outer class (`DataVisualMapping`), and it notifies other objects, whenever entries are changing. Its constructor takes two `AddonManager.Descriptor` instances, which describe the data and visual addon. It contains access methods to change the addons of the mapping and also to add and remove property conversions to the mapping.

Property This annotation is used to mark a property of an addon. A *property* is defined by adding the annotation `Property` to a public method, which returns an instance of type `IProperty`.

```
@Property("Color") public IProperty<Color> getColour()
  {return pColor;}

private Color fColor;
private IProperty<Color> pColor = new IProperty<Color>() {
  public Color getValue() {return fColor;}
  public void setValue(Color color) {pColor = color;}
}
```

Listing 5.3: Example of an implementation of an addon's property

For example there could be a property for changing and retrieving the color of a visual object. The implementation of the `IProperty` interface usually happens through an anonymous inner class. A full implementation of our example could look like it is shown in Listing 5.3. `pColor` is an implemented `IProperty`, while `fColor` stores the actual value of the property. `pColor` is a wrapper for accessing `fColor`. As a result, the concept of properties - known in other programming languages - is added to Java.

MappingListener A mapping listener registers with a `DataVisualMapping` object. This interface defines methods, which are called after a new mapping entry or a property mapping has been added, removed, or changed.

PropertyChangeListener This interface defines the `propertyChanged()` method, which is called whenever the value of any property from an addon is changed. The caller is a `Connector` object. The parameters of the `propertyChanged()` method are displayed in Table 5.3.

Parameter Name	**Parameter Type**
`sender`	`Connector`
`propertyName`	`String`
`oldValue`	`T`
`newValue`	`T`

Table 5.3: Parameters of the `propertyChanged()` method of the `PropertyChangeListener` interface

The model package

The package `at.ac.tuwien.j3dvn.model` contains data addon interfaces and the `Model` class, which keeps the data addons together.

DataAddon This is a direct descendant of `Addon`. It is the super-interface for all data addons. A data addon is a class, which describes an element of a data model. A data addon object has a name, a list of properties, and belongs to maximal one model. This interface itself has no own methods, but it exists because of inheritance purposes because the interfaces `Entity` and `Relation` extend this interface. So, a class should not implement the `DataAddon` interface, but rather one of its two descendants, `Entity` or `Relation`.
Data addons differ from other addon types in the way that properties of data addons needs not only be of type `Object`, but they must implement the `Comparable` interface. This restriction exists, since properties of data addons are used as input values for conversion addons.

Entity The `Entity` data addon represents a data entity. Since we are working with a graph based model paradigm, we can say that entities are the nodes of a graph.

Relation The `Relation` data addon is the second type of data addon. It represents a relation between two entities. In a graph the relations are the edges between the nodes. Two entities get connected by a relation by calling the `setEntity1()` and `setEntity2()` methods of the `Relation` interface.

Model A model is a data structure, which holds all information of a data model. It consists of two different units: entities and relations. Both can contain all kinds of data. The difference is that entities are connected to other entities through relations. An entity can have any number (or zero) of relations. A relation is always bidirectional.

The view package

The package `at.ac.tuwien.j3dvn.view` contains visual addon interfaces and classes to visualize and navigate the data model.

LayoutAddon A layout addon is responsible for creating the graph layout. The main functionality lies in its `calculate()` method, where the layout algorithm is executed.
The graph model may contain different types of edges. The layout algorithm may consider one type of edges (the "main relation") for its calculation. The `setEdgeType()` method sets that relation type.

VisualAddon A visual addon is a class, which implements the `VisualAddon` interface and holds any kind and number of 3D elements, which are connected through one scene graph. The root element of this scene graph must be a `Group` (this is a Java 3D class), to which there is direct access via the `getGroup()` method.

VisualEntityConnector The interfaces `VisualEntityConnector` and `VisualRelationConnector` are the only connectors not stored in the control package (Section 5.5.2). Just like the `Entity` addon this connector also doesn't have any special methods.

VisualRelationConnector This connector is a wrapper for a visualization of a relation. Via the methods `getEntity1()`, `getEntity2()`, `setEntity1()`, and `setEntity2()` the connectors of the two entities, which are connected through this relation, can be accessed.

GraphCanvas This is the class, where visualization actually takes place. It contains a `Canvas3D` object, on which all 3D objects are painted. Furthermore, this class contains the method `setModel()` to set the model to assign a reference to the existing data elements. The method `setDataVisualMapping()` sets the mapping rules between data addons and visual addons. Another important method is `calculate()`, which calls the `calculate()` method of the assigned layout connector. Besides these, there are additional methods to customize the visualization, for example to add lights or a background image.

Position This class is needed by the layout addon. Its constructor method expects a `VisualEntityConnector` variable, from which the position is retrieved and stored in the public variables `x`, `y`, and `z`. It stores a reference of the connector in the public variable `entity`. Layout addons will adjust the `x`, `y`, and `z` variables. This class has a **`synchronized`** method `adjustEntity()`, which will set the position of `entity` to its own values. This happens after the layout addon completed its calculation.

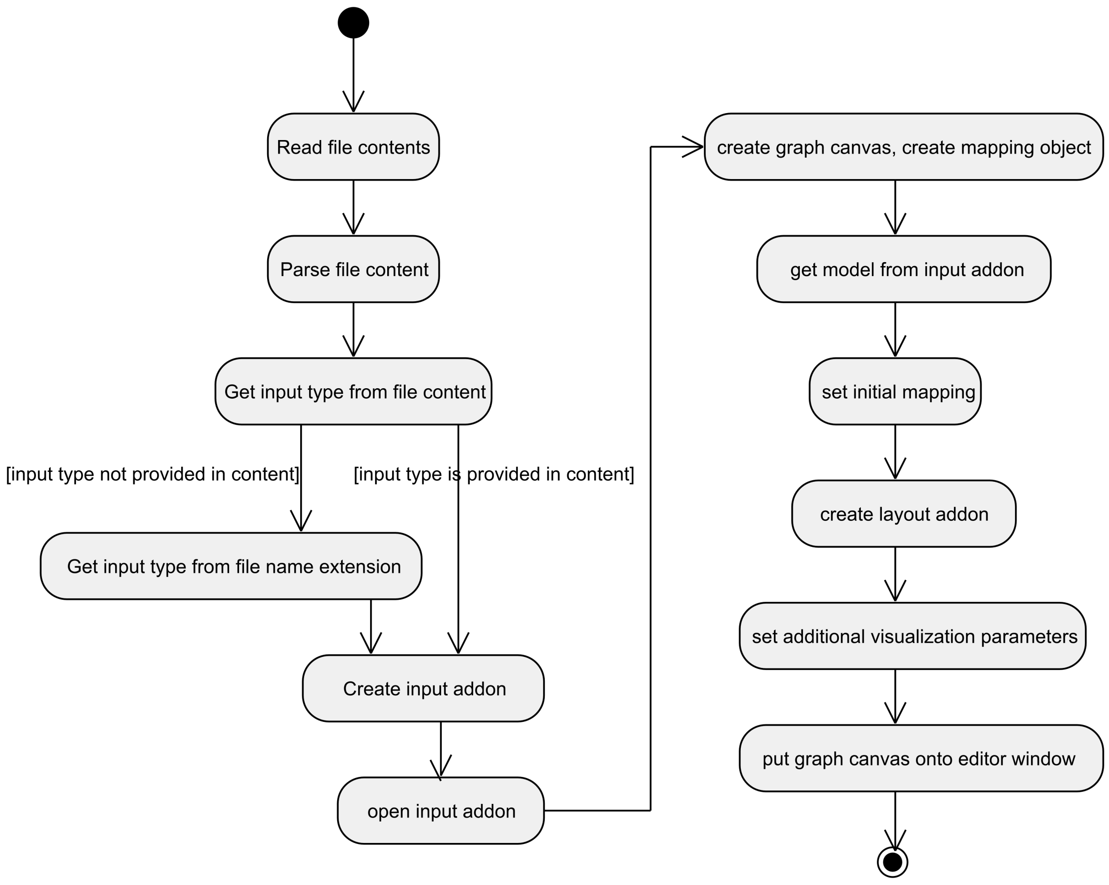
Read file contents
Parse file content
Get input type from file content
[input type not provided in content]
[input type is provided in content]
Get input type from file name extension
Create input addon
open input addon
create graph canvas, create mapping object
get model from input addon
set initial mapping
create layout addon
set additional visualization parameters
put graph canvas onto editor window

Figure 5.5: Activity diagram of the `createPartControl()` method of `Graph3DEditor`

LayoutEventListener This interface has one method: `layoutCompleted()`, which has one parameter of type `Collection<Position>` and a **`void`** return type. A layout event listener registers with a `LayoutConnector`. After a layout calculation the layout connector calls the `layoutCompleted()` method.

5.5.3 Project j3dvneclipse

This project is responsible to work with the Eclipse platform. Its main package is `at.ac.tuwien.j3dvneclipse`, which has additional sub packages. It contains dialogs to set mappings between data elements and visual elements. Particularly, it contains an editor window, `Graph3DEditor`, with which the 3D graphs are visualized. In its `createPartControl()` method the configuration file is loaded, and the visualization is created.

Name	**Default**	**Description**
attraction exponent	1.5	Exponent of the distance in the attraction energy. Is 1.0 in the LinLog model (which is used for computing clusters, i.e. dense subgraphs), and 3.0 in standard energy model of Fruchterman and Reingold [FR91]. Must be greater than 0.
repulsion exponent	0.1	Exponent of the distance in the repulsion energy. Exception: The value 0 corresponds to logarithmic repulsion. Is 0 in both the LinLog and the Fruchterman-Reingold energy model. Negative values are permitted.
gravity factor	2	Factor for the gravitation energy. Gravitation attracts each node to the barycenter of all nodes, to prevent distances between unconnected graph components from approaching infinity. A value of 0 can be used only if the graph is guaranteed to be connected.

Table 5.4: Properties of the `LinLog` addon

Figure 5.5 shows the activity diagram of this method. The first activity is to read the content of the input file, which is an XML document[1]. Its content is validated and parsed. After that, the type of input addon is retrieved. The input type information is provided as the `name` attribute of the `input` tag in the input file. By using the `AddonManager` a new input addon of desired type is created. To be precise, additionally an input connector is created, of which the input addon is a part. Then, the input addon is opened by calling the connector's `open()` method. This method reads and sets the input addon's properties and finally calls the input addon's `open()` method. After that a `GraphCanvas` object and `DataVisualMapping` object are created. The model, which was populated in the input addon's `open()` method is retrieved and the mapping object is initialized with this model. During that initialization process all data types which exist in the model are mapped to null. After that, the actual mapping information is retrieved. This can be set initially in an input file, so it doesn't have to be manually done each time the

[1]For the correct syntax of the input file please refer to the its schema in Appendix A.

visualization is created. Nevertheless, it can be changed at any time while the visualization is open. The next step is to create the layout addon as specified in the input file. Then, additional visualization parameters, such as custom light sources or background images, are read from the input file and set to the graph canvas. After this, everything can be connected to the graph canvas: the mapping object, the model, and the layout addon. As the next step, the `layout()` method of the `GraphCanvas` can be called, which in turn executes the calculation of the layout addon. As a last step the graph canvas can be put onto the editor window. This is done by creating an `SWT_AWT` bridge that allows creating AWT objects within a SWT `Frame`.

5.5.4 Project addoncollection

This project is not part of the framework but a contribution to it. Its main package is `at.ac.tuwien.j3dvnaddoncollection`. It contains a couple of general addons, which make it easy to start working with the framework.

Name	**Type**	**Exists In**	**Description**
color	`Color3f`	`Cube`, `Cylinder`, `Sphere`	Color of the element. Note that the type of the property is not a `Color` from the AWT package, but `Color3f`, a Java 3D class.
transparency	`Double`	`Cube`, `Cylinder`, `Sphere`	Transparency level of the visual element. A value of 0 makes the element solid, while 1 makes it invisible.
thickness	`Double`	`Cylinder`	The thickness of the cylinder. Cylinders are commonly used to visualize relations between two nodes. In such cases the length of the element will be calculated automatically, so that the edge touches both of the relation's nodes. The thickness can be used to display larger (thicker) or smaller (thinner) edges.

Table 5.5: Properties of the visual addons from the addoncollection project

Layout Addons

The addoncollection project provides two layout addons.

LinLog This addon uses the LinLog energy model [Noa06] to calculate the layout of the graph. The energy model specifies good graphs as graphs with little energy, i.e. basically with short edges. After minimum energy calculations in the LinLog model, clusters of a graph will be separated. Nodes with high degree[2] will be placed into the center of the graph while nodes with lower degree will be placed on the outer side.
Several properties can be changed to adjust the layout calculation and thus the appearance of the graph. These properties are explained in Table 5.4 on page 36.

Random The random layout algorithm is a very basic algorithm, which simply sets each node to a random position. This addon was added as a very basic proof of concept. It gives satisfactory results when there are very few (2-3) nodes.

Visual Addons

There are some very basic visual addons, which can be used for visualization. Please remember that the implementation of `VisualConnector` provides the "pseudo" properties *height*, *length*, *width*, and *text*, as explained in Table 5.1 on page 30.
The visual addons are *Cube*, *Cylinder*, and *Sphere*. Their implementations are quite similar. That's why we won't explicitly explain them separately. Table 5.5 on page 37 explains their properties.

[2]The degree of a node is the number of connected edges of this node.

Chapter 6

Case Studies

To further illustrate the workings of our model, we are providing a case study done on two open source software projects, *ArgoUML* and *Azureus*. By reading the case example, you should have a clearer understanding of how our model helps to analyze evolution information. *ArgoUML* is a leading open source UML modeling tool and *Azureus* is a powerful, full-featured, cross-platform bittorrent client. To gather the required evolution information, we accessed data through the *evolizerinput* project, which has been detailed below in Section 6.2.7. Further, due to the nature of the projects being cited, a time frame of six months was set as a functional parameter for the analysis of evolution information.

6.1 Foundation

Our research process will introduce you to our framework in the domain of software evolution and provide you with an overview of how it can be used to gather input data and visualize it. The steps followed, both evaluate and reveal the basic procedures. We implemented several addons as a tool which exposed the kind of data that is retrieved.

