Department of Computer Science, University of Leicester

CO7201 INDIVIDUAL PROJECT

Visual Zooming over Software Source Code

[SharpVisualizer]

By

SIVABALAN SUTHAKARAN

CFS ID: ss532@le.ac.uk

Student No: 089010150

MSc Dissertation

Project Supervisor: Dr. Artur Boronat

Second Marker: Dr.Laura Bocchi

Submitted On: 21-05-2010

Abstract

Visual Zooming over Software source code is a technique that supports software professionals to understand the existing legacy software artefacts. Visual zooming over a source code is generally a Software Visualization (SV) techniques, relevant facts are extracted as graph based data from source code using general or more specific fact extractors and render the retrieve data using appropriate graph rendering tools. But different task oriented SV tools need different approaches. There is no standard way for developing SV tools.

To proposed method to provide visual zooming for software source code, I approach the problem in two ways; firstly, I study in detail about existing SV techniques and tools available and identify the core requirement of such systems and the way they have been implemented. Then defines the requirements of the tool to be developed and propose a solution based on new standards and techniques. Objects Management Group's Architecture Driven Modernization Task Force [OMG's ADM] is working on new standards called Knowledge Discovery Meta-Model specification for technology and platform independent software modernization. I approach the problem of developing the tool with visual zooming, in a unique way based on OMG's ADM standards and Eclipse MoDisco which use the Model Driven Engineering approach to discover the models from existing software artefacts. I develop a tool which takes the model that are discovered by MoDisco, as input and visualize them as graphs like Package Dependency Graph, Control Flow Graph and Abstract Syntax Tree, with semantic zooming features to hide and display the information based on demand.

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do these amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Suthakaran Sivabalan

Signed:……..

Date: 21/05/2010

Table of Contents

Table of Figures

ABBREVIATION

SV – Software Visualization

OMG – Object Management Group

ADM – Architecture Driven Modernization

ZUI – Zoomable User Interfaces

AST – Abstract Syntax Tree

CFG – Control Flow Graph

PDG – Package Dependency Graph

KDM – Knowledge Discovery Meta-Model

XML - Extensible Mark up Language

XMI – XML Metadata Interchange

ASG – Abstract Semantic Graph

JDT – Java Development Toolkit

JaMoPP - Java Model Parser and Printer

GUI – Graphical User Interface

MDE – Model Driven Engineering

MDV – Model Driven Visualization

CFX – C Fact Extractor

API - Application Programming Interface

SDK – Software Development Toolkit

UI – User Interface

MVC – Model View Controller

ZEST – Zoom able Eclipse ShrimP Tool

Acknowledgement

I, Suthakaran Sivabalan take this opportunity to thank Dr.Artur Boronat for his continued guidance and support to me in delivering the project work. Your patience, encouragement and wealth of great ideas will never be forgotten. Your enthusiasm extends to whole batch.

I am grateful to Dr. Laura Bocchi in acting as the second marker for my dissertation work. Thank them for their valuable comments to achieve the project success. And also I am glad about the Department of Computer Science, University of Leicester for providing MSc taught courses and Individual project which is a valuable to chance to get involved in a project to exploit to sound theoretical and practical knowledge gathered during last two semesters. It is my great pleasure to acknowledge the contributions of all the people in the department and friends.

PART I
THE PROBLEM

CHAPTER 1

Introduction

Software visualization (SV) is widely accepting method for understanding software; most of the software visualization tools are mainly focused on software source code visualization. Researches show that during the maintenance phase 50% to 70% of the time is consumed for program comprehension. Most of source code visualization tools are based on program parser technology; a program parser gives the intermediate representation of source code which used to extract the facts for different types of visualization. Software visualization is not limited to visualizing program source code, for large code base, it is essential to visualize functional and architectural aspects of software as well for different stakeholders.

Visual Zooming of Software Source code is the main topic of this master's thesis, as my topic states this research is about developing a software application to visualize the source code as different graphs with information zooming feature. Developed tool generates Abstract Syntax Tree, Package Dependency Graph, Control Flow Graph and Source code Tree Map with semantic zooming features like nested nodes, fisheye viewer from java source code. Source code models generated by Eclipse MoDisco [2] which is new framework to discover model from existing software artefacts and based on Object Management Group's (OMG) Architecture Driven Modernization (ADM) standards [1] are used as input which is extracted and rendered as various graphs.

1.1Motivation

While the usefulness of Software visualization tools have been realized in many area of software development, the existing approaches don't have uniform structure to visualize the source codes and other artefacts. Most of the existing software visualization tools were developed based in different techniques and concept, in more specific ways; different programming languages need different approach to extract the facts from source code. Due to this, OMG is working on defining some software modernization standards that brings the uniform way for software modernization [1]. This concept motivated me to develop a source code visualization tool using these standards, so the tool can be used to visualize any programming language. In other hand the visual zooming features provided in most of the

existing tools to explore the source code information are different and mostly inadequate. In recent years effectiveness of Zoomable User Interfaces [ZUI] have been realized [4], use of this ZUI in visualization tools would bring more effective zooming mechanism to explore the huge code base. To support tool developer, I investigate the use of software modernization standards and ZUIs. However I could not find any application which uses these standards for code visualization.

Personally, I wanted to do this project because, I am always interested in software migration and software re-engineering techniques and technologies, and also attracted by module System Re-engineering (CO 7206) module provide by computer science department and a side benefit to explore more software development methodologies and technologies.

1.2 Aim

Main aim of the project is to develop a software application [named as *SharpVisualizer*] to visualize the software source code as different visual representations like Abstract Syntax Tree [AST], Control Flow Graph [CFG], Package Dependency Graph [PDG] and Tree Map, with rich user interfaces for easy navigation to make the program comprehension easy over the large code base.

1.3 Goals and Objectives

The main objective of this project is to develop a software visualization tool based on existing technologies and OMG's ADM standards such as Knowledge *Discovery Meta-model* [KDM] [3] specification and *Eclipse MoDisco* [2]. *MoDisco* provides an extensible framework to develop model-driven [13] tools to support to develop software modernization tools while KDM provides a common vocabulary of knowledge related software engineering source artefacts, regardless of the implementation programming language and operating platform. It defines a checklist of items that a software mining tool should discover and a software analysis tool can use. *MoDisco* provide Meta-Models to describe the existing artefacts and Discoverer to create the models from systems and generic tools to analysis and transform the created models.

Main objective of the project is explained as *SharpVisualizer's* requirements are listed below and each high level requirement is decomposed in their subsection for more details requirements.

1. Graphical representation of Abstract Syntax Tree [AST]

   AST is a tree representation of the abstract syntax structure of source code and each node in the tree denotes a construct occurring in the source code. ASTs can be obtained as models using *MoDisco*'s Meta-Models and Discoverers, using generic tools and other programming techniques these tools analyse these AST models and represent them as visual trees. Section 6.3 describes achieved result.

2. Package Dependency Graph

   Package dependency diagrams are integral to software development, outlining the complex, interrelationships of various packages in software. Section 6.4 describes the achieved result.

3. Source code Tree Map

   Tree map is a hierarchical visualization technique which is used to visualize large information base over millions of node using nested nodes and with varying size and colour [14]. SharpVisualizer visualize the java source code as tree maps defining nested boxes for Packages, Classes, and class members using different colour and size. Section 6.6 describes the achieved result.

4. Enable features to Study about visuals in more details

   In order to understand the software artefacts in details using these visual representations, this tool should provide features to do the following tasks.

   - Use the Graph actions, Zoom in, Zoom out, Zoom all (maximum zoom out), to size the graph.
   - Move the mouse pointer over a node to see an expansion of the type and name of the statement
   - Click a node to view source code related to that node, Ex: Click on Method node should show the source code that method.
   - Provide a Fisheye Viewer for exploring the data, the concept of Fisheye viewer is , when the is large number of nodes in a 2D space, if the user want zoom a specific node , and shrink other nodes, but still all the nodes will be visible to the user. Section 6.7 shows the output of Fisheye viewer.

Secondary Objectives which are listed below, and will be studied and implemented if time factor allowed. Again KDM models retrieved using *MoDisco* components will be used to extract the relevant information to achieve these objectives.

1. Graphical Representation of Control Flow Graph [CFG]

   CFG describe the logic structure of a method, in which the nodes represent element in the method and edges represent the flow between elements. KDM specification provides a package called Action Package which provides information about the control flow of a program. Again using *MoDisco* discoverers and generic tools we can extract the control flow data from KDM models and visualize them as graphs. Section 6.5 describes the output achieved.

2. Software pattern identifications

The tool should define a pattern-based approach and proceed to discover several layer of increasing abstraction such as extracting interfaces from a class or group of classes and indentifying design patterns like composite, strategy and bridge patterns can be discovered [15].

1.4 Challenges

Developing software visualization tools have challenges against size and complexity of software artefacts. Selection and abstraction are crucial in addressing large software; the challenge is to find appropriate and meaningful abstractions in order to provide a useful focused view. For an example when try to visualize the control flow diagram for huge a software artefact, it is the question, which part of the software should be visualized? Which level of abstraction should be visualized? In order to solve this problem, application should have data structure to store the information which should enable to extract different levels of information using normal programming techniques. Whenever user zooms in to a node, application should give more information about that node. Model which conforms to KDM specification, gives various aspects of software systems as separate packages for different level of granularity which can be browsed and extracted relevant information based on needs. The main challenges in developing *SharpVisualizer* application using KDM and Java Model are listed below.

- Use KDM, Java Model as source code repository and extract facts from these models for visualization.
- Visualize different levels of granularity for same aspect, show and hide information according to the user actions like when Zoom-In gives more details, Zoom-Out for more abstract view.
- Implementing smooth zooming feature for easy navigation.

- Provide link between different code representations and use different input models together for specific views.

Chapter 5 and Chapter 6 describe how these challenges have been tackled and solution developed.

1.5 Approach

I approach the problem of visual zooming over software source code by developing a prototype to visualize the software source code as different graphs and apply different levels of abstraction for graph, when user wants more information; they zoom in and get more information. In order to generate these graphs, I use two different methods. First, Use the source code models such as KDM Model and Java Models [16] which were generated by Eclipse MoDisco tools, as input to the application which extract the relevant facts and generate graphs .Second, I use the Extensible Mark up Language [XML] representation of source to retrieve the relevant facts to visualize (Chapter 6).

1.6 Evaluation

I evaluate the prototype with respect to project's goal and objectives. First, this work is evaluated in terms of practicality and usefulness of graphs generated and the extend of zooming feature which provides the information visualization in customized way. Each graph is constructed using object oriented programming. Second, to which extend this approach can be used further in order to add new kind of diagram and visual zooming. (Chapter 8)

1.7 Contribution

This research directly contributes to the software modernization and visualization community. This works demonstrates the feasibility of using model driven engineering for source code visualization with the use of Eclipse MoDisco, Eclipse Modelling Framework [17] and Eclipse Graphical Editing Framework [18] projects with high level programming. This is somewhat new when comparing to the existing tools and researches related to software visualizations.

1.8 Organization of Thesis

The dissertation is organized into three parts, **The Problem (Chapter 1 to 3), The Solution (Chapter 4 to 6)** and **The Evaluation (Chapter 7 to 8).**

Chapter 2 describes the domains study about software visualization and visual zooming over code base. And also explain the study about some popular software visualizations tool and critical evaluation with defined objectives.

Chapter 3 explains about used frameworks and tools. Importantly it describes about Knowledge Discovery Meta-Model 1.0 specification [3] and Java Meta-Model [16] which plays the major role in report.

The part II describes the solution proposed and as evidence it describes the design and implementation of SharpVisualizer application. **Chapter 4** describes functional and non-functional requirements of the tools. **Chapter 5** explains software architecture and structural design of the tool.

Chapter 6 describes the implementation, achieved results and system testing.

Chapter 7 critically evaluate the implemented solution and point out the limitation and problem faces during the implementation.

Chapter 8 concludes this report and outlines some future works can be done.

CHAPTER 2

Domain Study and Literature Survey

2.1 Background Study

Software development is a complex task involving many subtasks such as requirement specification, design, implementation, maintenance and re-engineering etc. Each of these tasks demands different information about the software systems. Visualizing software artefacts makes these tasks relatively easy. Software Visualization (SV) became much important in today's software industry in terms of time and value. Software systems become increasingly large and complex and their development and maintenance involves collaboration of many people with different background and specialized in different areas. This make the task of programming , understanding and maintaining software more and more difficult, especially when working with the code written by others or developed using legacy technologies. In other hand reverse engineering legacy software systems needs intensive analysis of the existing code base in order to recover the high level design of the systems. Program comprehension tools helps to understand the software systems. Mostly program comprehension tools visualize the software code as visuals which make the software systems more visible. This type of program comprehension tools are categorised as software visualization tools. Package Explorer, fan in/ Fan Out viewers, dependency graphs and coverage analysis are some examples for code visualization. SV is not limited to visualizing source code, it also visualize functional and architectural information, runtime behaviour, component interaction or software evolution. Most of the SV tools visualize UML class diagram, fan in/ Fan Out viewers, dependency graphs and flow charts. But they don't have any uniform standard for visualization [5].OMG's ADM Task Force is developing set of standards for software modernizing which enable to visualize the software in uniform way [1].

There are many taxonomies for SV, have been proposed [6][19]. More interest one is, Price, Baecker and Small (1998) [6] define six major attributes for SV tools as follows, "*scope (the range of programs the system can take as input), content (the subset of information about the software that is visualised by the system), form (the characteristics of the output of the visualization), method (how the visualization is specified), interaction (how the user interacts with and controls the visualization), and effectiveness (how well the system communicates*

information to the user). Each of these categories in turn has subcategories" [6]. Section 2.2 details about software visualization domain and Section 2.3 and Section 2.4 will give some related work based on these taxonomies and compare them.

2.2 Software Visualization in General

2.2.1 What to visualize

As I stated above SV serves different areas in software development, such as software design, comprehension of legacy code, bug detection, maintenance, refactoring and re-engineering...etc. Each of these tasks needs different level of information [20]. For an example, Visualizing requirements of new software can be useful to refine the requirements and aiding collaboration between developers. Or when automating source code refactoring using some re-engineering tools, visualizing each class as a visual tree will enable the developer to indentify the code elements and write refactoring queries using specific code transformation techniques. It is really important to define the purpose of the software visualization tool, which will enable to identify what kind of information should be visualized in which way. The Figure 2.1 shows the different level of source code visualization for different purposes.

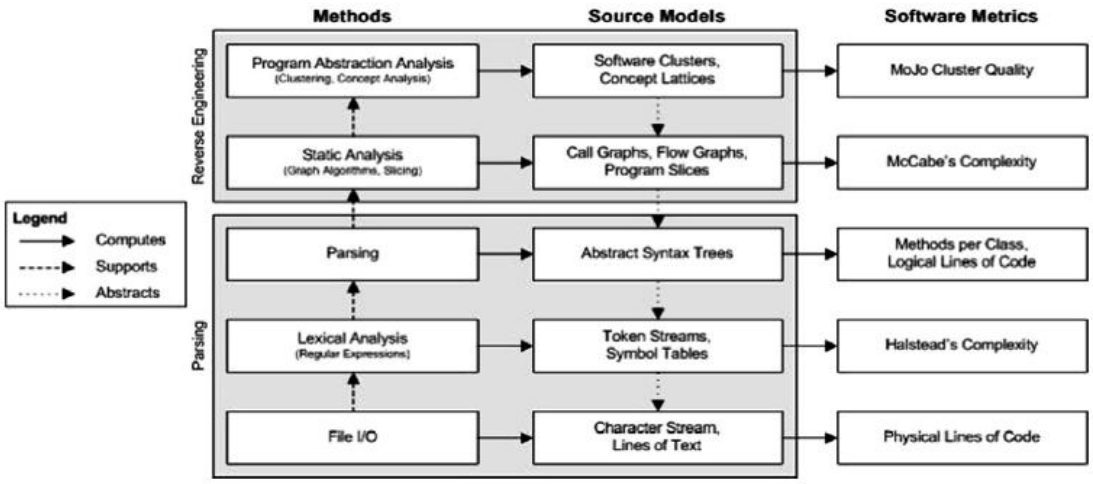


Figure 2.1 A Reverse Engineering Stack with different level of abstraction [21]

Program parsing is used to derive the AST which is the intermediate tree structure of any source code during the compilation, then followed by static analysis and program abstraction analysis which are used to extract Call graphs, Control flow graphs, and even more abstract level design information such as Software clusters and Conceptual designs.

Software systems are relatively large and complex, visualizing large software system as huge visual is not helpful. Selection and abstraction are crucial in developing software visualization tool addressing scale of software. Finding appropriate and meaningful selection and abstraction is big challenge.

2.2.2 How to visualize

Software visualization tool oriented for different tasks may use similar techniques or widely different one, but given visualization technique is interesting preliminary in terms of its appropriateness of the task. Figure 2.2 shows the basic block diagram of a simple software source code visualization tool.

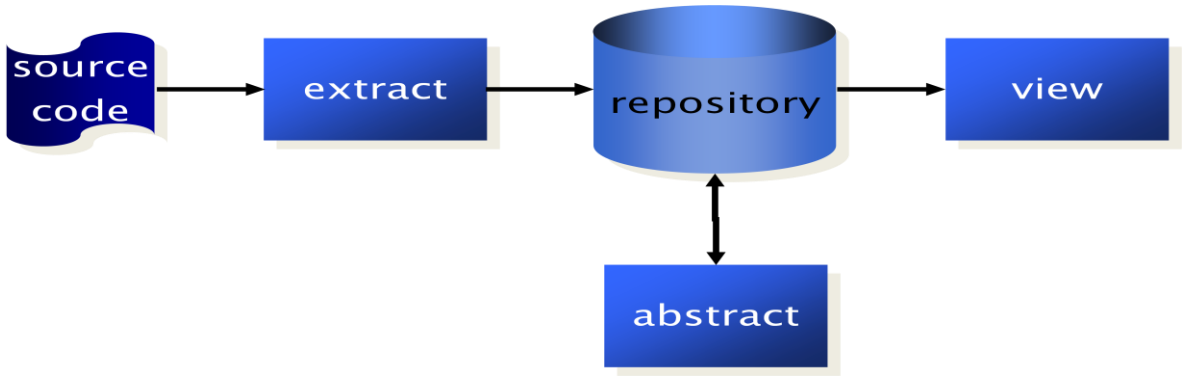


Figure 2.2 Block diagram of General Software Visualization Tool

- **Extract**

The roles of extract component is parse the source code using existing program parser or model discoverer [22] to extract the required facts as a defined data structure or model. If the tool is developed for multi language support, fact extractor should support for most of the programming languages or different parsers can be used for different languages. Generally output of Extractor component will be Abstract Syntax Tree [AST] or Annotated Abstract Semantic Graph [ASG] or a model which conforms to meta-modal. The Figure 2.3 shows a comparison between popular fact extractor for java source code.

Java Development Toolkit (JDT) is core of Eclipse java development environment, which generate AST of the source code. Deriving AST is basic feature of any reverser engineering or visualization tool, beyond this, there are other tools which do more than just deriving an AST. JaMoPP [Java Model Parser and Printer] [23] create exact and complete source code model from java source code. It defines its own Ecore meta-model [24] for Java. MoDisco is an Eclipse plug in to discover models from legacy systems for software modernizations [2]. It gives the extension to define meta-models for each language and other software artefacts and

provides relevant discoverer to extract models from source code. But MoDisco provides features to analysis and transform to targeted models (Section 3.3.3). We use Eclipse MoDisco in order to develop *SharpVisualizaer* application. Spoon [25] also based on Eclipse JDT, but it defines more detailed Ecore meta-model than MoDisco defines. Spoon also performs semantic analysis.

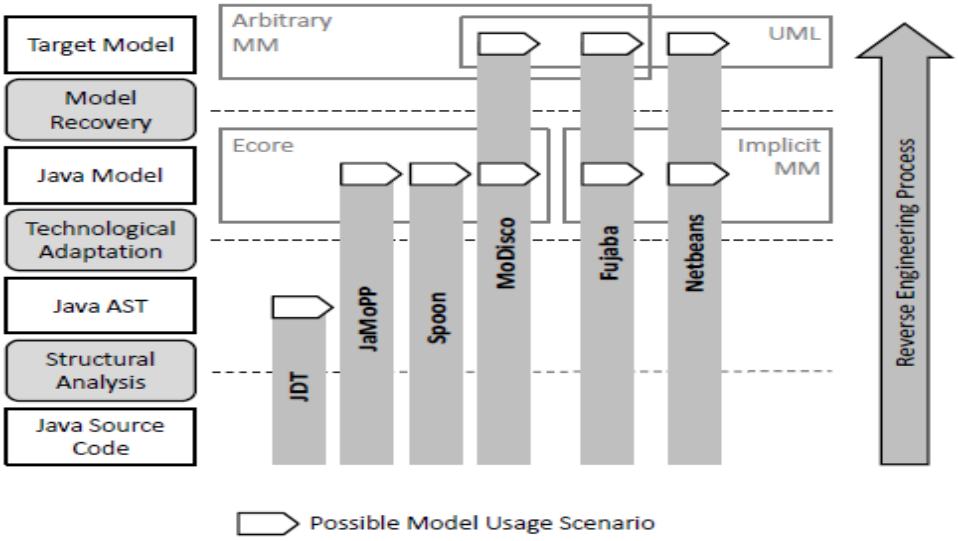


Figure 2.3 Tool Classification for Reverse Engineering Java Source Code [23]

Fujaba [26] also construct models from java source code, in order to analysis and visualize, But it does not define any explicit meta-model for java. It constructs models using plain java classes. Netbeans [27] is an IDE for development in java, which also generate models from java source code but, the generated model is implicit to Netbeans in order create UML diagrams from java source code But generated model restrict to extract other information except UML.