6.1.1 Versioning System

All data was gathered from the versioning system CVS. Versioning systems are used in software projects, where several team members can work on several files and several versions of a software system. A team member can check out the files of a software system, make modifications on some of them on his or her local computer, and commit the modifications in the end to the central store of the versioning system. The versioning system doesn't replace the existing files but only stores the changes made to them together with some additional information such as author, date and time, lines added, and lines deleted. CVS keeps a log of all that user (commit) activity. Only by analyzing this log file, it is possible to retrieve the necessary evolution data.

```
RCS file: /cvsroot/azureus/azureus2/com/aelitis/azureus/\
core/AzureusCoreException.java,v
Working file: com/aelitis/azureus/core/\
AzureusCoreException.java
head: 1.3
branch:
locks: strict
access list:
keyword substitution: kv
total revisions: 3; selected revisions: 3
description:
----------------------------
revision 1.3
date: 2006/02/10 03:43:09;  author: tuxpaper;  \
state: Exp;  lines: +2 -2
Copyright and Licensing mass update
----------------------------
revision 1.2
date: 2004/07/17 20:28:50;  author: parg;  state: Exp;  \
lines: +1 -1
bit of refactoring
----------------------------
revision 1.1
date: 2004/07/13 08:19:19;  author: parg;  state: Exp;
started work on core abstraction
=============================================================
```

Listing 6.1: Snippet of log file of Azureus source tree

Listing 6.1 shows the log file information of one single file (in that case namely *AzureusCoreException.java*) of the Azureus project[1]. This is a typical section of a CVS log file. Each section ends with a line of equal characters ('='). Not all the information of a section is taken into consideration, but only the following fields are retrieved:

- *RCS file*: The value of this field identifies the file in the CVS repository. From this information the file name and path can be obtained.
- *branch*: The branch information is stored here.
- *description*: Lists the modifications of the file. Every commit creates a new revision. Each revision entry starts with a line of minus characters ('-'). After that the following information is recorded:

[1]For layout reasons we word wrapped the file content. We marked a soft return with the backslash character ('\'). These characters are not actually part of the file contents.

- *revision*: The revision number identifies the revision of the file. After every commit the (minor) revision number is incremented by one.
- *date*: Date and time of the commit are stored in this field.
- *author*: This field contains the name of the author who made the modification. An example of this can be found in Listing 6.1, where two authors (*parg* and *tuxpaper*) had worked on the file.
- *state*: This field contains the state of a file. Its value is usually *Exp* (for experimental). If the file is deleted, the state changes to *dead*.
- *lines*: The numbers of added and deleted lines are stored in this field. The first revision doesn't contain this field, and also the number of lines that were modified cannot be seen.
- *comments*: The previous revision fields are stored in the first two lines of a revision entry. The remaining lines make up the comment of the revision. A comment of a revision is in no way unique as single revisions don't have comments. During one commit transaction several files are committed. Each of those files gets a new revision. And each of those new revisions gets the same commit message, which is a textual comment by the author.

6.1.2 Evolution Metrics

To analyze evolution of a software project the following evolution metrics were extracted:

- *Lines added*: The number of added lines gives an estimate of the extent of changes.
- *Lines deleted*: The number of deleted lines reflect a certain aspect of clean-up mentality. Code, which isn't in use any longer, may be deleted from the source code, thus reducing the number of lines in a file.
- *Lines changed*: As we pointed out before, the number of changed lines cannot be determined exactly. Instead, it is only estimated by computing the minimum of lines added and lines deleted.
- *Authors*: The number of different authors, who work on a single file, will possibly influence the bug count in this file.
- *Bugs*: The number of bugs tells us how many bugs were found in a file.
- *Refactorings*: Just like lines changed, the number of refactorings is a fuzzy metric, which is computed by analyzing the commit messages.

- *Revisions*: The number of revisions is a metric, which can change heavily over time, since development usually concentrates only on some modules. During the period, where a file is committed very often, the bugs in that file might increase.

- *Different commit messages*: The number of different commit messages might be a valuable metric to get a better idea of the number of issues of a file. For example, there might be several consecutive commit messages, which say "Updated search algorithm." This updated search algorithm might be so complicated that an author worked on it for several days. Each day the author commits the changes made to the source code and each day writes this same commit message. In the end, after several revisions, there was actually only one big change.

- *Change coupling*: Change coupling is defined as follows [RSVG07]:

> *Two entities (e.g. files) are coupled, if a modification to the implementation affected both entities.* The intensity of coupling between two entities a and b can be determined easily by counting all log groups where a and b are members of the same transaction, i.e. $C = \{\langle a, b\rangle \,|a, b \in T_n\}$ is the set of change coupling and $|C|$ is the intensity of coupling.

There is great variability in using CVS as a versioning system. But basically, even with the diversity that there is, there are some limitations. One disadvantage of CVS is the inability to record the process of renaming or moving of a file. For example, a Java class may be moved to another package or the name gets altered during refactoring. Unfortunately, this kind of refactoring cannot be recognized automatically. Instead, we would have to rely on the comments of the authors who made the refactorings. Another point that is worthy to note is that when modifying one line of a file in CVS means deleting one line and adding another one. An example of this can be found in Listing 6.1 on page 40, where we can assume that in revision 2 one single line and in revision 3 two lines were modified. Though these are some of the problems associated with using CVS, it has proved to be a very useful tool in gathering evolution information.

6.2 Methodology

In this section, the emphasis will be mainly on how input data is handled.

6.2.1 Data Extraction into Database

From the CVS server a full log file, including all the entries of the main branch only, is retrieved. The log file contains all the information we need (see Section 6.1.1. Then, the log file is parsed and the following information is extracted:

- file name
- path
- revision number
- date and time
- author name
- lines added and deleted
- comments (i.e. commit message)

As we pointed out in Section 6.1.1, the log entry of the first revision of a file in CVS doesn't contain the number of added lines. For this reason, we are not able to get the size of a file. What we get though, is the number of added and deleted lines. Since we want to find out about changes in a file, the absolute number of lines is not necessarily important for our purposes.
The data from the log file is written into an SQL database.

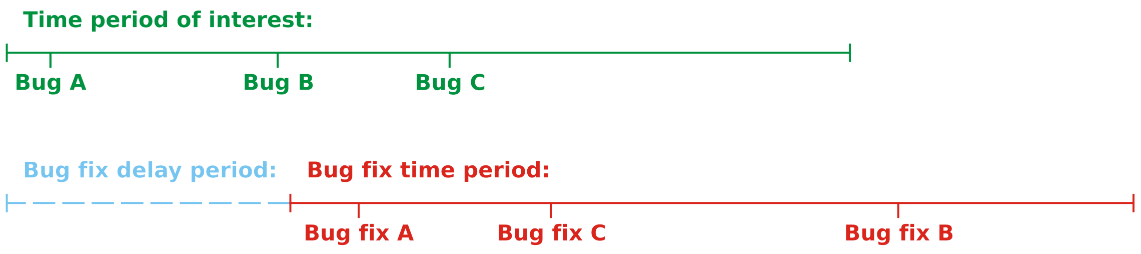

Figure 6.1: Time periods for analysis

6.2.2 Computation of Logical Coupling

The definition of change coupling (see Section 6.1.2) says that two files are coupled if a modification to the implementation affected both files. That means a coupling exists between two files if the two files were modified by one author and written back in one single commit transaction. To compute change coupling, the existing data information on the file revisions that were created within the same commit transaction is essential. Unfortunately, as stated earlier, we don't have this information and CVS does not support us in that aspect.

Metric	Visual Attribute
Number of authors	Length of cube (x axis)
Number of bugs	Color of cube
Number of different commit messages	Width of cube (y axis)
Name of class	Title of cube
Number of refactorings	Height of class (z axis)
Change coupling between two classes	Thickness of connecting edge

Table 6.1: Mapping from metrics to visual attributes

In spite of this disadvantage, we want to make a point that it is possible to recreate the information about commit transactions. One commit transaction takes a reasonable amount of time, usually several seconds or minutes. Each committed file of such a transaction gets a new file revision along with a revision date and time. So, different revisions of the same transaction will have different revision times. In addition to this, each revision of the same transaction will have the same author and the same comments (i.e. commit message). At the most, there can be one revision of each file per transaction.
Supposing that each transaction lasts 60 seconds, all the revisions within that time period that have the same author and commit message are considered to have happened within the same transaction. If another 60 seconds are provided, we can search for revisions within that new time frame with the same author and commit message. Those revisions can then be added to the transaction again. This can be done until there isn't a single revision to add. Though this procedure takes time, in the end it enables us to reconstruct the transaction entirely. This paves the way for us to advance towards the next step of determining change coupling.
To compute change coupling between two files within a specified time period, we first need to collect all the revisions of the first file, which have a revision time within the specified period. For each of these revisions, we find out if a revision exists of the second file in the respective transaction. If yes, we can proceed to increase change coupling by one[2].

6.2.3 Identifying Refactorings and Bugs

Determined only by using heuristics, refactorings can be identified within the commit messages. For example, a modification to a file is probably a refactoring if its commit message contains the word "refactor." However, it is probably not a refactoring if it contains the term "ready for refactoring." Generally, around 20 different SQL queries are used to identify refactoring out of the commit messages. The method of identifying refactoring is taken from Ratzinger et al. [RSVG07], who also proved that this method works sufficiently well.

[2]We start to count at 0: Change coupling of two files is 0, if no single transaction exists, which contains revisions of both of the two files.

Name	**Type**	**Description**
bugfix delay	`Integer`	Specifies the bug fix delay, which is used to shift the bug fix time period away from the time period of interest. See Section 6.2.4.
dialect	`String`	The SQL dialect of the database. This is *Hibernate* [RHM08] specific. For example *org.hibernate.dialect.PostgreSQLDialect.*
driver class	`String`	Specifies, which driver class to use to access the database. For example *org.postgresql.Driver.*
end date	`Date`	End date of the time period of interest.
file extension	`String`	This filter criteria is used to only look up files with the specified file extension, for example *java.*
minimal coupling	`Integer`	Another filter criteria, which is used to only retrieve classes, which have a change coupling of *minimal coupling* or higher with other classes.
minimal number of bugs	`Integer`	This filter criteria is used to reduce the retrieved classes to only show classes, which have at least *minimal number of bugs* bugs.
password	`String`	The password to access the database.
start date	`Date`	Start date of the time period of interest.
url	`String`	The URL to the location of the database.
username	`String`	The username to access the database.

Table 6.2: Properties of the `Evolizer` addon

In the same way, bug fixes are identified, as we look at words like ”fix,” ”correct,” ”problem,” ”workaround” and so on. We can assume that, per bug there is approximately one bug fix. So, it is possible to estimate the number of bugs by looking at the number of bug fixes.
A more precise way to identify bugs would be to gather additional data from a bug tracking system. However, this complicates the process of retrieving input data as it would be essential to have the log file of the versioning system and have access to a bug tracking system. Besides, this would be achievable only if the software project of interest uses a bug tracking system.

6.2.4 Time periods for Analysis

There is value in maintaining time periods for developing accurate results on data analysis. A wider period gives more data, but reduces dynamic relations between data. A shorter period reduces the overall amount of analyzed data but shows better relation between data.
For example, keeping a time period for the coding of bug fixes will lend quality to the analysis when deducing the number of bugs by counting the number of bug

Name	**Type**	**Description**
authors	`Integer`	The number of different authors who have worked on this class during the time period of interest.
bugs	`Integer`	The number of bugs, which crept into this class during the time period of interest. As we stated before, this is estimation. To be more precise, the value of this property specifies the number of (detected) bug fixes, which happened during the bug fix time period.
commit messages	`Integer`	The number of different commit messages of transactions, which include revisions of this class file during the time period of interest.
lines changed	`Integer`	The number of lines that have been changed within the time period of interest. This is the minimum of lines added and lines deleted.
name	`String`	The name of this class. This is the file name of the class file without path and extension.
package	`String`	The name of the package of this class. This information is taken from the path of the file.
refactorings	`Integer`	The number of refactorings on this class during the time period of interest.
revisions	`Integer`	This property informs us about the number of times this class has been revised during the time period of interest.

Table 6.3: Properties of the `EvoClass` addon

fixes. In Figure 6.1 on page 43, we list the scheme of the time periods we are dealing with. The top line shows the *time period of interest*. For a period of six months, we collected all metrics except *number of bug fixes*. The *bug fix time period* is delayed by the *bug fix delay*[3]. By setting the bug fix delay to 60 days, the most significant results could be achieved.

6.2.5 Filtering of Data

It is highly important to filter out irrelevant data for data analysis. In the version control system, there can be all kinds of files (text files, images, spreadsheets, etc.) that might reflect the evolution of a software system, but our primary concern is with the source code files that we consider as the relevant data. For classes to be included in the analysis they must have a minimal number of bugs and a minimal

[3]The bug fix delay period (the dashed line in the figure) itself is actually no period that is investigated. Only the bug fix delay itself, which is the length of the bug fix delay period, is of importance.

Name	Type	Description
coupling	`Integer`	This property represents the change coupling between the two entities of this relation within the time period of interest.

Table 6.4: Property of the `Coupling` addon

number of change couplings between each other. In our case studies, we use filter values for the minimal number of bugs between 3 and 5, and we chose minimal numbers of change couplings between 4 and 6.

6.2.6 Visualization Approach

After filtering out irrelevant data, we represented the remaining interesting classes as cubes and coupling between two classes as edges, where each edge connects two cubes. The visual addons (cube, edge) are taken from our *addoncollection* (`Cube`, `Cylinder`, see Section 5.5.4). For every visualization we use the same metrics. The mapping from metrics to visual attributes can be seen in Table 6.1[4].

6.2.7 Implementation

For the realization of our approach some addons were contributed to our framework. This happened in the *evolizerinput* project. There, we needed two data addons, `EvoClass` (for the class entities) and `Coupling` (for the relation between them), and an input addon, `Evolizer`.
The *evolizerinput* project is built on top of the *Evolizer*platform [RPG07].

Evolizer

This input addon uses the *Evolizer* platform to access a database, where the source data is stored. This database must have been prepared in an earlier stage (see the previous Sections 6.2.1, 6.2.2, and 6.2.3). It creates the data addons `EvoClass` and `Coupling` and fills the assigned model with those data elements. See Section 5.3 for general information on the data model.
Table 6.2 on page 45 explains the properties of this addon.

EvoClass

This `Entity` represents one class file containing metrics of the software evolution domain. The `Evolizer` input plug-in creates instances of `EvoClass`. Table 6.3 shows the properties of this addon.

[4]The higher the number of bugs the brighter the color.

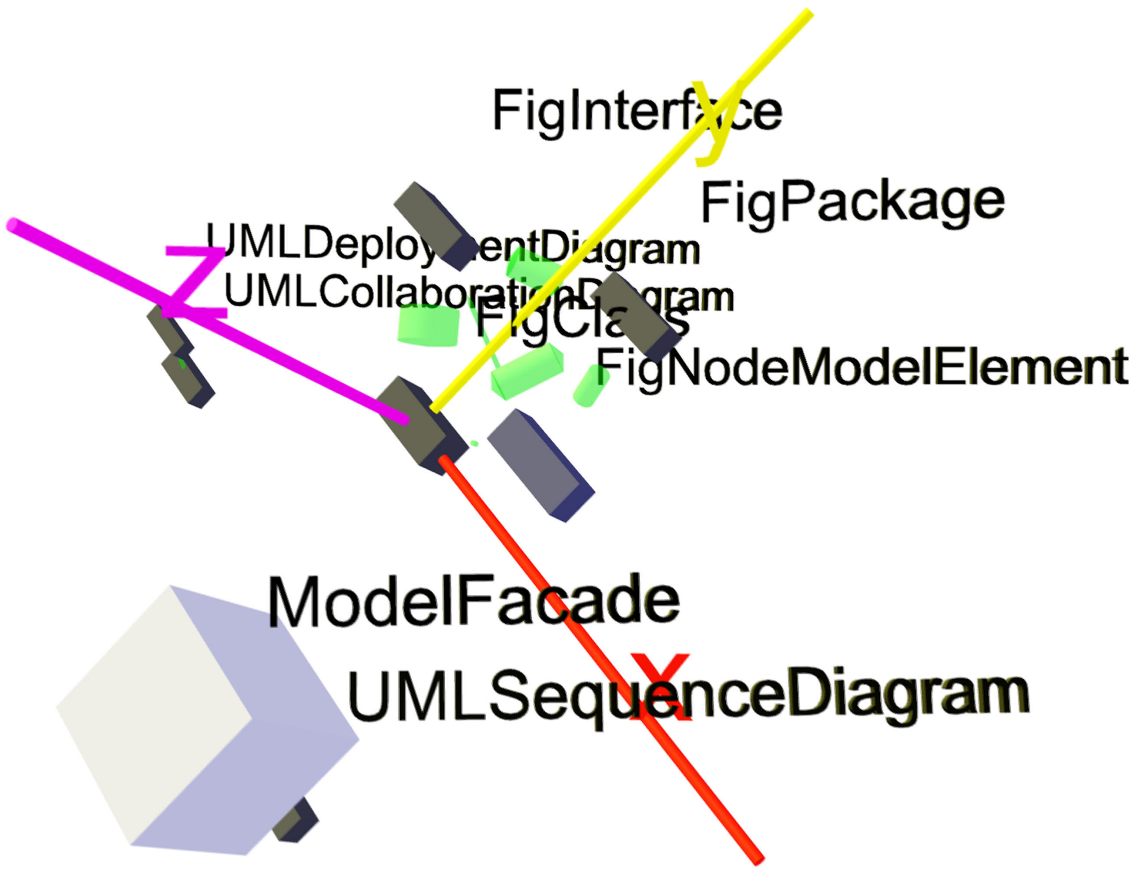

Figure 6.2: ArgoUML, min bugs=4, min coupling=6, period = 07.2003 - 12.2003

Coupling

This `Relation` represents the change coupling between two `EvoClass` addons. It has only one single property, which is described in Table 6.4. The two entities of this relation are the two `EvoClass` entities.

6.3 ArgoUML

ArgoUML [Arg08] is a UML modeling tool written in Java. Being under development since 1998, around 40 different authors have worked on approximately 4.400 java source files. This makes ArgoUML an interesting software project for analysis. To begin with, we analyzed its evolution data between 2003 and 2005. For filtering input data, we used three different criteria, namely data, number of bugs, and number of coupling. To achieve accurate results, we maintained a time period that covered classes with at least four bugs and that had a coupling of at least six. Figures 6.2 to 6.8 show the graphs for the whole analyzed period.

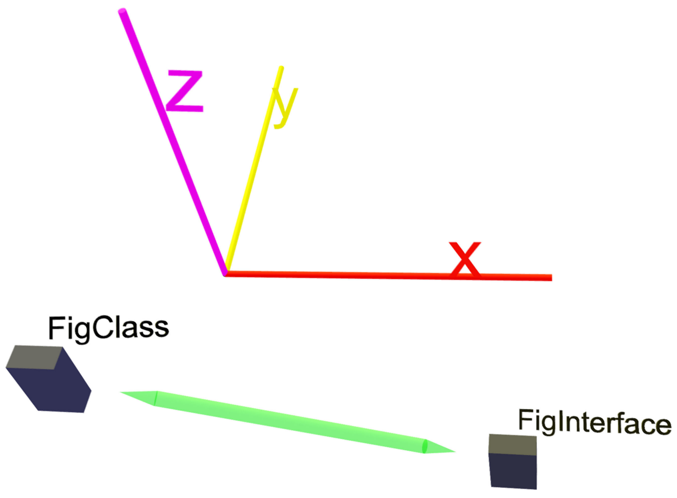

Figure 6.3: ArgoUML, min bugs=4, min coupling=6, period = 01.2004 - 06.2004

Figure 6.2 discloses that there is one problematic class, i.e., *ModelFacade*. The work of many authors on it resulted in lots of changes (refactorings and different commit messages), and many bugs.
The next period (01.2004 - 06.2004, Figure 6.3) shows very little activity. It could mean that there are only a few couplings and bugs, or more likely, no activity at all.
To get a better visualization of those two classes, we decided to implement a different layout algorithm. We quickly discovered that the LinLog algorithm didn't work well in those two classes. To overcome this, a random layout algorithm was chosen to set every node onto a random position in space. Usually this random layout algorithm would not be considered a very practical tool, but in this instance it proved to be successful by becoming the first (prototype) implementation of a *LayoutAddon*.
Figure 6.4 (07.2004 - 12.2004) shows a period of excessive work. Apart from displaying a lot of activity, the particulars of the graph are ambiguous. Presented with the dilemma of either changing our filter parameters or taking smaller snapshots, we decided to take the smaller snapshots approach. The time period was divided into two separate time intervals - the first one from 07.2004 to 09.2004 and the second one from 10.2004 to 12.2004. This served to reduce a lot of coupling between classes. This reduction is so significant that the number of details can be increased substantially. Our response was to decrease the filter to show classes with a minimum bug count of 3 and a minimal coupling of 4.
Figures 6.5 and 6.6 on pages 51 and 52 throw some light on the results. There, we have clustered graphs. In fact, such groupings are very appropriate as it separates possible sub-projects. Especially Figure 6.6 is a good example for that: The *PropPanel...* as well as the *Action...* classes are clustered altogether.
In Figure 6.7 and 6.8 on pages 53 and 54 the evolution information for the year 2005 is presented. It reveals *FigClass* as being problematic when it comes to the number of bugs. Additionally, the number of refactorings in both the periods is high. The

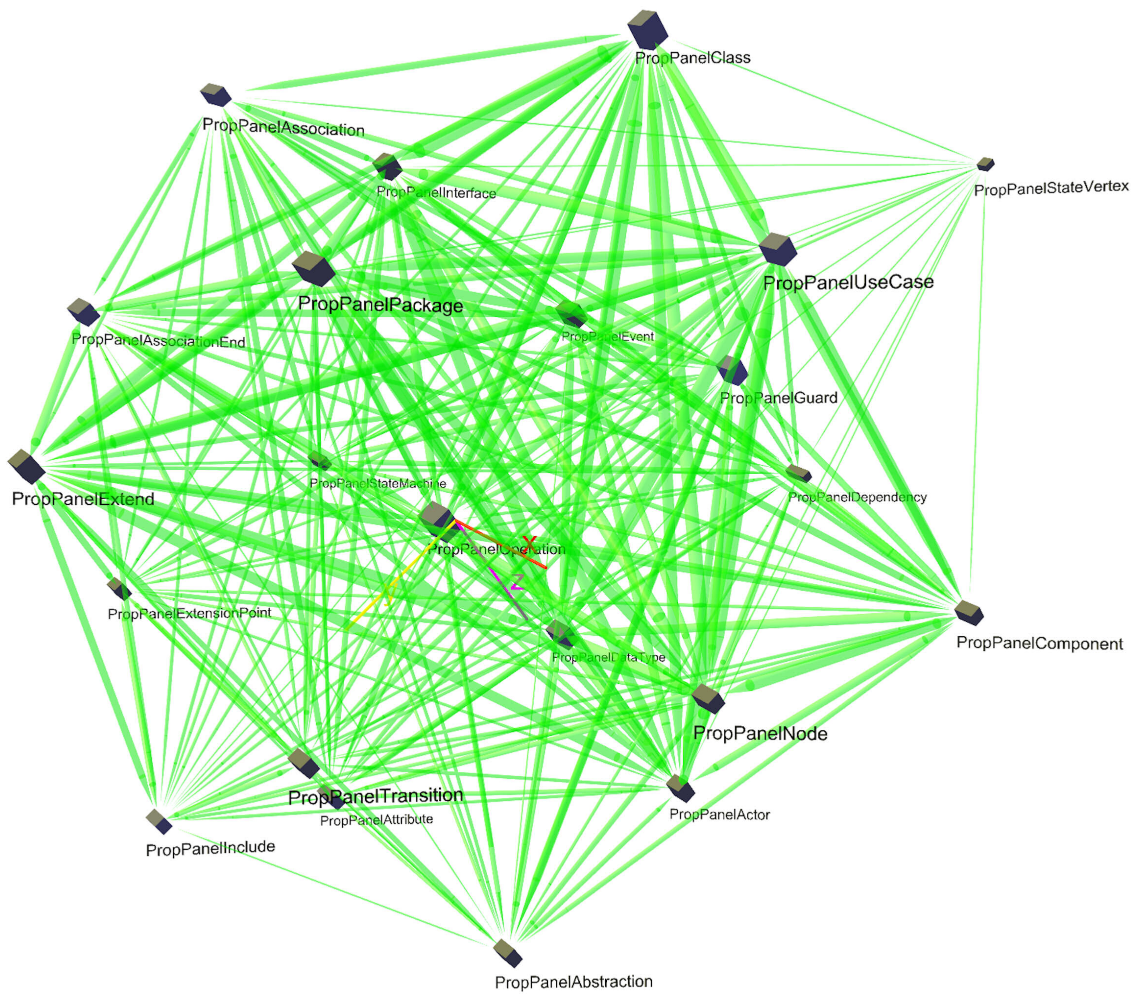

Figure 6.4: ArgoUML, min bugs=4, min coupling=6, period = 07.2004 - 12.2004

other dimensions (number of authors, number of different commit messages) are also broad with a lot of activity present in this class.
As the name denotes, the class with the highest possibilities of coupling with many other classes is *FigNodeModelElement*. Not only having an exceptionally high number of different commit messages in both time periods, this class also records a high number of authors and refactorings. In this part of the graph, in the interval from 07.2005 to 12.2005, the bug rate is by far the highest among all the classes.

6.4 Azureus

Azureus [Azu08] is a popular BitTorrent client written in Java. The project started in 2003. Since then 25 authors have worked on it. The project contains approximately 2800 java files. The time period in which we analyzed its evolution data was from 2003 to 2006. We set the minimal bug count for each class to five, and the

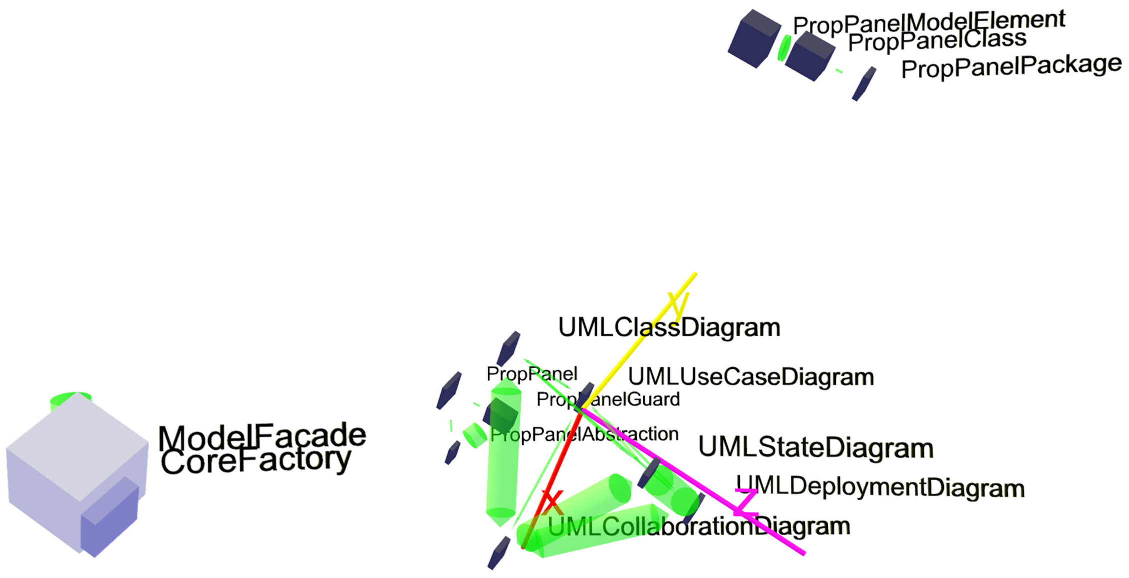

Figure 6.5: ArgoUML, min bugs=3, min coupling=4, period = 07.2004 - 09.2004

minimal coupling of two classes to five as well. Figures 6.9 to 6.15 show snapshots of the project in the analyzed time period.
Figure 6.9 on page 55 shows the first six months of the project. One class that catches your eye is *MainWindow*. A huge coupling exists between *MainWindow* and *ConfigView*, and also between *MainWindow* and *MyTorrentsView*. The number of different commit messages is notably high in the *MainWindow* class, compared to the other classes in the graph. The bug count for that class is also very high.
DiskManagerImpl has quite a high number of bugs as well, and a huge number of refactorings.
In the next time period (01.2004 to 06.2004, Figure 6.10, page 56) some more refactorings took place in the *DiskManagerImpl* class. This time, though, more interesting is the high number of bugs and the high number of different commit messages.
There are a couple of other classes/cubes with similar evolutionary data/shape and color. A handy feature of the LinLog layout algorithm is that the nodes with a high degree, like *MyTorrentsView* will be placed in the center of the graph. The connected edges of the cube denote that coupling exists with many other classes.
Figure 6.11 on page 57 gives the same impression: *DiskManagerImpl* is tightly coupled with many other classes.
The highest coupling can be seen between the two classes *CacheFileImpl* and *CacheFileManagerImpl*. Since those two nodes themselves are very small in every dimension, the coupling is emphasized. Interestingly, the number of different commit messages is low. However, the actual number of revisions, in which the classes were changed, cannot be seen in the graph. Any node in the graph can be clicked to see all its properties in a textual way. The following facts are visible:

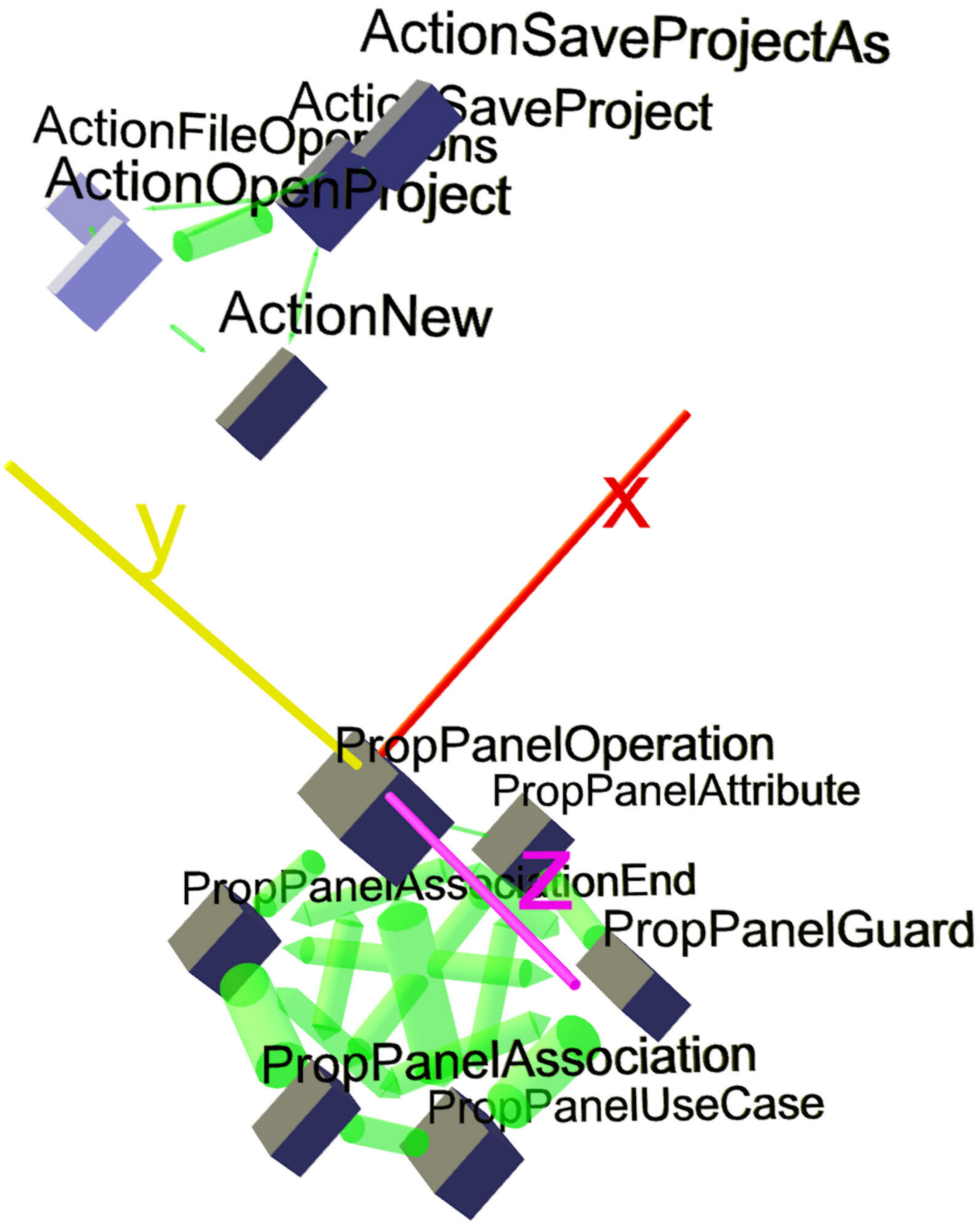

Figure 6.6: ArgoUML, min bugs=3, min coupling=4, period = 10.2004 - 12.2004

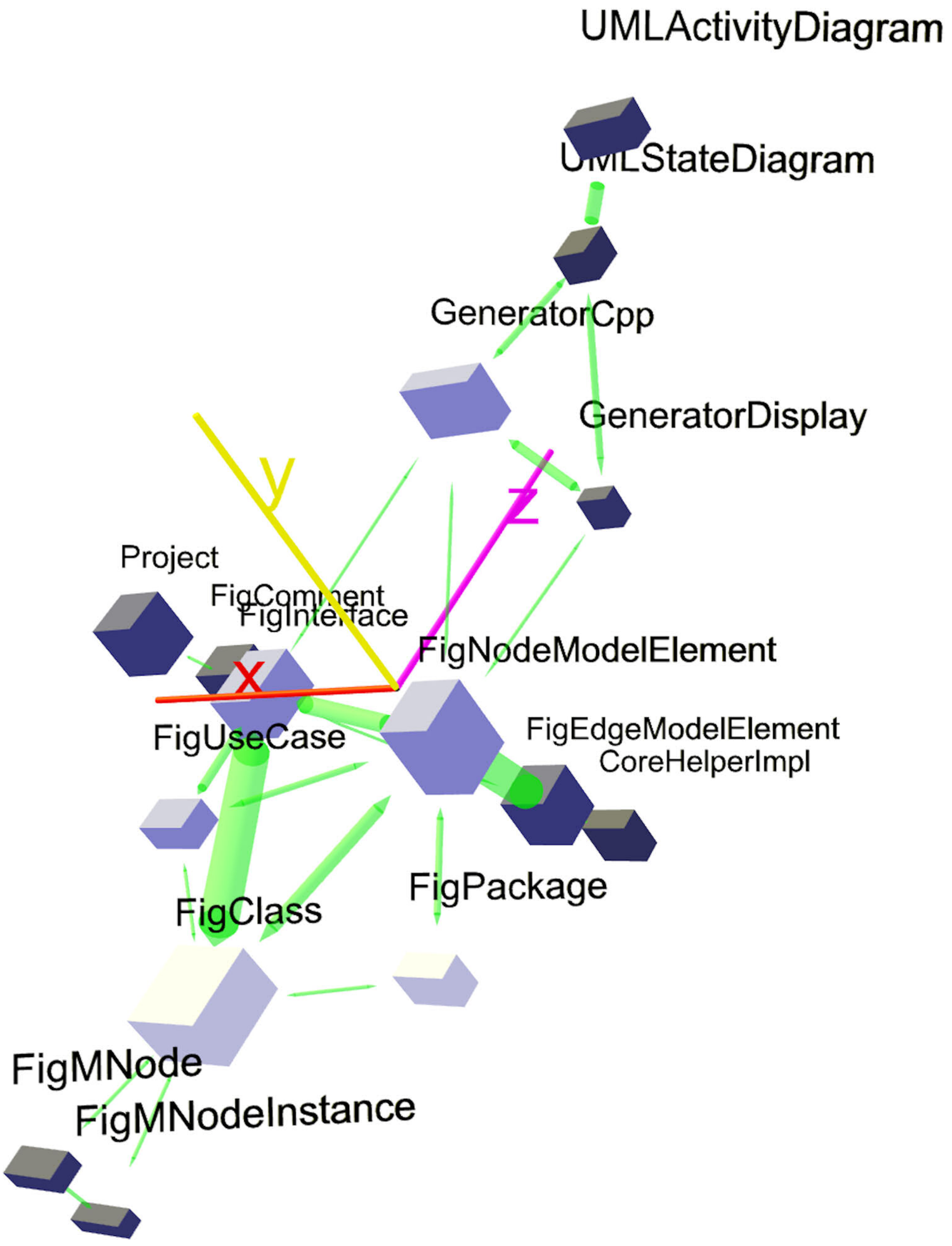

Figure 6.7: ArgoUML, min bugs=4, min coupling=6, period = 01.2005 - 06.2005

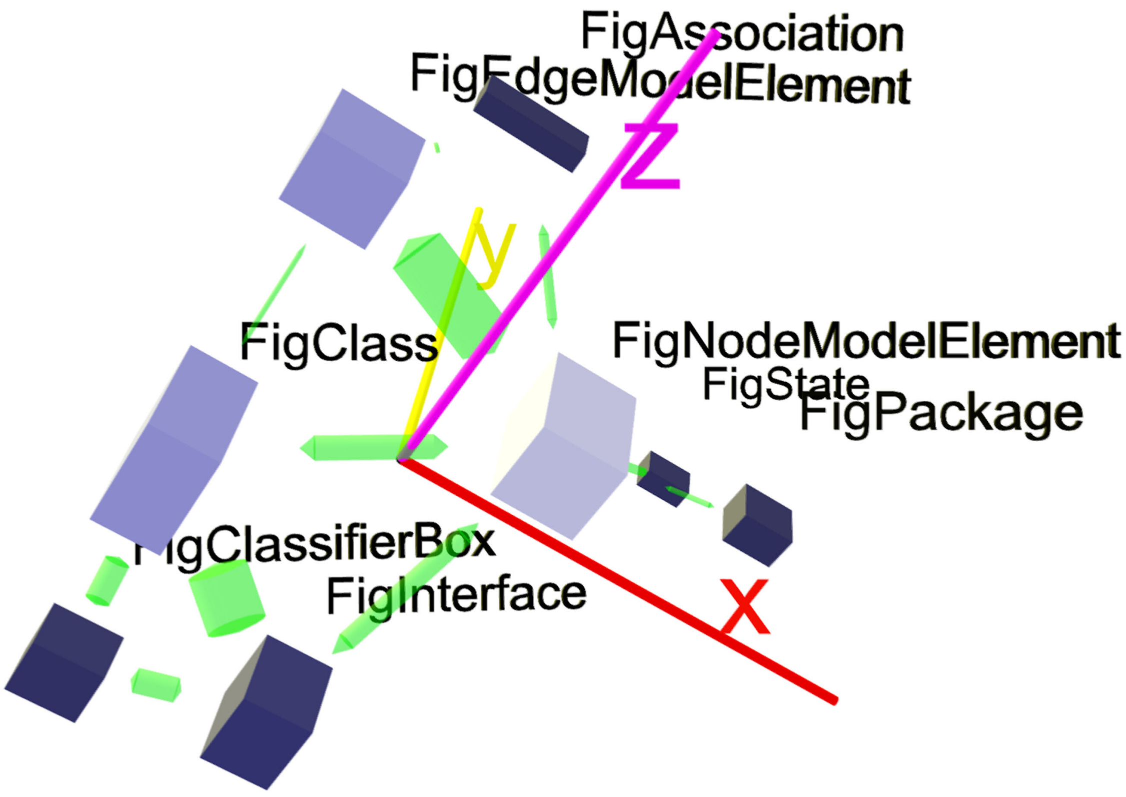

Figure 6.8: ArgoUML, min bugs=4, min coupling=6, period = 07.2005 - 12.2005

- *CacheFileImpl* has been changed in 41 revisions.
- *CacheFileManagerImpl* has been changed in 35 revisions.
- There are 25 revisions where both classes were changed at the same time.

These facts tell us that in 71% (or for *CacheFileManagerImpl* 61%, respectively) of the changes on one class the other class was changed as well. The visualized graph gives us an easy understanding of these high relative numbers.
The same effect can be seen in Figure 6.12 (01.2005 - 06.2005, page 58) for the classes *DHTControlImpl* and *DHTTransportUDPImpl*. Furthermore, only one author is working on every of the two classes. We assume that it is the same author for both classes.
In *PEPeerTransportProtocol* we see many bugs, refactorings, and commits that have taken place. We can also see heavy coupling with *PEPeerControlImpl*, another class with many different commit messages. If we look back at Figure 6.11, we can see that those two classes had a high number of bugs, refactorings, and commit messages from earlier on. In Figures 6.12 and 6.13 we come across those two classes having a lot of activity and a high number of bugs. We have not applied the actual size of a class in our visualization. Instead, we have dynamic numbers of changes in a certain

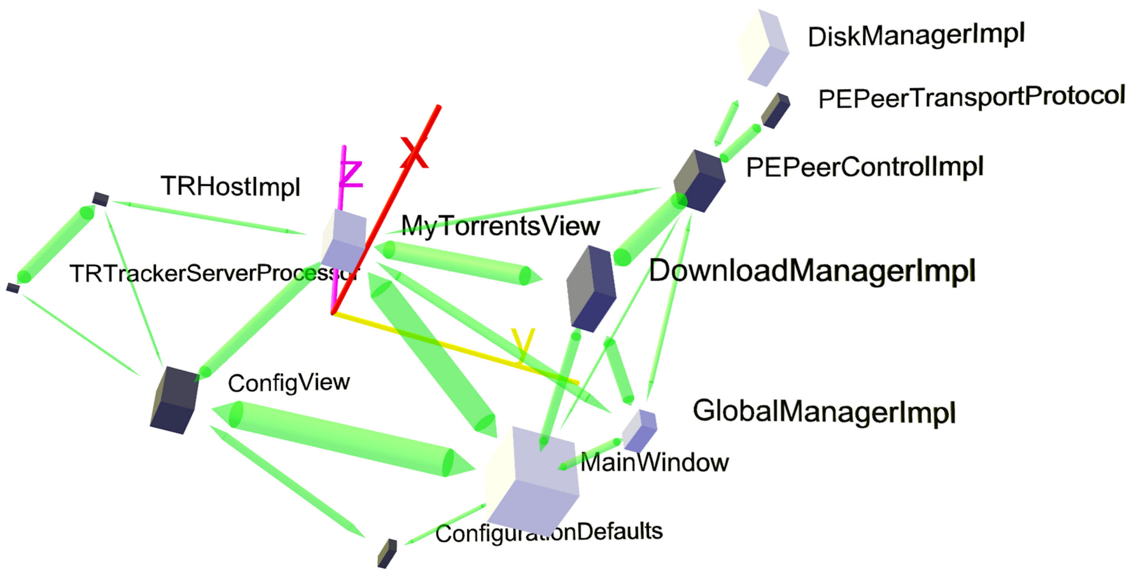

Figure 6.9: Azureus, min bugs=5, min coupling=5, period = 07.2003 - 12.2003

time interval. In Lanza's categorization of classes [Lan03], there are patterns, which may comply with the classes *PEPeerControlImpl* and *PEPeerTransportProtocol*:

- A *Pulsar* class grows and shrinks during its lifetime.
- A *White Dwarf* is a class which used to be of a certain size, but due to varying reasons lost the functionality it defined to other classes.