- **Repository**

This is the repository to store the information derived from the extractor. Repository defines data structure to keep this information in the form AST or ASG. This information is saved as XML/XMI representation which enabled to develop flexible front end using various technologies for different purposes. Because XML is standardize, enable interpretability, provide better tool support and also can easily be translated to various format like Graph Exchange Language [GXL], Hyper Text Mark up Language [HTML], Text and also can reconstruct the original source code. In other hand repository contains model generated by fact extractor which discover models which conforms to the meta-models defined them.

- **Abstract**

This component provides the libraries/ algorithms and model transformation queries for analysing and extracting required information for visualization. For an example different analysing methods for Data Flow analysis, architecture recovery, cluster analysis, browsing and querying...etc can be implemented in different ways, put them as a library. Application's other components can use this library functions for visualizing purpose.

- **View**

This is visual representation component which visualizes the data extracted from repository using relevant methods from abstract library for specific views. For an example, visualizing UML class diagram with relationship and references, or showing AST as a visual tree structure needs graph based data. AST Extractor generates AST graph contents as nodes and edges, then passed to graph content providers which will render the graphs. Views also this will provide flexible and intuitive Graphical User Interfaces [GUI] for easy navigation and information extraction from visuals.

2.2.3 General features of Software visualization tools

The past researchers identified most important features of any information visualization tools based on experiences [20].

- *Information for Detail on Demand*: This gives varying level of details, like fine granularity, abstraction, different information content and type, to satisfy user's interest.
- *High Information content*:"*Visualization should present as much information as possible without overwhelming the user*".
- *Integration to original artefacts*: It is useful to give link between the visuals and original information it represents.
- *Approachable user interfaces*: Flexible and intuitive user interfaces without unnecessary overheads with low visualization complexity and simple navigation. A good user interfaces should provides structures information visualization to reduce the user's chance of becoming lost.
- *Good use of Human compute interaction*: Interaction provides mechanism for gaining more information and maintaining attention. Zoom able User Interface like (SHrimP views) [8] gives really effective interaction mechanism using semantic zooming over information visualizations.
- *Visualization Customization*: Good visualization tool should enable the user to customize the visualization based on their requirements, such as presentation customization, data customization and behavioural/ control customizations.

2.2.4 Advances in Recent years

- Use of 3D : The introduction of 3D is largely technical advance which provides a third dimension which can better represent the visuals than previous 2D views. 3D visualization will overcome some of the limitations of 2D visuals. For example it is relatively easy to visualize number of attributes of class and size of class in terms of number of lines of code, and different types of relationships. Figure X shows a sample of 3D visualization using sv3D [28] application.

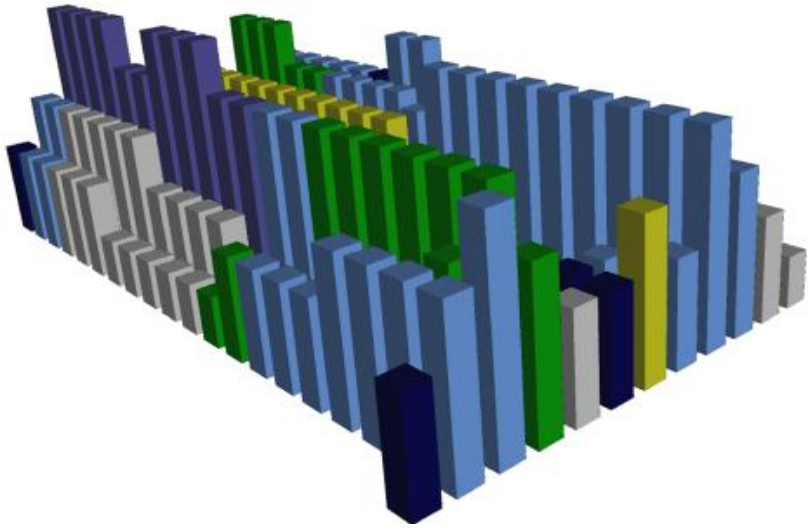


Figure 2.4 3D view of C++ source code generated by sv3D [28]

- Model Driven Visualization [MDV]: Model Driven Engineering [MDE] is a recognized technology in which a system is designed, built and deployed through series of models. Recent advances in MDE enable to generate models from software artefacts as well as interactive visualizations. Eclipse MoDisco is an example for discovering models form source code and visualizing as UML diagrams [29]. Main advantage of using MDE is, creating platform independent, program language independent software artefact models and visualization models, which leads to more reusability. The Figure 2.5 shows the process of creating Model Driven Visualizations using meta-modelling and model discovering and model transformation techniques.

- Attention to software as evolving: Software is not a solid product, which is evolving with time. Software configuration management is in place to track the software evolution. But this configuration management is not enough to understand the

accumulated changes on software architecture and design due to the changes in the source code. Visualizing software evolution is becoming popular topic in information visualizations community [20].

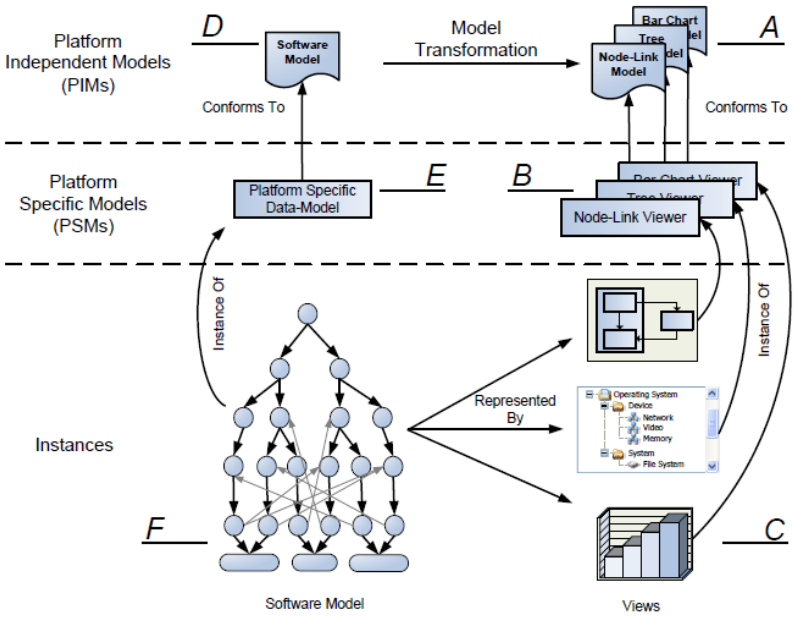


Figure 2.5 Overview of Model Driven Visualization [7]

2.2.5 Continuing Issues

Even technology has advanced to give solution to limitation in existing software visualization; still there are issues in software visualizations. Main issue is with matching visualizations to human needs.

- Scale: software is large and complex, as I stated in earlier sections, "simply repackaging massive textual information into massive graphical representation is not helpful". When visualize the software source, what is being visualized for whom is the main question. Different people who working on different areas needs different information about the software systems. It is difficult to create visualization tool which satisfy every one's requirements. "Most tools are generic and hence low level". When the abstraction level rises, tools need to be more specific for the purpose and should give abstract but more information about large scale software. [30]
- Intelligent Selection: A good visualizing should be coupled with good analysing algorithms, visualizing is not just representing data as visuals; visualization should provide meaningful abstraction and distillation of data in order to identify desired information. Algorithms to support meaningful visual representations are still evolving.

2.3 Related Work

There are many software visualization tools in industry Ex. SHrimP [8], Crole[9], Portable Bookshelf [10] and Fujaba[26]. Each of these software visualization tools use different methods and technologies to visualize the software system, some are customized for specific purpose. Ex. Creole was developed to visualize java code base. Most of the modern IDEs support visualizing the source code up to some extends mainly structures information as UML diagrams. However other graphical visualization like Abstract Syntax Tree, Control Flow/ Data Flow diagrams, software pattern identification ...etc are rarely supported by IDEs. The following sections will describe about some of these existing tools. To understand the process of developing code visualization tool, I analyse the main components of these tools and finally defined the requirement of SharpVisualizer application.

2.3.1 SHrimP

The SHrimP [Simple Hierarchical Multi-Perspective] is a domain independent visualization technique for enhanced visualizations of complex information space with nested graph view with smooth zooming features. SHrimP was originally developed for visualizing software programs for program comprehension with nested graph view architecture where user zoom in to find more information about graph node. Program source code and documentation are presented with graph by embedding code snippet within the nodes of the graph.

SHriMP's design was based primarily on the Java Bean approach [31]. In this approach each bean has a specific purpose and communicates with other beans solely through listeners and events; a bean does not contain any references to other beans. Within SHriMP, all beans were connected together and contained by the *SHriMP View*. In essence, the *SHriMP View* was SHriMP; most of the control and initialization of the program happened within the *SHriMP View*. Any data to be displayed by SHriMP needs to be converted to a graph. More specifically, data is converted into a directed typed graph in which the vertices and edges in the graph have a variety of attributes such as short name, a long name, a type, etc. SHrimP provides Hierarchical views for nested node link diagrams as well as flat views. Filmstrip view for take snapshot a view while navigating visual, that snapshots can be used gather the information.

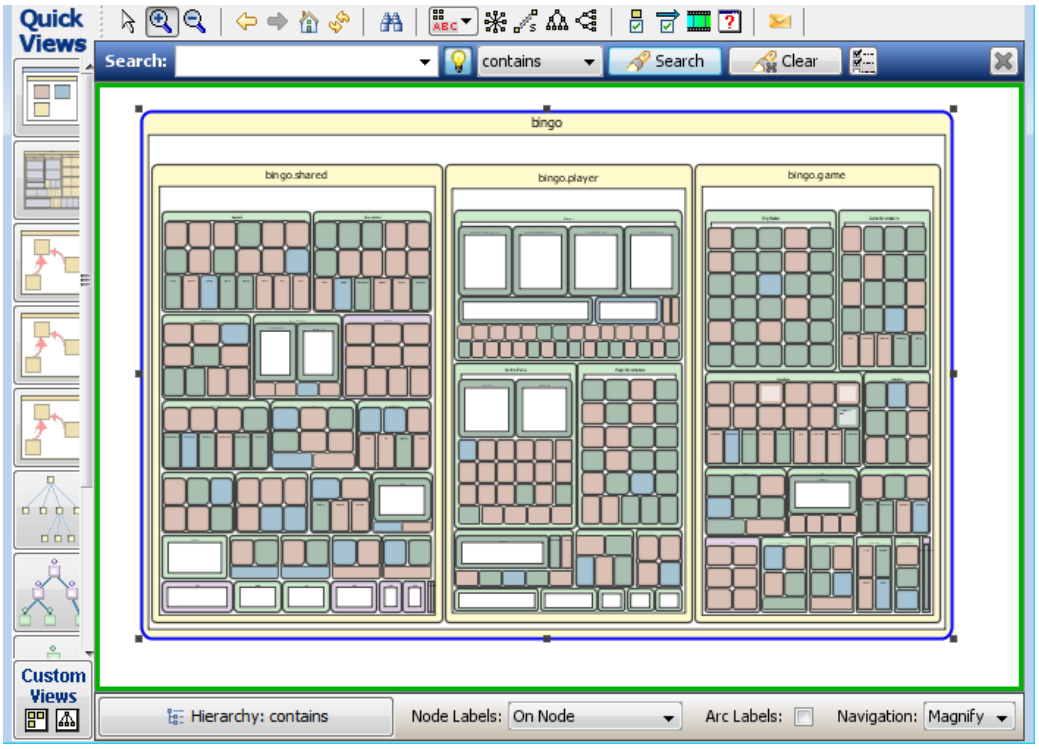


Figure 2.6 Tree Map views of Java project generated by SHrimP.

Most striking features of SHrimP were ZUIs which provides smooth zooming to enables the user to zoom in for details and zoom out for most abstract view of the information. The Figure 2.6 shows the Tree Map for java program visualized by SHrimP where each rectangle block specify a element of the program like package, class, method or field ....etc. Size and colour of the rectangle is defined based on language element in size Ex, Number of lines of code in a Class defined the size of rectangle for a class. SHrimP provides different visualization for software programs like Tree Map, Call Graph, Package Dependency Graph based on method call and field variable call. And also provide facility to customize the visualization like filtering nodes and edges, and edges or assign different colours for nodes and edges...etc and different graph layout algorithms like Spring Layout, Vertical Tree, Horizontal Tree...etc. SHrimP visualizes any data which is graphed based using GXL, XML, and XMI.

2.3.2 Portable Book Shelf

Portable Bookshelf [10] is a web based software visualization tool to visualize for architecture re-construction and re-document information about large software systems to make the comprehension easy. PBS contains three components 1.The C Fact Extractor [CFX] 2. A Domain Specific Language for manipulating Graphs (Grok) [11]. A Graph Rendering

tool (LSEdit)[12]. Low level software artefacts such as files, source code and configuration files are analysed to extract information using CFX fact extractor and further analysed to produce high level information such as, the dependency between components and sub systems using Grok. Grok is a domain specific language for manipulating graphs based on algebra. Finally LSEdit is used to render the graphs.

PBS uses relation model to generate the graphs. First it defines the relation between source, destination and relationship name which gives the nodes and edges. Then check the attributes of element and add relationship between elements, attribute and attribute value. The figure X shows the data model used in PBS for source code elements written in C language.

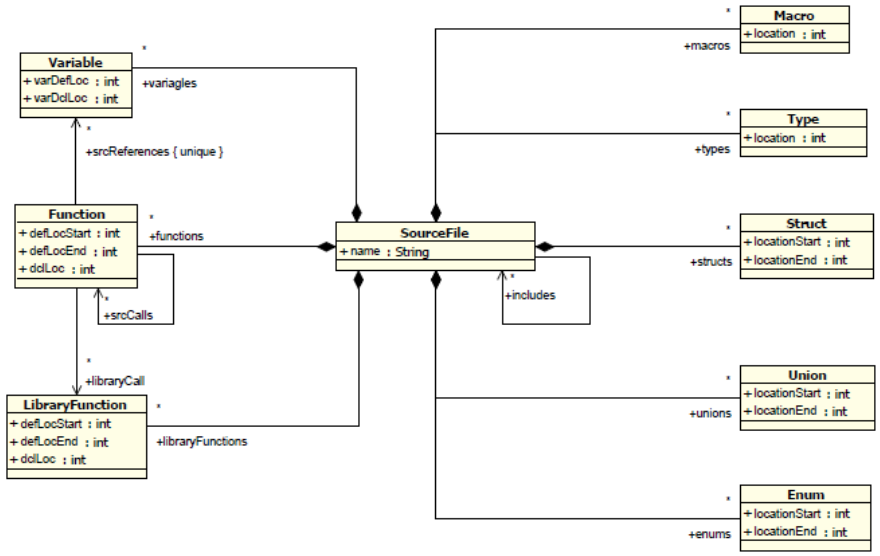


Figure 2.7 Model to Which Artefacts Extracted from CFX conforms. [7]

PBS also supports a nested graph viewer similar to SHrimP; Figure 7 shows the example produced using PBS. Architectural reconstruction is multi step process in PBS, first map each files or modules to subsystems, then subsystems are grouped hierarchically. Grok is used to identify and add relationships between two subsystems if there is a relationship between them. PBS provides two different methods for customizing visual attributes, one ways is each element can be annotated with an attribute describing its shape , label and colour and other one using dialogs to further configure the visual attributes. And also PBS supports user interaction features, such as, navigation, zooming, panning and history panels.

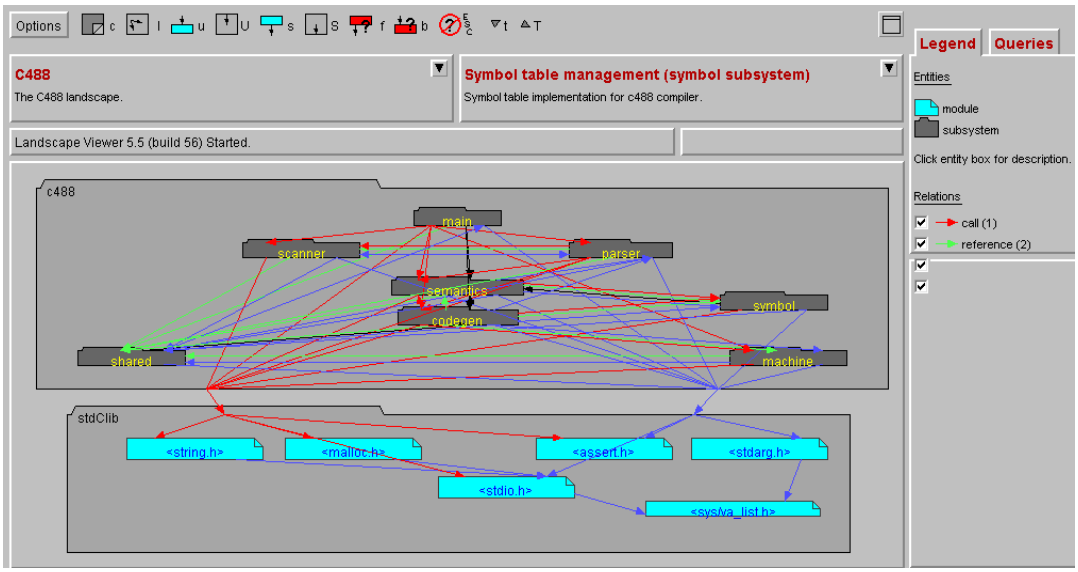


Figure 2.7 Nested Package Dependency Graph generated by PBS

2.3.3 Creole

Creole is domain specific version of SHrimP. Creole is a plug-in for the Eclipse integrated development environment that supports number of different visualization option for java source code. Like other source code visualization tool, it was developed with the intention of helping programmers with program comprehension. Creole extract number of artefacts from a java program and visualize those as different graphs such as Tree Map, Package dependency graph, call graphs and Class Hierarchy...etc with hierarchical visualization with nested graph nodes and views. Figure 2.8 shows PDG of java source code generated by Creole.

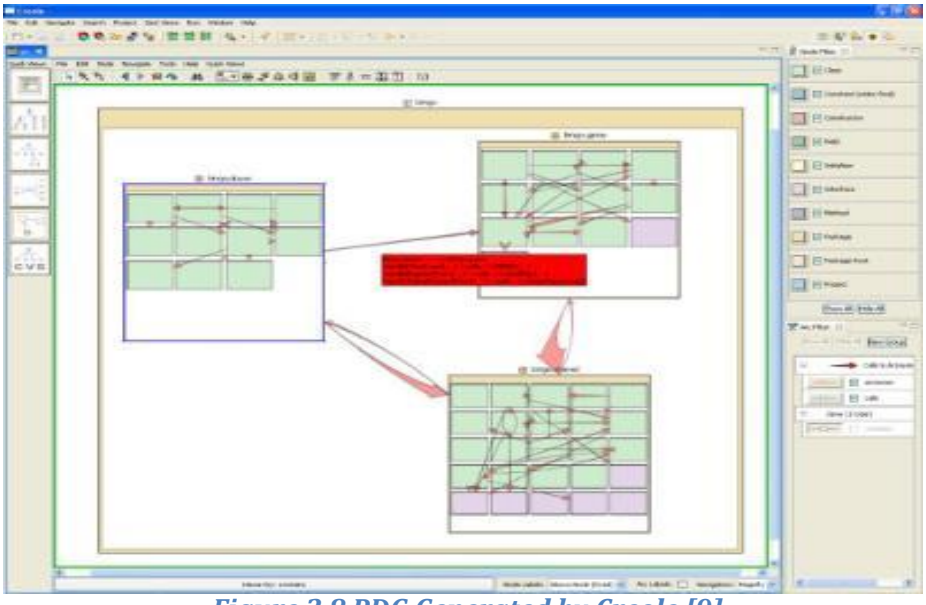


Figure 2.8 PDG Generated by Creole [9]

Creole provides facility to trailer the visualization using customization facilities such as series of UI dialog boxes to configure visual attributes like size, shape, colour, label, width and line styles can be configured for node and edges. And also Creole gives facility to filter the information using filtering nodes and connection. Like SHrimP, Creole also provides semantic zooming feature using ZUIs

2.3.4 Fujaba Toolkit for Reverse Engineering

Fujaba RE toolkit [26] is a plug-in for reverse engineering activities such as driving UML class diagrams, detecting design patterns, idioms, anti patterns and bad smells ...etc. But most existing feature of Fujaba is, design pattern detection from implemented java code. Design patterns are used in forward engineering as good solutions to recurring problems and form a common vocabulary among developers. Thus, recognizing implementations of design patterns in existing software systems helps the reverse engineer to understand the system. Fujaba identify design patters using graph transformation. Fujaba extract ASG and AST using java parser and also provide other two components called Pattern Specification which is used to write graph transformation rules to identify patterns, other one is Interference Engine which is used to process the rules. Pattern specification component is a graphical editor for writing transformation rules.

The Figure 2.9 shows a screen shot from Fujaba which recognize design patterns from Java.awt. Package.

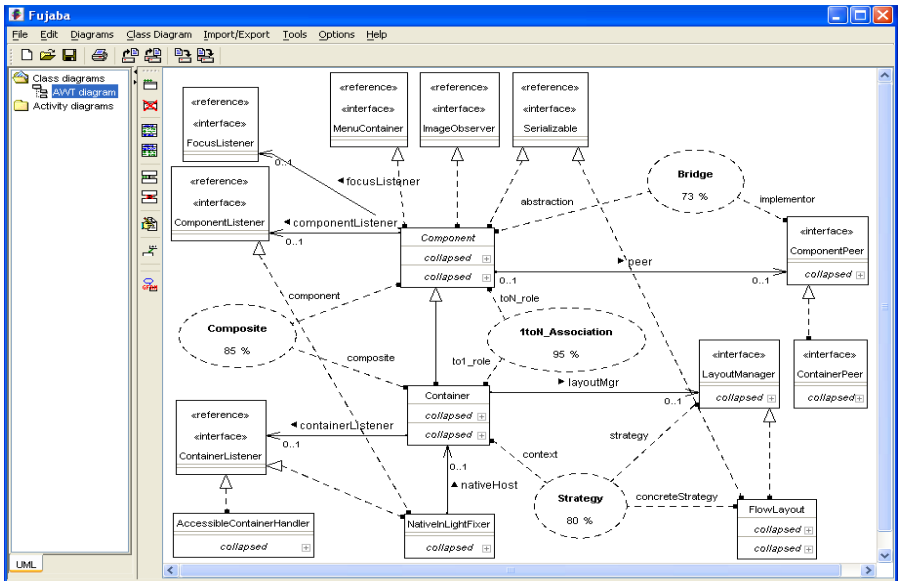


Figure 2.9 Screen shot of Pattern Detection using Fujaba [26]

2.4 Comparison of Some Software visualization tools

The Table 2.1 shows the comparison between the listed software visualization tools in previous section.

	Domain Specific Tool	Scope	Form of Visualization	Method of Visualization	Presentation Customization	Data Customization	Behaviour Customization
SHrimP	NO	Visualize any graph based data as graphs	Tree Map, Nested Views, Hierarchical views	Mapping data elements into graph elements programmatically	Different views, Zooming, Panning, Customizing visual attributes	Filter nodes ,Semantic zooming over data for more information	Give features to customize the behaviours.
PBS	Yes	Visualize C code base as graphs	Nested Graphs,	Mapping source code modules to graph nodes, use DSL for extract information	Annotated node with visual attributes,	Customize the data using Grok DSL language	---------
Creole	Yes	Visualizing Java Code	Tree Map, PDG, Call Graph, Class hierarchy	Mapping data elements into graph elements programmatically	Different views, Zooming, Panning, Customizing visual attributes	Filter nodes ,Semantic zooming over data for more information	Give features to customize the behaviours
Fujaba	Yes	Reverse engineering Java Code	UML class diagram, Design pattern detection and Bad smells	Provides feature to write rules to extract information	Provide Visual Attribute customization	Customize data using pattern rules	--

2.5 Discussion

To better understand the concept of based on this background study, creating a visualization tool for specific domain is not an easy task. Data that should be visualized, visual attributes and features to explore the information space must be designed appropriately for end user. The result of study gives the idea how a visualization tool should be implemented and what kind of features should be give for particular domain. The requirements of SharpVisualizer which is domain specific tool are derived based on this background study and related works [Chapter 4]. SharpVisualizer is specific to user who needs program comprehension techniques. To give better program comprehension, more specific graphs such as PDG, CFG, AST and Tree Map are identified from these related works and implemented.

CHAPTER 3

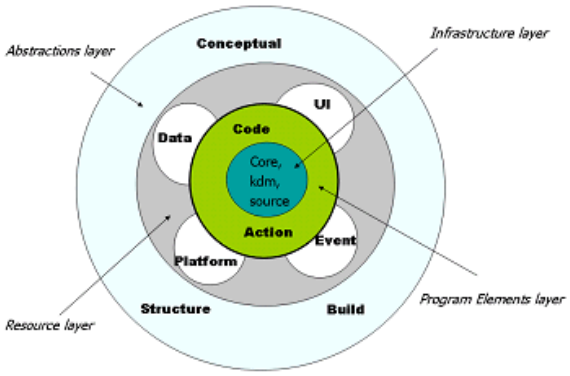
Used Frameworks and Tools

Selecting the Frameworks and tools is a big issue before starting the development. A good selection of framework and tool will make the development job easy, and allow the developer to focus the domain issues. As I stated in earlier chapters, SharpVisualizer application uses KDM and Java Model which was generated by Eclipse MoDisco, as primary inputs which is the model contains the information about source code. Following sections describes about KDM and Java meta-model which is crucial to understand SharpVisualizer construction explained in Chapter 5 and 6. SharpVisualizer do analyse these source code models and generate the graphs. I chosen Eclipse as main development framework because it provides functionality to explore the models and extract the relevant facts using Eclipse Modelling Framework [17]. Other hand I used some external APIs [Application Programming Interface] called Zest [Zoom able Eclipse Shrimp Tool] [32] and Piccolo2D [33] in order to visualize extracted facts. Following subsections will give details about these frameworks and tools.

3.1 Knowledge Discovery Meta-Model

The Object Management Group (OMG) Architecture Driven Modernization (ADM) Task Force is developing a set of modernization standards. They have published a specification call Knowledge Discovery Meta-Model (KDM) [3] for common intermediate representation of existing software artefacts, independent of used technology and operating environment, to facilitate the exchange of existing system meta-data for various modernization tools. KDM represent not only the source code, but entire physical and logical software artefacts. KDM is basically as entity relations diagram.

KDM separates knowledge about existing systems into several orthogonal facets that are well-known in software engineering and are often referred as Architecture views. KDM contains nine packages which are defined as separate models; these packages are organized into four layers as entities and relationships. Figure 3.1 and 3.2 show the organization of packages in different layers and their level of compliance.



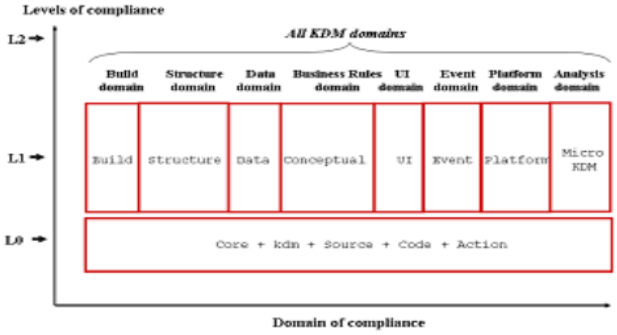


Figure 3.1 Separation of concerns in KDM [3] & 3.2 Domain and levels of Compliance [3]

KDM Infrastructure Layer contains KDM, Core and Source Packages. KDM and Core packages define common meta-model elements which provide the infrastructure for other packages. Other packages extend the elements available in infrastructure, add the requires feature for their specific purposes. The Figure 3.3 and 3.4 show "coreEntities" class diagrams of KDM specification.

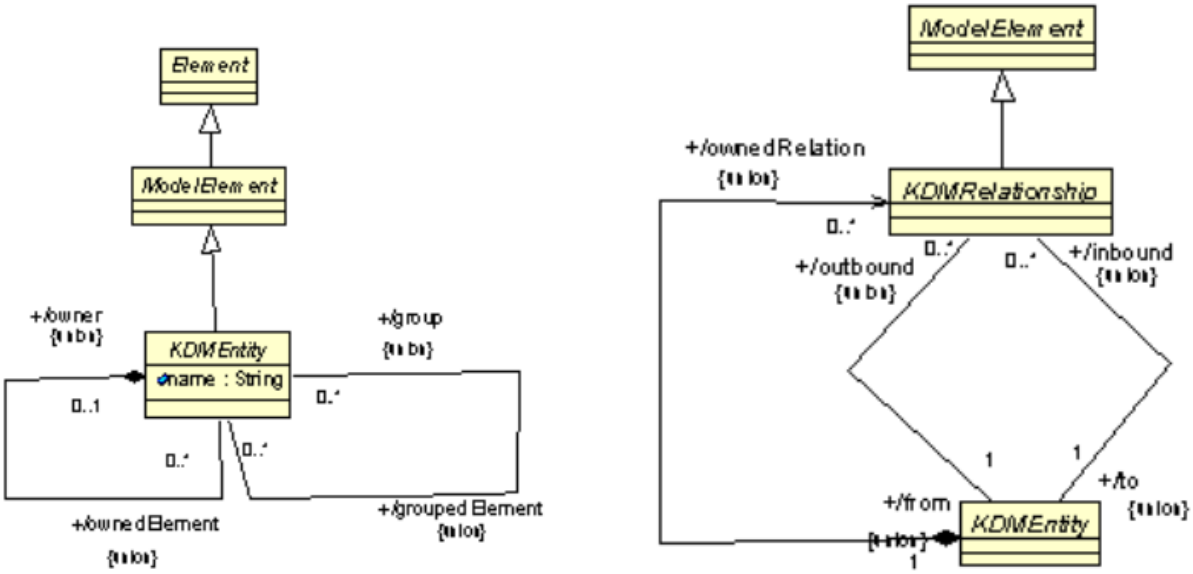


Figure 3.3 & 3.4 Core Entities class diagram of KDM [3]

KDMEntity and *KDMRelationship* which are abstract classes are the base classes for other KDM model elements. Each *KDMEntity* defines two special relationships container and grouping. Some KDM entities are containers for other entities which are defined using container ownership between container and the entities which are directly owned by this container. Some KDM entities are groups of KDM entities .This relationship is defines using special grouping association between the group and entities which are directly grouped into

this group. *KDMRelationship* is the base class for the relation between elements in other packages; *KDMRelationshhip* defines following properties and operation to identify the relations. To and From refers the source and destination of the relationship.

to: KDMEntity[1]	The target entity (also referred to as the to-endpoint of the relationship). This property determines a meta-level interface to KDM relationships. This property is a derived union. Every specific KDM relationship redefines the to-endpoint to a particular subtype of KDMEntity. In KDM this is represented by the CMOF "redefines" mechanism. Concrete properties redefine the "union" properties of the parent classes, defined in the Core package.
from:KDMEntity[1]	The origin entity (also referred to as the from-endpoint of the relationship). This property determines a meta-level interface to KDM relationships. This property is a derived union. Every specific KDM relationship redefines the from-endpoint to a particular subtype of KDMEntity. In KDM this is represented by the CMOF "redefines" mechanism. Concrete properties redefine the "union" properties of the parent classes, defined in the Core package.

getTo(): KDMEntity[1]	This operation returns the KDM entity which is the to-endpoint (the target) of the current relationship
getFrom():KDMEntity[1]	This operation returns the KDM entity which is the from-endpoint (the origin) of the current relationship.

Figure 3.5 Associations in with KDMRelationship class [3]

Source package define the Inventory model, which enumerates the artefacts of the existing software system and defines mechanism of traceability link between KDM elements and their original representation in the source code of the existing software system. Figure X shows the class diagram which shows the how each element is linked with original artefacts using *SourceRef, SourceRegion* and *SourceFile* Classes. *SourceRegion* class contains the path of the source file and position of the element in that source file.

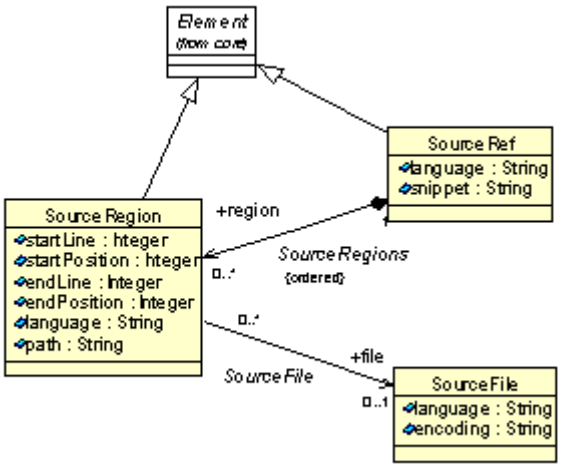


Figure 3.6 SoureRef Class Diagram [3]

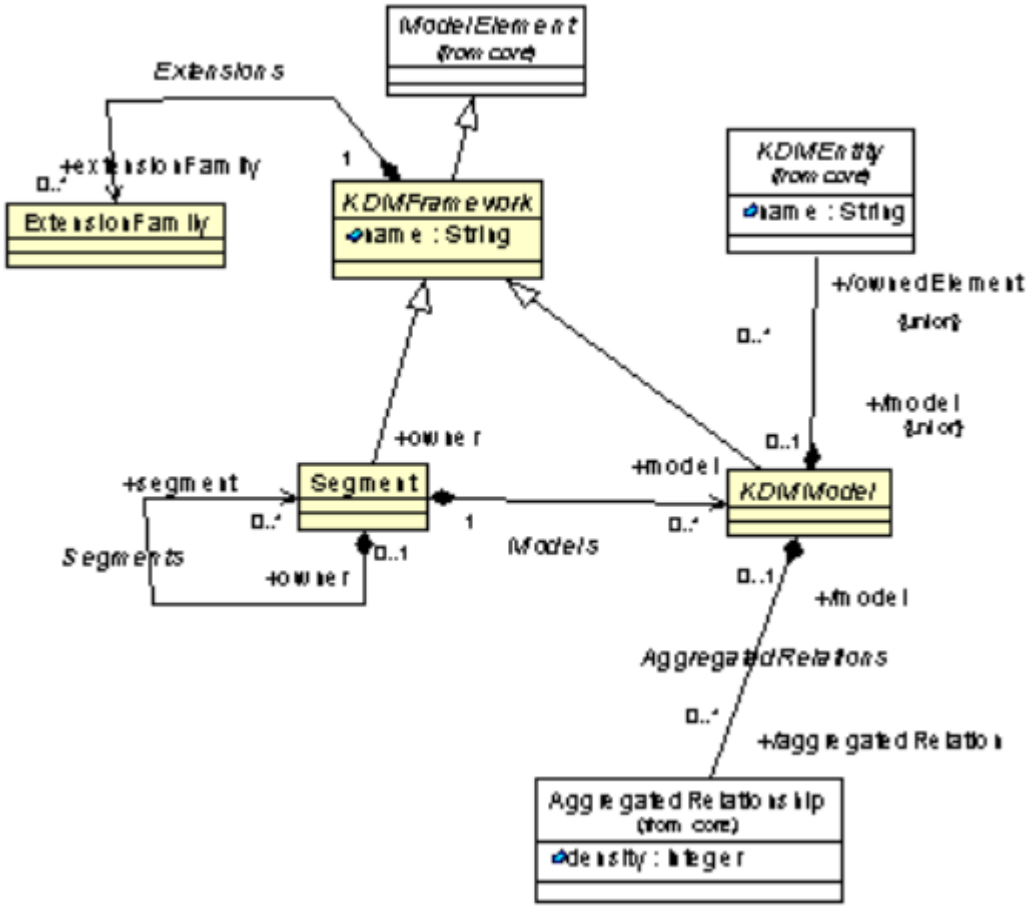


Figure 3.7 Framework Class Diagram [3]

Figure 3.7 shows the class diagram of KDM model structure defines in KDM package. Each package defines their own KDM model by extending *KDMModel* class. Generate KDM model contains one or more segment which contains collection of *KDMModel* defined by respective package. For example, Program Element Layer defines CodeModel by extending *KDMModel*.

Other layer defines by KDM specifications as follows Program Elements Layer contains code, action packages , Runtime Layer contains Platform, User Interface, Data and Event packages and Abstraction Layer contains Structure, Conceptual and Build packages. SharpVisualizer uses L0 KDM model which was generated by Eclipse MoDisco plug-ins. For more details about KDM specification refers APPENDIX B that gives important class of KDM specifications.

3.2 Java Meta-Model

Eclipse MoDisco plug-ins defines a meta-model for Java programming language and model discoverer called Java Discoverer which will generate java model from java project[16]. A Java model contains the full abstract syntax tree of the java program; each statement in the source code is described. In addition, links between elements are resolved. The model can thus be seen as an abstract syntax graph (ASG). The Figure 3.8 shows the basic class diagram of Java meta-modal.

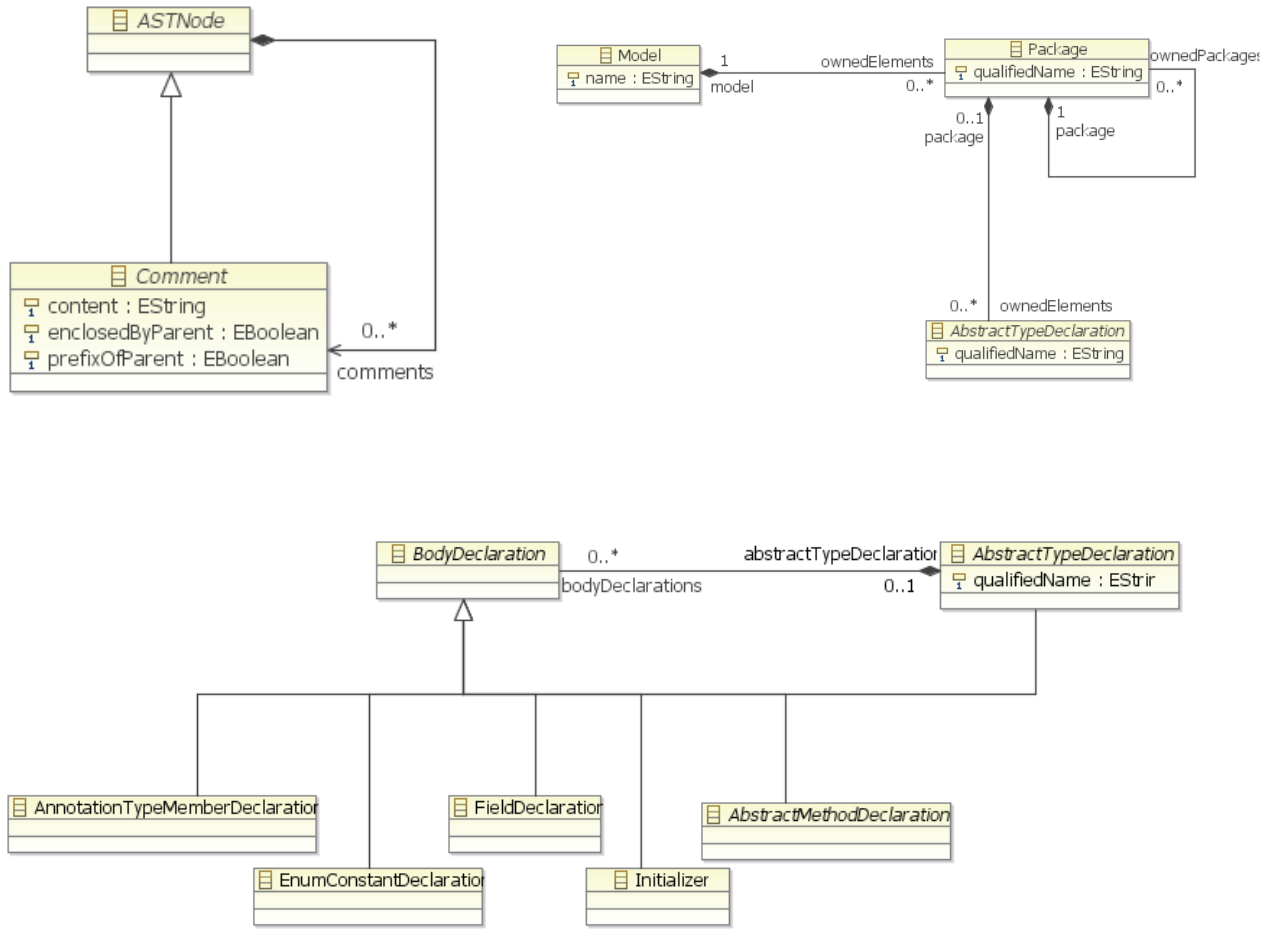


Figure 3.8 Java Meta-Model core class diagrams [16]

ASTNode is the base class for every element in the java model. Each Java Model contains the Packages and packages contain the Classes and so on which defined by *AbstractTypeDecleration*. Each type declaration contains Body Declarations. More information about Java Meta-Model can found in Appendix B.

3.3 Eclipse

Eclipse is not a huge single java program, but rather a small program which provides the functionality of typical loader called plug-in loader. Eclipse plug-in loader is surrounded by hundreds and thousands of plug-ins. A plug-in is a java program which extends the functionality of Eclipse in some way.

Figure 3.9 shows the architecture inside the Eclipse SDK. Each eclipse plug-in can either consume services provided by other plug-in or can extend its functionality to be consumed by other plug-ins

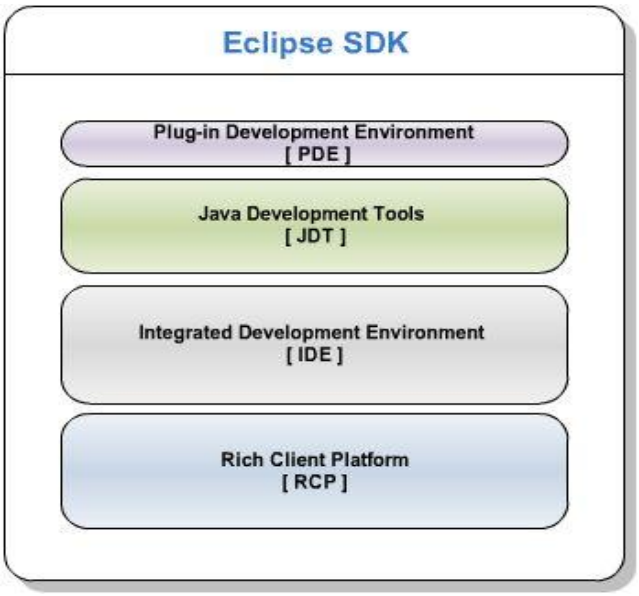


Figure 3.9 Core Architecture of Eclipse

These plug-in are dynamically loaded by eclipse at run time on demand basis. Also I studied about the Eclipse platform. It is an open platform designed to be easily extensible by third parties. Therefore we can build various products and tools around the Eclipse SDK and also we can extend those products and tools further and further. Thus all the extensions to the Eclipse IDE can be made via plug-ins and the extreme extensibility is achieved through that plug-ins. Plug-in developments to the Eclipse can be performed via Plug-in Development Environment (PDE). The PDE is a set of tools designed to assist developing, testing, debugging, building, and deploying Eclipse plug-ins while working inside the Eclipse workbench. PDE is not a separately launched tool. PDE integrates itself in the workbench by providing platform contributions, such as editors, wizards, views and a launcher, which users can easily access from any perspective without interrupting their work flow.

3.3.1 Eclipse Rich Client Platform

Eclipse RCP is the core of Eclipse IDE, which is build upon the Eclipse RCP with many plug-INS on top of it. Eclipse RCP provides a high quality end user experience for Eclipse domain by providing rich Eclipse native User Interfaces (UI) as well as high speed local processing. Eclipse RCP provides the plug-in extension mechanism to create new

applications with Eclipse native UIs. To better understand the components Figure 3.10 gives an overview of components.