The two classes, *PEPeerControlImpl* and *PEPeerTransportProtocol*, change drastically during a long time interval. The nature of the change, i.e., their steady decrease or increase during the development process is not discernable. They could be either *Pulsars*or *White Dwarfs*. On exploring the possibility of the classes being *White Dwarfs*, where the bug rate mostly decreases over time, we encounter no decrease in the bug rate. This eliminates the possibility of them being *White Dwarfs*. They could also be the opposite of *White Dwarfs* - normal classes, which increase steadily. If we then examine the revisions of those classes, we find out that *PEPeerControlImpl* grew significantly during that time (which corroborates the latter assumption), while *PEPeerTransportProtocol* didn't have much growth, which makes it a *Pulsar* candidate.
The graph of Figure 6.13 on page 59 has the shape of a single *ribbon*[5] (forming a half-torus). In an ideal ribbon-shaped graph, every node (except the two outer nodes) is only connected to two other nodes. The presence of a ribbon-shaped graph signifies that there are not many couplings between classes. Additionally, there is no sign of a "god class" as defined by Riel [Rie96] as a class that performs most of the work, leaving minor details to a collection of trivial classes.

[5]For the sake of demonstrating the ribbon shape we ignore that there is a lump of nodes (including *DiskManagerImpl*) at the one end of the ribbon.

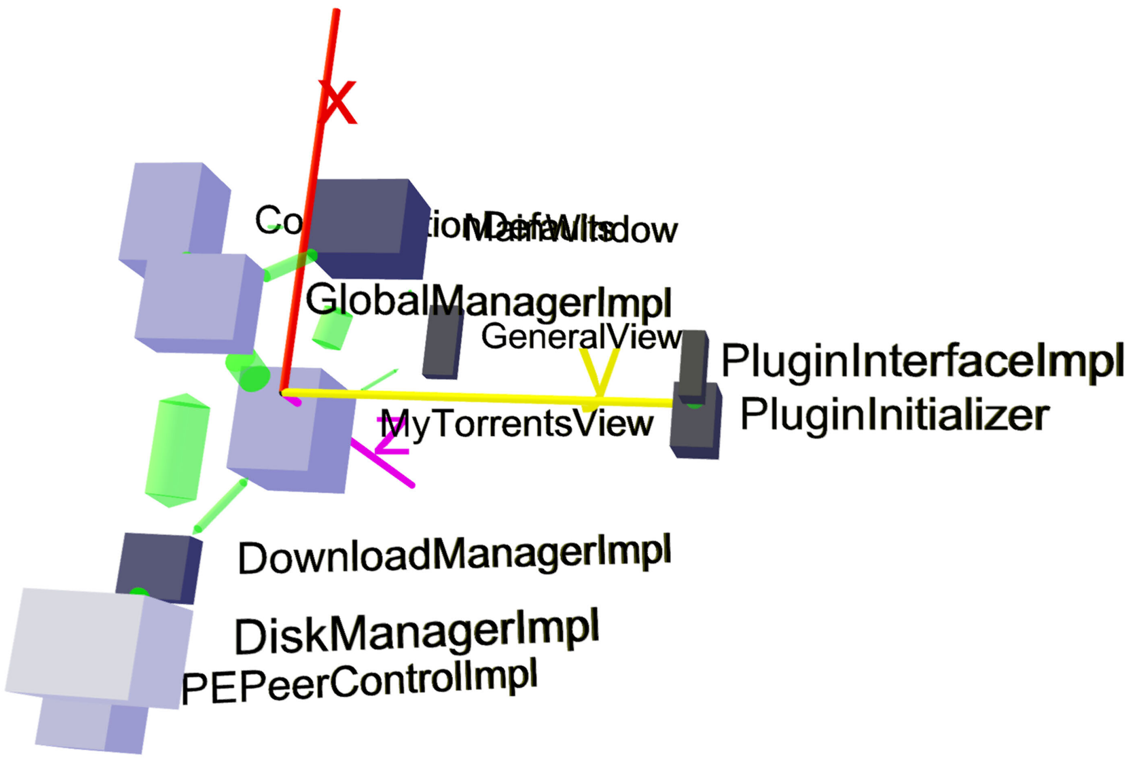

Figure 6.10: Azureus, min bugs=5, min coupling=5, period = 01.2004 - 06.2004

Figure 6.14 (01.2006 - 06.2006, page 60) shows a graph with a lot of coupling, i.e., the average degree of the nodes is high. In other words, every node is connected to many other nodes. Due to the nature of the LinLog algorithm such a graph will have the shape of a *Globe*.
There is an important point to note about the *Test* class in this figure. It couples with many other classes. This should not be taken into consideration, because for test classes it is a normal behavior to have a high coupling with other classes - this time speaking of the common definition of coupling in computer science as defined by Stevens [SMC74]: Coupling is the measure of the strength of association established by a connection from one module to another.
Figure 6.15 on page 61 shows only little activity. The *DownloadManagerImpl* class has experienced many refactorings, commit messages, and bugs. This class is constantly under development. When we look back at the figures of earlier time periods, we can find this class in almost every graph. Though it is present most of the time, it is very inconspicuous. It is constantly growing up to some point where it will possibly be split into several different classes.

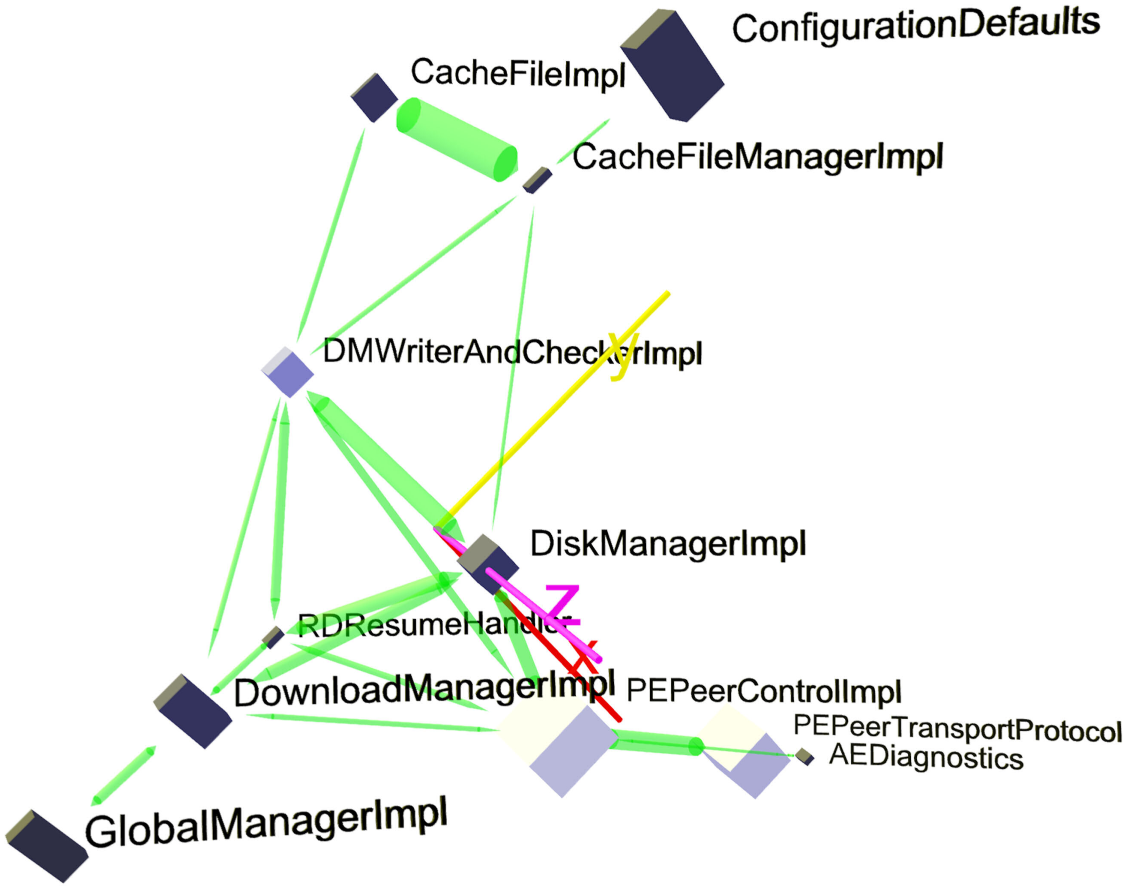

Figure 6.11: Azureus, min bugs=5, min coupling=5, period = 07.2004 - 12.2004

6.5 Results

Testing several functions of our framework on two software projects, *ArgoUML* and *Azureus*, has proven to be extremely useful. We were able to filter out problematic classes. We could recognize advantages and disadvantages of different layout algorithms.
Filtering the details out of the graph makes it more comprehensible. Another idea would be to modify an input addon in a way that it would only insert the top x nodes into a model. Instead of displaying all classes with a minimum of, say, four bugs, it would for example display the 15 buggiest classes. This way we could avoid both too small graphs (like in Figure 6.3) and too big graphs (see Figure 6.4). But, however, the drawback of such a filter is that two graphs become completely incomparable.
Problems of color and size of the graph elements exist. Since data of every graph is normalized, we cannot compare data values of different graph elements. Let's have a look at Figures 6.12 and 6.15, for example. In both graphs, there is one single class with a high number of bugs. In Figure 6.12 it is *PEPeerTransportProtocol*,

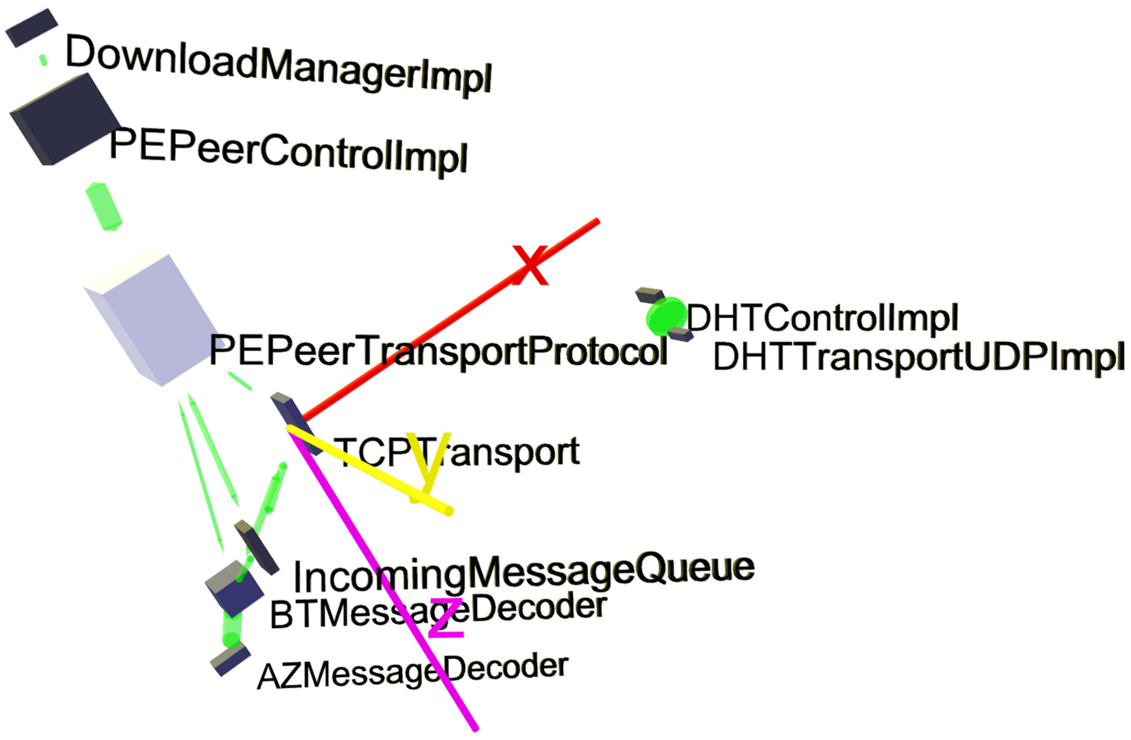

Figure 6.12: Azureus, min bugs=5, min coupling=5, period = 01.2005 - 06.2005

in Figure 6.15 *DownloadManagerImpl*. When comparing those two classes in the graphs, the reader might get the impression that they have approximately the same number of bugs. In fact, despite having the same color, *PEPeerTransportProtocol* had 19 bugs between 01.2005 and 06.2005, while *DownloadManagerImpl* had only 9 bugs between 07.2006 and 12.2006. This is a normal occurrence as every graph's properties are normalized. The minimum and maximum values of each property of the elements in a model are applied to the conversion addons for each visualization property. Due to the fact that every graph has its own model, the extreme values of the properties of two different graphs differ. We could change this behavior by manually setting extreme values for properties. This allows for changing the way the visualizations are normalized. We didn't do this in our case studies because we were more interested in temporary evolution. When comparing two graphs, the important thing to keep in mind is that the visualization uses relative values.
We could find certain patterns in graphs, which help us in understanding the evolution information:

- *Clusters*: A graph, which is split into several Clusters (we have seen this in Figure 6.6, is a sign that the project is split into several sub projects, where classes couple with classes of the same sub project but not with other classes.