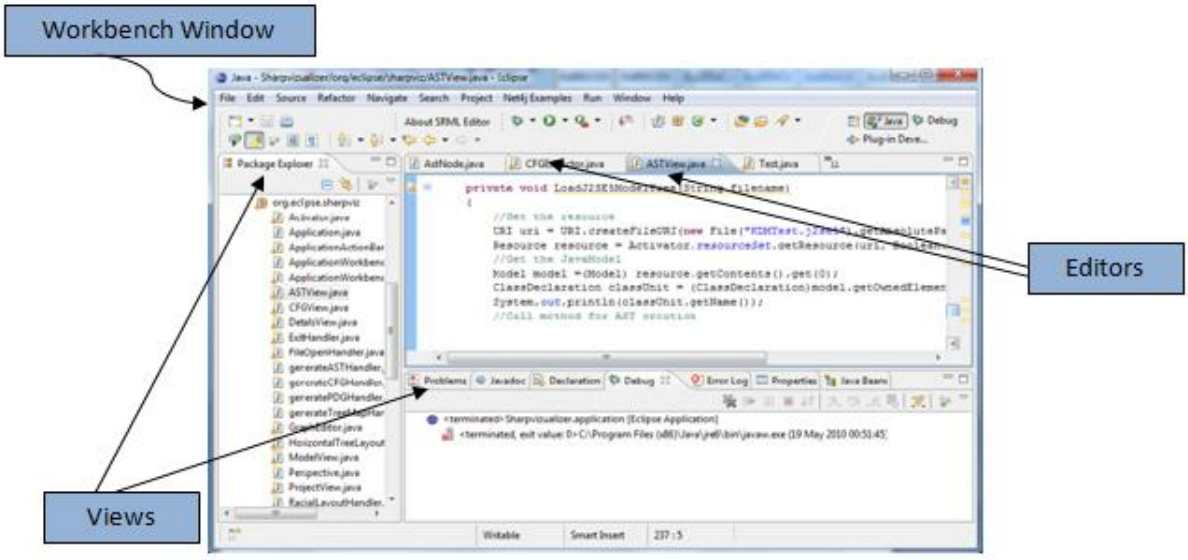


Figure 3.10 Basic components of an Eclipse RCP Application

Workbench window is topmost window, each workbench windows provides one or more perspectives which provide the layout for views and editors.

3.3.2 Eclipse Modelling Framework [EMF]

EMF is an Eclipse based modelling framework and code generation facility for building tools and other application based on a structure data model. Eclipse provides the feature to define models using XML, XMI, and Java. And also EMF provides native meta-model called Ecore for defining structural models. Eclipse MoDisco [Section 3.3.3] generates Ecore models which are manipulated using EMF inside SharpVisualizer application.

3.3.3 Eclipse MoDisco

MoDisco provides an extensible framework to develop model-driven-tools to support for software modernization. Model driven approach provides flexible and mature modernization solutions. MoDisco enables to defines meta-models describing software systems and provides abstract discovers from which concrete discovers can be derived to discover models conforms defined meta-models, from original source artefacts.

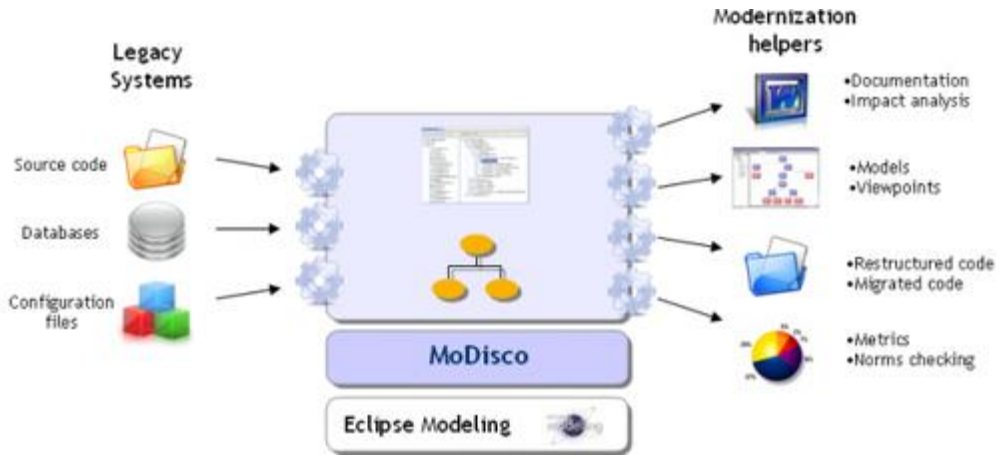


Figure 3.11 Overview of MoDisco [2]

The figure shows the sequence how to discover models from existing software artefacts.

3.3.4 ZEST

ZEST (Zoom able Eclipse ShrimP Tool) is a set of visualization components built for Eclipse. The Figure X shows the architecture of ZEST.

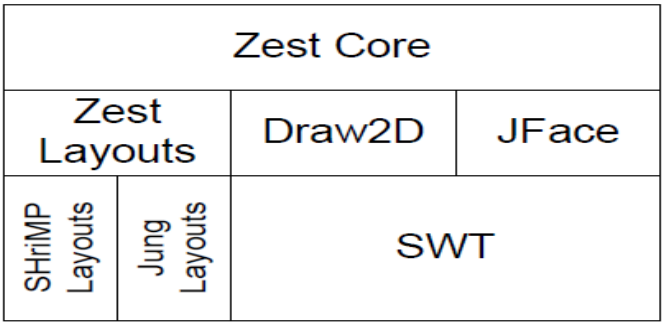


Figure 3.12 Architecture of Zest within Eclipse [7]

Zest library has been developed to support Model-Driven-Visualization [7], but it can be used with normal programming. It was developed in SWT (Standard Widget Toolkit) [34] and Draw2D [35]. Zest has been modeled after JFace [36], and all the Zest views conform to the same standards and conventions as existing Eclipse views. It enables to use the existing applications with Zest. Zest also provides different graph layout algorithms such as Tree Layout, Spring Layout...etc.

3.4 Piccolo2D Graphics Library

Piccolo2D is a 2D graphics library which supports development of 2D structural graphs, but it provides canvas with Zoomable User Interfaces (ZUI) to show the information or any 2D objects. A ZUI is a traditional computer display, but let the user to smoothly zoom in, to get more details and zoom out for abstract view. This ZUI features were developed based on Scene Graph models that is common is 3D environment. These ZUIs gives power to display objects in hierarchical structure and allows the developers to orient, group and manipulate objects in meaningful ways.

Part II

The Solution

CHAPTER 4

Software Requirements Specification

This chapter will describe the detail requirements of *SharpVisualizer* software visualization tool, after carefully analysing intended objective of the application. Following sub sections will list the functional and non-functional requirements of SharpVisualizer tool. The following chapters will explain the solution to achieve these requirements in detail.

4.1 Functional Requirements

This section describes the core functionalities of the SharpVisualizer applications with related to input data and internal data structure, presentation of visualization and behaviour of applications.

Data

- Input data: SharpVisualizer should take *KDM* models and *Java-Model* generated by *Eclipse MoDisco* and XML version of java model as input to the application.
- Internal Data Structure: SharpVisualizer should contain a proper data structure that must be able configured to visualize the relevant information and hide the rest from user. This can be achieved using data filtering, and data reduction techniques.
- Internal data structure and processing algorithms should be able to handle huge amount of data.

Views

- Simple and customized model viewer for KDM and Java Models which are input to the application. Model viewer should support for element selection.
- Visualization of original software artefact. Ex: Source code viewer
- Visualization of Graphs
    - Visualize AST of selected class as visual Tree
    - Visualize dependencies between components as PDG for whole project or selected part of project.
    - Visualize the control flow between statements of a method as CFG
    - Generate Nested Tree Map of whole project.

Presentation

- Provide rich user interfaces to present the graphs to user with zoom able interfaces and easy navigation features.
- Provide different zooming facility for different graphs and visuals
    - Nested nodes inside a container node for PDG. Child nodes can be showed or hide.
    - Semantic zooming for Nested Tree Map using Zoom able User Interface.
    - Mouse Double click on AST, or CFG node will pop-up a new window which contains code snippet related to selected node.
    - Fisheye viewer for observe the dependencies between large number of nodes in same window.
- Enable the user to customize the graph layout as wanted
    - Tree Layout
    - Spring Layout
    - Radial Layout
    - Horizontal Tree Layout
    - Composite Layout
- Different colours for different types of graph nodes which helps the user to easily identify the program elements.

Behaviour

- The visualization environment should support an efficient method for action selection and navigating information space. Mouse click on a nested node should give more information about that node or shows the child nodes if there is any.
- Link between different views of the application for easy navigation.

4.2 Non-Functional Requirements

- **Efficient view** : *SharpVisualizer* tool should have an efficient way to view for information visualization of data, presentation and support the behaviour and supporting complex operation like linking multiple views and link to actual artefacts...etc.
- **Performance:** Performance issue must to be considered, because SharpVisualizer should be able to load and analysis huge input model.
- **Familiar notation**: While providing efficient view, *SharpVisualizer* should use the standard notation or familiar graphical representation language which will give much user friendly. For an example use the UML notation for Control Flow Graph.
- **Use existing viewers**: When developing visualization tools like *SharpVisualizer*, it is suggested to use the existing viewers and graph rendering engines that will provide behaviour of standard widgets which user used to that. For example Use of JFaceViewers , SWT, and Zest viewers.

4.3 User Interface Requirements

Design and implement an intuitive, flexible and well structured GUI for visual representation of extracted information.

- Low visualization complexity and well structured with high information content and easy navigation. For example using techniques such as landmarks to reduce the user's chance of becoming 'Lost'.
- Use of Zoom able user interfaces to provide zooming over the information
- Support navigation features like Overview, Smooth Zooming feature, Filtering graph nodes and edges for easiness.
- Use standard navigation features and notations for usability.

CHAPTER 5

SharpVisualizer Software Design

This chapter gives details about architecture of SharpVisualizer application, detailed structural design of each component and Graphical User Interface design. As information visualization software, to support different views from same data, SharpVisualizer adopts the Model-View-Controller (MVC) architecture patterns. Section 5.1 explains about the architecture, Section 5.2 explains about structural design and Section 5.3 details about GUI design.

5.1 Architecture of SharpVisualizer

As I stated above SharpVisualizer application was developed based on MVC architectural pattern, following sections will give more details about it.

5.1.1 Model-View-Controller

Model-View-Controller (MVC) is a classic design pattern often used by applications that need the ability to maintain multiple views of the same data. Figure 5.1 shows architecture of MVC pattern. The MVC pattern hinges on a clean separation of objects into one of three categories, models for maintaining data, views for displaying all or a portion of the data, and controllers for handling events that affect the model or view.

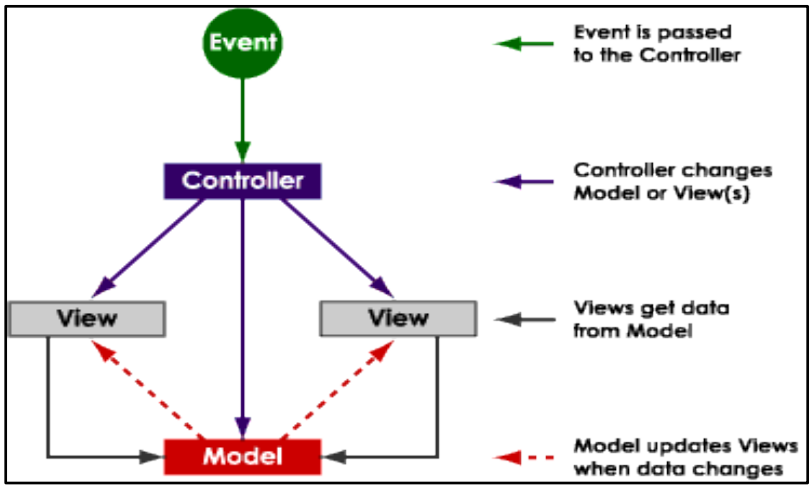


Figure 5.1 Overview of MVC architecture

5.1.2 SharpVisualizer with MVC Architecture

The Figure 5.2 shows the MVC architecture which is adopted to develop the SharpVisualizer application. Whenever user requests different type visualization or different level of information, controller part analysis the data models and passed the relevant data to the viewers.

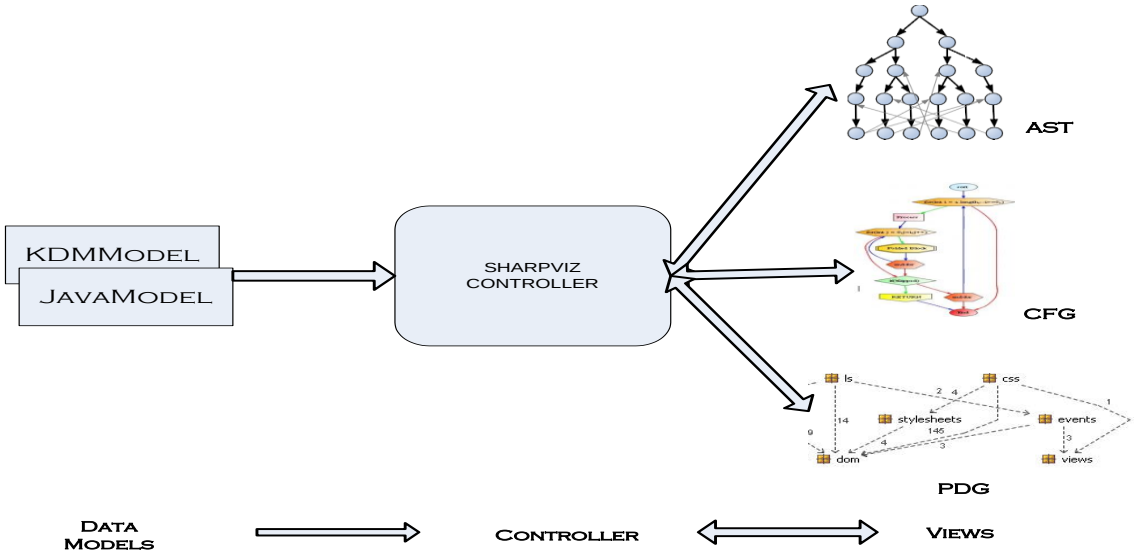


Figure 5.2 SharpVisualizer MVC Architecture

SharpViz controller implements algorithms and methods to explore data in the KDM and Java-Models and extract relevant facts required for visualization and generating view models according to user request and make the connection between data models and view by passing the view models to the graph viewers. Whenever user requests any actions in view, like Zoom IN or Zoom Out or Filtering nodes, controller does take the responsibility. Views are very thin presentation layers without any processing power. Viewers just render the content provided by controller.

View component contain different viewers for different type of graphs, such as, Node Link Type viewer for AST, Nested Node Link Type Viewers for Package Dependency Graph...etc. And also Viewer can contain the visual attributed which can be customized by user. For an example assigning different colours for different level of nodes in AST or CFG. And also few viewers use ZUIs provides by Piccolo2D graphics library which offers smooth zooming features.

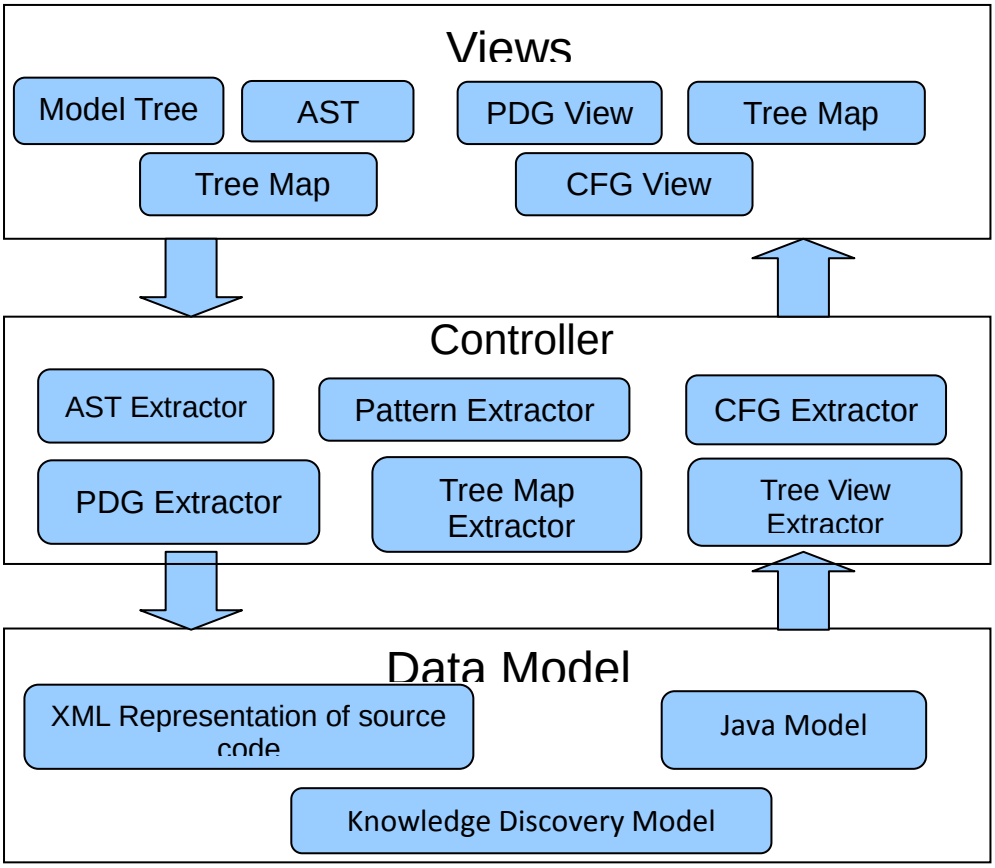


Figure 5.3 SharpVisualizer Layered Architecture

The Figure 5.3 shows the detailed view of SharpVisualizer architecture which conforms to MVC pattern. Each subcomponent in the diagram shows every low level module in the developed system. These separations of components enable to facility add more fact extractor and views easily. Following section will describe about structural design of complete application with and give more details about each components defined in Figure 5.3

5.2 SharpVisualizer Structural Design

5.2.1 Overview of SharpVisualizer Structure

Figure 5.4 shows the overall class diagram of SharpVisualizer application. This was developed based on background reading about existing visualization tool's structure and MVC architecture stated above. It is platform independent design except the Zest and Piccolo2D graphics components which are specific to java related technologies. This class diagram is an abstract view of the systems; this is divided into three parts, as Source model, and Model Extractor and Viewer which are mapped to MVC pattern components Data Model, Controller and Views respectively.

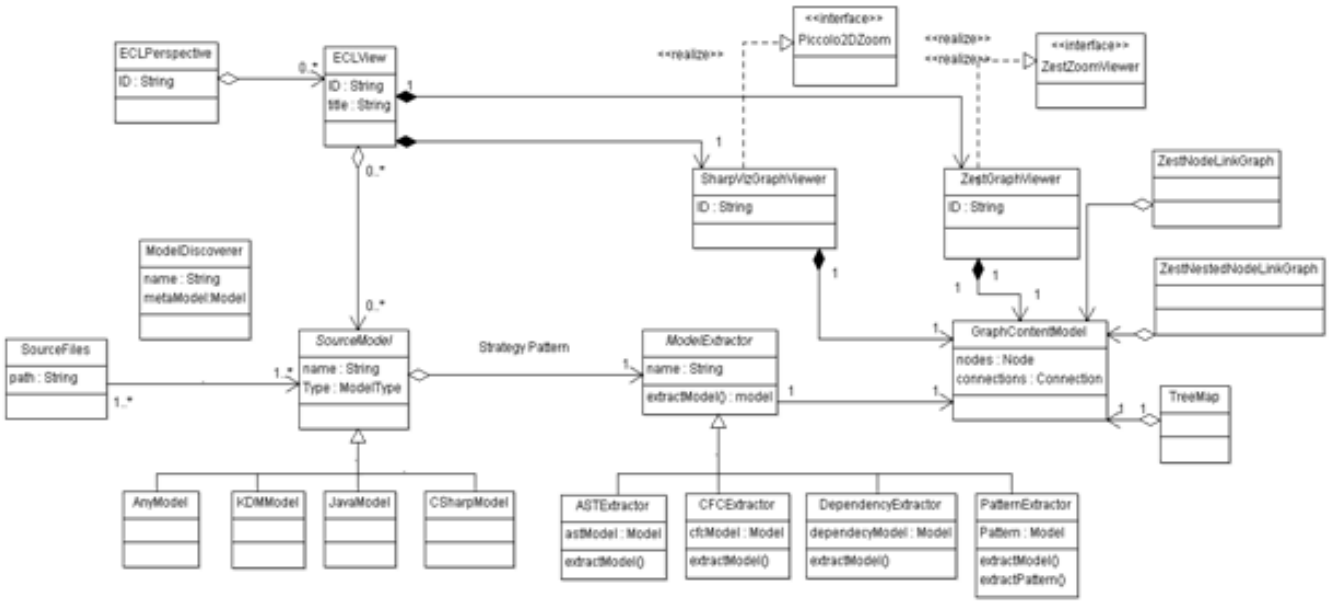


Figure 5.4 SharpVisualizer Class Diagram

*SourceMode*l class in referred to the source code models such as KDM model and Java model which is basically input to current SharpVisualizer application. But implementing the model discoverer workflow within the SharpVisulizer application will make it as complete which will generate visualization directly from raw source code. But internally discovering these source code model from source code will be implemented using Eclipse MoDisco plug-ins and workflow.

ModelExtractor part is algorithm factory which used to explore the information available in source code models. Other component is viewer part, which is responsible for rendering graphs and provides interfaces for zooming feature. Graph Viewer classes implement zooming algorithms provided by third parties [32][33] to make zoom able graph viewers. Following sections gives more detailed class diagrams about these components.

5.2.1 Controller for Extract Facts

In Figure 5.4, *ModelExtractor* class is an abstract class which is super class for all other most specific fact extractor that extends the *ModelExtractor* class. Each class implements its own fact extracting algorithms. For an example *ASTExtractor* implements extractModel method to generate AST graph contents from XML version of Java-Model. Other extractor are *CFGExtractor*, *PDGExtractor*, and *TreeMapExtractor* which are used to extract the control flow ,package dependency, and tree map graph contents respectively. Any new graph

extractor can be added easily by just extending *ModelExtractor* class with appropriate graph viewers. This enables the future changes and extensions easily. Chapter 6 explains the implementation of these algorithms.

5.2.2 Graph Viewers

The main purpose of SharpVisualizer application is providing visual zooming over the java source code, it defines different graphical representation of graphs with zooming features. The following subsections will gives the class diagrams of each different graphs used. Node Link Diagram and Nested Node Link diagrams are derived using Zest while Tree Map viewer and Fisheye viewer are derived using Piccolo2D which provides ZUIs that supports semantic zooming over the nodes.

5.2.2.1 Node Link and Nested Node Link Diagram using Zest

AST view and CFG view defined in overall architecture [Figure 5.3] use Node Link Diagram type to visualize AST and CFG while PDG View uses Nested Node Link diagram which can visualize nested class nodes within a package node. Figure 5.5 shows the class diagram for the graph viewer component. The classes with prefix "zest" are classes provided by Zest graphics framework. This prefix is used to differentiate the core classes in this diagram.

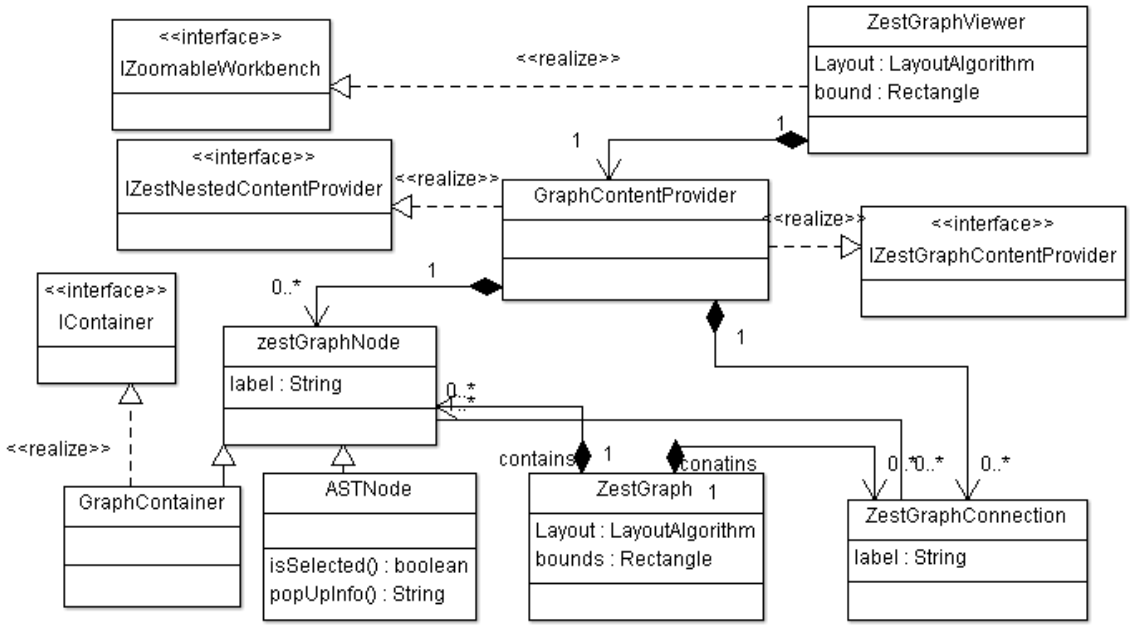


Figure 5.5 Class Diagram for Graph viewers using ZEST.

GraphNode, GraphConnection, Graph, GraphContentProvider , GraphContainer and *GraphViewer* classes are reused from ZEST. They give the direct meaning of graph components. *ASTNode* class extends the GraphNode class to customize the behaviour of the node zooming feature. Click on AST node should pop-up the source code snippet related to that node.

GraphContentProvider which provide the graph model to graph viewer, implements the *IGraphContentProvider* interface. *GraphViewer* implements the *IZoomableWorkBench* interface in order to provide physical zooming for graphs. Each graph defines its own *GraphContentProvider* and *GraphViewer.* GraphContentProvider for PDG view implements *INestedContentProvider* interface which provided by ZEST in order to create nested nodes graphs. *GraphContainer* class is another option to create nested node, *GraphContainer* extends the GraphNode class as well as implements the *IContainer* interface in order to create the containers.

5.3.2.3 Tree Map using Piccolo2D

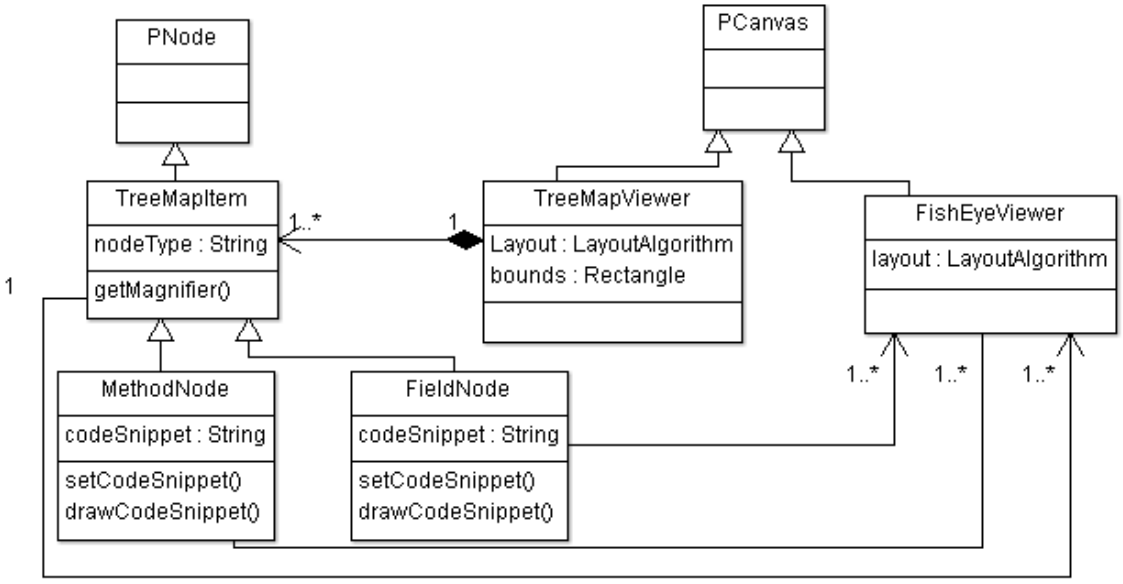


Figure 5.6 Class Diagram for Graph viewers using PICCOLO2D

SharpVisualizer also provides Nested Tree Map and Fisheye viewer in order to visualize large source code base in single view effectively. These viewers are developed using Piccolo 2D Graphics library. Classes with prefix "P" are provided by Piccolo. *TreeMapViewer* and *FisheyeViewer* extends the *PCanvas* class with proper layout algorithms to accommodate child nodes. *TreeMapItem* class extends the *PNode*, in order to customize the properties and

behaviour which is relevant to visualize the source code with semantic zooming. *MethodNode* and *FieldNode* classes extend the *TreeMapItem* class, these nodes are user visualize code snippet of a method unit and field declarations when the node is magnified.

5.3 User Interface Design

The Figure 5.7 shows the proposed user interface of the SharpVizualizer application. The figure show the main interface components which used to visualize different graphs and source code and input models.

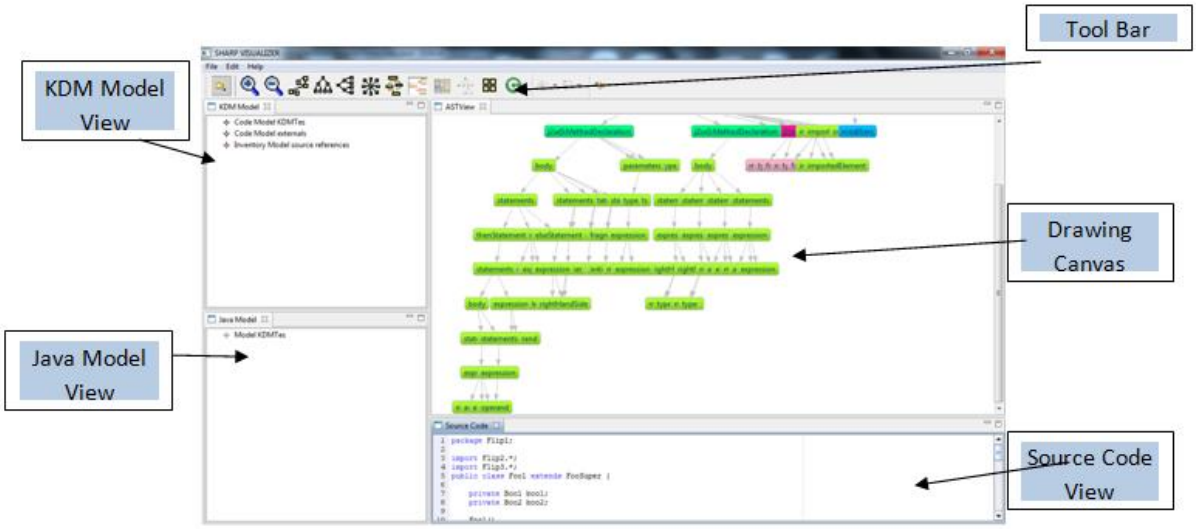


Figure 5.7 Graphical User Interface of SharpVisualizer.

KDM and Java-Model views are used to view the input models, Drawing canvas is used to render the graphs, both graph viewers that implemented by SharpVisualizer are rendered in same canvas. Source code view is display the original artefacts of selected item. Tool Bar provide some visual attribute customization features, such as changing graph layouts and zooming as well as commands to render the graphs for loaded models.

CHAPTER 6

Implementation and Testing

This chapter explains how SharpVisualizer was developed using given frameworks and tools. Development was based on Eclipse and Java. Section 6.1 shows the eclipse plug-in extension for SharpVisualizer and following sections explains the algorithms and output of each view defined in design. SharpVisualizer uses EMF to load and explore the information available in the model. Most of the fact extracting methods are developed using EMF utilities and information given respective meta-models for KDM and Java Models except AST extractor which use the XML processing. Finally section 6.9 shows the some system test cases carried out for implemented features.

6.1 Eclipse RCP Extension for SharpVisualizer

Eclipse RCP which is core of Eclipse IDE's GUI, provides an extension mechanism to create Eclipse native applications. Figure 6.1 shows the plug-in extension diagram of SharpVisualizer.

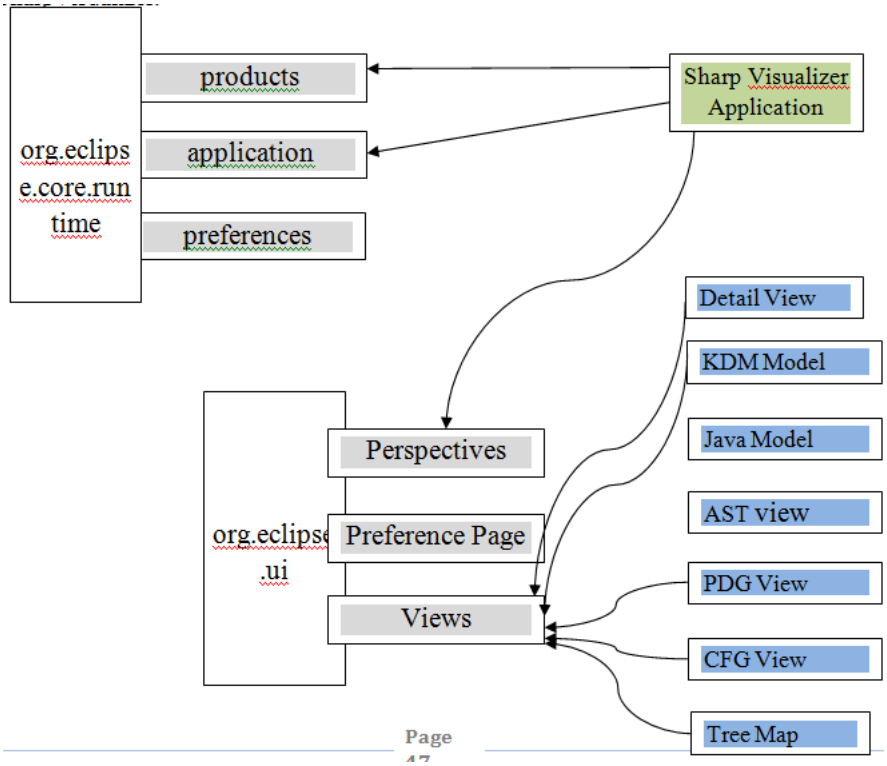


Figure 6.1 SharpVisualizer RCP plug-in Extension Diagram

SharpVisualizer uses extension points of the *org.eclipse.core.runtime* and the *org.eclipse.ui* plug-in. This diagram shows an overview of views provided by this application. There are separated views for each graphs and other form of visualizations. For example PDG view is used to visualize the package dependency graph. *Org.eclipse.ui.views* is the extension point where all the views are registered. Multiple views can be registered using single extension point. In order to save preferences; the application should use the *org.eclipse.core.runtime.preferences* plug-ins extension point. In order to use Eclipse native preference dialog, the application should extend the preferencePage of *org.eclipse.ui* plug-in. The current SharpVisualizer doesn't implement these preference features. But it can be easily added in future.

6.2 KDM and Java Model Tree View

KDM and Java models which are Ecore models discovered using Eclipse MoDisco plug-ins, input to the SharpVisualizer application. Discovered model contains information about software artefacts, program elements. MoDisco plug-in provides specific model viewer and editor for these models, but those viewers more advanced for this application. Due to this, SharpVisualizer implements its own, very simple model viewer to show the tree structure of models. User should import generate these models using MoDisco and import to SharpVisualizer application to generate the views. When the model is imported, initially model will be loaded to common *ResourceSet* in *Activator* class which act as utility to other part of applications. Figure 6.2 shows the code snippet for loading the models.

```
{
//Load Resource
ResourecSet resourceSet = new ResourceSetImpl();
URI uri_j2se5 = URI.createFileURI(new File("KDMTest.j2se5").getAbsolutePath());
URI uri_kdm = URI.createFileURI(new File("KDMTest.kdm").getAbsolutePath());
    Resource resource_j2se5 = resourceSet.createResource(uri_j2se5);
    Resource resource_kdm = resourceSet.createResource(uri_kdm);

    //Loading option to increase the performance while loading models
    Map loadOptions = new HashMap();
    loadOptions.put(XMLResource.OPTION_DEFER_ATTACHMENT,true);
    loadOptions.put(XMLResource.OPTION_DEFER_IDREF_RESOLUTION, Boolean.TRUE);
    loadOptions.put(XMLResource.OPTION_USE_DEPRECATED_METHODS, true);
    loadOptions.put(XMLResource.OPTION_USE_PARSER_POOL, new XMLParserPoolImpl());
    loadOptions.put(XMLResource.OPTION_USE_XML_NAME_TO_FEATURE_MAP, new HashMap());
    //Saving Option, to increase the performance while saving models
    Map saveOptions = new HashMap();
    saveOptions.put(XMLResource.OPTION_CONFIGURATION_CACHE, true);
    saveOptions.put(XMLResource.OPTION_USE_CACHED_LOOKUP_TABLE, new ArrayList());
        try
        {
            resource_j2se5.load(loadOptions);
            resource_j2se5.save(saveOptions);
            resource_kdm.load(loadOptions);
            resource_kdm.load(saveOptions);
        }catch(IOException ex)
        {
            ex.printStackTrace();
        }
}
```

Figure 6.2 Code Snippet to load Ecore models in a Resource set

Loaded models are visualized using a tree viewer provides by *org.eclipse.jface.viewers* plug-in [36]. Figure 6.3 shows the Tree viewer for KDM and Java Model.

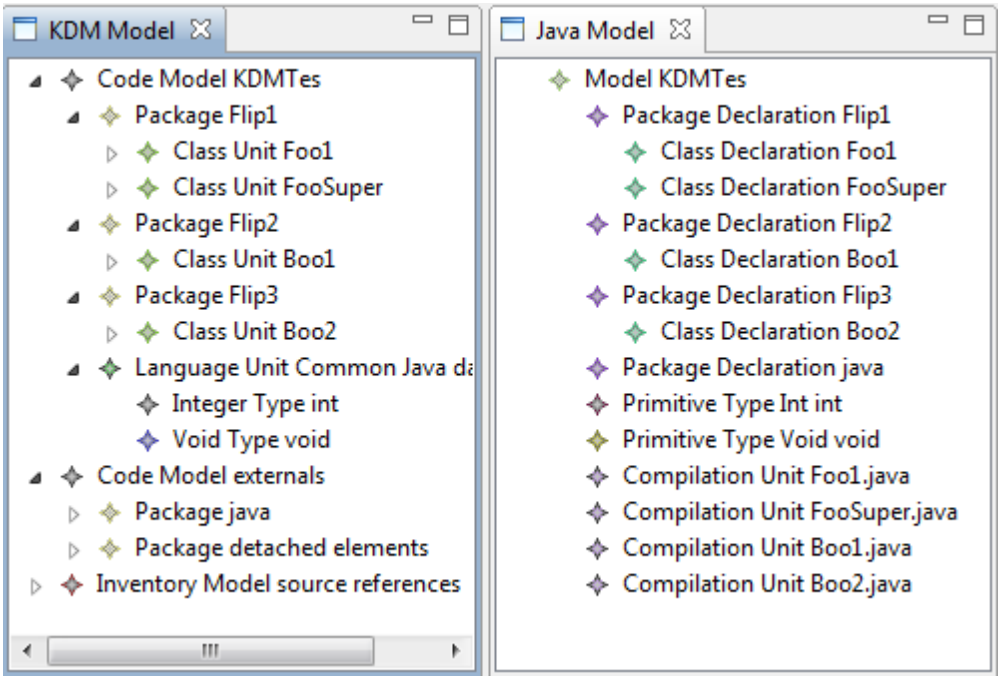


Figure 6.3 Tree Viewer for KDM and Java Model

In order to use the jface treeviewer, SharpVisualizer create *ModelViewcontentProvider* which provides the contents from loaded model and *ModelViewLabelProvide*r which provides the label to the tree view elements. Figure 6.4 code snippet how to use the Tree viewer to visualize the model.

```
public void createPartControl(Composite parent) {
    // TODO Auto-generated method stub
    viewer = new TreeViewer(parent);
    viewer.setContentProvider(new ModelViewContentProvider());
    viewer.setLabelProvider(new ModelViewLabelProvider());
    //LoadKDMModelTree("");
    URI uri = URI.createFileURI(new File("KDMTest.kdm").getAbsolutePath());
    Resource resource = Activator.resourceSet.getResource(uri, true);
    viewer.setInput(resource.getContents().get(0));
    getSite().setSelectionProvider(viewer);

}
```

Figure 6.4 Code Snippet to use the Tree Viewer

createPartControl() method is very import method in each views extended from *org.eclipse.ui* plug-in. Whatever need to be visualized in the view should be defined within this method.

6.3 Abstract Syntax Tree View

Java-Model which contains AST of complete java source code is used to visualize the AST. Following algorithm shows how to extract an AST for a specified class and generate the graph AST graph content. Java-Model is an Ecore model which contains data in XML format. This model can be browsed using EMF model browser or using simple XML processing.

```
<ownedElements xsi:type="j2se5:ClassDeclaration"  Name="Flip1.Foo1">
    <modifiers visibility='public'/>
    <bodyDeclarations xsi:type="j2se5:FieldDeclaration" name='f1'>
      <modifiers visibility="private'/>
    </bodyDeclarations>
    <bodyDeclarations xsi:type="j2se5:FieldDeclaration" name ="f2">
      <modifiers visibility="private'/>
    </bodyDeclarations>
    <bodyDeclarations xsi:type="j2se5:MethodDeclaration"  name="Foo1" >
      <body >
        <statements xsi:type="j2se5:ExpressionStatement' '>
          <expression xsi:type="j2se5:Assignment" >
            <leftHandSide xsi:type="j2se5:NamedElementRef' >
            <rightHandSide xsi:type="j2se5:ClassInstanceCreation' >
              <method element='/>
            </rightHandSide>
          </expression>
        </statements>
      </body>
    </bodyDeclarations>
    <superClass  o element='//@ownedElements.0/@ownedElements.1"/>
</ownedElements>
```

Figure 6.5 XML view of Java-Model

```
Nodes[];
Connections[];
getRoot(); Nodes.add(root)
for(Node n: root.DirectChilds())
{
    e=  New Connection(root, n);
    Nodes.add(n);Connections.add(e)
    If(n.hasChild()
      {
       //Recursive Call
        Nodes.add(n);
        Connections.add(e)
          -------------
      }
}
```

Figure 6.6 AST Extractor Algorithm

Figure 6.5 shows the simple Java-Model generated using *MoDisco*, and Figure 6.6 Shows the algorithm to process the data model and retrieves nodes and connections for a tree like graph structure. This collection of nodes and connection will be passed to graph viewers for rendering. Figure 6.7 shows the resulting AST visualized using SharpVisualizer. Visualizing AST is achieved using XML representation of Java-model instead of using generated Ecore model. When visualizing AST it was easy to used XML processing than exploring java model using EMF utilities. Generated AST shows different types of node with different visual attributes mainly using different colour. And also SharpVisualizer provide information zooming feature like double click on a specific node will pop-up up a new window with code snippet related to that node. Generated result is shown in Figure 6.8.