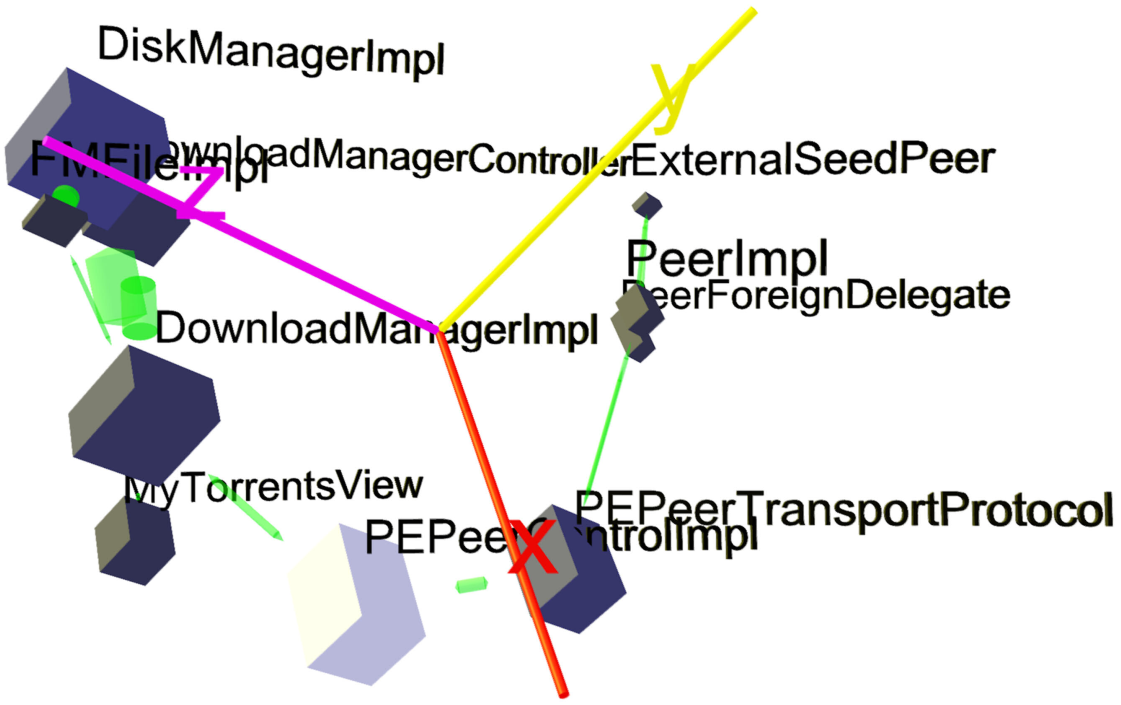

Figure 6.13: Azureus, min bugs=5, min coupling=5, period = 07.2005 - 12.2005

- *Globe*: A Globe (see Figure 6.14) is quite the opposite[6] of a clustered graph. Every class couples with every other class of the graph. Speaking of coupling, this is the exact anti-pattern of how not to develop.

- *Ribbon*: A graph, where every node (except the two outer nodes) is connected to exactly two other nodes, forming a ribbon. A ribbon-shaped graph tells us that there is little coupling between classes.

6.5.1 Comparison of related work

In this section we will compare the abilities of *J3DVN* with the projects we presented in Chapter 3. Table 6.5 summarizes the differences between those tools.

2D/3D

2D visualization is possible with almost all the tools. *J3DVN* and *sv3D* can achieve that by simply setting one dimension (either height, length, or width) to 0. Only with White Coats, 2D visualizations are not possible. Nevertheless, in addition to its 3D cubic view, it displays textual data.
Only *White Coats*, *sv3d*, and *J3DVN* can create 3D visualizations.

[6]The exact opposite of a Globe would be a graph without edges. Because only problematic classes are of interest, usually a filter is applied to only show classes where the coupling is greater than 0. That's why in the graphs of our case studies there are no nodes without any edges.

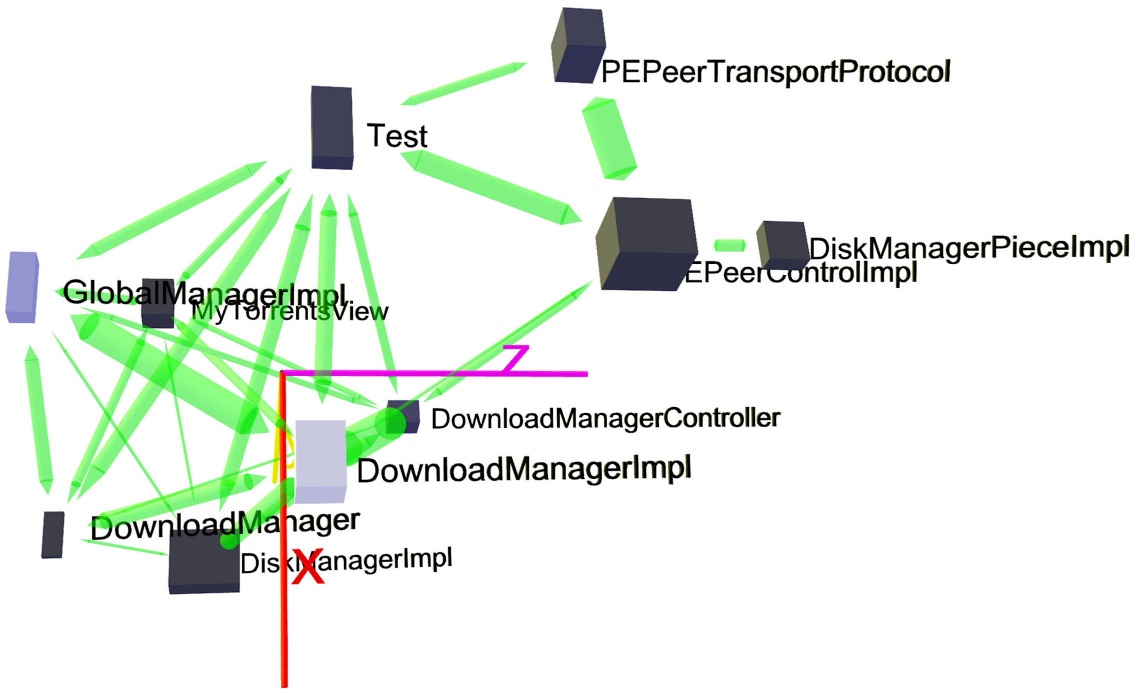

Figure 6.14: Azureus, min bugs=5, min coupling=5, period = 01.2006 - 06.2006

Visualization types

Graph-based visualizations are possible with *Rigi*, *SHriMP*, G^{SEE}, *EvoLens*, *Mondrian*, and *J3DVN*.
Other non-graph based visualizations like lines, rectangles or blocks can be done with *SeeSoft*, *SeeSys*, G^{SEE}, *VCN*, *White Coats*, *sv3D*, *Mondrian*, and *J3DVN*.
Comparing *J3DVN* with *Mondrian* regarding 2D visualizations, *Mondrian* has more possibilities like spectrographs and scatter plots. However, such visualizations could be realized with *J3DVN* by creating respective layout algorithms.

Navigation

Simple navigation (zoom and pan; rotate in 3D) is possible with most of the tools. *SeeSoft* and *SeeSys* developed the concept of showing all the information on one screen. Thus, there is no zooming and panning within those two approaches.
Rigi supports multiple views for navigation. This feature is sometimes considered as one of its drawbacks, because navigation has to happen by opening one view after the other. Although it is not the basic way of usage, *J3DVN* can use multiple views, since Java 3D allows multiple views in a universe, and since there can be more than one reference to a model in *J3DVN*.
SHriMP and *EvoLens* support a fisheye view - an advantage over *J3DVN*. They also know the concept of nested graphs, which does not exist with *J3DVN*. However, this can be achieved by creating and using more sophisticated layout addons.

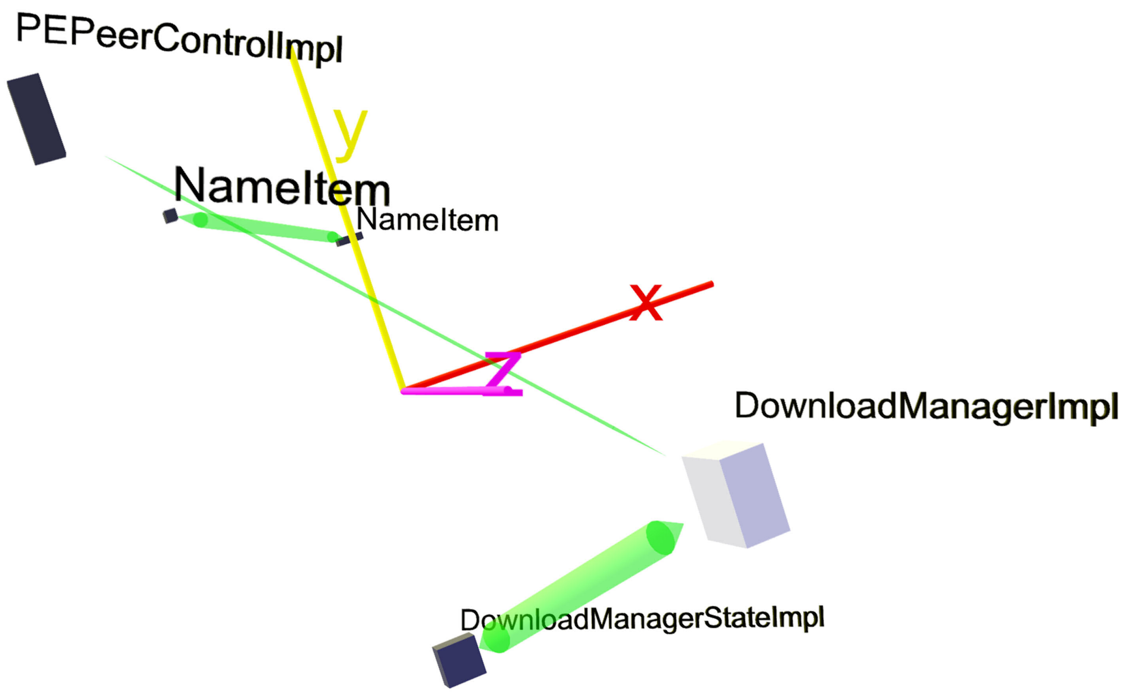

Figure 6.15: Azureus, min bugs=5, min coupling=5, period = 07.2006 - 12.2006

White Coats has some techniques that make navigation easier. One is the use of the reference cube. All information blocks are rendered within that cube. This is one way of helping the user realize the orientation of the current visualization. This reference cube is one unique feature of *White Coats*, which doesn't exist in *J3DVN*. However, it could easily be added to it, because the framework allows access to the underlying Java 3D universe. There, a wireframe cube could simply be added to the existing scene graph.
Another technique of *White Coats* is the horizon, which also helps the user to orientate within the visualization. Although in *J3DVN* an image of a virtual horizon could simply be added as a background image, this would not be the same, because rotation happens only for the visualized graph, and not for the viewer. The viewer doesn't "fly" around the graph, but he or she turns and pushes it around[7]. The orientation aid *J3DVN* uses a 3D Cartesian coordinate system in the center of the visualization.
Predefined viewpoints are another navigation aid of *White Coats*, which can't be found in any other tool. *J3DVN* could easily be extended to have such a feature. All it needs is to access the underlying Java 3D objects.

[7]In *J3DVN*, the user interface metaphor to the real world is not that of a spaceship flying through the universe. Instead, it is a human sitting on his or her desk, having an object at hand that consists of many connected nodes, and turning and moving the object around and bringing it closer to the eyes or further away from them.

Genericness

The ability to be used for many different problems (which we call "genericness") is high for *Rigi* and *SHriMP*, because both allow the use of a programming language to adapt to different problem domains.
The genericness of G^{SEE} and *J3DVN* is also high, because of their framework architecture. Functionality can be added easily.
We classify *sv3D*'s genericness as medium. It is built as a front-end independent of its data source. However, the representation abilities are limited.
Mondrian's abilities to adapt to different problem domains are very high because of two reasons. One is its framework architecture. The other one is the way it works directly on the objects, which it visualizes.
The other tools (*SeeSoft*, *SeeSys*, *VCN*, *White Coats* and *EvoLens*) are really only tools and thus tailored to specific problem domains. Their genericness, however, is low.

Filtering

Filtering possibilities exist in *Rigi* through the use of its programming language. The same is possible in *SHriMP*. Moreover, it has abilities to map input data differently to visualized objects. There even exist many filter types for live filtering.
SeeSoft allows different mappings, as well as a color slider to filter out entities below a certain level of the mapped metric[8].
G^{SEE} is programmable in such a way that only interesting data might be visualized, while other data may be filtered out.
Besides the possibility to have different mappings, *White Coats* includes a query engine that allows comfortable live filtering.
The focus of *sv3D* doesn't lie on filtering but rather on representing. Since it is only a front-end, filtering capabilities have to be implemented in the underlying content provider.
EvoLens provides live filters for a coupling threshold (to only show edges between classes/packages with a certain amount of change coupling), as well as navigation in time. The latter allows visualizing data within a user-defined time frame only.
Mondrian has scripting support, where visualization happens immediately. Live filters are not a separate issue to look at, but it is a part of the whole concept.
J3DVN allows filtering by setting values of (filter) properties of input addons. Moreover, different mappings for live filtering are built-in.
The only way to filter out data in *SeeSys* and *VCN* is to use different mappings of input data to visualization objects.

[8]For example, to show only those lines of source code files, which have been modified at least a certain number of times.

Name	**2D**	**3D**	**Visualizations**	**Navigation**	**Genericness**	**Filtering**
Rigi	Yes	No	Graph with boxes	Multiple views, zoom, pan	High through programming language	Via programming language
SHriMP	Yes	No	Graph with boxes	Fisheye views, nested graphs, zoom, pan	High through programming language	Different mappings possible, many different filter types for live filtering
SeeSoft	Yes	No	Lines	Complete info on screen, no zoom	Low	Different mappings possible, slider for 1 metric for live filtering
SeeSys	Yes	No	Cushions	Complete info on screen, 4 linked views, zoom	Low	Different mappings possible
G$^{\text{SEE}}$	Yes	No	Many	Different views	High through framework architecture	programmable
VCN	Yes	No	Cushions	3 different views	Low	Different mappings possible
White Coats	No	Yes	Blocks	Zoom, rotate, pan, predefined viewpoints, reference cube, horizon	Low	Different mappings possible, query engine for live filtering
sv3D	Yes	Yes	Containers with poly cylinders	Zoom, rotate, pan	Medium (built as a front-end independent of data source)	Dependent of data source
EvoLens	Yes	No	Graph with boxes and ellipses	Fisheye views, nested graphs, zoom, pan	Low	Coupling threshold and navigation in time for live filters
Mondrian	Yes	No	Graph, spectrograph, scatter plot, etc.	Zoom, pan	Very high through framework architecture, works directly with represented objects	script for (live) filtering
J3DVN	Yes	Yes	Graph, different layouts and elements possible	Zoom, rotate, pan	High through framework architecture	Via input addon properties, different mappings for live filtering

Table 6.5: Comparison with related work

Chapter 7

Conclusion and Future Work

Displaying data in a graphical way can facilitate comprehension of large volumes of data and detection of patterns [LRB03].
Many tools exist, which do a great job in visualizing certain types of data in a certain kind of way. The trouble with them is that they are restricted to their problem and cannot be used for anything else.
In this thesis, we proposed a generic data model that can be applied to various domains by extending it. Visualization of data is dynamic - it can be changed at any time, thus making the model flexible to use.
We created an implementation of the model. This implementation is a general tool, designed as a framework that can be integrated into the Eclipse platform. In a nutshell, it can be used for, but is not restricted to visualization of software evolution. When developing the framework one requirement was to make it as easy as possible to use it for custom visualization needs. That meant to design it in a way that additional coding would be minimal.
We added basic contributions to the framework, so it can be used rapidly for simple problems. Nevertheless, the framework is useful for complicated problems as well. We showed this by developing the *evolizerinput* project, which contributes addons for analyzing evolution of software systems.
We evaluated our model by running two case studies, in which it proved to be a useful tool in that area. The case studies showed the power of the framework, as well as its limitations.
Future work will concentrate on the following issues:

- More addons to demonstrate the genericness even better.
- Nesting of data. Right now we always only deal with exactly one data model. Nesting of data entities within other data entities is yet possible by using distinguished relation types. Still, the comprehensibility would most definitely be higher if we could use the concept of a "model within a model." Another thing that would become possible - and can't be done right now - is the ability to use different layout addons for different "clusters" of the whole model.