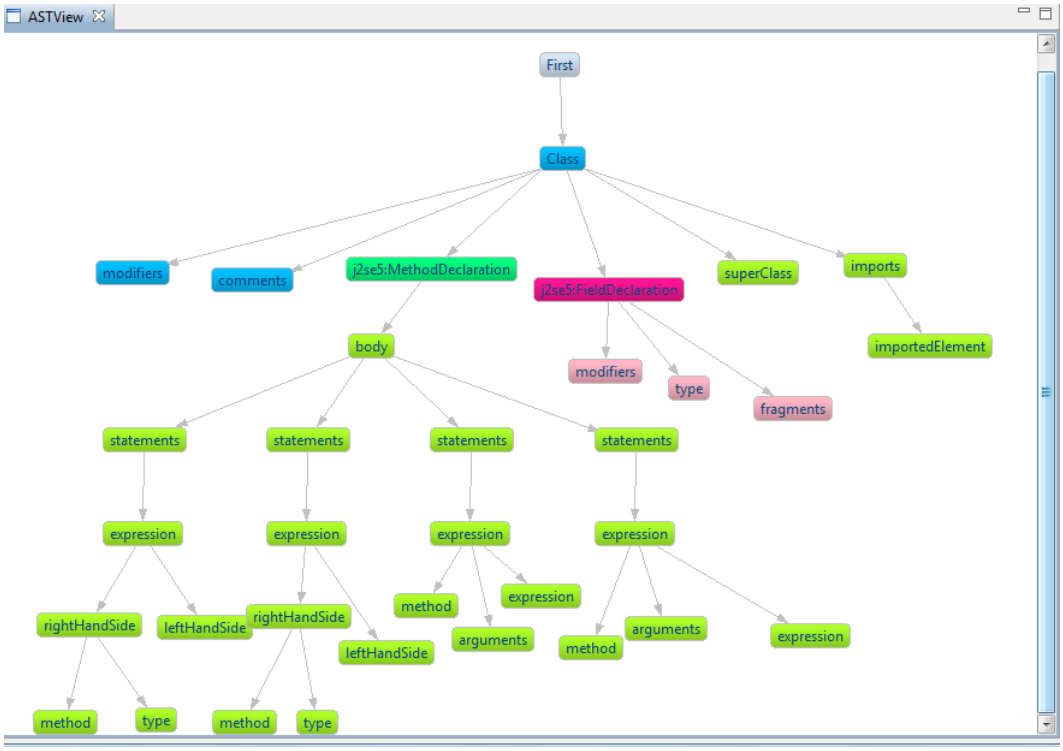


Figure 6.7 AST generated by SharpVisualizer

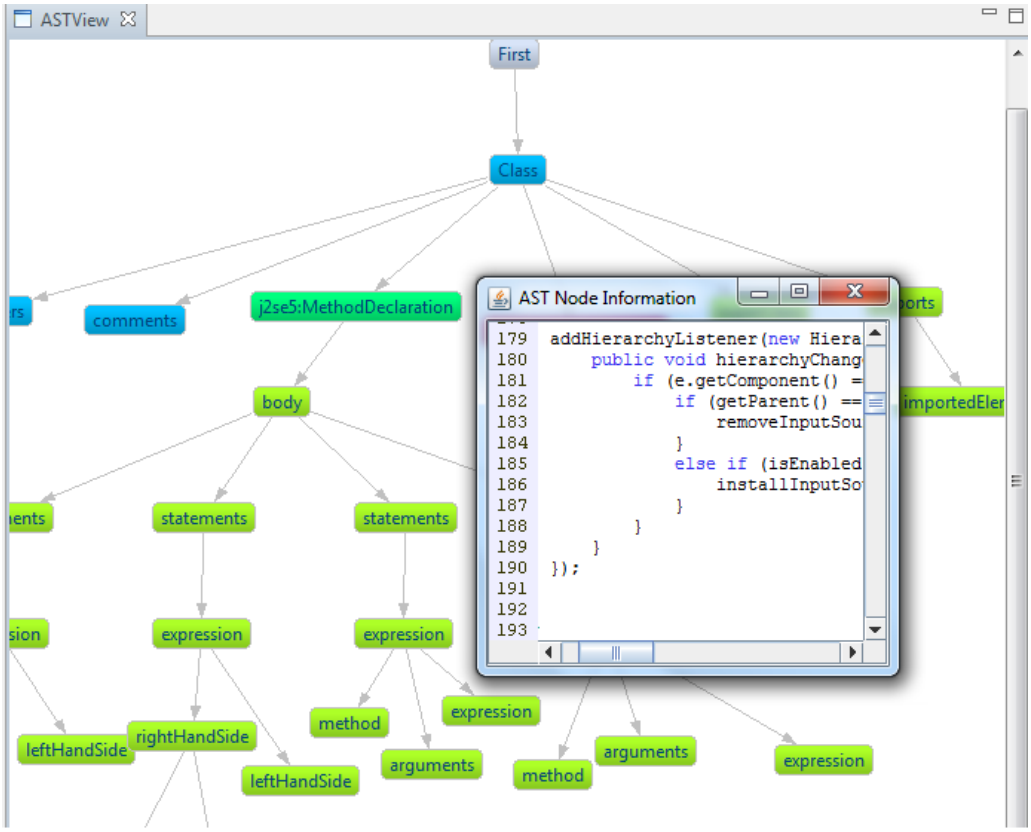


Figure 6.8 AST with Pop-Up, generated by SharpVisualizer

6.4 Package Dependency Graph View

SharpVisualizer visualize PDG using KDM. As explained in Section 3.1, generated L0 KDM models come with a Segment which usually contains two CodeModels and InventoryModel. CodeModel contains the information in hierarchical order as follows, class member such as StorableUnit, MethoUnit are nested into a ClassUnit or InterfaceUnit, these ClassUnit and InterfaceUnit are nested into Packages, Packages also can be nested to Packages. The entire root Packages are child elements of Code Model. Dependencies between model elements are derived using CodeRelation which is subclass of KDMRelationship is defined in KDM infrastructure layer [Section 3.1] and Appendix B.

```
The relation "x in* C" means that x is in container C or in some sub-container of C, transitively.
For relation R, let R' be the corresponding aggregated relation.
Given containers C1 and C2 and the relation R, let
P = {(x,y) : x in* C1 and y in* C2 and x R y}
That is, P is the set of pairs such that x is in* C1 and y is in* C2 and x R y.
Then
C1 R' C2 iff |P| > 0
C1 and C2 are related by the aggregated relation R' if and only if there is at least one pair in the set P.
The density of C1 R' C2 is then simply |P|, the size of the set P.
```

6.9 Algorithm to detect dependencies KDM elements [3]

The Figure 6.9 shows the algorithm to identify the dependencies based on AggregatedRelationship defined in KDM core package specification. See Appendix B for more details. Figure 6.10 shows theoretical the output of the algorithm.

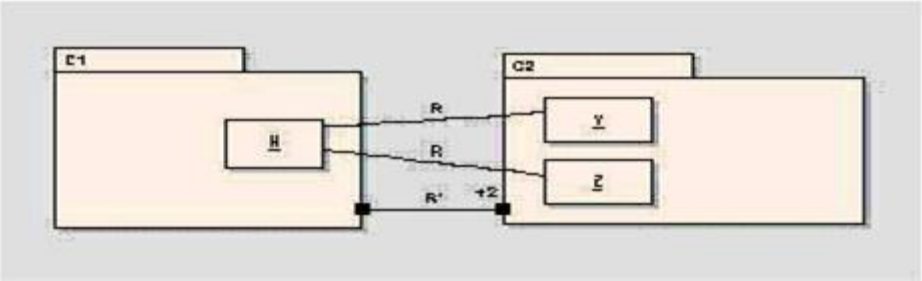


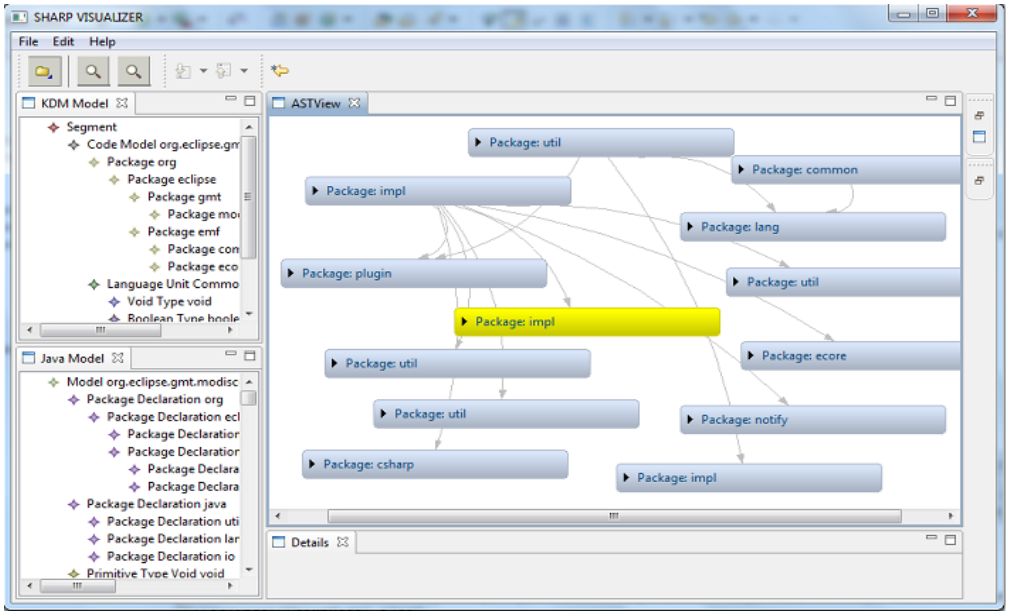
Figure 6.10 Theoretical Output of algorithm defined in Figure 6.9 [3]

But due to the limitation in generated KDM model which doesn't generate the Aggregated Relationship properties, I used a different method identify the relationships.

```
private void getCodeRelationsFromClassUnit(ClassUnit classUnit)
    {
        List relations = new ArrayList();;
        EList<AbstractCodeRelationship> codeRelations = classUnit.getCodeRelation();
        for(int z=0;z< codeRelations.size();z++)
            {
                if(codeRelations.get(z) instanceof Imports || codeRelations.get(z) instanceof Extends)
                {
                    List<EObject> connections = new ArrayList<EObject>();
                    if(codeRelations.get(z).getFrom()instanceof Package){
                        connections.add(codeRelations.get(z).getFrom());
                    }else{
                        connections.add(codeRelations.get(z).getFrom().eContainer());
                    }
                    if(codeRelations.get(z).getTo() instanceof Package){
                        connections.add(codeRelations.get(z).getTo());
                    }else{
                        connections.add(codeRelations.get(z).getTo().eContainer());
                    }
                    this.relations.add(connections);
                    }
            }
        return relations;
        }
}
```

Figure 6.11 Algorithm to identify the dependencies

Figure 6.11 shows the algorithm which was developed in order to derive the package dependency diagram using KDM model. Each ClassUnit and InterfaceUnit is analysed, if there is any Code Relation elements with type of import, extends or implements, then retrieve them to and from elements from that relationship. After that find out the direct or indirect container of that elements make the connection between then. Direct and Indirect package are linked using nested nodes. Actual algorithm doesn't shows the nested nodes and nested packages due to the limitation in ZEST due this, just shows the complete relationship this algorithm brings out all inner packages as separated nodes. Figure 6.12 shows the output generated by SharpVisualizer.



6.12 PDG generated by SharpVisualizer

6.5 Control Flow Graph View

SharpVisualizer visualize the CFG using Java-Model , as explained in Section 3.2 java model contains each element of java program. Each statement of a selected method of a class is analysed and flow of control between statements is identified based on Java Meta-Model which defines 22 different types of statements, such If, for, While, and 26 different type if expression such as prefix, method access...etc. Refer the Appendix B for more details about these statements and expression.

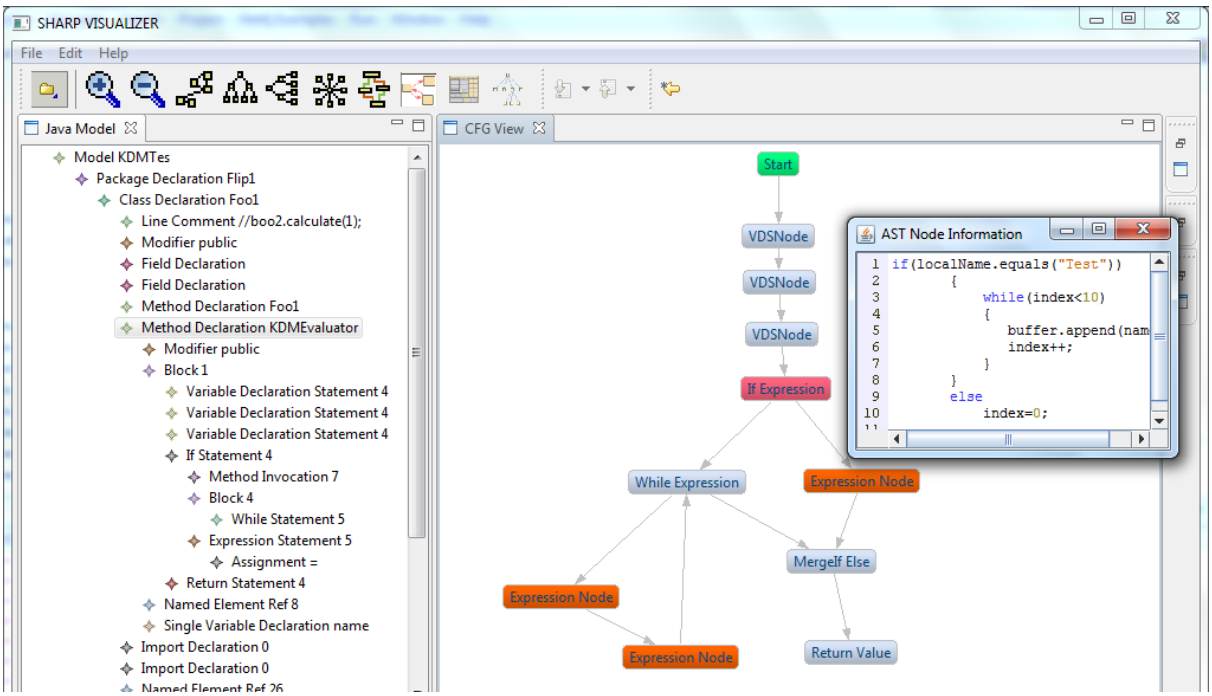
```
<bodyDeclarations xsi:type="j2se5:MethodDeclaration" originalColumn
  <modifiers visibility="public"/>
  <body originalColumnNumber="1" originalLastLineNumber="36" origin
    <statements xsi:type="j2se5:VariableDeclarationStatement" origi
    <statements xsi:type="j2se5:VariableDeclarationStatement" origi
    <statements xsi:type="j2se5:VariableDeclarationStatement" origi
    <statements xsi:type="j2se5:IfStatement" originalColumnNumber="
      <expression xsi:type="j2se5:MethodInvocation" originalColumnN
      <thenStatement xsi:type="j2se5:Block" originalColumnNumber="4
        <statements xsi:type="j2se5:WhileStatement" originalColumnN
      </thenStatement>
      <elseStatement xsi:type="j2se5:ExpressionStatement" originalC
    </statements>
    <statements xsi:type="j2se5:ReturnStatement" originalColumnNumb
  </body>
  <returnType originalColumnNumber="8" originalLastLineNumber="20"
  <parameters originalColumnNumber="28" originalLastLineNumber="20"
</bodyDeclarations>
```

Figure 6.12 Sample Java-Model Contents

```
Nodes[];
Connections[];
for(Statements s: MethodDeclaration.getStatements())
{
    checkStatementType; //ex: for, If, While
    CreatNode;
    setNodeProperty();
    CreateConnection;
    Add to nodes;
    Add to Connections;
}
```

Figure 6.13 More Abstract Algorithm to extract CFG graph

A simplifies algorithm is shown in figure 6.13, but actual algorithm is (java code is attached with Appendix C) based on Java-Model shown in Figure 6.12. Figure 6.14 shows the sample output. Mouse Double click on a node will pop up code snippet related that node.



6.14 CFG generated By Sharp Visualizer

6.6 Tree Map View

Tree Map view shows the all the source code elements in a single view with semantic zooming feature. It shows the hierarchy view of KDM model with nested nodes, root element will be Segment, and Packages are nested inside segment node, then each classes and interfaces in a package will be nested into that package node, then all the class members will be nested inside a class node. Nodes are retrieved by traversing the tree structure of the model. Figure 6.15 shows the part of the algorithms that used to extract the Tree Map from KDM model. It identifies the member of a class unit and identifies their types using KDM specification.

```
for(int i=0;i<codeItems.size();i++)
        {
            PSWTPath ciNode = PSWTPath.createRectangle(0,0,settings_0[1],settings_0[1]);
            if(codeItems.get(i) instanceof MethodUnit)
            {
                MethodUnit methodUnit = (MethodUnit)codeItems.get(i);
                switch(methodUnit.getKind())
                {
                case CONSTRUCTOR: ciNode.setPaint(java.awt.Color.red);break;
                case METHOD: ciNode.setPaint(java.awt.Color.ORANGE);break;
                case DESTRUCTOR: ciNode.setPaint(java.awt.Color.red);break;
                case ABSTRACT: ciNode.setPaint(java.awt.Color.red);break;
                case VIRTUAL: ciNode.setPaint(java.awt.Color.red);break;
                case OPERATOR: ciNode.setPaint(java.awt.Color.red);break;
                case UNKNOWN: ciNode.setPaint(java.awt.Color.red);break;
                }
                //Add to the class Node
                ccNode.addChild(ciNode);
            }
            if(codeItems.get(i)instanceof StorableUnit)
            {
                StorableUnit storableUnit = (StorableUnit)codeItems.get(i);
                switch(storableUnit.getKind())
                {
                case LOCAL: ciNode.setPaint(java.awt.Color.magenta);break;
                case GLOBAL: ciNode.setPaint(java.awt.Color.orange);break;
                case STATIC: ciNode.setPaint(java.awt.Color.blue);break;
                case REGISTER: ciNode.setPaint(java.awt.Color.magenta);break;
                case EXTERNAL: ciNode.setPaint(java.awt.Color.magenta);break;
                case UNKNOWN: ciNode.setPaint(java.awt.Color.magenta);break;
                }
                //Add to the classNode
                ccNode.addChild(ciNode);
            }
```

Figure 6.15 Part of Tree Map extracting algorithms

A complete algorithm is added in Appendix B. Figure 6.16.A to 6.16.D shows the different zooming level of Tree Map generated.

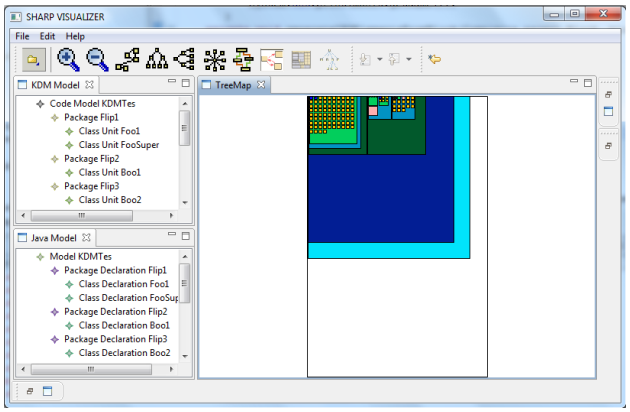


Figure 6.16.A Tree Map 1

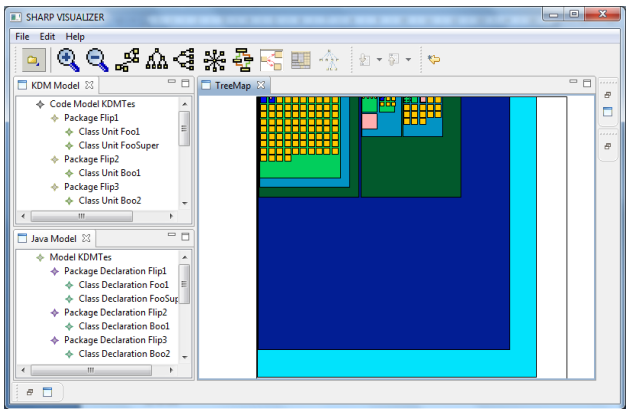


Figure 6.16.B Tree Map 2

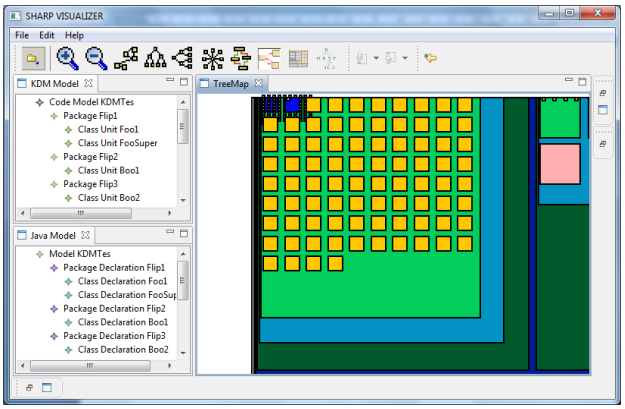


Figure 6.16.C Tree Map 3

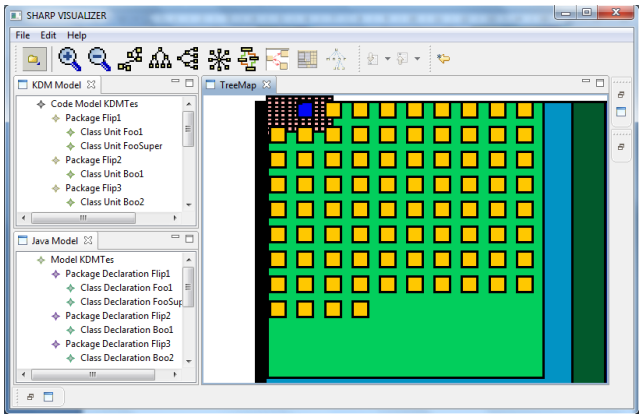


Figure 6.16.D Tree Map 4

Tree Map is implemented with semantic zooming feature provides by Piccolo2D graphics library, Mouse Click on a node will be magnified in order to get more information. Current Tree Map lowest level node is class members, but those nodes don't show the code snippet as defined in the requirements. There is bug in the layout algorithm to be fixed which cause node listed outside the parent node.