- Better filtering capabilities, which can be applied in real time. Right now, filters are defined in input addons only and the filter rules have to be set in input files. Using real time filters would even improve navigation of a visualized data model. The user could "jump" from one place in the model to the other and always only see the part of interest.

- Animation would add another dimension. Changes in time could easily be demonstrated to the viewer. Right now we have to take several snapshots of a software system, analyze each one separately, and compare them with each other.

- We didn't compare our 3D visualization to a 2D tool, which would make similar visualizations. The discussion whether 3D visualization is superior to 2D visualization is left open.

Appendix A

Schema of input file

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.tuwien.ac.at/j3dvn"
  xmlns:tns="http://www.tuwien.ac.at/j3dvn"
  elementFormDefault="qualified">

  <element name="j3dvn" type="tns:j3dvnType"></element>

  <complexType name="j3dvnType">
    <annotation>
      <documentation>
        The root element of every configuration file.
      </documentation>
    </annotation>
    <sequence>
      <element name="input" type="tns:inputType" maxOccurs="1"
      minOccurs="1" />
      <element name="mapping" type="tns:mappingType"
        maxOccurs="1" minOccurs="1" />
      <element name="layout" type="tns:layoutType" maxOccurs="1"
        minOccurs="1" />
      <element name="visualization"
        type="tns:visualizationType" maxOccurs="1"
        minOccurs="0">
        <annotation>
          <documentation>
            For beautification purposes there is an optional
            visualization element.
          </documentation>
        </annotation>
      </element>
    </sequence>
```

```
</complexType>

<element name="input" type="tns:inputType" />

<element name="mapping" type="tns:mappingType" />

<element name="layout" type="tns:layoutType" />

<element name="visualization" type="tns:visualizationType" />

<complexType name="inputType">
  <annotation>
    <documentation>
      The input element may contain properties, which are
      needed for the input addon to read input data.
    </documentation>
  </annotation>
  <sequence>
    <element ref="tns:property" minOccurs="0"
      maxOccurs="unbounded" />
  </sequence>
  <attribute name="name" type="string" use="optional">
    <annotation>
      <documentation>
        Specifies the input addon to use. If this attribute is
        omitted, then the input addon is chosen by the file
        extension of this file.
      </documentation>
    </annotation>
  </attribute>
</complexType>

<complexType name="layoutType">
  <annotation>
    <documentation>
      The layout element selects the layout addon to use. It
      can contain additional properties for the layout addon.
    </documentation>
  </annotation>
  <sequence>
    <element ref="tns:property" minOccurs="0"
      maxOccurs="unbounded" />
  </sequence>
  <attribute name="name" type="string" use="required">
    <annotation>
      <documentation>Name of the layout addon.</documentation>
    </annotation>
  </attribute>
</complexType>

<complexType name="mappingType">
  <annotation>
    <documentation>
      All the mappings between data addons and visual addons
      and their properties are defined here.
    </documentation>
  </annotation>
  <sequence>
    <element name="map" type="tns:mapType" minOccurs="0"
      maxOccurs="unbounded" />
  </sequence>
</complexType>

<complexType name="visualizationType">
  <annotation>
    <documentation>
      The visualization element can contain information on how
      the whole visual environment should look like.
    </documentation>
  </annotation>
  <sequence>
    <element name="background" type="tns:backgroundType"
      minOccurs="0" maxOccurs="1" />
    <element name="light" type="tns:lightType" minOccurs="0"
      maxOccurs="unbounded" />
    <element name="antialias" type="tns:antialiasType"
      minOccurs="0" maxOccurs="1" />
  </sequence>
</complexType>

<complexType name="propertyType">
  <annotation>
    <documentation>
      Any addon property. It is possible to parse strings,
      numbers, dates, and colors. If the property is of type
      color, then a color element is needed. If the property
      is a date, then the date has to be in the format
      yyyy-mm-dd.
    </documentation>
  </annotation>
  <sequence>
    <element ref="tns:color" minOccurs="0" maxOccurs="1" />
  </sequence>
  <attribute name="name" type="string" use="required">
    <annotation>
      <documentation>Name of the property</documentation>
    </annotation>
  </attribute>
</complexType>

<complexType name="mapType">
  <annotation>
    <documentation>
      Represents a mapping between a data addon and a visual
      addon.
    </documentation>
  </annotation>
  <sequence>
    <annotation>
      <documentation>
        Usually, a mapping between a data addon and a visual
        addon will not be sufficient. One wants to add
        mappings between properties of those addons.
      </documentation>
    </annotation>
    <element name="propMap" type="tns:propMapType"
      minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attribute name="data" type="string" use="required">
    <annotation>
      <documentation>Name of the data addon.</documentation>
    </annotation>
  </attribute>
  <attribute name="visual" type="string" use="required">
    <annotation>
      <documentation>Name of the visual addon.</documentation>
    </annotation>
  </attribute>
</complexType>

<complexType name="propMapType">
  <annotation>
    <documentation>
      A propMap represents a mapping between a property of a
      data addon and a property of a visual addon. The mapping
      from data to visual property happens by using the
      specified conversion addon. The data addon and the
      visual addon are specified in the parent map element.
    </documentation>
  </annotation>
  <sequence>
    <annotation>
      <documentation>
        The properties in this sequence are properties of the
        conversion addon.
      </documentation>
    </annotation>
    <element ref="tns:property" minOccurs="0"
      maxOccurs="unbounded" />
  </sequence>
  <attribute name="data" type="string" use="required">
    <annotation>
      <documentation>
        Name of the data property.
      </documentation>
    </annotation>
  </attribute>
  <attribute name="visual" type="string" use="required">
    <annotation>
      <documentation>
        Name of the visual property, to which the value of
        the data property will be mapped.
      </documentation>
    </annotation>
  </attribute>
  <attribute name="conversion" type="string" use="required">
    <annotation>
      <documentation>
        Name of the conversion addon, which will do the
        conversion from data property to visual property.
      </documentation>
    </annotation>
  </attribute>
</complexType>

<element name="property" type="tns:propertyType" />

<element name="color" type="tns:colorType" />

<complexType name="colorType">
  <annotation>
    <documentation>Defines a color value.</documentation>
  </annotation>
  <attribute name="r" type="float" use="required">
```

```
      <annotation>
        <documentation>Red value</documentation>
      </annotation>
    </attribute>
    <attribute name="g" type="float" use="required">
      <annotation>
        <documentation>Green value</documentation>
      </annotation>
    </attribute>
    <attribute name="b" type="float" use="required">
      <annotation>
        <documentation>Blue value</documentation>
      </annotation>
    </attribute>
  </complexType>

  <complexType name="backgroundType">
    <annotation>
      <documentation>Describes the background.</documentation>
    </annotation>
    <sequence>
      <element ref="tns:color" minOccurs="0" maxOccurs="1">
        <annotation>
          <documentation>
            Background color. Use either this or the image
            attribute.
          </documentation>
        </annotation>
      </element>
    </sequence>
    <attribute name="image" type="string" use="optional">
      <annotation>
        <documentation>
          File name and path of an image file, which will be
          used as background.
        </documentation>
      </annotation>
    </attribute>
    <attribute name="scaling" type="tns:scalingType"
      use="optional">
      <annotation>
        <documentation>
          Specifies how the background image should be scaled.
          If no image attribute exists, then this attribute is
          ignored.
        </documentation>
      </annotation>
    </attribute>
  </complexType>

  <simpleType name="scalingType">
    <list itemType="string">
      <enumeration value="all"></enumeration>
      <enumeration value="max"></enumeration>
      <enumeration value="min"></enumeration>
      <enumeration value="none"></enumeration>
      <enumeration value="center"></enumeration>
      <enumeration value="repeat"></enumeration>
    </list>
  </simpleType>

  <complexType name="lightType">
    <annotation>
      <documentation>
        There can be arbitrary many lights in one visualization.
      </documentation>
    </annotation>
    <sequence>
      <element ref="tns:color" minOccurs="0" maxOccurs="1">
        <annotation>
          <documentation>
            Color of the light. If this element is omitted,
            then white is used as color.
          </documentation>
        </annotation>
      </element>
      <element name="direction" type="tns:directionType"
        minOccurs="0" maxOccurs="1">
        <annotation>
          <documentation>
            This element is only used if type is directional. If
            it is omitted, then a default vector (1, -1, 0) is
            used as the light''s direction.
          </documentation>
        </annotation>
      </element>
    </sequence>
    <attribute name="type" type="tns:lightTypeType"
      use="required">
      <annotation>
        <documentation>
          The type of the light. This is either ambient or
```

```
          directional. If directional is used, then the
          direction element can be used.
        </documentation>
      </annotation>
    </attribute>
  </complexType>

  <simpleType name="lightTypeType">
    <list itemType="string">
      <enumeration value="ambient"></enumeration>
      <enumeration value="directional"></enumeration>
    </list>
  </simpleType>

  <complexType name="directionType">
    <attribute name="x" type="float" use="required" />
    <attribute name="y" type="float" use="required" />
    <attribute name="z" type="float" use="required" />
  </complexType>

  <complexType name="antialiasType">
    <annotation>
      <documentation>
        Specifies whether antialiasing should be used for the
        visualization or not. If this element is omitted, then
        no antialiasing will be used.
      </documentation>
    </annotation>
    <attribute name="value" type="boolean" use="required" />
  </complexType>
</schema>
```

Appendix B

How to contribute to the framework

Create Eclipse Plug-in

First of all, create a new, empty Plug-in project in Eclipse. It must contain an activator class. Add `at.ac.tuwien.j3dvneclipse` and `at.ac.tuwien.j3dvn` to the list of required plug-ins. In this project you can start to create your addons now. Every addon must be an extension element to an existing extension point. Table B.1 shows the straightforward way of what to add where.

Addon type	**Extension**	**Extension Element**
Conversion	at.ac.tuwien.j3dvneclipse.conversions	`conversion`
Data	at.ac.tuwien.j3dvneclipse.data	`data`
Input	at.ac.tuwien.j3dvneclipse.inputs	`input`
Layout	at.ac.tuwien.j3dvneclipse.layouts	`layout`
Visual	at.ac.tuwien.j3dvneclipse.visuals	`visual`

Table B.1: Properties of the visual addons from the addoncollection project

Addon commonality

Every addon type must implement the `getName()` method, which must return the name of the addon type as a string. The following is an example of such a `getName()` method:

```
public String getName() {
  return "my conversion addon";
}
```

Conversion Addon and Property

A conversion addon must have two generic types, which are the input and the output type of the addon. For example, the head declaration of a conversion addon, which converts from a `Long` value to a `Double` value looks like this:

```
public class LongToDouble implements
  ConversionAddon<Long, Double>
```

Note that you must not convert from or to primitive types like **`long`** or **`int`**. Instead you have to use the class types `Long` or `Integer`.
Next, it needs the `setMinMax()` method. It is up to the developer whether this method is genuinely implemented or only left as an empty method, which ignores the feature of setting an input value range. An example implementation could look like this:

```
private Long min = null;
private Long max = null;

public void setMinMax(Long min, Long max) {
  // Check for legal input values:
  if ((min != null) && (max != null) && (max - min >
    Double.MIN_VALUE)) {
    this.min = min;
    this.max = max;
  }
}
```

Now it's time to get acquainted with the process of adding a property. Every addon type can have properties. A useful property for a conversion addon with a numerical output type could be a multiplication factor that is applied to every output value to amplify output (if the multiplication factor is greater than 1). So, let's create a property of type `Double`:

```
private Double fMultiFactor = 1.;

private IProperty<Double> pMultiFactor =
  new IProperty<Double>() {
  public Double getValue() {
    return fMultiFactor;
  }

  public void setValue(Double value) {
    fMultiFactor = value;
  }
};
```

```
@Property("multiplication factor") public
  IProperty<Double> multiFactor() {
  return pMultiFactor;
}
```

First of all, we need a variable where we will store the property value. We call this variable `fMultiFactor` and give it a default value of 1. Next, we define a variable called `pMultiFactor`. Its type is a new `IProperty` interface of the generic type `Double`. The `getValue()` method simply returns the value of `fMultiFactor` and the `setValue()` method simply sets the value of the `fMultiFactor` variable. Finally we have to create a public method, which gives access to the property. We call this method `multiFactor()`. The return type has to be `IProperty<Double>`, since it gives access to a property of type `Double`. The name of this method is not very important, actually, since it will never be called manually, but only through the addon manager. Important, however, is the annotation `@Property` and its value, which is the name of the property. We call this property *multiplication factor*. Our method `multiFactor()` does nothing but return the `IProperty` variable `pMultiFactor`. Now our addon has a property. This is the general way of adding a property to an addon. Creation of properties becomes even easier, if you define a code template in Eclipse that looks like this:

```
private ${property_type} f${field_name};

private final IProperty<${property_type}> p${field_name} =
  new IProperty<${property_type}>() {
  public ${property_type} getValue() {
    return f${field_name};
  }

  public void setValue(${property_type} value) {
    f${field_name} = value;
  }
};

@Property(${property_name})
public IProperty<${property_type}> ${field_name}() {
  return p${field_name};
}
```

But now back to conversion addon specifics. You need to implement the `convert()` method. Here, the actual work of the addon will happen. This is an example of such a method:

```
public Double convert(Long inputValue) {
  Double normValue;
  // Check for illegal input value:
  if (inputValue == null)
    normValue = 0.;

  // Check if input range has been set. If yes, then
  // perform normalization, so that inputValue will be
  // in the range [0, 1]:
  else if ((min != null) && (max != null)) {
    normValue = (inputValue - min) / (max - min);
  }

  // No input range has been set:
  else
    normValue = inputValue.doubleValue();

  // Multiply output value with multiplication factor:
  return normValue * fMultiFactor;
}
```

The comments in the listing explain this method. Please note that in the last line we use the value of our property *multiplication factor*. Now we have all the necessary methods for a conversion addon - that's all it needs.

Data Addon

Creating a data addon is even easier than creating a conversion addon. Data addons only need to implement the `getName()` method - just like every addon needs to. Besides that they only need to have defined properties. You create a property for a data addon just like for any other addon. So, the example property of our conversion addon from the previous section can be used to create a data addon as well.

Input Addon

An input addon needs a method `getFileTypes()`, which returns a value of type `String[]`. We encourage you not to use this method but only return **`null`**. This method could be used to set file extensions for which the input addon can be used. Because this is quite a complicated process it is highly discouraged. Instead, we recommend to stick to the concept that input files always have to have the extension *.j3dvn*. Nevertheless the method `getFileTypes()` has to be implemented in some way.
The next easy method to implement is `setModel()`. This method is called by the connector, which will contain the input addon. It is only needed to set a reference

to the model variable, which the input connector created. The input addon will populate this model in its `open()` method. So, the `setModel()` method will usually only look like this:

```
private Model model;

public void setModel(Model model) {
  this.model = model;
}
```

First we need to define a private variable of type `Model`. We don't need to initialize it and we also don't need to do error checking in the `setModel()` method, because anyway any method will only be called by the input connector. And we can be assured that the connector will definitely call the `setModel()` method right after the addon has been created.
Finally we need an `open()` method, which will do all the work of reading and parsing input data. This method takes an `InputStream` as parameter, which will be an open input stream on the input file. Input data can come from the input file. In that case it must be somewhere within the *input* tag of the xml file. If input data cannot be stored in the input file (for example because it is binary data or data is retrieved from a database), then the input plug-in would ideally have properties, which will retrieve needed information to access the actual input data. We will not give a full example implementation of an `open()` method but sketch the most important parts:

```
public void open(InputStream source) {
  while (String value = getInput(source) != null) {
    MyEntity = new MyEntity();
    EntityConnector entityConnector = AddonManager.
      getInstance().createConnector(newEntity);
    entityConnector.setProperty("my property", value);
    model.addEntity(entityConnector);
  }
}
```

We assume that there is an imaginary method `getInput()`, which will return a string value for every entity. After the last entity it would return **`null`**. To create a new entity, we use the addon manager. First, we simply create a new instance of a data addon, in our case it is an imaginary entity type called `MyEntity`. Next, we call the static method `getInstance()` of `AddonManager` to access the singleton instance of the addon manager. Then we call its `createConnector()` method with the newly created `MyEntity` variable. This method can only return a proper result (in this case: a result of type `EntityConnector`), if we pass an entity, which is not **`null`**. That's why we had to create the addon manually before. The addon manager now creates a new entity connector, assigns the entity to the entity connector, and returns it. Now we have an `EntityConnector` variable, which can be used to access the properties of the entity. We call the connector's `setProperty()` method and we set an imaginary property called *my property* to `value`. This property needs to be of type `String`, otherwise an exception will be thrown when trying to set its value to a string. So, at this point it is up to the developer of the input addon to only pass correct values to the property of a data addon. Finally we need to add the new entity to the model. This happens by calling the model's `addEntity()` method. For simplicity reasons we left out creation and adding of relations.

Layout Addon

Next, we will create a layout addon. First of all we need a `getNodes()` method, which will return all the nodes, of which the position is calculated:

```
private final Collection<Position> nodes =
  new ArrayList<Position>();
public Collection<Position> getNodes() {
  return nodes;
}
```

We need a variable to store all the nodes first. It needs to be a collection of `Position` objects. We create it immediately and for security reasons we declare it as **`final`**, because we return the variable directly in the `getNodes()` method. What is the `nodes` variable good for? Well, it becomes useful after implementing the `addNode` method, which is responsible for adding nodes to the list of nodes. We also need a `removeNode()` method to remove a node again:

```
public void addNode(Position node) {
  nodes.add(node);
}

public boolean removeNode(Position node) {
  return nodes.remove(node);
}
```

Next, we need to give the ability to change the edge (i.e. relation) type, which will be used for layout calculation. Whether the edge type will be ignored or not is up to the layout addon developer. The getter and setter method must at least exist:

```
  private Class<Relation> edgeType = null;

  public void setEdgeType(Class<Relation> edgeType) {
    this.edgeType = edgeType;
  }

  public Class<Relation> getEdgeType() {
    return edgeType;
  }
```

In this example we use the edge type. We declare a variable, in which the edge type will be stored, and we create getter and setter methods for it.
Finally we need to implement the `calculate()` method, which contains the layout algorithm. In our example we use a very simple random layout "algorithm," which only sets every node to a random position:

```
public void calculate() {
  Random random = new Random();
  for (Position node : nodes) {
    node.x = 10 * (random.nextDouble() - .5);
    node.y = 10 * (random.nextDouble() - .5);
    node.z = 10 * (random.nextDouble() - .5);
  }
}
```

Visual Addon

The last addon type we need is the visual addon type. First of all we implement the methods to access the assigned data connector:

```
private DataConnector data;

public DataConnector getModelData() {
  return data;
}

public void setModelData(DataConnector data) {
  this.data = data;
}
```

So far, so good. Next we also need a `getGroup()` method, which returns the transform group to our visual object:

```
private final TransformGroup transformGroup;

public Group getGroup() {
  return transformGroup;
}
```

Because the transform group is returned directly, we declare the private variable as **`final`**.
Since `VisualAddon` extends `PropertyChangeListener`, we also need to implement `propertyChanged()`. We are not going to react to any change event, so we only add an empty method:

```
  public <T> void propertyChanged(Connector sender,
    String propertyName, T oldValue, T newValue) { }
```

We certainly could need some properties to set the appearance of the visual object. However, for simplicity reasons we'll leave that out. What we still need, though, is a constructor. Here lies one difference of visual addons compared to the other addon types: A constructor is needed, where the visual object is created. We show a minimal example of a visual addon constructor:

```
public MyBox() {
  Box box = new Box();
  transformGroup = new TransformGroup();
  transformGroup.addChild(box);
}
```

Our visual addon type's name is *MyBox*. It creates a simple box[1]. The group object is created and the box is added as a child to the group.

[1]`Box` is a class, which comes with Java 3D but is not part of the API.

Bibliography

[Arg08] ArgoUML Project. Argouml. `http://argouml.tigris.org/`, 2008.

[Art88] Lowell Jay Arthur. *Software evolution: the software maintenance challenge*. Wiley-Interscience, New York, NY, USA, 1988.

[Azu08] Azureus Project. Azureus. `http://azureus.sourceforge.net/`, 2008.

[Bae81] Ronald Baecker. Sorting out sorting. 30 minute color film (developed with assistance of Dave Sherman, distributed by Morgan Kaufmann, University of Toronto), 1981.

[BE95] Marla J. Baker and Stephen G. Eick. Space-filling software visualization. *Journal of Visual Languages and Computing*, 6(2):119–133, 1995.

[BF03] Moshe Bar and Karl Fogel. *Open Source Development with CVS*. Paraglyph Press, 2003.

[BNDL04] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz. Software landscapes: Visualizing the structure of large software systems. In *VisSym 2004, Symposium on Visualization*, pages 261–266. Eurographics Association, May 2004.

[Com08] Computer Human Interaction & Software Engineering Lab (CHISEL). Creole - eclipse plug-in. `http://www.thechiselgroup.org/creole`, 2008.

[Die07] Stephan Diehl. *Software Visualization*. Springer, 2007.

[Ecl08] The Eclipse Foundation. Eclipse. `http://www.eclipse.org/`, 2008.

[ESS92] S. G. Eick, J. L. Steffen, and E. E. Sumner. Seesoft- a tool for visualizing line-oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, Nov 1992.

[Fav01] Jean-Marie Favre. G^{SEE}: a generic software exploration environment. In *Proceedings of the 9th International Workshop on Program Comprehension*, pages 233–244. IEEE, 2001.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience*, 21(11):1129–1164, 1991.

[Fre08] Free Software Foundation, Inc. Data display debugger. `http://www.gnu.org/software/ddd/`, 2008.

[GB03] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins.* Addison-Wesley, 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[GRR99] Martin Gogolla, Oliver Radfelder, and Mark Richters. Towards three-dimensional animation of UML diagrams. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723, pages 489–502. Springer, 1999.

[GS94] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

[GYB04] Hamish Graham, Hong Yul Yang, and Rebecca Berrigan. A solar system metaphor for 3d visualisation of object oriented software metrics. In *APVis '04: Proceedings of the 2004 Australasian symposium on Information Visualisation*, pages 53–59, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[Har88] David Harel. On visual formalisms. *Commun. ACM*, 31(5):514–530, 1988.

[IEE98] IEEE standard for a software quality metrics methodology. *IEEE Std 1061-1998*, 1998.

[Int97] International Organization for Standardization. *ISO/IEC 14772-1:1997: Information technology – Computer graphics and image processing – The Virtual Reality Modeling Language – Part 1: Functional specification and UTF-8 encoding.* International Organization for Standardization, Geneva, Switzerland, 1997.

[KBK04] A. Kerren, F. Breier, and P. Kugler. Dgcvis: an exploratory 3d visualization of graph pyramids. In *Proceedings of the Second International Conference on Coordinated and Multiple Views in Exploratory Visualization, 2004.*, pages 73–83, 2004.

[Kof35] Kurt Koffka. *Principles of Gestalt Theory.* Harcourt, Brace, 1935.

[Lan03] Michele Lanza. *Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization.* PhD thesis, University of Bern, 2003.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Trans. Softw. Eng.*, 29(9):782–795, 2003.

[LNVT05] Gerard Lommerse, Freek Nossin, Lucian Voinea, and Alexandru Telea. The visual code navigator: An interactive toolset for source code investigation. In *INFOVIS '05: Proceedings of the 2005 IEEE Symposium on Information Visualization*, page 4, Washington, DC, USA, 2005. IEEE Computer Society.

[LRB03] Michael D. Lee, Rachel E. Reilly, and Marcus E. Butavicius. An empirical evaluation of chernoff faces, star glyphs, and spatial visualizations for binary data. In *APVis '03: Proceedings of the Asia-Pacific symposium on Information visualisation*, pages 1–10, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[MFM03] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3d representations for software visualization. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 27–ff, New York, NY, USA, 2003. ACM Press.

[MGL06] Michael Meyer, Tudor Girba, and Mircea Lungu. Mondrian: an agile information visualization framework. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 135–144, New York, NY, USA, 2006. ACM Press.

[MK88] H. A. Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 80–86, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[ML05] Cédric Mesnage and Michele Lanza. White coats: Web-visualization of evolving software in 3d. In *VISSOFT '05: Proceedings of the 2005 IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 40–45, 2005.

[Noa06] Andreas Noack. Energy-based clustering of graphs with nonuniform degrees. In Patrick Healy and Nikola S. Nikolov, editors, *Graph Drawing, Limerick, Ireland, September 12-14, 2005*, pages pp. 309–320. Springer, 2006.

[Obj08] The Object Management Group. Unified modelling language. `http://uml.org/`, 2008.

[PCJ96] Helen C. Purchase, Robert F. Cohen, and Murray James. Validating graph drawing aesthetics. In *GD '95: Proceedings of the Symposium on Graph Drawing*, pages 435–446, London, UK, 1996. Springer-Verlag.

[Ree73] Trygve Reenskaug. Administrative control in the shipyard. In *ICCAS conference, Tokyo, 1973*, 1973.

[RFG05] J. Ratzinger, M. Fischer, and H. Gall. Evolens: Lens-view visualizations of evolution data. In *Eighth International Workshop on Principles of Software Evolution*, pages 103–112, Dec 2005.

[RHM08] LLC. Red Hat Middleware. Hibernate. `http://www.hibernate.org/`, 2008.

[Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics.* Addison-Wesley, 1st edition, 1996.

[RPG07] Jacek Ratzinger, Marting Pinzger, and Harald Gall. Eq-mine: Predicting short-term defects for software evolution. *Proceedings of the European Joint Conferences on Theory and Practice of Software (ETAPS'07): Fundamental Approaches to Software Engineering (FASE 2007)*, pages 12–26, 2007.

[RSVG07] Jacek Ratzinger, Thomas Sigmund, Peter Vorburger, and Harald Gall. Mining software evolution to predict refactoring. *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 2007.

[SB92] Manojit Sarkar and Marc H. Brown. Graphical fisheye views of graphs. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 83–91, New York, NY, USA, 1992. ACM Press.

[SM95] M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using SHriMP views. In *Proceedings of the 1995 International Conference on Software Maintenance*, pages 275–284, Oct 1995.

[SMC74] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[Sun08] Sun Microsystems, Inc. Java3d. `https://java3d.dev.java.net/`, 2008.

[TL01] M. Tavanti and M. Lind. 2d vs 3d, implications on spatial memory. In *INFOVIS '01: Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS'01)*, page 139, Washington, DC, USA, 2001. IEEE Computer Society.

[Tuf90] Edward Tufte. *Envisioning Information.* Graphics Press, 1990.

[WC99] U. Wiss and D. A. Carr. An empirical study of task support in 3d information visualizations. In *Information Visualization*, pages 392–399, 1999.

[WCJ98] U. Wiss, D. A. Carr, and H. Jonsson. Evaluating three-dimensional information visualization designs: A case study of three designs. In *Proceedings of 1998 IEEE Conference on Information Visualization, IV'98*, pages 137–144, Jul 1998.

[WF94] C. Ware and G. Frank. Viewing a graph in a virtual reality display is three times as goodas a 2d diagram. In *Proceedings of the IEEE Symposium on Visual Languages, 1994*, pages 182–183, Oct 1994.

[WHF93] C. Ware, D. Hui, and G. Franck. Visualizing object oriented software in three dimensions. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 612–620. IBM Press, 1993.

[WL07] Richard Wettel and Michele Lanza. Program comprehension through software habitability. In *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 231–240, Washington, DC, USA, 2007. IEEE Computer Society.