6.7 Fisheye Viewer

SharpVisualizer provide other type of Zoom able viewer called Fisheye viewer. The UI is implemented successfully; Data binding with these grids have to be done. Added in future work.

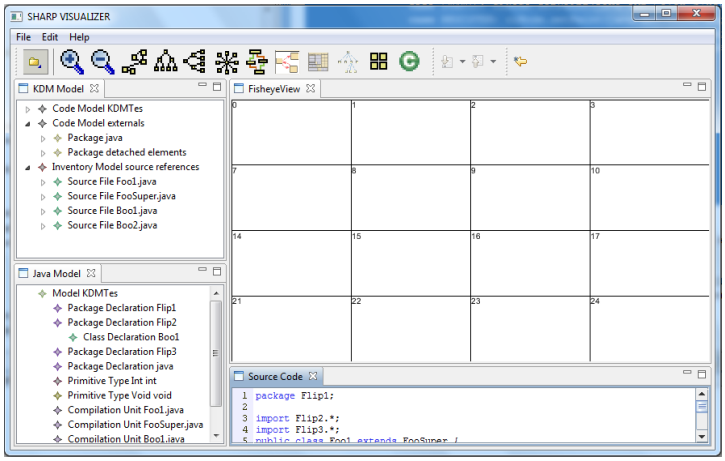


Figure 6.17 Fisheye Viewer (Initial)

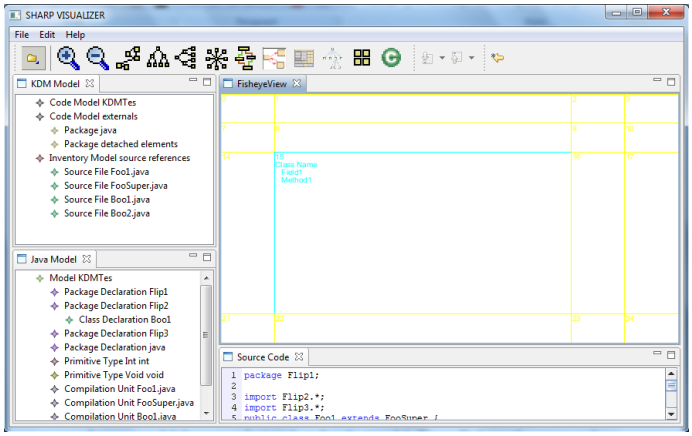


Figure 6.18 Fisheye Viewer (Zoomed In)

6.8 Source code Viewer

Source Code viewer shows original source file which is derived from KDM Inventory model or Java Model compilation unit or XML representation of source code. KDM's Inventory model provides the link between model elements and original artefact (Section 3.1). Figure 16.9 shows the source code viewer with syntax highlighter.

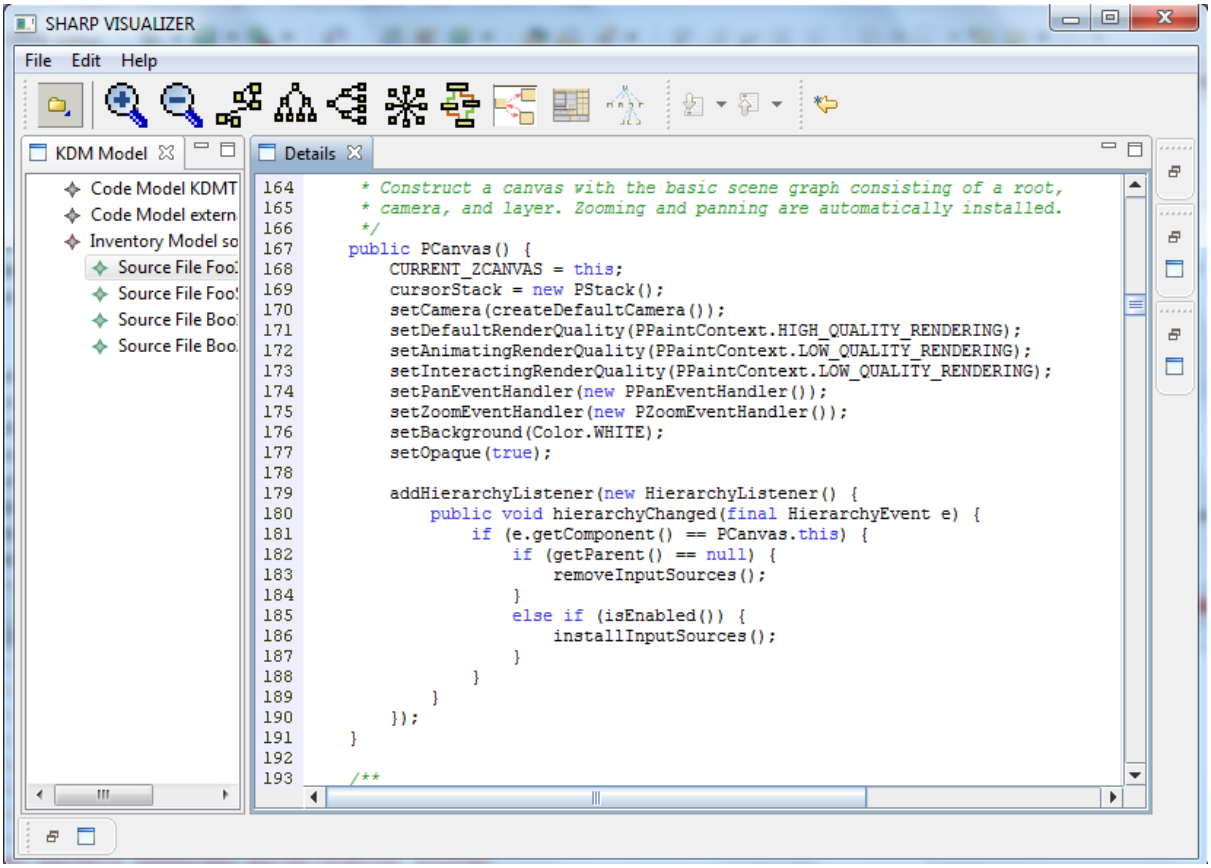


Figure 6.19 Source viewer in SharpVisualizer

Java syntax highlighting feature is implemented using external library called "jsyntaxpane" which provides the laxer to identify elements in java source code [38].

6.9 System Testing

This section gives the system test carried out in order to check the features implemented in SharpVisualizer. The Table 6.1 shows the test cases and their expected and actual outputs.

ID	Test Case	Description	Expected output	Actual Output	Result
01	Load KDM Model	User should load the existing KDM models via open file dialog.	KDM model should be loaded and shown in model viewer.	KDM model loaded successfully, and shown in viewer.	PASS
02	Load Java-Model	User should load the existing Java models via open file dialog.	Java model should be loaded and shown in model viewer	Java model loaded successfully, and shown in viewer.	PASS
03	Load XML version of Java-Model	User should load the XML version of existing Java models via open file dialog.	XML file should be loaded without any exception	XML file loaded successfully.	PASS
04	Generate AST	Click on AST command to visualize AST graph	AST graph should be rendered.	AST is generated successfully	PASS
05	Check code Snippet of Node	Double Click on AST node, should pop up the code snippet of the node	Mouse Double Click, should pop-up the code snippet.	Relevant code pop-up successfully	PASS
06	Generate PDG	Click on PDG command should generate PDG	PDG should be rendered	PDG is generated successfully	PASS
07	Generate CFG	Click on CFG command should generate CFG	CFG should be rendered	CFG is generated successfully	PASS
08	Generate Tree Map With zooming enabled	Click on Tree Map command should generate the Tree Map; click on a node should	Tree Map should be rendered. Click on node should	Tree Map rendered Issues with zooming and	Need Improvement.

		magnify the node.	expand that	layout	
09	View source code viewer	Click on source file command should display the selected source file	Click on source file command should display the selected source file	Source code displayed successfully	PASS
10	Different layout for Graphs	Click on different layout command should change the layout	Layouts should be changed according to the command.	NO created , issues	Failed.
11	Dynamic selection of model element	Selecting a element in model view should update the view	Not Implemented	Not Implemented	Not Implemented.

Table 6.1 System Test Cases

PART III

THE EVALUATION

CHAPTER 7

Evaluation

The goal of this project is to develop a software prototype to visualize the java source code as graphs with different information zooming mechanism for program comprehension. In the previous chapters I demonstrate the possibility of visualizing software source code with Knowledge Discovery Meta-Model specification defined by OMG's ADM and Eclipse MoDisco project which discover model from existing software artefacts. To prove that visual zooming over software source code base on defined solution is viable, I revisit the defined objectives.

- Graphical Representation of AST
- Graphical Representation of PDG
- Graphical Representation of CFG
- Graphical Representation of Tree Map

This chapter evaluate that to which extend, each objective has been met. In Section 8.1 I discuss about how the objectives have been met and critical evaluation of technical achievement. In section 8.2 explains the limitation of developed solution and in section 8.3 I discuss about reflective analysis about the project like time management, decisions taken and progress made.

7. 1 Objective's Evaluation

The main objectives were visualizing java code using graphics and proper zooming mechanism over the information. I have demonstrated the feasibility of creating visuals using source code models and high level programming with the help of some external graphics libraries. Mainly used components are KDM specification [3], Java Meta-model [16] which gives the details about software artefact models and ZEST and Piccolo2D Graphics libraries for rendering the graphs with proper zooming mechanism. These graphics libraries were selected after the detailed feasibility study. However later stages of implementation phase, I had face few issues with selected tools and technologies which lead to alternative approaches to develop the solution. The main issue was zooming feature provided by ZEST, based on documents , research papers published about ZEST [39], ZEST provides zoom able user interfaces like SHrimP views [Section 2.3.1], Initial plans about zooming algorithms was

based on this mechanism, but later found out current version of ZEST library doesn't contains the zoom able user interface features. Other problem faced during the development is incomplete KDM model, the current KDM model generated by MoDisco is incomplete, it doesn't contains some important features defines in KDM specification.

Due to that I had go with alternative approaches. I explain the technical problems I faced during implementation of each objective.

- **Graphical Representation of AST**

  Visualizing AST of java class is successfully achieved using XML version of Java Model as input and Zest graphics library for rendering graphs. Section 6.1 shows the output generated by SharpVisualizer. It was comparatively easily to map an XML documents in to visual tree by traversing each nodes recursively in the XML. In other hand there was another option using exploring Ecore model of input java model. But using that option I had to write long and lots coding to analyse each element in the program. Anyhow visualizing CFG was achieved using Ecore model concepts [Section 8.3]. The difficulty of using XML representation of source code is , attributes of each element in XML documents and link between elements had to be achieved using pure XML processing. I would be much easy to get the cross references using Ecore models. Zest graphics library is used to render the AST, which give the facility to set the node properties such as colour, image and shape. And also it gives interfaces to link the actual artefact with the nodes using setData methods. Due to the limitation about zoom able user interfaces, SharpVisualizer gives other mechanism to show the information about the node. Double click on a node will pop up code snippet related to that node.

- **Graphical Representation of PDG**

  Visualizing PDG of a java project is successfully achieved using KDM model and Zest graphics framework. Section 6.2 shows the output generated by SharpVisualizer application. KDM specification provides relationships between model elements [Section3.1, Section 6.2] using AggregatedRelationship [Appendix A], But KDM model generated by MoDisco doesn't contains information about this relationship. Due to this issue I went with the alternative approach to get the dependencies between packages using CodeRelation defined in CodeModel [Appendix A]. And another big

issue was a bug in Zest graphics library. Zest graphics library provides feature to create Nested node link diagram. But current Zest build doesn't show the nested nodes in a container but enables to add the child elements. Figure X and Y shows the code snippet and output which gives the evidence of this issue. Each graph container added child node, but there not visible. Due to this limitation, all the nested packages are shows as separate nodes in PDG.

```
GraphContainer c1 = new GraphContainer(g, SWT.NONE);
c1.setText("Canada");
GraphContainer c2 = new GraphContainer(g, SWT.NONE);
c2.setText("USA");

GraphNode n1 = new GraphNode(c1, SWT.NONE, "Ian B.");
n1.setText("dshgddhggdsdghsd");
n1.setSize(200, 100);
GraphNode n2 = new GraphNode(c2, SWT.NONE, "Chris A.");
n2.setTooltip(tooltip);

GraphConnection connection = new
 GraphConnection(g, ZestStyles.CONNECTIONS_DIRECTED, n1, n2);
connection.setCurveDepth(-30);
GraphConnection connection2 = new
 GraphConnection(g, ZestStyles.CONNECTIONS_DIRECTED, n2, n1);
connection2.setCurveDepth(-30);
```

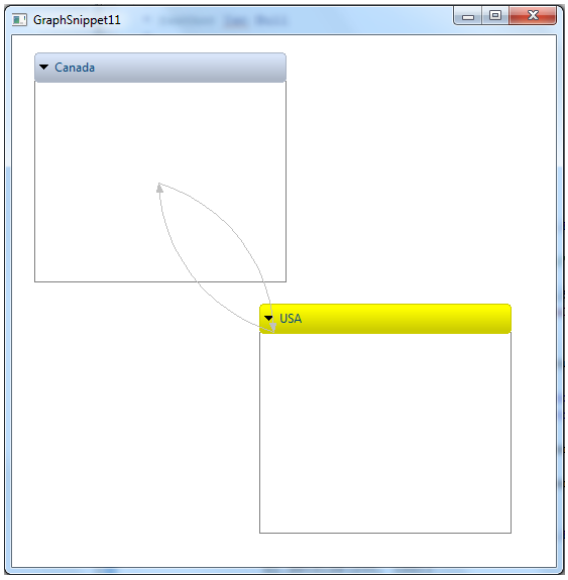


Figure 7.1 Zest Graph Container Example

- **Graphical Representation of CFG**

Visualizing CFG also successfully achieved. The proposed way to visualize the CFG by examining each statement in method declaration of java class using Java Meta-Model specification. But examining every possible statement is really huge task and takes much time. Because Java Meta-Model defines 22 types of statements [Section 3.2, Appendix A]. SharpVisualizer implements basic prototype of CFG using few types of statements to prove the feasibility of visualizing CFG by proposed method. Section 6.3 shows the CFG generated by SharpVisualizer for a method unit with If, While statements. CFG also visualized using ZEST. Due to the limitation in zooming mechanism in Zest, SharpVisualizer uses alternative approach to show code snippet like in AST.

- **Graphical Representation of Tree Map**

This objective was added later part of project due to the limitation in ZEST. SHrimP views are really popular in information visualization community. I was initially impressed with its zooming mechanism, and planned to develop that kind of zoom able user

interfaces in SharpVisualizer. But later only I found out that ZEST doesn't support that view anymore. So I decide to implement my own view to propose the semantic zooming algorithm using Zoom able user interfaces. I read about ShrimP and found out that SHrimP was developed using Piccolo2D graphics library. Tree Map view also successfully achieved except some issues in layout algorithms and magnifying issues. Which can be corrected but due to the time constraint I could not get it done. Section 6.5 shows Tree Map generated by SharpVisualizer. The figures X, Y, Z shows different level magnifying the nodes.

Except these core objectives other one was providing enabling navigation and information exploring feature like zooming, navigation, fisheye viewer and source code view. These objectives have been achieved while implementing above objectives. But different feature were developed with different objectives, could not get consistent throughout each graphs due to the issues faced with graphics libraries.

Most of the non-functional requirements also achieved such use of existing viewers such as Eclipse RCP views, ZEST graph viewers, to increase the performance while loading and serving models, SharpVisualizer use performance options provided by EMF. And SharpVisualizer offers different graph layout algorithms such as Tree layout, spring layout, Radial layout and Horizontal tree layout provided by ZEST. The graphs generated using ZEST supports these layout algorithms.

7.2 Limitation

If we consider SharpVisualizer as a complete product, it contains some important limitation. Mainly providing a selection provider in KDM and Java Model tree viewer is not implemented. Due to this issue current SharpVisualizer don't support the dynamic changes of graphs by selecting different model element in model view. But current application gives demo each type of graphs by selecting a specific model element from loaded model. Any model element can be passed to graph generating methods implemented in *"org.eclipse.sharpviz.extractor"* package in implemented source code.

Other limitation is in tree map, prototype of tree map is implemented, but as I defined in requirements the lower level node in the Tree Map hierarchy is Method Unit and Storable Units. These nodes are supposed to show code snippet of that node. But current Tree Map doesn't implement to shows the code snippet. Other limitation in tree map is layout algorithm

which was developed by me, have some issues which cause the layout problem when increase the hierarchy level. However this tree map provides the semantic zooming feature like SHrimP views. Other limitation is link between multiple views this functionality is not implemented according to the plan due to time constraint.

7.3 Reflective Analysis

Even though I initially started this project with program parser based approach using Microsoft Dot Net framework and C# programming language, after the literature studies I came across the OMG's software modernization standards and Eclipse's MoDisco to discover models, I started my study based on these tools and techniques, as these are new and evolving, it will challenging to learn and try new thing than going with old approaches. Actually using these tools and standards was much improved technique than parser based approaches that need extensive domain specific programming such as customizing program parser, defining XML Schema for represents source code. So I decided to switch the approach to develop this tool using Eclipse Framework and MoDisco components which completely based on java. Due to this there are some changes in initial objectives proposed, such as visualizing class diagram was removed, because MoDisco already provide feature to generate UML diagrams from KDM and Java models. Instead I added new objectives like visualizing CFG and Tree Map. Due to the switch to different technology, getting familiar with new tools and techniques took a reasonable period which heavily affected the project plan and time line. Due to the time constraint, identifying Software Design pattern, Visualizing call graph objectives have been left out. It can be developed in future.

Based on chosen approaches, technology and tools, main objectives of the project have been achieved successfully. But still achieved objectives have some minor issues due to issues and limitation in used tools and technologies.

Other main factor which affects the time line is issues in graphics libraries and model generated by Eclipse MoDisco. The used graphics libraries ZEST and Piccol2D are purely open source which contains incomplete documents. Some Initial plan was made based on feature provides by these libraries, but at later stages, issues and current limitation which are not available in documents lead for alternative approaches. Section 8.1 gives an example for such issue faced.

Finally, Eclipse MoDisco is still understudied area, standards and meta-models are still evolving, and most of the defined discoverers and meta-models are lacking in documentations which affected the implementation whenever face some problems or difficulties , it was difficult find the root cause.

CHAPTER 8

Conclusion

This report proposes an approach to develop a software visualization tool with visual zooming over the information. A number of software visualization tools and techniques have been proposed in the literature. Even though most of them were developed with the intention of visualizing software source codes, each of them uses different analysing and rendering techniques. I tried to use the evolving software modernization standards defined by OMG and discovering data models from existing software artefacts based on model driven approach in order to make the visualization of source code rather than going with existing parser and XML/XMI based techniques for software visualization.

I achieved most of core objectives successfully, mainly visualizing AST, CFG, and PDG with some information zooming mechanism as well as Tree Map and Fisheye view with more effective semantic zooming features like SHrimP views that is most popular visualization tool because of its ZUIs. But during the development phase, like in the real world software development, I faced many difficulties and limitation which was main obstacles to achieved defines goals more effectively. Due to the switching the technology and concept while project in progress, I had to tackle the time line, even though I achieved most of the defined objectives, I failed to make the tools as complete product. More preciously dynamic selection of contents, update the views, link between different views are not implemented, that's why end product lagging in usability. But the issues are not really big; they can be fixed using some extensive programming, due to the time constraint I could not make it.

Otherwise I personally feel that I came up with something new in software visualization area, I could not find any related studies using the software modernization standards and tools.

8.1 Future work

As I said the tools is not finished yet, still there are missing feature that has to be added in future. Importantly use of ZUIs for all the graphs and provides smooth zooming facility with hierarchical view of information. That will fulfil the main goal of the project. In other hand software design pattern detection, customization of presentation, data, and behaviour of the visualization can be added that will enable the user explore information in customized way.

9 References

[1] OMG's Architecture Driven Modernization *http://adm.omg.org/*.

[2] Eclipse MoDisco Project *http://www.eclipse.org/gmt/modisco/*.

[3] Knowledge Discovery Meta-Model http://www.kdmanalytics.com/kdm/.

[4] Benjamin , Jesse Grosjean and Jon Meer "Toolkit Design for Interactive Structure Grapics"

[5] Jean Marie Favre "*A Flexible Approach to Visualize large Software Products*".

[6] B.Price, Beacker, Small I.S *" A Principled Taxonomy of Software Visualization".*

[7] Robert Ian Bull "*Model Driven Visualization: Towards a Model Driven*"

[8] SHrimp *"http://www.thechiselgroup.org/shrimp".*

[9] Creole *http://www.thechiselgroup.org/creole*.

[10] Portable BookShelf *http://www.swag.uwaterloo.ca/pbs/*.

[11] Grok *http://www.yworks.com/en/products_yfiles_about.html*.

[12] LSEdit www.atxsoftware.co.uk.

[13] Model-Driven Architecture http://www.omg.org/mda/

[14] Tree Map http://www.cs.umd.edu/hcil/treemap/

[15] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides "*Design Patterns: Elements of Reusable Object-Oriented Software"*

[16] Java Meta-Model http://wiki.eclipse.org/MoDisco/Components/Java/Documentation/0.8

[17] Eclipse Modeling Framework "http://www.eclipse.org/modeling/emf/"

[18] Eclipse Graphical Editing Framework http://www.eclipse.org/gef/

[19] Maletic, J.I., Marcus, A., and Collard, M.L. (2002) *"A task oriented view of software visualization."*

[20] Marian Petre and Ed De Quincey "*Gentle over view of Software visualization*".

[21] Andrew Sutton, Jonathan I. Maletic "*Recovering UML class models from C++: A detailed explanation"*

[22]JavaDiscoverer

http://wiki.eclipse.org/MoDisco/Components/Java/Documentation/0.8#Java_Discoverer

[23] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, Christian Wende *"Construct to Reconstruct—Reverse Engineering Java Code with JaMoPP"*

[24] EMF http://www.eclipse.org/modeling/emf/

[25] Spoon http://spoon.gforge.inria.fr/

[26] Fujaba http://www2.cs.uni-paderborn.de/cs/ag-schaefer/Lehre/PG/FUJABA/

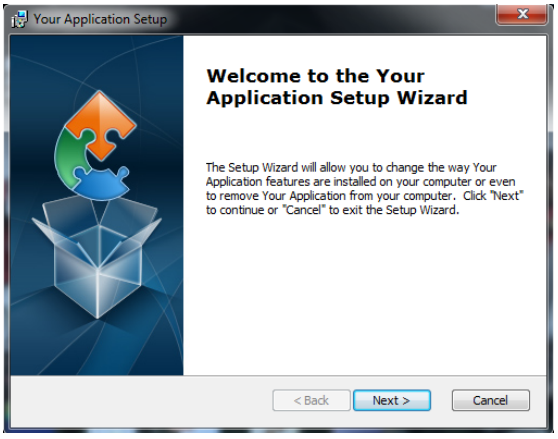
[27] Netbeans http://netbeans.org/

[28] Andrian Marcus Louis Feng Jonathan I. Maletic *"3D Representations for Software Visualization"*
[29] MoDisco KDM to UML Converter
http://www.eclipse.org/gmt/modisco/infrastructure/KDMtoUML2Converter/
[30] Petre, M. (2002) "*Mental imagery, visualisation tools and team work*"
[31] Java beans http://java.sun.com/j2se/1.4.2/docs/api/java/beans/package-summary.html
[32] Zest http://www.eclipse.org/gef/zest/
[33] Piccolo2D http://www.piccolo2d.org/
[34] SWT http://www.eclipse.org/swt/
[35] Draw2D http://www.eclipse.org/gef/overview.html
[36] jFace http://wiki.eclipse.org/index.php/JFace
[38] jsyntaxpane http://code.google.com/p/jsyntaxpane/
[39] R. Ian Bull, Casey Best, Margaret-Anne Storey "*Advanced Widgets for Eclipse*"

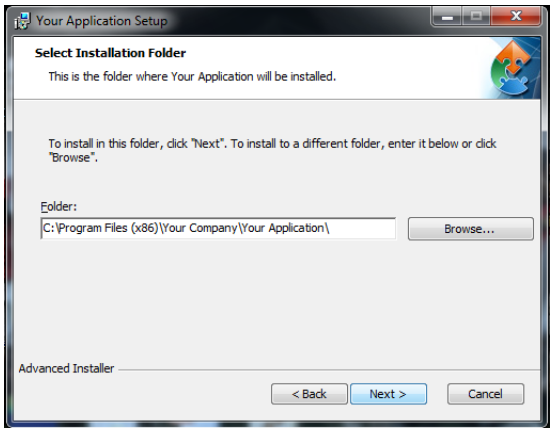
Appendix A

SHARPVISUALIZER USER GUIDE

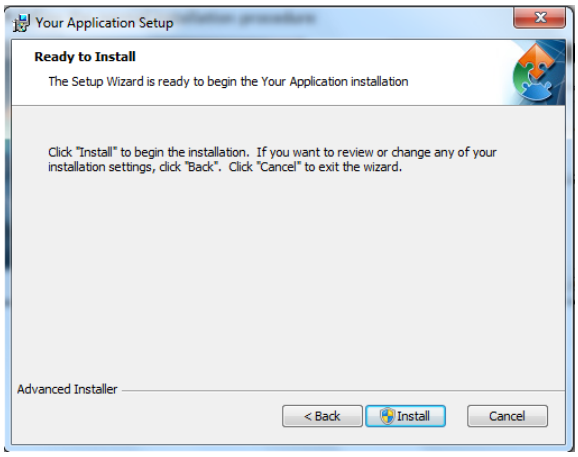
- Get Set Up file in SVN code folder
  - SVN→Code→trunk→SharpViz-Binaries→setup.msi
- Run→ follow the normal installation procedure
-



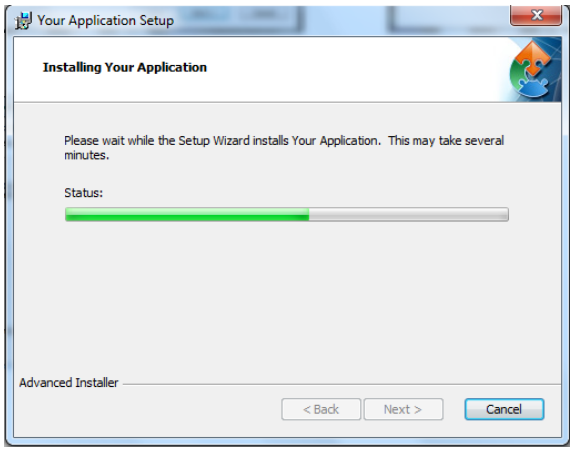
1



2

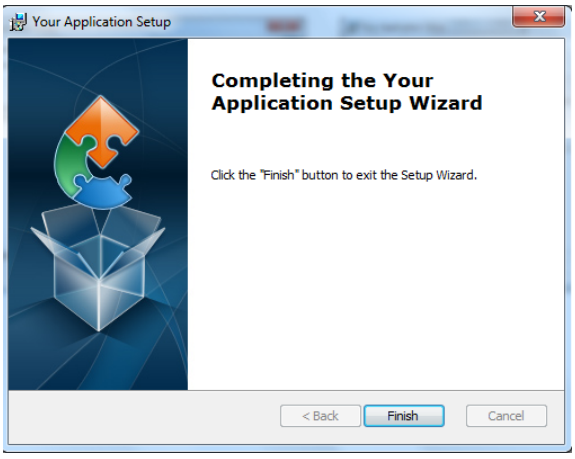


2

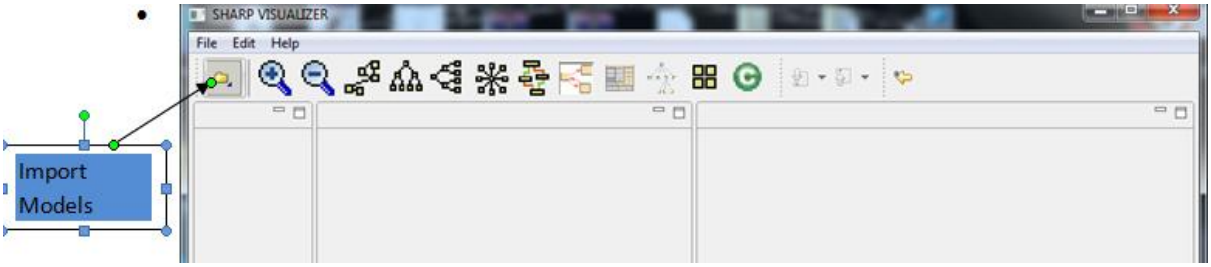


4

-



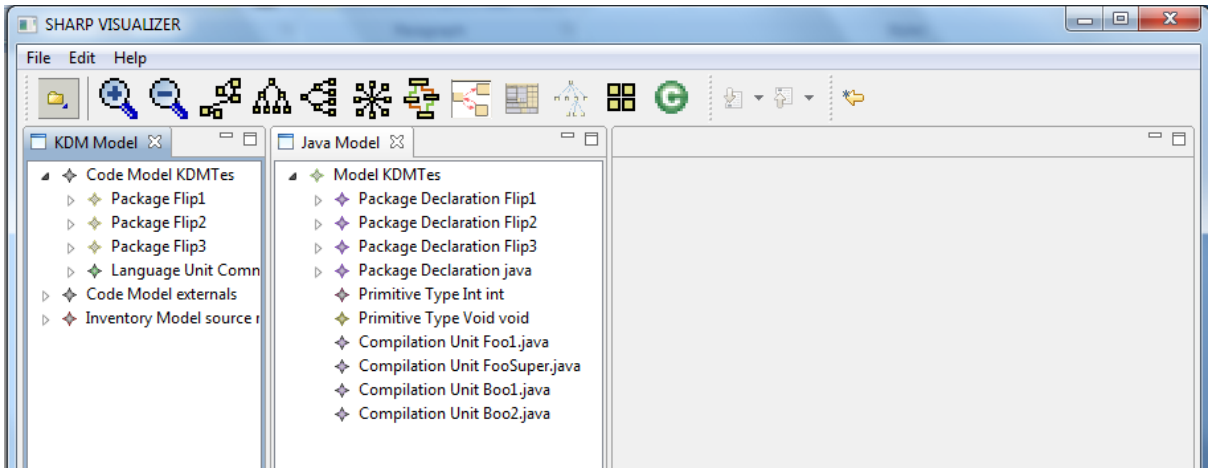
- Once the installation finished, you will find eclipse.exe in the installed location;
- Double Click on this: U will start the application:
-



- Using Open file option load input models which is in SVN →other→input or create your own model in MoDisco, make the copy of java model , changes the extension .j2se5 to .xml for XML data or(SVN→code→trunk→SharpViz-Binaries)
  The load one by one
  - KDMTest.kdm -to visualize the PDG, TreeMap
  - KDMTest.j2se5 -to visualize the CFG
  - KDMTest.xml [XML version of Java Model] This file is must, in order to visualize the AST.

  These are sample inputs in SVN.

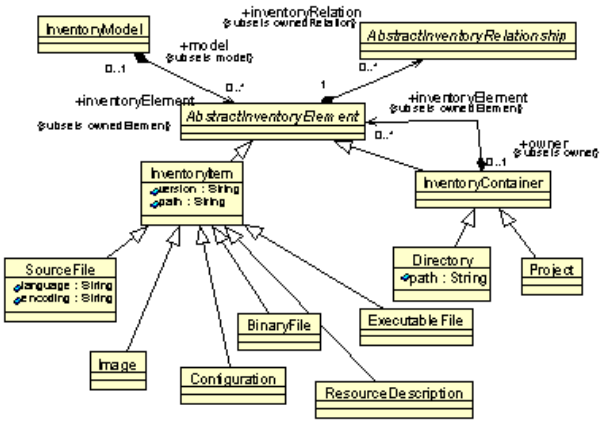
- Once U loaded successfully, Model viewers will display the model tree view
-



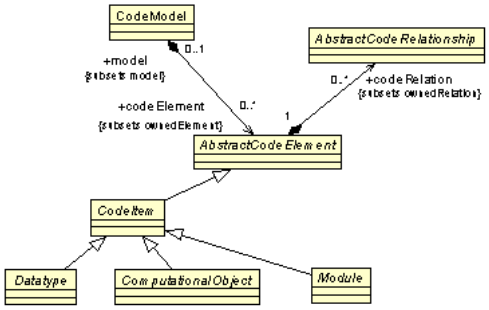
- The click the required command in Cool Bar to view the Graphs.
- AST and CFG → double click on graph node will pop up the source code.
- Mouse Click on Tree Map or Fisheye viewer will magnify the selected node.
- Click on source file command will display the source file.

Appendix B

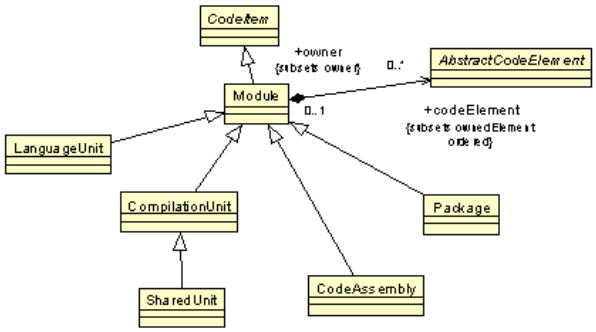
1. KDM Specification Class Diagrams [3]



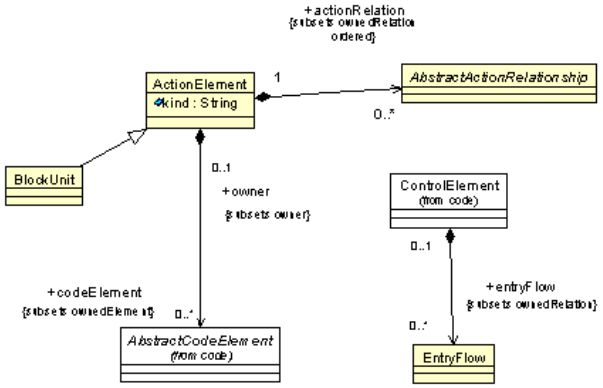
KDM Source Package: Inventory Mode Class Diagram



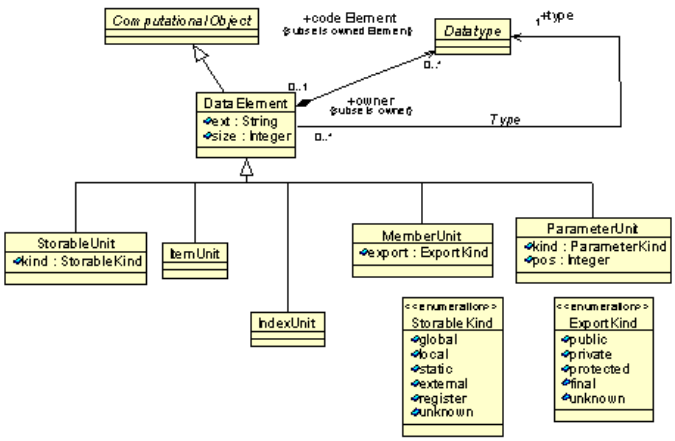
KDM: Code Package : CodeModel Class Diagram



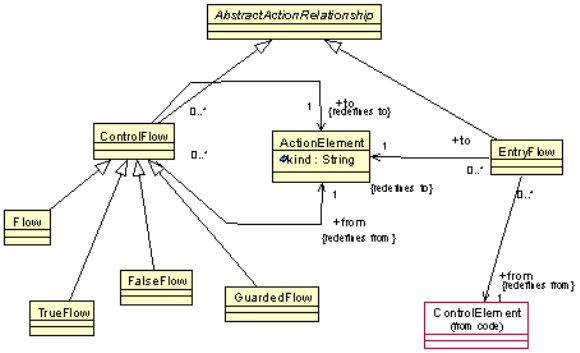
KDM Code Package: Module Classes



KDM Action: Action Element

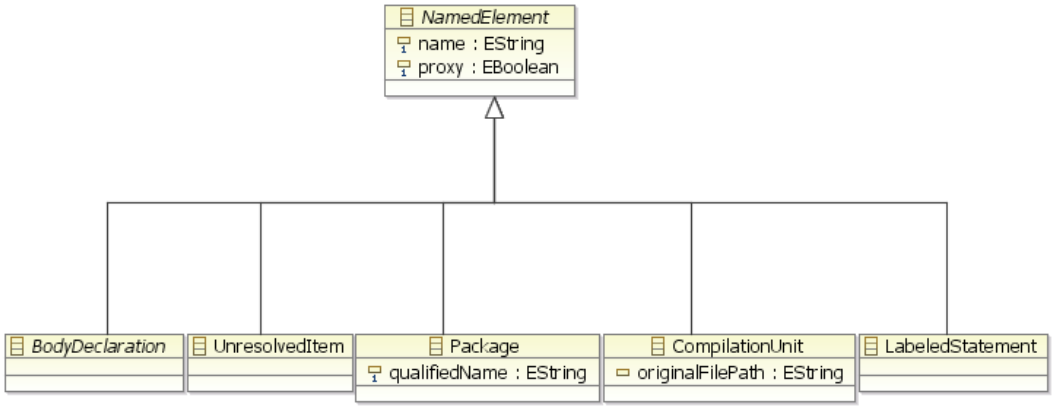


KDM: Code: Data Elements

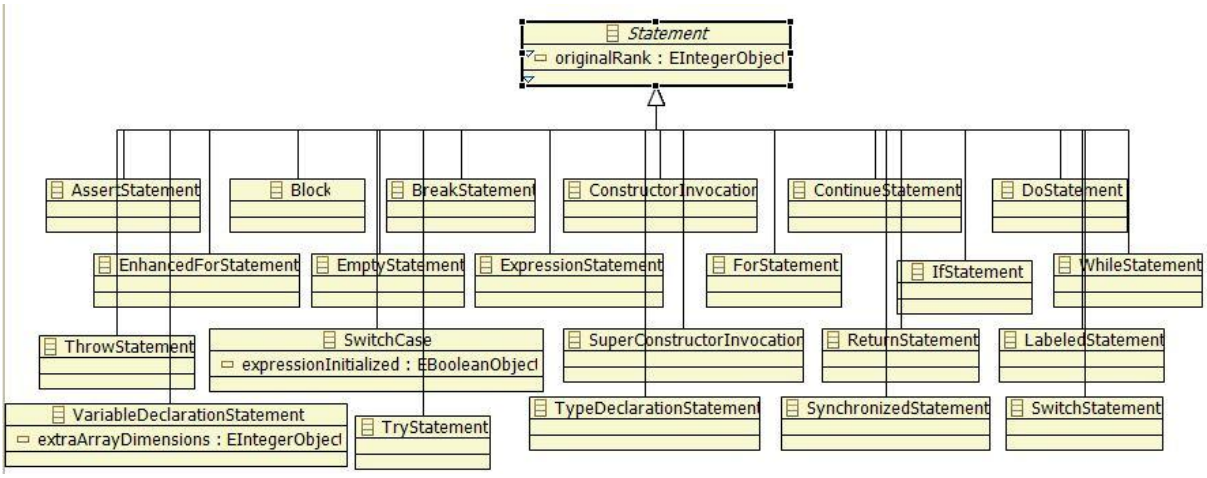


KDM : Action: Control Flow

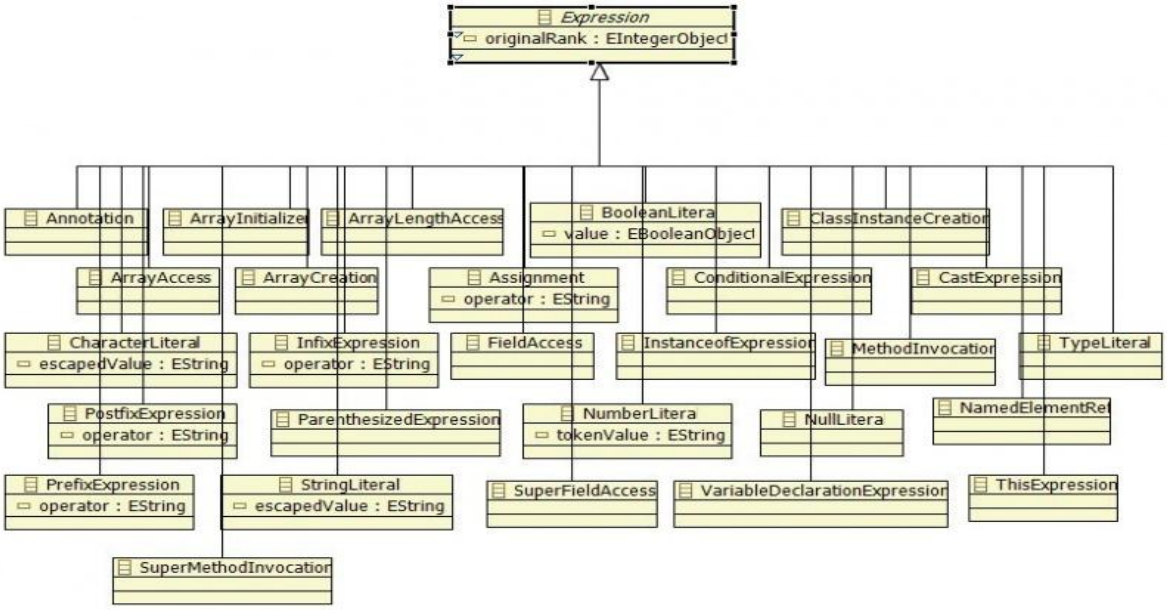
3. Eclipse MoDisco : Java Meta-Model Class Diagrams.



Named Elements & its heirarchy



Java Statements class diagrams



Java Expressions

Appendix C

CFG Extractor, Checking each type of statement:

```
private GraphNode generateCFGElementsFromStatements(Composite parent,Statement statement,Graph graph)
    {
        if(statement instanceof Block){}
        if(statement instanceof DoStatement){}
        if(statement instanceof ForStatement){}
        if(statement instanceof WhileStatement){}
        if(statement instanceof IfStatement){}
        if(statement instanceof ReturnStatement){}
        if(statement instanceof LabeledStatement){}
        if(statement instanceof SwitchStatement){}
        if(statement instanceof SuperConstructorInvocation){}
        if(statement instanceof TypeDeclarationStatement){}
        if(statement instanceof AssertStatement){}
        if(statement instanceof BreakStatement){}
        if(statement instanceof ConstructorInvocation){}
        if(statement instanceof ContinueStatement){}
        if(statement instanceof SynchronizedStatement){}
        if(statement instanceof ExpressionStatement){}
        if(statement instanceof EmptyStatement){}
        if(statement instanceof SwitchCase){}
        if(statement instanceof TryStatement){}
        if(statement instanceof VariableDeclarationStatement){}
        if(statement instanceof ThrowStatement){}
        if(statement instanceof EnhancedForStatement){}

        return lastConnectedNode;
    }
```